Parallel and distributed algorithms for pattern matching in big graphs

Sarra Bouhenni

N° d'ordre NNT : 2021LYSE1260

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée au sein de
l'Université Claude Bernard Lyon 1 en cotutelle avec **l'Ecole Supérieure d'Informatique - Alger**

Ecole Doctorale N° 512
Informatiques et Mathématiques (InfoMaths)

Spécialité de doctorat : Informatique

Soutenue publiquement le 01/12/2021, par :

Sarra BOUHENNI

Parallel and Distributed Algorithms for Pattern Matching in Big Graphs

---

Devant le jury composé de :

Fatima SI-TAYEB Professeure, École Supérieure d'Informatique, Algérie	**Présidente**
Yahya SLIMANI Professeur, Université de la Manouba, Tunisie	**Rapporteur**
Olivier TOGNI Professeur, Université Bourgogne Franche Comté, France	**Rapporteur**
Salima HASSAS Professeure, Université Claude Bernard Lyon 1, France	**Examinatrice**
Hamamache KHEDDOUCI Professeur, Université Claude Bernard Lyon 1, France	**Directeur de thèse**
Nadia NOUALI-TABOUDJEMAT Directrice de recherche, CERIST, Algérie	**Co-directrice de thèse**
Saïd YAHIAOUI Maître de recherche, CERIST, Algérie	**Invité**

This work is dedicated to my mom

You have always wished that I become a doctor (precisely a physician)

I hope that becoming a doctor in computer science counts

I love you

Acknowledgements

I would like to thank my three supervisors without whom the completion of this project would not have been possible. Thank you Prof Hamamache Kheddouci and Dr Nadia Nouali for your support and guidance throughout my four years of PhD. Your vision and expertise helped me not only to progress as a researcher but also as a person.

My deepest appreciation and gratitude goes for my mentor and supervisor Dr Saïd Yahiaoui who taught me the real meaning of research. Thank you for being there during the hard times to appreciate even the little progress I made... Thank you for reminding me every time that there is always room for improvement.

I would also like to acknowledge the programs PHC Tassili and IDEX Lyon for funding this PhD.

I would also like to thank my colleagues in DTISI Lab and LIRIS lab for their interest and valuable comments on my work.

To my friends and family, you should know that your support and encouragement was worth more than I can express on paper.

To Nacera and Ouidad, thank you for your unconditional love and support.

To Anis, you were always there with a word of encouragement and listening ear. Thank you for your presence along the way and your curiosity and willingness to help me solve the problems I encountered.

Abstract

Graph Pattern Matching (GPM), usually evaluated through subgraph isomorphism, finds subgraphs of a large data graph that are similar to an input query graph. It has many applications, such as pattern recognition and finding communities in social networks. However, besides its NP-completeness, the strict constraints of subgraph isomorphism are making it impractical for GPM in the context of big data. As a result, relaxed GPM models such as graph simulation emerged as they yield interesting results in polynomial time. Moreover, massive graphs generated by mostly social networks require distributed storing and processing of the data over multiple machines. Therefore, the existing algorithms for relaxed GPM need to be revised to this context by adopting new paradigms for big graph processing, e.g. Think-Like-A-Vertex and its derivatives.

In this thesis, we investigate the use of distributed graph processing paradigms and systems in the evaluation of GPM queries. Our goal is to identify the programming models that are best suited for this problem. Furthermore, we study the existing GPM approaches, with more emphasis on the relaxed ones in the aim of proposing new parallel and distributed algorithms for relaxed GPM that guarantee linear scalability. Our contributions are summarized as follows. First, we propose a taxonomy of prior work on distributed GPM based on multiple criteria, such as the GPM model and the programming paradigm. Next, we introduce BDSim as a new model that captures more semantic similarities compared to the existing models while being feasible in cubic time. Besides, we design distributed vertex-centric algorithms that are adapted to the context of massive graphs for evaluating BDSim. Furthermore, we propose the first fully distributed and scalable approach for strong simulation, a relaxed GPM model that strikes a balance between flexibility and tractability. Finally, we propose the first efficient parallel edge-centric approach for evaluating graph simulation and dual simulation in distributed graphs. We validate the effectiveness and efficiency of our approaches through theoretical guarantees and reliable testing over synthetic and real-world graphs.

We confirmed in this thesis that different paradigms can be used in designing distributed GPM algorithms depending on the GPM model adopted. Indeed, algorithms for neighborhood-based models such as subgraph isomorphism and strong simulation perform better with a vertex-centric or subgraph-centric paradigm as the latter involves some data locality, while the most efficient algorithms for graph simulation and dual simulation are edge-based and offer linear scalability guarantees.

Keywords: *Big data, Big graphs, Pattern matching, Subgraph isomorphism, Graph algorithms, Parallel and Distributed computing.*

Résumé

L'appariement des sous-graphes (ASG) est un problème classique, souvent modélisé à l'aide de l'isomorphisme de sous-graphes. Il est utilisé dans différents domaines d'application tels que la reconnaissance de motifs et la détection de communautés dans les réseaux sociaux. Néanmoins, en plus du fait qu'il soit NP-Complet, l'isomorphisme de sous-graphes s'avère très strict pour l'ASG dans le contexte actuel des grands graphes. Par conséquent, de nouveaux modèles d'ASG relaxé ont apparu comme la *Graph Simulation*, permettant d'avoir des résultats intéressants dans un temps polynomial. De plus, les graphes massifs qui sont issus des réseaux sociaux requièrent un stockage et un traitement distribués sur plusieurs machines, d'où la nécessité de revisiter les algorithmes d'ASG relaxé en adoptant de nouveaux paradigmes, dédiés au traitement des grands graphes, notamment le Think-Like-A-Vertex et ses variantes.

Dans cette thèse, nous étudions l'intérêt des systèmes et paradigmes distribués de traitement des grands graphes dans l'évaluation des requêtes d'ASG. L'objectif est d'identifier les modèles de programmation qui sont les mieux adaptés pour ce problème. Par ailleurs, nous visons à proposer de nouveaux algorithmes d'ASG qui sont parallèles, distribués et offrant une scalabilité linéaire. Nos contributions se résument en quatre points : (1) nous proposons une nouvelle classification des approches distribuées d'ASG, en nous basant sur plusieurs critères tels que le modèle d'ASG et le paradigme de programmation, (2) nous introduisons le nouveau modèle d'ASG relaxé *BDSim* qui permet de mieux capturer les similarités entre les graphes, tout en ayant une complexité cubique. En plus, nous proposons des algorithmes distribués centré sommet pour l'évaluation de *BDSim* sur des grands graphes, (3) nous développons le premier algorithme scalable et complètement distribué pour évaluer *Strong Simulation*, un modèle d'ASG relaxé offrant un compromis entre la flexibilité et la faisabilité, (4) enfin, nous proposons la première approche parallèle et centrée arêtes pour évaluer *Graph Simulation* et *Dual Simulation* dans les graphes massifs et distribués. Nous validons les différents algorithmes proposés théoriquement et expérimentalement sur des graphes massifs synthétiques et réels.

A travers ce travail de recherche, nous avons confirmé que différents modèles de programmation peuvent être utilisés pour la conception d'algorithmes d'ASG et cela dépend du modèle d'ASG adopté. Effectivement, l'isomorphisme de sous-graphes et *Strong Simulation* sont des modèles basés sur la localité et le voisinage à plusieurs sauts, ce qui nécessite un paradigme centré sommet ou encore centré sous-graphe. En revanche, les algorithmes les plus efficaces pour évaluer *Graph Simulation* et *Dual Simulation* effectuent des traitements centrés arêtes et garantissent une scalabilité linéaire.

Mots clés: *Big data, Big graphs, Pattern matching, Subgraph isomorphism, Graph algorithms, Parallel and Distributed computing.*

Contents

List of Figures

List of Tables

List of Acronyms

BDSim Bounded Dual Simulation.

BFS Breadth First Search.

BSIM Bounded Simulation.

BSP Bulk Synchronous Parallel.

CC Connected Component.

D3S Distributed Scalable Strong Simulation.

D3S+ Distributed Scalable Strict Simulation.

DAG Direct Acyclic Graph.

DBS Double Simulation.

DFS Depth First Search.

DRS Dual Relaxation Simulation.

DSIM Dual Simulation.

GAS Gather-Sum-Apply-Scatter.

GPM Graph Pattern Matching.

GSIM Graph Simulation.

LSIM Limited Simulation.

Max-PG Maximum Perfect Subgraph.

MCS Multi-Constrained Simulation.

MPI Message Passing Interface.

PDSim Parallel Dual Simulation.

PGSim Parallel Graph Simulation.

RDD Resilient Distributed Dataset.

RDF Resource Description Framework.

RPC Remote Procedure Call.

SGSIM Strong Simulation.

SIMD Single Instruction Multiple Data.

SJS Surjective Simulation.

SSSP Single Source Shortest Path.

STSIM Strict Simulation.

STwig Star Twig.

SubISO Subgraph Isomorphism.

TLAV Think Like A Vertex.

TRS Time-Respecting Simulation.

TSIM Tight Simulation.

TXS Taxonomy Simulation.

URS Unidirectional Relaxation Simulation.

WSN Wireless Sensor Network.

Introduction

Motivations and challenges

With the rapid growth of the Internet, huge amounts of data are being generated with every passing minute. The challenges brought about by what is commonly known today as *"big data"* have led to the research community revising traditional techniques and algorithms that were designed for collecting, storing, analyzing, and visualizing data. According to an IBM report published in the year 2017 [139], 90 percent of the data in the world has been generated in the last two years alone; i.e. during 2015–2016. Incredibly, there are 2.5 quintillions new bytes of data produced every day, and such huge quantities of information cannot be stored by traditional forms of centralized storage. Indeed, distributed file systems are used everywhere from large scale databases to big data analytics platforms.

An important share of this data is represented in the form of graphs, which are known to be natural and flexible data structures that are effective in modeling complex relationships, interactions, and interdependence between objects. They are used for modeling datasets in a broad range of domains, such as biological networks, the World Wide Web and social media networks. There are a variety of graph problems that can be used in modeling use cases of different application domains. For example, we have the shortest path problem, graph traversals such as BFS or DFS, connected components, triangle counting, minimum spanning tree, graph pattern matching or the page rank problem used to rank the different nodes of a graph.

The very large graphs have been dubbed *"big graphs"*, which results in bringing new challenges in the different stages of the big graphs pipeline, whether it is in terms of distributed storage, distributed processing or visualization. Distributed processing of big graphs is the most crucial stage of this pipeline. It involves solving a specific problem through highly optimized and efficient algorithms that are distributed and parallel.

Therefore, they support data scalability and are able to harness the computing resources available on every computing machine.

In this thesis, we address Graph Pattern Matching (GPM), which refers to the problem of finding occurrences of a small pattern graph (a.k.a. query graph) in a relatively larger data graph. It is among the most analyzed problems in graph theory, and it has a wide range of applications like in-database analytics, biometric identification and social networks analysis. GPM queries are traditionally evaluated based on subgraph isomorphism that finds a bijective mapping between the vertices of the query graph and a subgraph of the data graph, such that query edges are preserved by this mapping. Since several decades, different subgraph isomorphism algorithms have been put forward, aiming to reduce time and space complexity and addressing the problems of specific graph classes, e.g. trees, bipartite graphs, and direct acyclic graphs. Nevertheless, subgraph isomorphism is known to have several limitations. It is an NP-Complete problem [49], which makes it impractical for massive graphs that may contain up to billions of nodes and trillions of edges. Indeed, the strict constraints imposed by this model are not interesting for the current real-world applications especially in social networks. Consequently, graph simulation [57, 32] has been considered as an alternative model that relaxes the matching constraints imposed by subgraph isomorphism, hence making it feasible in polynomial time. Furthermore, several other relaxed models emerged, aiming to reduce the time and space complexity while keeping an acceptable accuracy depending on the application domain and its requirements in terms of flexibility and response time. We cite for example bounded simulation [34], dual simulation and strong simulation [84]. These newly introduced models consider not only the topological structure of the graphs, but also the semantic information carried by labels and attributes on both the graph vertices and edges.

Besides, programming models are being used to design distributed graph algorithms such as Hadoop MapReduce or the Bulk Synchronous Parallel model (BSP). However, recent studies have shown that the general-purpose programming models, especially Hadoop MapReduce, are inefficient in processing distributed graphs due to the complexity of graph data as opposed to other forms of data. Big data processing in Hadoop MapReduce is based on I/O operations that are time consuming. In addition, the data locality is an intrinsic property of iterative graph algorithms. It increases the number of I/O operations making this category of frameworks unsuitable for graph processing at large scale. This is a reason why new paradigms, specially designed for big graph processing, appeared such as the Think-Like-A-Vertex (TLAV) model and its derivatives adopted

by Pregel [86], Giraph [44], GPS [108] and GraphX [143]. These paradigms consider the different properties of graph data and allow designing linearly scalable algorithms intended to run on clusters composed of dozens to hundreds of machines.

With the emergence of these distributed systems, important challenges need to be addressed in the field of graph pattern matching. The first challenge is distributed subgraph matching in general, while the second one is efficient and linearly scalable algorithms for relaxed subgraph matching. Even though the relaxed GPM models are appropriate for social network analysis due to their polynomial time complexity, only few works address them from a scalability point of view. In this thesis, we study the programming models that are used for distributed graph processing and their adaptation to the problem of graph pattern matching. We aim to propose, based on these models, new efficient parallel and distributed algorithms for evaluating relaxed GPM while considering the intrinsic properties of big graphs that are issued from real-world applications. The linear scalability of these algorithms will be proved through a detailed theoretical validation in addition to an experimental evaluation of their performance on several real graphs.

Contributions

To achieve the above mentioned objectives, we investigate in our first contribution the existing relaxed GPM models and compare them to subgraph isomorphism. This study allowed us to draw the relationships between these models and show how close they are to graph simulation and subgraph isomorphism. In addition, we study the existing graph processing paradigms to answer the following question; which programming models are better suited for the problem of GPM and whether the differences between these models affect the selection of the programming models that should be adopted or not? Furthermore, we conduct a thorough study over prior distributed graph pattern matching approaches to spot and discuss their shortcomings, notably in terms of scalability. Finally, we propose a new taxonomy of GPM algorithms based on multiple criteria, such as the GPM model involved and the programming paradigm adopted in designing the approach.

One conclusion drawn from this study is that strong simulation is an interesting extension of graph simulation that captures the topological structure of the query graph by bringing duality and locality. However, even though strong simulation allows preserving most of the important information of a query graph, this model has not been sufficiently covered in the context of massive graphs. Apart from the proposal of alternative models that remain limited in terms of scalability, i.e. strict simulation or tight simulation [38], few

research papers have paid attention to distributed strong simulation. For these reasons, we tackle the problem of scalability of strong simulation from three different angles. First, we propose an alternative GPM model that answers relaxed GPM queries in distributed graphs while capturing duality and locality (properties present in strong simulation) and offering scalability guarantees. Second, we design a new distributed and scalable algorithm for evaluating strong simulation with performance guarantees on attributed and dynamic data graphs. Finally, since dual simulation is an important building block in the evaluation of strong simulation, we propose the first parallel and highly scalable algorithm for dual simulation in distributed graphs (in addition to graph simulation). These contributions are detailed below.

Our main contribution in this thesis is the proposal of a new relaxed GPM model called Bounded Dual Simulation (BDSim), and that sits between graph simulation and subgraph isomorphism. BDSim strikes a balance between tractability and accuracy of the returned results as it captures perfectly the cyclic structure of the query graph while being feasible in cubic time. We validate the scalability of BDSim through theoretical guarantees in addition to the design and implementation of a distributed vertex-centric algorithm that evaluates BDSim on massive data graphs. Finally, we conduct extensive experiments on synthetic and real data graphs and compare our approach to prior GPM models.

The existing approaches of strong simulation are either centralized or distributed. However, even the distributed ones do not provide scalability guarantees. In our second contribution, we propose the first fully distributed and vertex-centric algorithm for scalable evaluation of strong simulation on massive graphs, called D3S. An extension of this approach to evaluate strict simulation, referred to as D3S$^+$, is also proposed. In addition to the theoretical guarantees on the convergence and correctness of this approach, the conducted experiments prove that D3S and D3S$^+$ can reduce the response time by up to five times compared to the state-of-the-art distributed algorithm.

Our last contribution is designing an efficient parallel edge-centric approach for evaluating graph simulation and dual simulation. First, we introduce a new distributed data structure called ST whose elements can be processed in parallel. Next, we propose the first parallel and edge-based algorithm called PGSim to answer relaxed GPM queries through graph simulation. Based on ST and PGSim, we develop PDSim, a highly scalable split-and-combine approach to efficiently evaluate GPM queries via dual simulation. Furthermore, we provide guarantees on the convergence and correctness of the two algorithms. The conducted experiments show that our approach runs faster than the state-of-the-art

vertex-centric algorithm of graph simulation by more than an order of magnitude. The obtained results also prove the linear scalability of the two parallel algorithms.

Thesis organization

Chapter 1 (Preliminaries) provides important preliminaries in addition to an overview of the two categories of graph pattern matching models that exist. The first category is structural GPM and includes mainly subgraph isomorphism, while the second category groups the relaxed GPM models such as graph simulation, dual simulation, strong simulation and many other extensions.

Chapter 2 (*Graph Pattern Matching in Massive Graphs: State-of-the-art*) is divided into two parts. The first part introduces and discusses the different programming models and systems used in distributed graph processing, whereas in the second part, we review the state-of-the-art distributed approaches of GPM on massive graphs. Finally, we classify these works and provide a new taxonomy based on multiple criteria.

In Chapter 3 (*Distributed Graph Pattern Matching via Bounded Dual Simulation*), we present our new model BDSim, along with details of the distributed vertex-centric algorithms proposed to evaluate GPM queries via BDSim.

Next, in Chapter 4 (*A Distributed and Scalable Approach for Strong Simulation*), we present and evaluate our new distributed vertex-centric algorithm for the scalable evaluation of the two relaxed models strong simulation and strict simulation.

Chapter 5 (*An Efficient Parallel Edge-Centric Approach for Relaxed GPM*) presents a new approach for parallel evaluation of relaxed GPM queries on distributed graphs. We introduce the new distributed data structure ST and the two parallel edge-centric algorithms *PGSim* and *PDSim* for graph simulation and dual simulation, respectively.

Finally, we conclude this thesis with a summary of the carried works and some future perspectives.

Part I

Context and Background

Chapter 1

Preliminaries

1.1 Introduction

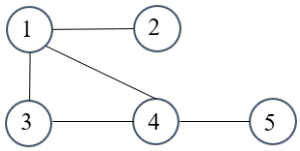
Graphs, being a set of nodes connected to each other by links, are a natural way of representing information in a broad range of domains. Graphs are everywhere, they can be found in chemistry to model the molecule structure, in physics to study the three-dimensional structure of atoms, in social networks to model users and the relationships between them, in the study of the World Wide Web where web pages and the hyperlinks connecting them are represented as a graph, and finally, in computer systems to model computation flows. This chapter is an introduction to the world of graphs and graph pattern matching in particular. First, we define important terms and concepts. Next, we introduce the problem of graph pattern matching and its application domains. Finally, the largest part of this chapter is dedicated to reviewing the existing formulations and models of graph pattern matching and drawing the relationships between them.

1.2 Terminologies

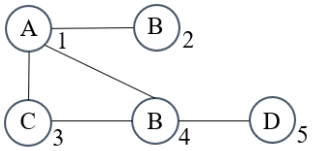
A graph is a structure that groups a set of objects where some pairs are related to each other. These objects are identified by unique numbers and can be characterized by labels or attributes. They are known as the graph vertices (or nodes). When two vertices are related, the link between them is referred to as an edge (or an arc). Moreover, the edges can be labeled with values in $\mathbb{R}$. In this case, the graph is called a weighted graph. An edge can also have a specific direction (the graph is therefore directed), or it can be undirected (we talk about an undirected graph). Moreover, when an edge between two

vertices v and v' is directed from v to v', v is considered the source vertex and v' is the destination vertex of this edge. Finally, a graph is said to be simple when there is at most one edge between any pair of vertices and there are no edges from a vertex to itself. In other words, the graph does not contain parallel edges or loops.

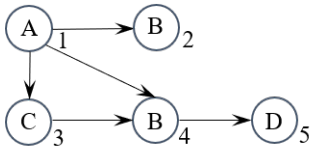
Graphs are illustrated with a set of circles connected with simple lines (in undirected graphs) or arrows (in the case of directed graphs). Examples of an undirected unlabeled graph, undirected labeled graph and directed labeled graph are given in Figures 1.1a, 1.1b and 1.1c, respectively. A weighted graph is illustrated in Figure 1.1d.



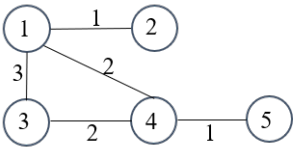
(a) Undirected simple graph.



(b) Undirected labeled simple graph.



(c) Directed labeled simple graph.



(d) Weighted simple graph.

Figure 1.1 Illustrative examples of different categories of graphs

In what follows, we give the formal definition of a directed labeled graph, a subgraph, induced subgraph and other important terms.

Definition 1 (Directed labeled graph)**.** *A directed labeled graph* $G = (V, E, f)$ *is a simple graph where:*

(1) V *is a finite set of all the vertices in* G*,*

(2) E *is the set of edges* $E \subseteq V \times V$ *in which* (v, v') *denotes an edge from* v *to* v'*,*

(3) f *is a function that maps each vertex* $v \in V$ *to a label value* $f(v)$ *in* Σ*, the set of all labels.*

Definition 2 (Subgraph)**.** *A directed graph* $G' = (V', E', f')$ *is considered subgraph of a directed graph* $G = (V, E, f)$ *if and only if*

(1) V' *is a subset of* V*,*

(2) E' *is a subset of* E*,*

(3) $\forall v \in V', f'(v) = f(v)$.

Definition 3 (Induced subgraph)**.** *Given a data graph* $G = (V, E, f)$ *and a set of vertices* $V' \subset V$*, the induced subgraph of* G *with respect to* V' *is the subgraph of* G *that is composed of* V' *and all the edges connecting them from* E*.*

Next, we define a path, shortest path, graph distance, eccentricity, graph diameter and its center.

Definition 4 (Path)**.** *A path in a graph* $G = (V, E, f)$ *is a sequence of vertices* $\rho = (v_1, v_2, \ldots, v_n)$ *such that* $(v_i, v_{i+1}) \in E, \forall i \in 1, 2, \ldots, n-1$*. An elementary path is a path where no vertex is repeated twice;* $\forall v_i, v_j \in \rho,\ v_i \neq v_j$*.*

Definition 5 (Shortest path)**.** *Given an unweighted graph* $G = (V, E, f)$ *and two vertices* v *and* v' *such that* $v \neq v'$*. A shortest path between* v *and* v' *is the path* $\rho = (v, \ldots, v')$ *with the least number of edges.*

Definition 6 (Graph distance)**.** *The distance* $d(v, v')$ *between two vertices* v *and* v' *in a connected graph* $G = (V, E, f)$ *is the minimum length of the paths connecting them. The length of a path* $p = (v_1, v_2, \ldots, v_n)$ *is the number of its edges.*

Definition 7 (Eccentricity of a vertex)**.** *Eccentricity* $\epsilon(v)$ *of a vertex* $v \in V$ *in graph* $G = (V, E, f)$ *is defined as the maximum graph distance between* v *and any other vertex in* G*.* $\epsilon(v) = \max d(v, u), \forall u \in V - \{v\}$

Definition 8 (Graph diameter)**.** *Given a graph* $G = (V, E, f)$*, the diameter of* G *is its maximum eccentricity.*

Definition 9 (Graph center and graph radius)**.** *Given a graph* $G = (V, E, f)$*, the centers of* G *are the vertices having the minimum eccentricity, while its radius is the value of minimum eccentricity.*

Finally, a cycle is simply a closed path, and it is formally defined as follows.

Definition 10 (Cycle)**.** *A cycle is a path* $\rho = (v_1, v_2, \ldots, v_n)$ *where* $v_1 = v_n$*. A simple cycle is a cycle formed by an elementary path; the first vertex is the only one repeated twice.*

1.3 Problem definition

Graph pattern matching is the problem of finding occurrences of a small input graph called *pattern* (a.k.a. *query*) *graph* Q such that $Q = (V_q, E_q, f_q)$ in a larger graph called *data graph* G where $G = (V, E, f)$.

The problem of GPM has several formulations or representations. We find the structural graph pattern matching that includes graph isomorphism and subgraph isomorphism. Other models based on simulation represent a flexible alternative, and are widely used in today's real-world applications. Under this category, referred to as relaxed GPM, we find graph simulation and its extensions such as bounded simulation, dual simulation and relaxation simulation.

Structural matching considers the structural information contained in the graph, i.e. the information available is the edges between the graph vertices and it does not capture matches that are close to the query. Vertices and edges may contain semantic information in the form of labels making the matching semantic. On the other hand, the relaxed graph matching considers less the structural information and more the semantic one carried by labels on the data graph vertices or edges. These labels are very informative as they participate in the matching process, such that two nodes or edges with the same label are considered similar. A node/edge can have multiple labels, they are generally represented as a set of attributes in the form of *(key, value)* where *key* is the label type and *value* is the label itself. In this type of graph pattern matching, the structure of the answers may vary from that of the input query because the structural matching constraints are relaxed, seeking lower complexity and to capture more similarities.

A common property of these different GPM models is returning a match graph, which can be either the whole graph G or one to several parts of it. A match graph is built from a binary match relation R that maps the vertices of Q to the vertices of G *w.r.t.* the GPM model. The maximum match graph is the one containing all the possible match graphs. It is formally defined as follows.

Definition 11 (Maximum match graph)**.** *Let* $G = (V, E, f)$ *be a data graph,* $Q = (V_q, E_q, f_q)$ *a pattern graph and* $R \subseteq V_q \times V'$ *the maximum match relation of* Q *in* G *w.r.t. a given GPM model. The maximum match graph is* $G' = (V', E', f')$*, a subgraph of* G *verifying:*

(1) R *maps every vertex* v *in* G' *to a vertex* u *in* Q*, i.e.* $\forall v \in V', \exists u \in V_q$ *such that* $(u, v) \in R$*.*

(2) $\forall (v, v') \in E', \exists (u, u') \in E_q$ *such* $(u, v) \in R$ *and* $(u', v') \in R$*.*

1.4 Applications of graph pattern matching

Graph matching and graph pattern matching have been used since many decades in multiple application areas. In document processing, we find two important applications. First, in handwritten recognition where documents are analyzed and the text is extracted and modeled as a graph. Then, these graphs are queried against an existing database of already identified texts and characters to recognize the written text [82, 1, 41, 118]. Similarly, graph matching was also used in symbols recognition [80, 142].

Biometric identification is an important task in security to determine the identity of a person. It uses distinctive features of the person in the authentication process. A large set of identification metrics are extracted from images and modeled as graphs. Then, this set is queried with graph matching algorithms to find a correspondence with the input feature of the person to be identified. Among these digital features, we find fingerprints [22, 161], retina [71] and the face [140]. In computer vision, graph matching is used for image analysis and image databases for both indexing and retrieval [30, 2, 19, 19, 64]. Moreover, it is also used in video analysis for object tracking [123, 95, 150, 159, 56].

Furthermore, different approaches of GPM are used for community detection in the World Wide Web graph [42, 90] and community mining in social networks [13, 58]. Community outlier detection is addressed through graph matching in [87]. In social networks, the problem of expert finding can be solved with graph pattern matching such that a pattern of the desired qualifications and their relationships is used as an input query graph [33]. Fraud detection through the analysis of cyclic patterns in social networks and graph databases is addressed in [107, 99, 9]. These techniques employ GPM algorithms to find predefined patterns in the form of rings that are suspicious of fraudulent behaviour in banks and other data sources.

Finally, graph pattern matching has always been highly adopted in the field of information security for intrusion and anomaly detection in programs, networks and cyber-physical systems [67, 5, 75, 155, 88].

1.5 Structural Graph Pattern Matching

We define graph isomorphism and subgraph isomorphism, two of the most studied models in the literature. They are purely structural, i.e. emphasize the relations between vertices.

1.5.1 Graph isomorphism

Two graphs $G = (V, E, f)$ and $G' = (V', E', f')$ are isomorphic if and only if there is a bijective mapping $R(R : V' \to V)$ that satisfies the following condition: $\forall u, u' \in V'$, if $R(u) = v \in V$ and $(u, u') \in E'$ then $\exists v' \in V$ such that $R(u') = v'$ and $(v, v') \in E$.

While we are interested in techniques and models that find answers to a query graph in a larger graph, graph isomorphism is mainly used for comparing graphs of the same size (a.k.a. graph matching).

1.5.2 Subgraph isomorphism

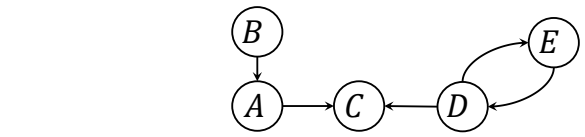
Subgraph isomorphism (SubIso) is defined as an injective mapping between the vertices of the two graphs $Q = (V_q, E_q, f_q)$ and $G = (V, E, f)$. In other words, it is the problem of finding all the subgraphs of a data graph G that are isomorphic to a pattern graph Q. Formally, G matches Q via subgraph isomorphism if and only if: $\forall u, u' \in V_q$, if $R(u) = v \in V$ and $(u, u') \in E_q$ then $\exists v' \in V$ such that $R(u') = v'$ and $(v, v') \in E$, we note here that the mapping is not bijective.

Subgraph isomorphism can also be used in graph matching where the two graphs are similar in size.

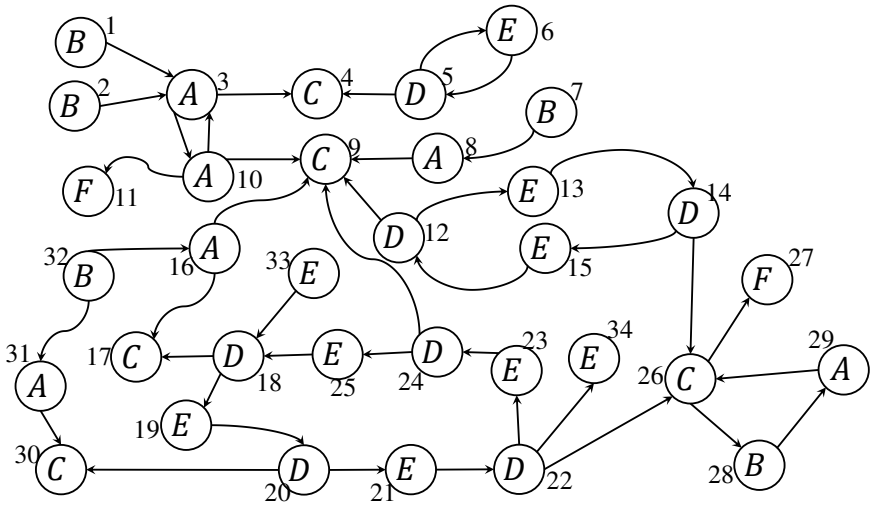
Example 1. *Figure 1.2 shows an example of a pattern graph Q_1 and a data graph G_1. G_1 represents a social network, where vertices in G_1 refer to employees, while an edge between vertices v and v' means that v has endorsed v' on this social network (they worked together in the past). A label on a data node indicates the job of the employee represented by this vertex. In the process of hiring a new team, a pattern graph modeling the relations required between the members of such team is used to find eligible candidates. For example, an edge (A, B) in the pattern graph is translated to the following requirement: the person who takes job A should have a past experience working with the employee getting job B, A should also have recommended B on this social network. When applying subgraph isomorphism, we can extract two matched subgraphs from G_1. Their corresponding vertices are $\{1, 3, 4, 5, 6\}$ and $\{2, 3, 4, 5, 6\}$ as it is shown in Figure 1.3.*

1.5.3 Algorithms for subgraph isomorphism

Subgraph isomorphism is an NP-Complete problem [49] that has been the subject of several studies over the past decades. We distinguish two types of algorithms to evaluate it; exact and approximate algorithms. The exact algorithms find all subgraphs isomorphic



(a) Pattern graph Q_1



(b) Data graph G_1

Figure 1.2 An example of a data graph G_1 containing employees in a professional social network, where labels refer to their jobs and an edge between two vertices means that the source endorsed the destination. The pattern graph Q_1 represents the desired connections between the employees that should be retrieved.

to the input query which leads to an exponential time complexity, whereas the other algorithms provide approximate matches yielding to some improvements in the required running time and space but with a loss in the quality of results. The comprehensive Surveys [23, 46, 43, 131] give an overview of the different techniques used to evaluate exact and inexact subgraph isomorphism since the seventies until 2015.

In its basic definition, subgraph isomorphism does not capture the semantic similarities that are generally represented by labels on both graph vertices and edges. However, recent approaches consider the semantic information in addition to the pattern structure.

In [128], Ullmann suggested a tree-based algorithm for solving graph isomorphism. It enumerates all the possible ways of matching query vertices with the data graph vertices starting from a mapping matrix. It also uses a refinement procedure that prunes the search space by eliminating unfruitful matches. It is based on the observation that a query vertex u and a data vertex v that are mapped together must also have their

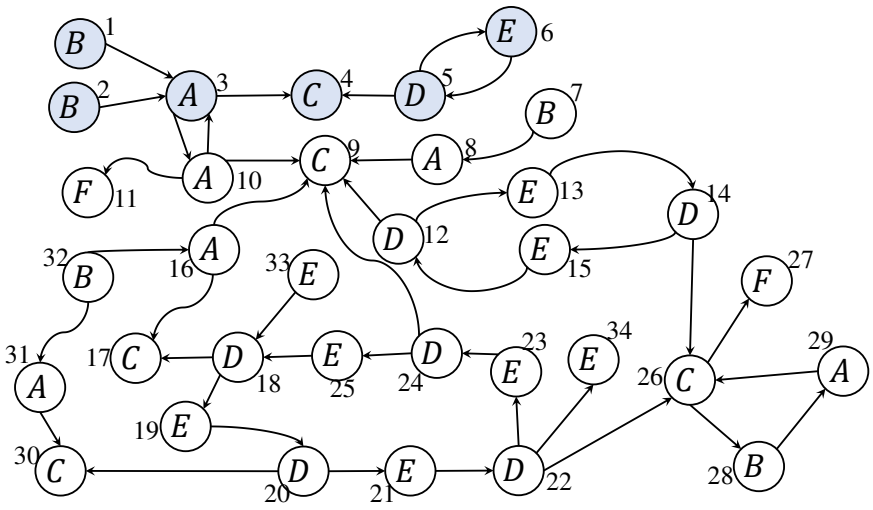


Figure 1.3 Results of applying subgraph isomorphism between query graph Q_1 and data graph G_1 are colored in blue.

neighbors matched together. Thus, any match that does not satisfy this constraint is eliminated from the beginning.

All the works that followed are based on this seminal work. They address two common challenges. First, how to minimize the size of intermediate results by ordering the query vertices. Second, finding pruning strategies that avoid useless computations.

The first algorithm is VF2 [24]. It is based on the state space representation for solving graph isomorphism. VF2 explores the search space starting from the neighbors of each matched query vertex. VF2Plus [17] improves the ordering of VF2 by picking the query vertices with the lowest chance of finding matches in the data graph and the highest number of neighbors among prior vertices in the ordering. VF3 [16] also improves VF2 by proposing heuristics for reducing the search space size and the time used for processing each state. Moreover, VF3-Light [15] is another algorithm that eliminates some of VF3's heuristics which results in shorter running time.

Furthermore, we find algorithms that use exploration of the data graph to evaluate subgraph isomorphism such as QuickSI [111], RI [10] or VF2++ [60]. Others create tree/graph indices for storing the candidates' sets like GraphQL [55], TurboIso [53], BoostIso [100], CFL-Match [8], TurboFlux [66], DAF[52] and VC [121], while some construct indices based on the data graph and use them to assist the embeddings enumeration; GADDI[157] and SPath[160]. Finally, MQO-iso [101] constructs a Direct Acyclic Graph (DAG) based on the relationships between multiple queries to guide subgraph isomorphism search.

As opposed to Ullmann that does not define an ordering of the query vertices, QuickSI [111] proposed to process vertices with infrequent vertex labels in the data graph as quickly as possible, which allows avoiding useless computations. QuickSI processes the edges/vertices having a lower number of possible embeddings first, hence, reduces the size of explored search space. It first selects a spanning tree that minimizes the number of possibly generated embeddings in the data graph. Then, it selects an order of processing by setting the first edge to explore as the one having minimum size of candidates for its source and destination. Finally, a Depth First Search (DFS) traversal of the spanning tree is conducted to enumerate all the possible embeddings. The two algorithms RI [10] and VF2++ [60] are similar to QuickSI as they all explore the data graph directly without generating any candidate sets. However, RI picks query vertices based on their degrees by first selecting the one with the highest degree, and then the ones with most neighbors in the current ordering. VF2++ picks query vertices with the least frequent labels in G and the highest degree in Q.

To improve the existing pruning rules, GraphQL [55] proposed to use a neighborhood signature of a query vertex u which is the set of its neighbors' labels. A data vertex v is pruned out if its neighborhood signature does not include the neighborhood signature of its matched query vertex. GraphQL picks the query vertex that is connected to an already matched vertex and that has a minimum number of candidates. Similarly, SPath [160] used an improved neighborhood signature to minimize the size of the candidates' set. It also matches a path-at-time.

TurboISO [53] is another algorithm that groups query vertices having the same label and the same neighborhood into one node to reduce the duplicated generation of unfruitful matches. It divides the data graph into separate Candidate Regions (CRs), where the root of every CR is a data node matched to the query tree root. For each CR, TurboISO enumerates the possible mappings based on a combine/permute strategy. Given a sequence order and a candidate region CR, only one query vertex is processed and if it leads to an accepted answer, other combinations are generated from other vertices in its group. Otherwise, they are all eliminated.

Inspired by the query compression in TurboISO, BoostISO [100] compresses the data graph based on the relationships between data vertices. It groups the data vertices having the same matches to one hypernode in a new adapted graph that has a smaller size. BoostISO also groups similar query vertices which allows it to prune unfruitful matches as quickly as possible. A follow up revision of the filtering phase was made by the same authors in [134].

Furthermore, in [101], Ren et al. propose a multi-query optimization plan (MQO-iso) to reduce the processing time of multiple queries. The input query set is processed to detect the common subgraphs, then a DAG is built based on the isomorphic relationships between the queries such that there is an edge from q to q' if q is a subgraph isomorphic to q'. This DAG guides the subgraph isomorphism search while exploiting the already generated intermediate results. The parents in the DAG are processed first and their results are cached then used to process their children.

CFL-Match [8] proposes to decompose the query graph into three substructures. It matches the cores first because they are less likely to be highly present in the data graph. Second, it processes the forest substructures that have high chances of being present in the graph, while leaves (one-degree vertices) are left to the end as they are most likely to generate most of the unnecessary embeddings that must be avoided. The core is a dense connected subgraph containing every single edge that is not present in a spanning tree of the query. Such substructure, when processed earlier, reduces the number of unpromising partial mappings by avoiding generating unnecessary Cartesian products. The forest is the complement of the core in the query graph without the leaves. To evaluate subgraph isomorphism, CFL-Match uses a Compact Path Index (CPI) having the same structure as a Breath First Search (BFS) tree q_t of the query graph. Its nodes represent a query vertex u with its label and candidate set. For every edge (u_1, u_2) in q_t, the CPI keeps the edges between the candidate sets of u_1 and u_2. This CPI is then traversed to enumerate subgraph isomorphism embeddings by processing the core, then the forest and finally the leaves. For the core, it uses a path-based ordering that minimizes the number of generated embeddings while favoring the paths with non-tree edges.

On the other hand, DAF [52] addresses the limitations of TurboISO and CFL-Match by converting the query graph into a DAG instead of a spanning tree. The algorithm uses dynamic programming to find the candidate set for all the subgraph isomorphism embeddings then follows an adaptive, DAG-based ordering to build the possible embeddings.

Moreover, VC [121] improves the path-based ordering with a cost function that considers edges also between the paths. VC uses a bigraph index for holding the candidate edges instead of a spanning tree as it is the case for CFL-Match.

Finally, to detect patterns in a dynamic graph, TurboFlux [66] uses a data-centric graph (DCG) representation of the intermediate results. A DCG is composed of the data graph such that for each data node, its candidate query vertices are represented by incoming edges. This DCG is then traversed to find subgraph isomorphism matches. TurboFlux defines transition rules on every edge in the DCG, then, whenever an edge is inserted or

deleted, it first checks whether it matches an edge in the query graph or not and then updates the results according to the transition rules.

1.5.4 Parallel algorithms for subgraph isomorphism

In order to scale up the existing algorithms, two categories of parallel approaches showed up addressing the problem of subgraph isomorphism, namely relational join-based and exploration-based. We also find the GPU-friendly algorithms that use joins or exploration to implement subgraph isomorphism enumeration on massively parallel architectures.

Join-based approaches

The data graph edges are considered as relational database tables, such that each pair of labels forms a table, the attributes represent the two labels and the edge identifiers are the attribute values. These tables are filtered and a series of joins is applied to get the matching answers. However, joins are very costly and many works addressed this problem through different categories of joins: binary joins where the first two tables (edge candidates) are joined, then, the result is joined again with a third table of edge candidates until obtaining the end result [97, 152, 51, 127]. These works propose different join orders that aim at optimizing the size of intermediate results. Other works attempted to join substructures larger than edges which resulted into reducing further the size of intermediate results. In [122], the pattern graph was decomposed into stars (two level trees) that can be found through data graph exploration (they require only information about direct neighbors of each data vertex). After that, the matched substructures were joined to form a complete subgraph match. More recent works used different joining units such as the TwinTwig [68], Crystal [98] or Clique [69, 70].

On the other hand, Afrati *et al.* [3] used the multiway joins where all the edge candidate sets are joined at the same time. Furthermore, worst-case optimal join algorithms are also used for multiway joins. Two common algorithms are GenericJoin [91] and LeapFrog TrieJoin [130]. Parallel approaches for finding subgraph isomorphism based on these algorithms were proposed by [4] and [45] respectively. Mhedhbi *et al.* combined both binary joins and the GenericJoin algorithm to evaluate subgraph isomorphism in [89].

In [70], Lai *et al.* surveyed and conducted an experimental comparison of a set of join-based GPM algorithms and classified them into three categories based on the join strategy applied.

Exploration-based approaches

These are approaches based on data graph exploration, which naturally fit into the newly TLAV paradigms for distributed graph processing. We discuss this category more in detail in Chapter 3 after introducing the different concepts and programming models for distributed graph processing in Chapter 2. However, as an exception, we present here the four works QFrag [110], CECI [7], PSM [25] and BENU [138]. First, QFrag and CECI replicate the data graph on multiple machines where each processor runs subgraph isomorphism search on different parts of the data graph. Moreover, PSM evaluates subgraph isomorphism in parallel on a single machine, while BENU is presented here because it is also a parallel system that uses a similar approach to answer GPM queries.

QFrag [110] replicates the data graph on k workers and parallelizes TurboISO to evaluate subgraph isomorphism while using *task fragmentation* to achieve load balance. QFrag decomposes the first super-step of processing into two phases. In the first phase, each worker is assigned a list of roots from which it builds the corresponding CRs. It considers the embeddings enumeration of each CR as one subtask and estimates the processing time of each tree to detect any outliers that cannot be processed locally. Embedding enumeration of regular trees is conducted locally in the first super-step, while the outlier trees are split into equal subtasks and distributed over the workers to be processed in the second super-step.

CECI [7] is based on a spanning tree index constructed from a BFS traversal of the query graph. Each node in this tree maps a query node u to a data node v matched to it and to its neighbor for representing the tree edge candidates. It also keeps candidates of u's neighbors that are connected by a non-tree edge. This index is then used to enumerate subgraph isomorphism embeddings in parallel. Every candidate of the query tree root is used as a pivot of an Embedding Cluster (EC) that will be processed in parallel. To achieve load balancing, CECI estimates the cost of evaluating each EC, then decomposes the costly ones into multiple ECs. Each worker processes an equal number of ECs. Finally, CECI keeps the candidates of the non-tree edges in its index and uses an intersection between the two sets, i.e. tree-edge candidates and non-tree edges candidates, to enumerate the embeddings.

Furthermore, PSM [25] parallelizes backtracking based algorithms that adopt a DFS traversal of the candidates tree index on a single machine. PSM decomposes the tree index into search regions and each worker is assigned a region that it can expand independently in parallel. Load balancing is conducted dynamically where busy workers split their search regions into two equal subtasks.

Finally, BENU [138] also uses backtracking and decomposes the search tree into regions to be processed by different workers. The workers use adjacency lists of the data graph to enumerate possible embeddings through the application of set intersections. Load balancing is achieved by splitting a tree task into subtasks if the degree of the current tree root vertex is larger than a certain threshold.

GPU-friendly approaches

Under this category, we find the approaches aiming to accelerate graph processing tasks with the help of GPU friendly algorithms. A CPU is characterized by a limited number of powerful cores, whereas a GPU has thousands of cores with lower individual performance. Graphics processors are highly used nowadays to implement solutions for graph problems, e.g. shortest path [54, 63], BFS [83, 156], and minimum spanning tree [132, 105]. Each core in the GPU works in a single-instruction-multiple-data (SIMD) fashion, i.e. the same instruction is executed on different input data in parallel. For speeding up the computing process, GPUs use a global memory with a large amount of device memory and a shared memory for each multiprocessor with a limited amount of device memory. A comprehensive survey on the challenges of GPU-based graph processing models and algorithms can be found in [116].

Many GPU based algorithms were suggested tackling the GPM problem. GpSM [127] is a graph pattern matching algorithm based on subgraph isomorphism that was designed for massively parallel architectures. The authors proposed a filter-and-join approach that takes on edges as a basic processing unit. A selection order is firstly obtained for the query vertices based on some selection criteria. They get such order by extracting a query tree from the query graph. Edges are added to the tree iteratively by selecting at each step the ones maximizing the ratio between the vertices degree and their labels frequency. This selection order was proven to reduce the size of the search space. Then, for each query vertex in the tree, a *filter phase* is conducted in parallel to check the candidacy of every data vertex against it. To filter out more data vertices, the neighborhood of each query vertex is explored in parallel. A candidate vertex v is pruned out if there is a query vertex u' adjacent to u such that u' does not have any candidate vertex in the adjacency list of v. Another refinement strategy is also executed to prune low-connectivity vertices with a degree less than a certain threshold. In the *joining phase*, candidate edges are gathered then combined to form partial subgraph matches incrementally. They collect candidate edges for each query edge, given a query edge (u, u') such that $C(u)$ and $C(u')$ are the respective candidate sets of u and u', candidate edges of (u, u') are all the edges

(v, v') verifying $v \in C(u)$ and $v' \in C(u')$. At the end of this phase, each query edge is assigned a set of candidate edges. Then, the algorithm combines partial matches starting from the smallest set of candidate edges to the largest one. Here, subgraph matches are constructed iteratively by picking up at each step an edge from the candidate edges, then joining it to the already built partial matches. One of the main assets of GpSM is the graph representation in memory, it uses a different representation for storing the candidate vertices and candidate edges.

GpSense [126] is an algorithm that enhances the performance of GpSM allowing to find subgraph isomorphism efficiently on large common-sense knowledge graphs. The main issue with knowledge graphs is their huge size not allowing them to be stored on the memory of a single GPU, which led the authors to propose a new graph compression technique that permits to reduce the graph size while preserving the same subgraph isomorphism results. In a knowledge graph, they noticed that multiple data nodes playing the same role have the same adjacency list, based on this observation, the authors propose a multiple-level compression technique that merges such data nodes into one hyper-node.

In [136], the authors proposed a GPU-friendly algorithm for finding subgraph isomorphism in a large graph dataset. The initial problem that consists of counting triangles in a large network is converted into a subgraph matching problem. They built a filter-and-join strategy for subgraph isomorphism based on the Gunrock [137] programming model.

Gunrock allows programmers to design primitives for graph processing on the GPU with a high level of abstraction and higher performance. It's a parallel, high-level data-centric model manipulating a basic subset of edges or vertices that are actively participating in the computation (such subset is called frontier). A program within the Gunrock framework consists of three steps: *advance*, *filter* and *compute*. The *advance* step generates a new frontier from the current frontier by exploring the neighbors of its vertices or edges . The *filter* step generates a new frontier from the current one by extracting a subset of it based on some criteria defined by the programmer. Finally, the *compute* step that consists of a programmer-specified function to be executed in parallel on the elements of current frontier.

To enumerate subgraph isomorphism matches, this approach proceeds as follows. First, the candidate vertices initialization is executed where the candidate set is filled according to the information held by every data vertex, this phase is executed within an advance operator. In the second step, candidate edges are collected using the advance and filter operators. Finally, the joining step combines together candidate edges is executed to incrementally build subgraph matches in parallel. This approach showed better

performances in the running time compared to the GpSM implementation of subgraph isomorphism when the size of the query graph increased.

Other works such as [78, 153, 21] also propose new algorithms, partitioning strategies or data structures that are GPU-based to solve the problem of subgraph isomorphism in massively parallel architectures.

1.5.5 Limitations of subgraph isomorphism

Although discussions regarding the approximate solutions have dominated the research on subgraph matching over the recent years, their performance is not yet able to manage the actual size of the GPM problem. We are facing today a growing need for more performing techniques and tools that can process big graphs; that are highly dynamic and distributed over hundreds of machines because of their enormous size. For example, Facebook is the largest social network with 2.4 billion monthly active users in June 2019 [31]. Furthermore, subgraph isomorphism requires the input graph to be structurally identical to the data graph. In its stringent formulation, it does not consider the intrinsic deformations of an object from its ideal model. Such drawbacks have led the research community to investigate other models of GPM that are less expensive in time and space and exploit better the semantic information held by labels. Graph simulation and its extensions are considered nowadays the best fit techniques for GPM in many of the emerging applications like analyzing associations in social networks [93].

1.6 Relaxed Graph Pattern Matching

In this section, we review graph simulation and the other models proposed as an alternative to structural GPM. Models in this category aim to capture more similarities by relaxing the matching constraints imposed by structural GPM, which may result into answers that do not have the same structure of the query. This way of defining similarity is highly needed in the current real-world applications such as plagiarism detection. For example, Chao *et al.* proposed an approach based on graph pattern matching to detect plagiarism in programs having thousands of lines of code [79]. They use the program dependence graph to find similar graphs in an available code base.

1.6.1 Graph simulation

Graph simulation (GSIM) is a flexible model that allows one query vertex to be mapped to multiple data vertices. It maps separately every edge from the pattern graph to edges in the data graph resulting into one-to-many matching as an output. A vertex v in the data graph $G(V, E, f)$ is mapped to a vertex u in the query graph $Q(V_q, E_q, f_q)$ if they have the same labels and the vertices on the outgoing edges of u have matches among the children of v. In a formal way: given a data graph G and a pattern graph Q, a binary relation $R \subseteq V_q \times V$ is said to be a match if:

(1) $\forall\,(u, v) \in R, f(u) = f_q\,(v)$,

(2) $\forall (u, u') \in E_q, \exists (v, v') \in E$ such that $(u', v') \in R$.

Graph G matches pattern Q via graph simulation, if there exists a total match relation R where $\forall u \in V_q, \exists v \in V$, such that $(u, v) \in R$. It was proved that a total match can be found in polynomial time (quadratic time) using a refined version of the first algorithm of graph simulation, called HHK [57]. Another algorithm for graph simulation was proposed in [32].

Given the data graph G_1 in Figure 1.2b and the pattern graph Q_1 in Figure 1.2a, the application of graph simulation returns a match set R containing all the vertices of G_1 except three vertices $\{11, 27, 34\}$, as it is shown in Figure 1.4. We notice that several vertices appearing in R should not have been conserved, notably $\{10, 33\}$.

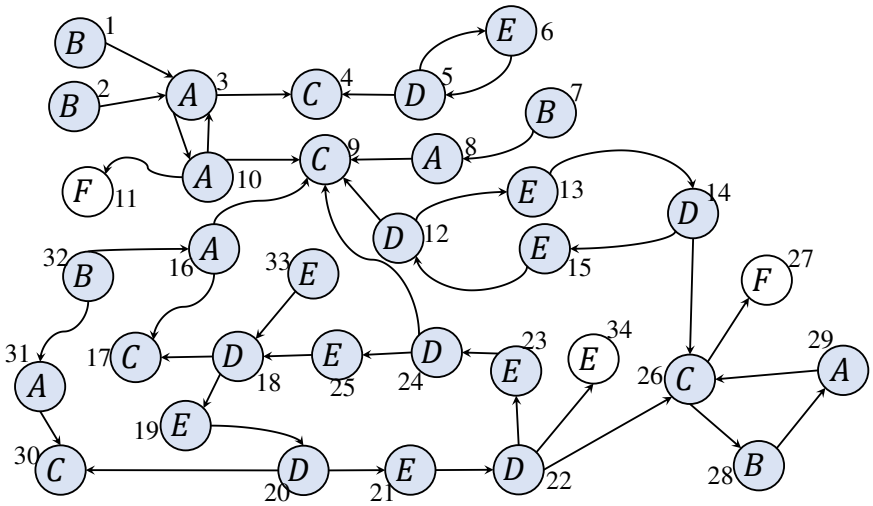
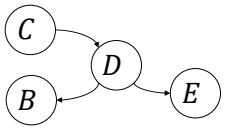
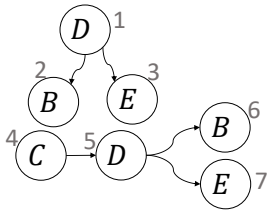


Figure 1.4 Results of applying graph simulation between query graph Q_1 and data graph G_1 are colored in blue.

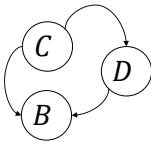
Graph simulation may appear like inexact graph pattern matching, but this is not true. Error-tolerant approaches allow mismatches while graph simulation does not, i.e. the answers returned by graph simulation respect the constraints defined by the model when inexact matching tolerates incorrect matches to some degree. Besides its low complexity, graph simulation is a natural fit for performing graph matching on a fragmented data graph [32]. However, its weak part is failing to capture topological similarities between matched graphs, since a disconnected graph can be correctly set as a match for a connected one (a query example is given in Figure 1.5a and its answer in Figure 1.5b), and a cyclic pattern can be matched to tree subgraphs (a cyclic pattern example is shown in Figure 1.5c and its answer in Figure 1.5d).



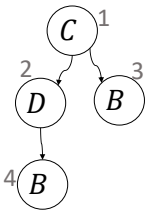
(a) Connected query



(b) Disconnected answer



(c) Cyclic query



(d) Tree answer

Figure 1.5 An example showing the limitations of graph simulation.

1.6.2 Dual simulation

Dual simulation (DSIM) [84] considers parent relationships in addition to the child relationships imposed by graph simulation. A vertex v from $G = (V, E, f)$ is mapped to a vertex u in $Q = (V_q, E_q, f_q)$, if they have the same labels and the vertices on their incoming and outgoing edges are also mapped together. Formally, a data graph G matches a pattern graph Q via dual simulation if there exists a binary match relation $R \subseteq V_q \times V$ such that:

(1) $\forall (u, v) \in R$, $f_q(u) = f(v)$, i.e. u and v have the same label,

(2) $\forall u \in V_q, \exists v \in V$ such that $(u, v) \in R$ and:

(a) $\forall (u, u') \in E_q, \exists (v, v') \in E$ such that $(u', v') \in R$ (child relationship),

(b) $\forall(u'', u) \in E_q, \exists(v'', v) \in E$ such that $(u'', v'') \in R$ (parent relationship).

Moreover, the maximum dual match set is defined as follows.

Definition 12 (Maximum dual match set)**.** *Given a data graph $G = (V, E, f)$ and a query graph $Q = (V_q, E_q, f_q)$. Let $R \subseteq V_q \times V$ be a match relation between Q and G w.r.t. dual simulation (also called dual match relation). The maximum dual match set R_d is a dual match relation such that for any dual match relation R, $R \subseteq R_d$.*

The results of evaluating dual simulation on the example of Figure 1.2 are shown in Figure 1.6. It removes vertices $\{10, 33\}$ from the match set returned by graph simulation. The dual match graph is the induced subgraph containing only vertices of the data graph G_1 that are already in the dual match relation. For example, even though vertices 26 and 28 belong to the dual match set, the edge linking them is not in the dual match graph because there is no edge with labels (C, B) in Q_1.

Considering that a connected graph is said to be an answer to a certain query if and only if $\forall u \in V_q, \exists v \in V$ that matches u. Dual simulation brings duality into graph simulation i.e. it eliminates cases where disconnected graphs are considered answers to connected graph queries (back to the example in Figure 1.5, the partial answer composed of vertices $\{1, 2, 3\}$ in Figure 1.5b is not valid as an answer on its own to the query in Figure 1.5a in the case of dual simulation due to the absence of label C).

Even though dual simulation returns answers of better quality, it does not take into account the data locality compared to graph simulation. Data locality indicates the scope in which we should look for answers to our query. It could be the distances between vertices in the resulting graph or its size. It is obvious that dual simulation does not take into account data locality since cycles in the query graph can be matched to very long cycles in the data graph. The data locality problem becomes relevant when the returned answers are very large, thus difficult to analyze and also when the distances between data vertices are not kept, making the relationships between distant vertices lose their meaning. That is why this model was reinforced by introducing strong simulation.

1.6.3 Strong simulation

Strong simulation (SGSIM) [84] improves dual simulation by adding data locality such that matches are only searched in a ball b of center v and having as radius d_q, the diameter of the pattern graph Q. The subgraph of G contained in this ball is denoted $\hat{G}[v, d_q]$. Formally, G matches Q via strong simulation, if there exists a vertex $v \in V$ such that:

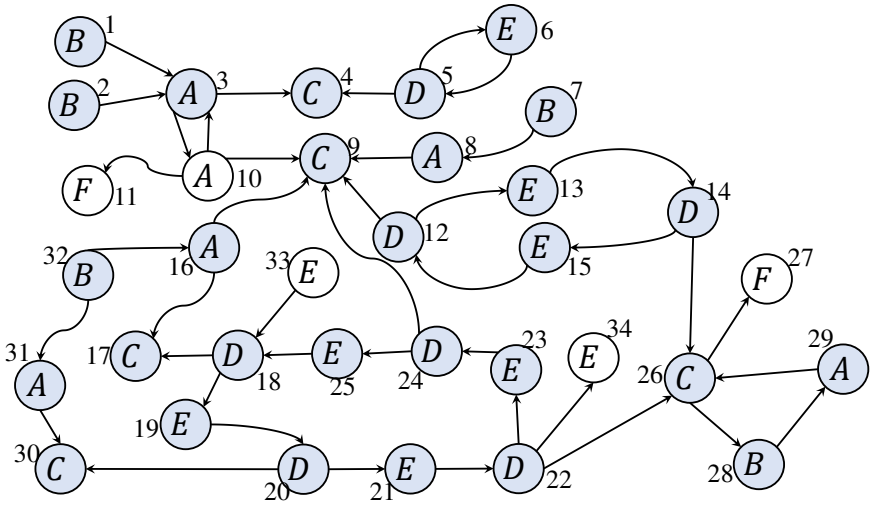


Figure 1.6 Results of applying dual simulation between query graph Q_1 and data graph G_1 are colored in blue.

(1) Q matches $\hat{G}[v, d_q]$ with maximum dual match set $R^b_{d_q}$ in ball b of radius d_q,

(2) v is member of at least one of the pairs of $R^b_{d_q}$.

For every vertex v in the data graph, the connected part of the result match graph of each ball with respect to its $R^b_{d_q}$ containing v is called a maximum perfect subgraph (Max-PG) of G with respect to Q. By proposing a cubic time algorithm, the authors proved that strong simulation not only succeeds to capture more topological similarities, but it also keeps the same polynomial time complexity as graph simulation.

Applying strong simulation to the example given in Figure 1.2 eliminates matches that fall within the long cycle composed of $\{18, 19, 20, 21, 22, 23, 24, 25\}$, because the distance between vertices 18 and 25 is greater than the pattern's diameter, which is equal to 4. The maximum match set contains the vertices $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16, 26, 28, 29, 32\}$ as shown in Figure 1.7. However, strong simulation is limited in terms of scalability since it incurs very long processing time and memory consumption for locality balls creation, especially when the data graph is very large.

1.6.4 Strict simulation

Strict simulation (STSIM) improves strong simulation with a revised definition of locality which enables achieving better quality of results with performance enhancements [39]. Locality balls are created from the members of the dual match graph instead of generating them directly from the dual match set in the initial data graph as it is the case in strong

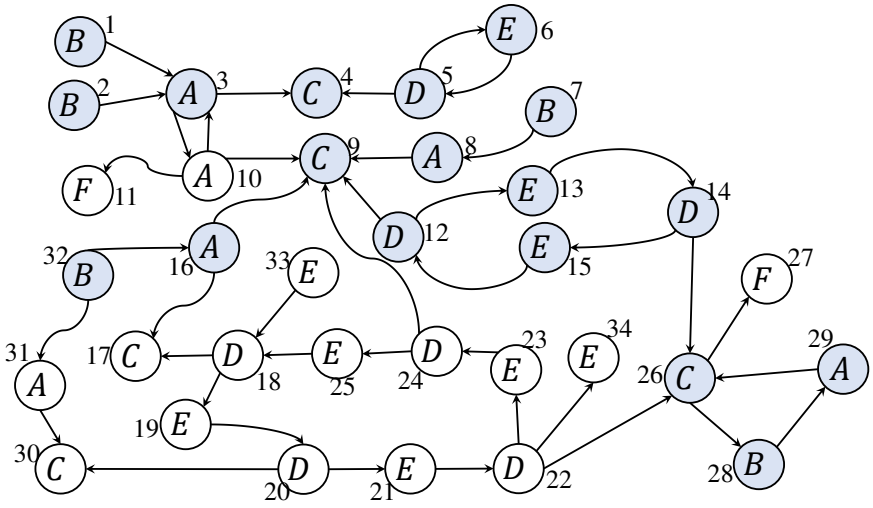


Figure 1.7 Results of applying strong simulation between query graph Q_1 and data graph G_1 are colored in blue.

simulation. The difference is that in strong simulation the balls may contain data vertices that are not in the dual match graph while in strict simulation the balls contain only members from the dual match graph.

Let us define the dual match graph first.

Definition 13 (Dual match graph)**.** *Given a data graph* $G = (V, E, f)$ *and a query graph* $Q = (V_q, E_q, f_q)$*. The dual match graph is the subgraph* $G' = (V', E', f)$ *of* G *composed of* V' *and* E' *such that :*

(1) V' *is the set of all vertices in* R_d *(the maximum dual match relation for* Q *in* G*),*

(2) $E' \subset E$ *such that* $\forall (v, v') \in E', \exists (u, u') \in E_q$ *verifying* $(u, v) \in R_d$ *and* $(u', v') \in R_d$*.*

Given a data graph $G = (V, E, f)$ and a query graph $Q = (V_q, E_q, f_q)$ such that $G_d = (V_d, E_d, f_d)$ is the dual match graph. G matches Q via strict simulation if there exists a vertex $v \in V_d$, such that:

(1) $\hat{G}_d[v, d_q]$ matches Q where $\hat{G}_d$ is a subgraph of G_d centered at v with radius d_q,

(2) v is member of the resulting Max-PG.

Strict simulation results into better efficiency by decreasing the size of the locality balls, while obtaining better matches. The authors proposed an algorithm for calculating the Max-PGs in cubic time. When applying this model on the example of Figure 1.2, it results into the same match graph as strong simulation. However, we notice that the size of the balls created is smaller than in the case of strong simulation because the latter

contains also non candidate vertices, the results are shown in Figure 1.8. We notice that the final Max-PGs are the same as in strong simulation, but this is only valid for this particular example. Even though strict simulation improved strong simulation, it still incurs considerable processing time and may result into duplicate answers. Actually, two adjacent vertices will most likely give the same Max-PG, a limitation that motivated Fard *et al.* to propose tight simulation (TSIM) in [38].

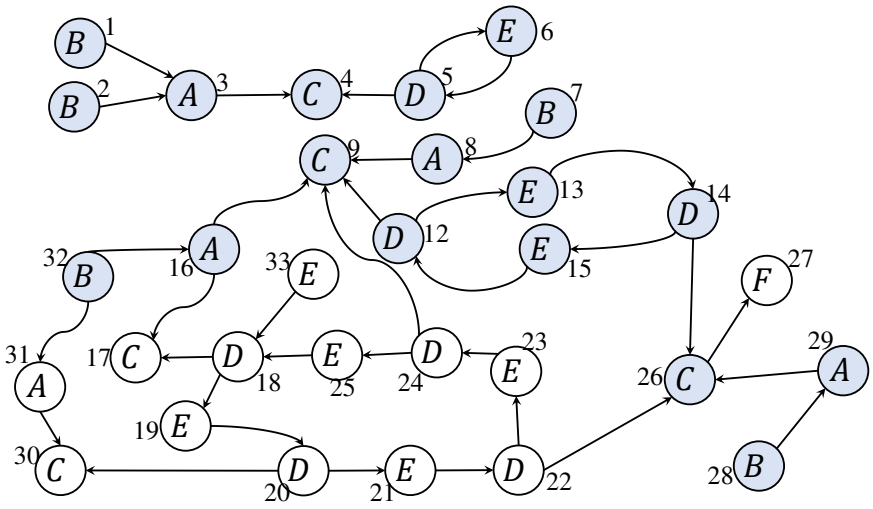


Figure 1.8 Results of applying strict simulation between query graph Q_1 and data graph G_1 are colored in blue.

1.6.5 Tight simulation

It is based on strict simulation where only the most important vertices of the query graph are considered in locality balls creation [38]. If strong simulation generates a ball centered at each data vertex, and strict simulation creates a ball for each member of the dual match graph, then tight simulation picks only data vertices candidate to the query vertex with the highest selectivity score. The selectivity score here is a measure computed for each query vertex based on its degree and label frequency. This measure allows us to choose the most important vertex among the graph vertices. Tight simulation reduces the number of locality balls, resulting in shorter processing time, because the size of intermediate results is reduced and duplicated partial answers are minimized. It is formally defined as follows: Given a pattern graph $Q = (V_q, E_q, f_q)$ and a data graph $G = (V, E, f)$. G matches Q, if there are vertices u in Q and u' in G such that:

(1) u is a center of Q with highest defined selectivity,

(2) $(u, u') \in R_D$, where R_D is dual match set between Q and G,

(3) Q matches $\hat{G}_D[u', d_q]$ where $\hat{G}_D$ is a ball extracted from G_D (result of dual simulation) with radius d_q,

(4) u' is member of the result Max-PG.

Among the centers of the query graph, the authors pick the one with the highest ratio of its degree to its label frequency ($\max_{u \in C}(deg(u)/freq(f_q(u)))$, where C is the set of query centers). The locality balls are created with a radius equal to the eccentricity of the query graph. They proposed a distributed algorithm with cubic time complexity for finding all the Max-PGs with respect to tight simulation [38].

We apply tight simulation on the example of Figure 1.2, the list of eccentricities for each query vertex is $\{(A : 3), (B : 4), (C : 2), (D : 3), (E : 4)\}$. There is only one center in Q_1 having a minimum eccentricity equal to 2 which is C. The data vertices $\{4, 9, 17, 26, 30\}$ are part of the dual match graph between Q_1 and G_1. Hence, according to tight simulation, we only create balls centered at these vertices with radius equal to 2. Then, for each ball, we verify the dual match relation between the ball members and the query vertices. The ball centered at 4 gives a match set containing the following data vertices: $\{1, 2, 3, 4, 5, 6\}$. The ball centered at 9 gives us the same previous match set. Then, the balls centered at 17, 26 and 30 result all in zero matches. We notice that tight simulation here gives the exact same result as subgraph isomorphism (results are shown in Figure 1.9), while it can be computed in cubic time. The limitation of tight simulation is that it is too strict for some applications of GPM where zero answers are not welcomed.

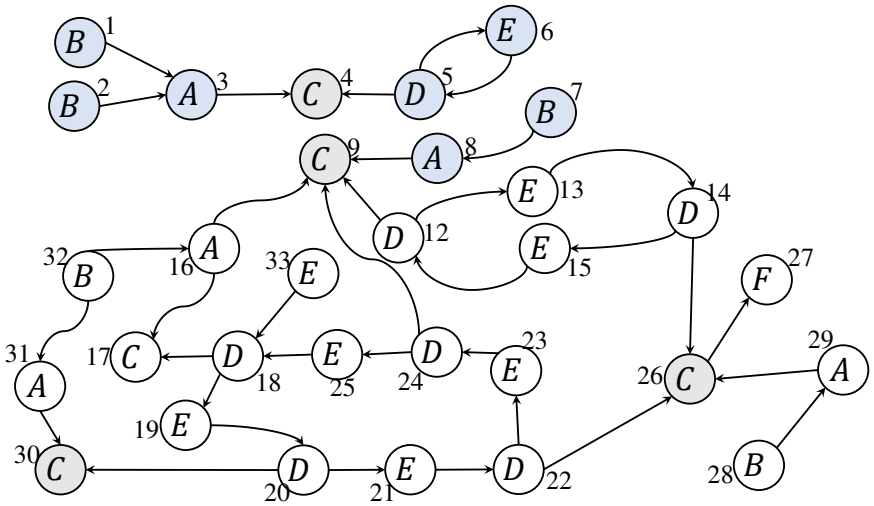


Figure 1.9 Results of applying tight simulation between query graph Q_1 and data graph G_1 are colored in blue while the selected centers are gray.

1.6.6 Bounded simulation

Bounded simulation (BSIM) [34] is an extension of graph simulation that came as a revision for the traditional subgraph isomorphism model, that falls short of capturing similarities in the actual emerging applications, e.g. detecting communities in social networks. A bounded path p in G is a sequence of vertices $v_1, v_2, \ldots, v_n \in V$, such that $\forall i \in \{1, \ldots, n-1\}, (v_i, v_{i+1}) \in E$.

In this model, a data graph is defined as $G = (V, E, f_A)$, such that f_A is used instead of the labeling function f. For each node u in V, $f_A(u)$ is a tuple of attributes $(A_1 = a_1, \ldots, A_n = a_n)$. Furthermore, a pattern graph is given as $Q = (V_q, E_q, f_v, f_e)$ where two functions f_v and f_e are introduced. First, $f_v(u)$ is a conjunction of predicates on the attributes of u. f_e is another function defined on E_q to limit the length of paths matched to each edge of E_q ($f_e(u, u') = \{k, *\}$ where * means no requirement on the path length).

A data graph G matches a pattern graph Q via bounded simulation if there exists a binary relation $R \subseteq V_q \times V$ such that:

(1) $\forall u \in V_q, \exists v \in V$ such that $(u, v) \in R$,

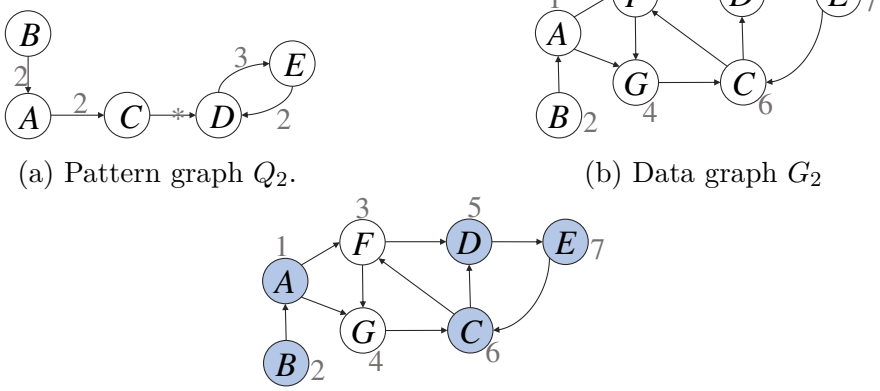
(2) $\forall (u, v) \in R$:

  (a) The attributes $f_A(v)$ of v satisfy the predicate $f_v(u)$ of u,

  (b) $\forall (u, u') \in E_q, (u, u')$ is mapped to a non-empty, bounded path p of length k from v to v' in G such that $(u', v') \in R$ and $len(p) \leq k$ if $f_e(u, u')$ is a constant k.

Graph simulation is a special case for bounded simulation where only edge-to-edge mappings are allowed. In the same paper, the authors proposed an algorithm for computing bounded simulation that runs in cubic time. Incremental algorithms for bounded simulation were also proposed for the case of a continuously updated data graph.

Example 2. *We present in Figure 1.10 another example of, respectively, a pattern graph Q_2 and a data graph G_2. By applying bounded simulation, we get the edge (B, A) mapped to the edge $(2, 1)$, edge (A, C) mapped to the path $(1, 4, 6)$ of length 2, (C, D) mapped to paths $(6, 5)$ and $(6, 3, 5)$. Also, the edges (D, E) and (E, D) are mapped respectively to $(5, 7)$ and $(7, 6, 5)$. Note that (D, E) cannot be mapped to the path $(7, 6, 3, 5)$ because this latter is of length 3, which does not satisfy the constraint on the edge (D, E) in Q_2. The result of applying bounded simulation is shown in the same figure.*



(a) Pattern graph Q_2.

(b) Data graph G_2

(c) Results of applying bounded simulation between Q_2 and G_2

Figure 1.10 An example of a labeled data graph G_2, a query graph Q_2 and the results of bounded simulation where the matched vertices are colored in blue.

1.6.7 Relaxation simulation

This model is an extension of graph simulation that allows partially absent vertices with the condition of substituting them with their children and / or parents [47]. To avoid the problem of zero answers, the authors relax the constraints imposed by graph simulation and dual simulation with respectively two models: unidirectional relaxation simulation (URS) and dual relaxation simulation (DRS).

Given a data graph $G = (V, E, f)$, a pattern graph $Q = (V_q, E_q, f_q)$ and a matching relation $R \subseteq V_q \times V$ with $(u, u') \in R$. If graph simulation requires all children of $u \in V_q$ to be matched with some or all the children of $u' \in V$, then URS relaxes this constraint by accepting also that grand-children of u' replace their absent parents if they match the children of u. The same goes for DRS such that in addition to children constraints, if there are no parents to u' matching some parents of u', the grand-parents of u' that respect the matching constraints are considered in the final match relation.

1.6.8 Surjective simulation

Surjective simulation (SJS) [114] extends bounded simulation by relaxing the matching constraints for the context of multi-labeled graphs. Vertices may have multiple labels and this model redefines the problem of graph pattern matching by instead of imposing that all the labels of a query graph exist on its matched data vertex, surjective simulation is defined such that a query vertex u from the query graph can be correctly matched to a data vertex v if the two share a minimum number of common labels. Moreover, the

answers to surjective simulation must fall within a ball that has at most the diameter of the input query graph. Shemshadi et al. [114] proposed an algorithm for finding the top-k matches *w.r.t.* surjective simulation.

1.6.9 Taxonomy simulation

Taxonomy simulation (TXS) [76] relaxes the label constraints in graph simulation by allowing the mapping of different labels. This model introduces taxonomy to the context of relaxed GPM through the use of a taxonomy graph in the matching process. A taxonomy graph is a labeled rooted forest T that represents the existing relationships between different labels. It defines a specialization-generalization hierarchy in the data graphs. The edges of T model an *is-a* relationship between two vertices.

Taxonomy simulation redefines the problem of graph pattern matching by allowing a query vertex u in Q (labeled with l) and a data vertex v in G (labeled with l') to be matched if and only if l' is a descendent of l in the taxonomy hierarchy tree T.

1.6.10 Multi-constrained simulation

Multi-constrained simulation (MCS) [115] addresses the problem of multiple constraints on edges by providing a non-trivial extension to bounded simulation. As opposed to bounded simulation that supports only one constraint on the edge, i.e. the path length, multi-constrained simulation allows for multiple constraints on the edge attributes. In the same paper, a multi-threading heuristic algorithm was proposed for evaluating GPM queries based on multi-constrained simulation.

1.6.11 Limited simulation

Limited simulation (LSIM) [29] is a model that considers the attributes on both vertices and edges. It uses a similarity measure between vertices called *k-similarity*. Given a data graph $G\,(V, E, f_V, f_E)$ and a pattern graph $Q\,(V_q, E_q, f_{Vq}, f_{Eq}, w)$ where (i) f_V and f_{Vq} are the respective vertex labeling functions associated to G and Q, (ii) f_E and f_{Eq} are the respective edge labeling functions associated to G and Q, (iii) w is a weight function that maps each $u \in Q$ to an element in $\mathbb{N} \cup \infty$. This weight is defined based on the application domain, it could be for example the diameter of a circle of authority of a manager in a social network.

Definition 14 (k-limited similarity)**.** *k-limited similarity is a relation between two nodes v in G and u in Q denoted by $v \geq_k u$. Formally, v is k-limited similar to u if:*

(1) $f_V(v) = f_{Vq}(u)$ *when* $k = 0$,

(2) $v \geq_0 u$; $\forall e_q = (u, u') \in Eq, \exists e = (v, v') \in E$ *with* $f_{Eq}(e_q) = f_E(e)$ *and* $v \geq_{k-1} u$ *when* $k > 0$.

G matches Q via limited simulation if there exist a node mapping relation $R_v \subseteq V_q \times V$ and an edge mapping relation $R_E \subseteq E_q \times E$ such that:

(1) $\forall (u, v) \in R_v, v \geq_{w(u)} u$,

(2) $\forall (e = (u, u'), e' = (v, v'))$ in R_E, we have $f_{Eq}(e) = f_E(e')$, $(u, v) \in R_V$ and $(u', v') \in R_V$.

This model is better suited for applications in social media since it considers also the types of vertices and edges. The authors proposed also a polynomial algorithm to evaluate limited simulation.

1.6.12 Time-respecting simulation

Time-respecting simulation (TRS) [158] also revises bounded simulation for the context temporal data graphs. The data graph edges are timestamped which implies that the matching process not only looks for vertices satisfying bounded simulation constraints but also verifies the time validity of the answer. Given a query graph $Q = (V_q, E_q, f_q)$ and a data graph $G = (V, E, f, f_t)$, where f_t maps edges to timestamps. A path $p = (e_1, e_2, .., e_n)$, such that $\forall i, e_i \in E$, is considered a match to a query edge $e \in E_q$ if and only if p is a time-respecting path. A time-respecting path is a sequence of edges with non-decreasing time.

1.6.13 Double simulation

The most recent model in the relaxed GPM category is double simulation (DBS) [141]. It extends bounded simulation by imposing the matching constraints on both children and parents of a data vertex. On the other hand, double simulation can be seen as an extension of dual simulation that maps edges in the query graph to paths in the data graph.

1.7 Chapter summary

To sum up what has been presented in this chapter, graph simulation was the first model that came to relax the matching constraints imposed by subgraph isomorphism.

Three extensions to this model were proposed, bringing a new definition of similarity; dual simulation that captures more topological structures (connected components and undirected cycles), bounded simulation that matches edges with paths of bounded length, and limited simulation that considers a new similarity information called *k-similarity*. Multi-constrained simulation and surjective simulation are extensions to bounded simulation that support multiple labels. Time-respecting simulation is another extension designed for temporal graphs. On the other hand, relaxation simulation relaxes further the constraints of graph simulation and dual simulation and hence allows for larger results. Similarly, taxonomy simulation allows mapping even nodes of different labels but that are related in terms of taxonomy. Strong simulation is an extension to dual simulation that imposes locality to reduce the answer size while double simulation relaxes dual simulation by mapping query edges to reachability paths. Finally, strict and tight simulation shrink the size of the resulting match graphs to keep only answers closer to subgraph isomorphism in terms of quality.

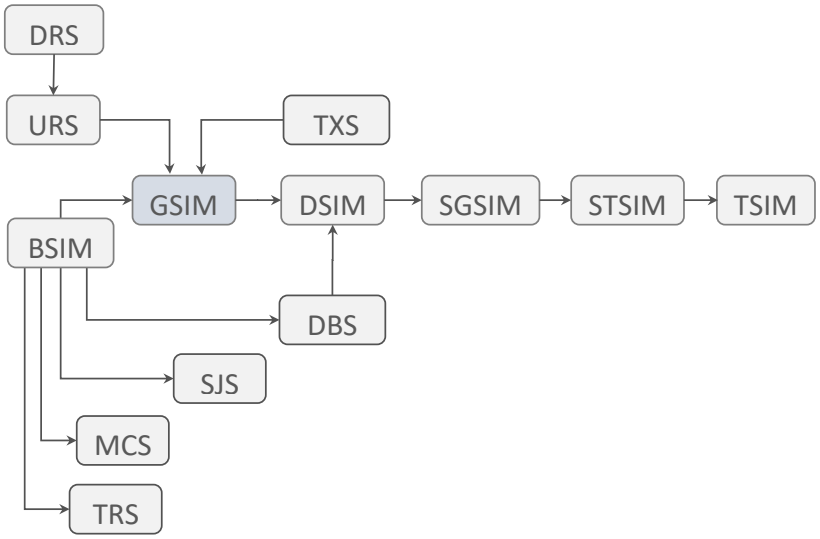


Figure 1.11 Extensions of graph simulation where the direction of the arrows indicates whether this extension came to restrict further the constraints imposed by graph simulation or to relax them when the arrow is reversed (e.g. URS).

Figure 1.11 illustrates how these extensions, based on graph simulation, make the constraints more or less relaxed or more stringent. The schema does not consider limited simulation as an extension since it redefines the notion of similarity rather than adding or omitting some constraints to / from the original model.

The following propositions were given in [39, 84, 38] to describe the relations between some GPM models:

(1) If Q matches G via subgraph isomorphism, then Q matches G via tight simulation.

(2) If Q matches G via tight simulation, then Q matches G via strict simulation.

(3) If Q matches G via strict simulation, then Q matches G via strong simulation.

(4) If Q matches G via strong simulation, then Q matches G via dual simulation.

(5) If Q matches G via dual simulation, then Q matches G via graph simulation.

Furthermore, Figure 1.12 depicts the inclusion relationships between the different models. Based on the application needs, we should decide to opt for graph simulation when interested in flexibility and short response time. Moving from graph simulation toward tight simulation and subgraph isomorphism will result in longer response time, but will allow us to get better quality results compared to dual simulation and graph simulation.

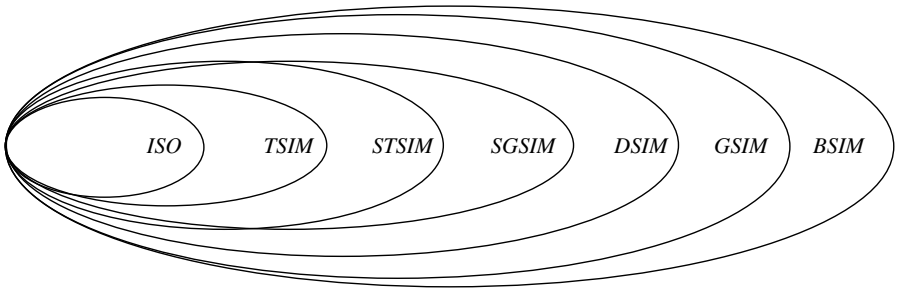


Figure 1.12 The inclusion relationships between results obtained by the different GPM models discussed.

However, even with the progress made in reducing subgraph matching complexity from exponential to a polynomial time, massive graphs containing billions of nodes and trillions of edges cannot be stored or processed in a single machine. The following chapter discusses this issue and presents the different mechanisms used in handling graphs distributed over multiple machines.

Chapter 2

Graph Pattern Matching in Massive Graphs: State-of-the-art

2.1 Introduction

Huge amounts of raw data that need to be analyzed are collected every minute. It is common to partition and distribute them over multiple machines, so that queries can be efficiently evaluated in a distributed fashion. Therefore, sequential algorithms used in finding subgraph isomorphism or graph simulation need to be adapted for this context. In this chapter, we provide an overview of the paradigms used in distributed graph processing. In a second part, we dive deeper into how these paradigms for massive graph processing are used in the context of graph pattern matching. Finally, we draw a taxonomy of the distributed GPM approaches based on multiple classification criteria.

2.2 Distributed graph processing

The most common distributed architecture used for handling massive graphs is the master-slave architecture. The master is a machine of the cluster executing tasks of coordination between different slave machines (a.k.a. workers) that are responsible for parallel computations. The master machine also performs some monitoring tasks to maintain a green state of the cluster so that all workers keep running smoothly. According to Yan *et al.* in [144], a cluster is composed of mainly four components starting from local storage, to distributed storage, communication layer and finally the computing module. (this architecture is sketched in Figure 2.1).

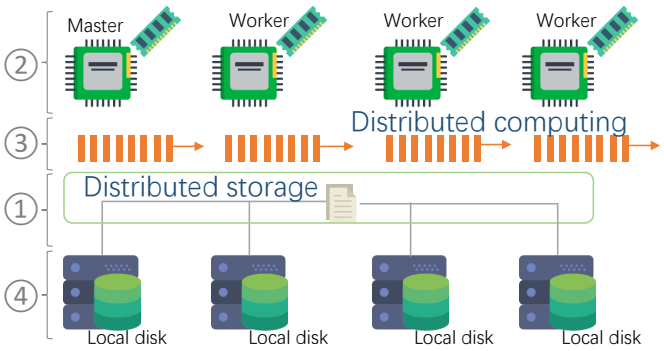


Figure 2.1 Components of a distributed architecture.

Component (1): Data graphs reside physically on distributed storage. HDFS [11] and Google's file system (GFS) [50] are two of the most popular distributed file systems used currently by several high-scale graph processing systems. These two file systems share the same aspects but do have some differences including the block size, node division and the deletion strategy adopted.

Component (2): This part of the distributed architecture is responsible of loading graphs from the distributed file system to memory. In a parallel manner, the computation module performs the computations on each machine.

Component (3): The communication layer defines the technique used to exchange messages between machines in the cluster, e.g. by using the paradigms Message Passing Interface (MPI) or Remote Procedure Call (RPC).

Component (4): Each machine is responsible of its own local storage, which can be used in out-of-memory systems that need to offload data graphs from memory to local disks during computations.

Since data graphs must be split over multiple data stores, we define a distributed graph along with other related terminologies, i.e., fragment and partitioning strategies.

Definition 15 (Distributed graph)**.** *A distributed data graph is a graph G partitioned into k fragments $(F_1, F_2, \ldots, F_k)$, each fragment is located on a separate machine. $(V_1, V_2, \ldots, V_k)$ is a partition of $G(V, E, L)$ if and only if: (1) $\cup_{i=1}^{k} V_i = V$ and (2) for any $i \neq j \in [1,\ k] V_i \cap V_j = \emptyset$.*

Fragments of a data graph can depend on each other because of the crossing-edges between vertices in different partitions and such vertices are called boundary nodes.

Definition 16 (Fragment). *A fragment F_i is denoted as $(G[V_i], B_i)$ where $G[V_i]$ is the subgraph of G stored at machine i and B_i is the set of all boundary nodes in this subgraph. We denote B_V as the set of all boundary nodes and B_E as the set of all crossing-edges.*

The partitioning strategy we have just described is commonly used for distributing graphs in the context of GPM and it is called Edge-Cut partitioning. The example shown in Figure 2.2 has three fragments $\{F_1, F_2, F_3\}$, the global boundary set B_V is composed of the vertices $\{3, 5, 8, 9, 11\}$. The alternative to Edge-Cut strategy is the Vertex-Cut partitioning where edges of the data graphs are split between the different machines. In this case, we have crossing vertices that are shared between multiple machines.

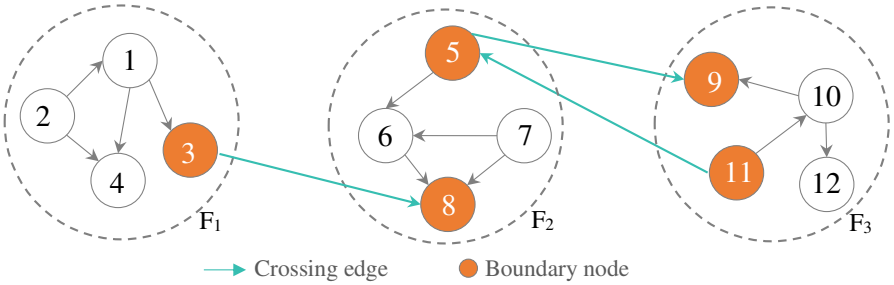


Figure 2.2 Graph fragmentation based on edge-cut partitioning strategy.

2.3 Programming models for distributed graph processing

A programming model specifies an abstraction of how a parallel algorithm is designed and executed. MapReduce [26] is one of the leading paradigms employed for processing large data sets. However, algorithms that deal with graphs are different since they incur poor data locality and little processing on every vertex of the graph, which results into MapReduce job spending most of the time in I/O operations. This limitation of MapReduce makes it unsuitable for handling distributed graphs. As a result, Pregel framework was proposed in [86], along with other paradigms that have emerged later challenging the programmer to think like a vertex. In Figure 2.3, we give a classification of the commonly employed programming models for each layer in a big graph processing system. These models are classified based on three criteria namely execution model, communication and timing. Each one offering two or more ways of defining how certain tasks are performed by the system.

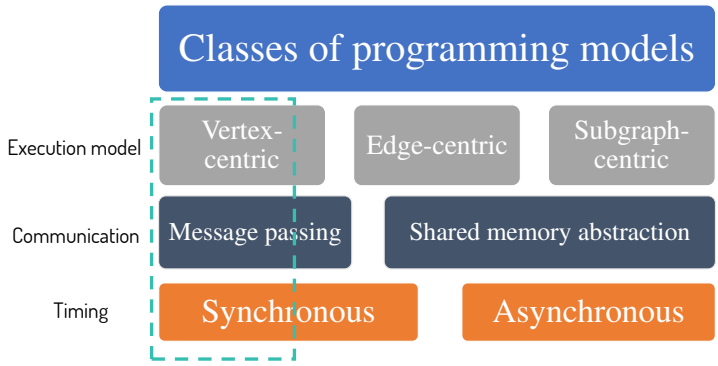


Figure 2.3 Programming models for distributed graph processing.

2.3.1 Timing

The timing defines the synchronization type used to schedule tasks in parallel. Thus, a parallel program can be either synchronous or asynchronous. In synchronous programming models like Bulk Synchronous Parallel (BSP) [129], programs are composed of super steps separated with synchronization points. After finishing a super step, a process is blocked until all the other processes, executed in parallel, terminate their tasks before moving to the next super step. Asynchronous programming models do not set any constraints on the order in which processes are executed, every process can send any message or execute a task whenever CPU and bandwidth are available.

Considered as the most common model used for parallel processing, the bulk synchronous parallel was proposed by Valiant in 1990 [129]. A BSP computer is composed of three parts: (1) Components that can perform computations locally, (2) A router for exchanging messages between each pairwise of such components, (3) A module allowing for synchronization of these components. Every component carries out a computation in a sequence of global super steps: (a) concurrent computation, (b) communication and (c) barrier synchronization. Figure 2.4 illustrates the functioning of BSP.

2.3.2 Communication

This criterion defines the way messages are exchanged between distant machines in the cluster. We use either a message passing approach or shared memory abstraction. In the first communication type, a message with its ID, containing information about the source, destination and the needed information is sent from one machine to another while in shared memory abstraction, vertices (programs) are considered as shared variables that are accessed directly from any machine in the cluster.

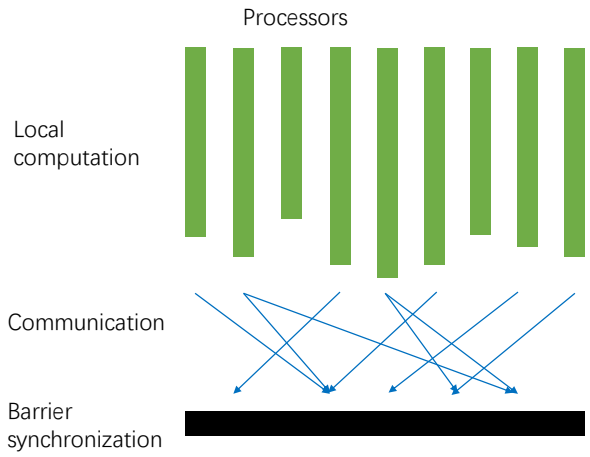


Figure 2.4 Bulk Synchronous Parallel model.

2.3.3 Execution model

The execution model determines the way a data graph is seen and iterated over. It could be vertex-centric, edge-centric or subgraph centric.

Vertex-centric

The vertex-centric concept was introduced by Google in their paper on Pregel system [86]. A graph algorithm is executed from the perspective of a vertex that can perform a list of operations as getting its ID, getting/setting its value or getting/counting its edges. A vertex is analogous to a process in the BSP model. We also define the notion of neighboring processes, i.e. a process can send or receive messages only from its neighbors (An illustration of the vertex-centric model is given in Figure 2.5).

Pregel defines the way a graph processing algorithm is executed in parallel. It is based on the BSP model, and adopts a message passing mechanism to communicate between vertices. The data graph vertices are partitioned among several workers where each vertex executes a *compute()* function, which is the same for all the vertices. A vertex can have an active or inactive state. A Pregel program consists of a number of synchronized iterations conducted in parallel by several vertices, and that can terminate only if all the processes vote to halt and there are no pending messages waiting to be achieved.

Since its apparition, this model has revolutionized the world of graph processing. Ever after, several open source frameworks were developed, adopting the same model while providing improvements on the communication mechanism, load balancing or fault tolerance. For example, Giraph [44] is the most popular framework with a large community

working on it, including engineers from Yahoo, Facebook, LinkedIn and Twitter. This system enhanced the initial model of Pregel with three improvements by first, introducing parallelism at a fine grain using multi-threading on each worker. Also, to reduce the communication cost, Giraph serializes Java objects before sending them from one machine to another. Finally, to reduce the out-of-memory problems that occur generally with longer super-steps with too many messages to send, Giraph adopts splits large super-steps into smaller ones. We can cite other systems like GPS [108], MocGraph [162], Mizan [65], GraphX [143], Pregel+[146] and GraphD [149] that propose an improvement of Pregel's model in a way or another.

Edge-centric

In this class of programming models, the parallel program iterates over the graph edges. Therefore, the computing unit is an edge that can fulfill some basic operations, e.g. get/set its value or ID, or get the value of its target vertex. An edge can also communicate with its neighboring edges by sending and receiving messages (see the illustration provided in Figure 2.6). This paradigm is well-suited for data flows, and X-Stream [106] is one of the frameworks adopting it.

Subgraph-centric

Unlike vertex-centric programming where vertices cannot know whether a vertex is located at the same machine or not, subgraph-centric programming considers the fragment of vertices located at the same machine as one computing unit. Here, communications between vertices of the same subgraph are not delayed to the next super-step but executed immediately. Such improvements result to better performance in computing time, number of messages and total super-steps (see illustration of this model in Figure 2.7). In [144], Yan *et al.* recommended this programming model in solving graph pattern matching. Several frameworks support this paradigm including Giraph++ [125], GoFFish [117] and Blogel [145], but they are basically vertex-centric.

Subgraph mining systems

On the other hand, there are subgraph mining systems that are dedicated for specific graph applications like subgraph matching, triangles finding or frequent subgraphs listing, e.g. Arabesque [124], Quegel [147], G-miner [20], Rstream [135], Fractal [27] and G-thinker [148].

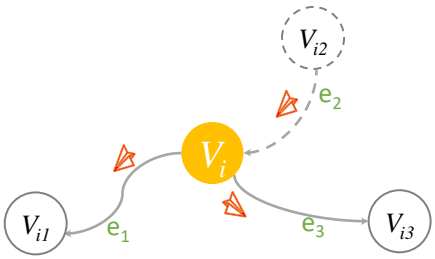


Figure 2.5 The vertex-centric paradigm.

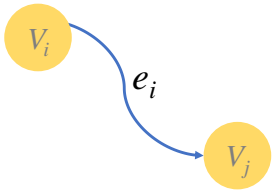


Figure 2.6 The edge-centric paradigm.

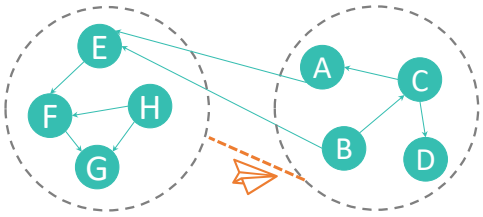


Figure 2.7 The subgraph-centric paradigm.

The most recent one is G-thinker which differs from previous frameworks that are known to be IO-bound, i.e. their processing time is mostly dominated by I/O operations. G-thinker is CPU-bound thanks to always keeping CPUs busy by switching to tasks that have their data ready instead of waiting for the current task to receive its requested data through the network. It handles two data tables: one for storing the graph available locally and another cache table for remote vertices. The latter is shared by the different subgraphs without the need to duplicate data within each subgraph. A subgraph is handled by one thread independently and hence no synchronization is needed between the different threads. Each thread, called *comper*, will construct its subgraph and work on it to evaluate subgraph isomorphism or to count the number of local triangles, etc.

GRAPE [37] is a system that is different from the above mentioned ones. It allows parallelization of sequential graph computations without adopting a TLAV paradigm. Given a data graph G that is composed of k fragments $\{F_1, F_2, \ldots, F_k\}$, GRAPE assigns each fragment to a processor. It uses partial evaluation to solve the graph problem on one fragment of the data graph, i.e. a processor P_i evaluates the problem based on the known data available in fragment F_i and exchanges messages based on the BSP model to get the missing information updated. Then, each processor updates its partial answer based on the received updates. When no processor emits new messages, an assemble procedure is triggered at a coordinator machine to combine the partial answers.

Each one of the programming models presented in this section is well adapted for designing algorithms that respond to specific problems but not to others. For example,

the vertex-centric programming model is adapted for algorithms based on communications where the vertex program does not need to access neighborhood information, e.g. Single Source Shortest Path (SSSP), Page Rank. On the other hand, the subgraph-centric programming model is better suited for algorithms where the vertex-program requires its neighborhood information in addition to its local information to update its local context, which is the case for GPM. Finally, the edge-centric model is used to design flow-centric algorithms where the main processing is executed on the graph edges and not on its vertices.

Other programming models also exist, but they are not as commonly used as the ones discussed above. For example, the partition-centric model is a special case of subgraph-centric where all nodes and edges that are located in the same machine are considered as one computing unit. Moreover, neighborhood-centric is another model considering that the scope of a single vertex is composed of its local state and the states of its multi-hop neighborhood. We also find other programming models similar to the vertex-centric one such as Scatter-Gather and Gather-Sum-Apply-Scatter (GAS). These two models are slightly different from the vertex-centric model. For example, Scatter-Gather defines for each vertex two functions Scatter and Gather. The Scatter function defines and sends messages to its neighbors, while the Gather function allows a vertex to read the received messages and edit its local information. Both functions are executed in the same iteration respecting the BSP model. A comprehensive survey of the programming abstractions used for distributed graph processing can be found in [61].

2.4 Distributed graph pattern matching approaches

There are several previous works that reviewed and proposed different taxonomies of sequential subgraph isomorphism algorithms such as [23, 46, 43, 131]. Other works like [73, 14, 120] conducted a benchmarking to compare some of the very well known subgraph isomorphism algorithms on the same environment. Unlike previous works, we provide in this section a taxonomy of the distributed GPM approaches based on graph exploration and which adopt the emerging TLAV paradigm and its derivatives. We also discuss these programming models and how they improve solving GPM in massive graphs. This section synthesizes the work conducted recently in GPM on distributed graph processing systems with a narrow focus on relaxed matching that has gained an increased interest during the past ten years.

2.4.1 Querying distributed data graphs

When evaluating a subgraph matching query, fragments of the data graph can be processed separately. Actually, the real challenge here is how we should evaluate GPM for the boundary nodes. Because, as we mentioned earlier in Chapter 1, to verify if a data vertex is a match to a certain query vertex, the matching information of its neighborhood is necessary in the evaluation of the model's constraints. Several answers to this question have been proposed in the literature, we present in this section some of the most relevant.

To find matches of a query graph in a data graph that is fragmented over multiple sites, a general algorithm follows these steps: a coordinator machine receives the pattern graph Q, then transfers it to the workers which, in turn, perform a local evaluation of the query resulting into a partial matching set. Such partial matches are then sent to the coordinator to combine them into one final result. Different mechanisms and techniques are involved in the evaluation of GPM for the set of boundary nodes. This can be achieved by either communicating intermediate results between workers or even shipping parts of the fragmented graph from one worker to another. We notice that some GPM propositions rely on the existing frameworks of distributed graph processing while others simply provide a *from-scratch* implementation of the distributed algorithms.

Different evaluation criteria are used to measure the performance of distributed graph algorithms. Firstly, we have *data shipment* as the amount of data exchanged between distinct machines during the execution of the algorithm. The second criterion is *visit times*, the number of messages that are exchanged between workers in the cluster. Finally, there is the *make-span* (a.k.a. response time) that refers to the duration between receiving the query and finding the result.

2.4.2 Distributed structural graph pattern matching

In this section, we review the latest works proposed for finding subgraph isomorphism in a distributed environment. We classify them based on the timing and execution model applied. The studied works are divided into four classes: synchronous master-slave, asynchronous master-slave, synchronous vertex-centric and asynchronous vertex-centric. The works presented under this section are considered structural because they aim to find answers that have the exact same structure as that of the input query.

Synchronous master-slave

This model synchronizes the computation through super-steps separated by communications between workers or between workers and a coordinator. The main difference between an algorithm using a master-slave execution approach and the one adopting subgraph-centric is that the latter is fully centric while the first one involves a coordinator for exchanging data between different workers or to orchestrate the graph matching process. Both consider the subgraph as one computing unit on which all the operations are made at the same time.

Peng *et al.* [96] adopt partial evaluation to find matches locally to each partition of the distributed data graph. For the cross matches that include data vertices between multiple fragments, each worker evaluates subgraph isomorphism based on its local information and leaves some matches unanswered. Then, the partial matches are assembled on a coordinator site to get cross matches. The assembly algorithm collects all the partial matches, joins them by starting with two partial matches from connected fragments and considers a new match at a time. If the match is complete, another assembly is started, otherwise, other partial matches are joined to form a final answer. The authors introduce a partitioning of the partial matches to avoid unfruitful joins. Partial matches that share an internal data vertex are collected in the same partition and the assembly is made between different partitions. Another distributed assembly that follows a BSP master-slave was proposed: in the first super-step the partial-evaluation part is conducted locally, then partial matches are shared between workers to assemble the cross-matches in a distributed way such that an intermediate result is communicated each super-step. A one-direction communication is used to avoid duplicate answers.

To parallelize TurboISO on a fragmented data graph. Fan *et al.* [37] expand each fragment F_i with data vertices from other fragments such that subgraph isomorphism can be evaluated locally on each worker. Subgraph isomorphism is determinable locally for a fragment F_i if and only if, for every vertex $v \in F_i$, v can reach all its neighborhood within d hops such that d is the diameter of the query graph. This work is based on GRAPE that offers the three functions: partial evaluation (PEval), incremental evaluation (IncEval) and assemble. In PEval, each worker gathers the d-neighborhood of its boundary nodes. Then, it sends them to the coordinator that, in turn, sends each d-neighborhood to where its center resides. IncEval is simply a sequential implementation of TurboISO. Finally, the assemble function takes the union of the subgraph isomorphism matches returned by each worker. The limitation of this approach is that large parts of the data graph

may end up duplicated among workers, especially for small diameter graphs and when dealing with complex queries of large diameters.

Asynchronous master-slave

We present the asynchronous approaches that allow workers to communicate with the coordinator machine without synchronization barriers, hence allowing to reduce their waiting time and consequently optimizing the overall response time.

The first approach under this category is COSI [12]. COSI relies on the RDF format[1] for describing both the data graph and query graph. The architecture used for evaluating subgraph isomorphism consists of a set of workers that communicate directly with each other and a master node behaving as an interface with the end user. This system uses a partitioning strategy that minimizes the edge cuts between different sites of the distributed architecture. It is based on the idea of keeping on the same machine every two vertices that are likely to be retrieved together by a random query. The query is received by the master that routes it to one or more workers. The worker process the query locally and send back a complete subgraph matching answer to the master node. The local algorithm uses a local index for retrieving neighbors of a data vertex. It defines an order for processing the query variables, finds data candidates of the current query variable, substitutes it with a data vertex v then searches for candidates of the next variable in the set of neighbors of v. It continues executing this operation until no candidates are found in the current partition. At the end of this step, the algorithm verifies if a valid match set is obtained or not to send back an answer to the master. So here, the algorithm explores the tree search in a depth first search fashion where each branch is traversed in parallel to reduce the processing time. This approach for finding subgraph isomorphism performs well in terms of time and data shipment. However, even though neighborhood indexes accelerate and facilitate the access to the data graph vertices, they have generally a super-linear creation time.

In [122], a GPM system developed on top of Trinity [112] was proposed that is based on query decomposition and graph exploration. The aim is to reduce the amount of data shipment involved in the processing and also limit the size of intermediate results by avoiding joins. It includes three major steps: *Query decomposition and STwig ordering*, *Exploration* and finally a *Joining phase*. The query graph is firstly decomposed into STwigs (trees of two levels). After that, a selection algorithm is executed to find the best

[1]The RDF format stands for Resource Description Framework, which is a graph data model for describing graphs using a set of triples of the form: *subject-predicate-object*.

order in which these STwigs will be processed minimizing further the size of intermediate results (the output is an ordered sequence of STwigs $S_1, S_2, \ldots, S_N$). The STwigs sequence is next delivered to each worker which will process it, explore the graph while joining the partial matches obtained for each STwig $(M_1, M_2, \ldots, M_N)$, then send back the result R to the coordinator machine. The most important contribution here, is using exploration instead of joining whenever it is possible, which considerably reduces the size of intermediate results. The local evaluation of subgraph isomorphism requires accessing STwig root children located on different workers, and hence involves network communication. Compared to the propositions from previous approaches that rely on indexes, this work has shown performances in response time when handling a graph of one billion nodes. The only index required was intended to accelerate the access to each data vertex by mapping identifiers to their labels.

Synchronous vertex-centric

We review under this category the synchronous vertex-centric approaches proposed for GPM.

Lighthouse [40] is a system that evaluates subgraph isomorphism and accepts as input a graph query language equivalent to Cypher (the one used by Neo4J [59]). It is an improvement for the approach proposed in [122]. Similarly, the query graph is decomposed into STwig data structures that will be matched individually using the vertex-centric approach, then, the matching results are joint to build the match result. This work discusses the possibility of simply processing the STwigs in parallel then joining them sequentially. Such optimization may require more memory to hold the intermediate results but will definitely reduce the number of super-steps and hence the query processing time. So here, the main contribution was taking the query decomposition approach of [122] and transforming it into a vertex-centric one and thus, enabling the system to achieve better performance and gain in scalability.

PSgL [113] evaluates subgraph isomorphism based on a parallel traversal of both the pattern graph Q and the data graph G to map vertices in Q with vertices in G. Each data vertex v is first mapped to a query vertex u (selected based on a cost function that minimizes the size of intermediate results). Then, in the next super-step, the mapping is expanded by mapping neighbors of u to the neighbors of v if the mapping satisfies the matching constraints. In this step, u is colored black, its neighbors are gray, and the rest of the pattern is white. In the next super-step, this mapping (tree) is forwarded to the neighbors of v that can expand it further. A neighbor v' will expand a gray vertex (that

becomes black), its white neighbors (not yet mapped) are now mapped to neighbors of v'. Every mapping is expanded in the same way until all the query vertices are mapped (it returns the subgraph) or there is no way to expand it, i.e. no remaining neighbors of v'. The authors propose heuristics that guide the tree forwarding among the data vertices. This approach that adopts a BSP-based vertex-centric paradigm was implemented on top of Giraph.

In [48], the authors use a Single Sink DAG extracted from an undirected pattern to guide the evaluation of subgraph isomorphism in a distributed data graph. The sink of the DAG is a query vertex that has no outgoing vertices, only incoming vertices. Every data vertex candidate to this sink will be responsible for collecting a subgraph match if it exists. Transition rules are attached to every edge in the DAG, then each data vertex considered a candidate to a non sink query vertex will generate and transfer the received messages that end at the sink data node. The data node accumulates the received messages to form a correct match or drop the messages if there are non respected subgraph isomorphism constraints. Furthermore, to support the detection of the same query on an evolving graph, any update event triggers a message broadcast received by the data vertices that will verify the transition rules and transfer these messages to their neighbors if the transition rules become unsatisfied. The receiving nodes will do the same recursively until reaching the sink nodes that will update their matching information accordingly.

Asynchronous vertex-centric

Reza *et al.* [102] proposed recently a distributed algorithm for evaluating subgraph isomorphism on one trillion-edge graphs based on HavoqGT, a platform for asynchronous vertex-centric graph processing [94]. The algorithm is composed of two phases. Firstly, a Local Constraint Checking procedure eliminates aggressively the vertices that do not satisfy the subgraph isomorphism constraints, several iterations can be executed until no vertex is eliminated. Then, a Cycle Checking procedure is executed only for cyclic queries. The first phase can successfully eliminate all the data vertices that do not participate in answering the input query. But when the query graph contains cycles, there can still be remaining vertices that do not verify the whole matching constraints. The idea proposed by the authors consists of checking the existence of cycles based on some cyclic constraints introduced by the user to keep only matches respecting them in the data graph. If the algorithm finds a cycle having a defined length d in the query graph then, each data vertex already matched to a member of this cycle will execute a token passing

routine. This routine is used to verify if it is also part of a cycle with the same length in the data graph. The data vertex sends a token to its neighbors in a diameter of d such that if the data vertex does not get back its token at the end of d iterations, it will be eliminated because it is not part of any cycle. This approach showed good results for handling undirected graphs of more than one trillion edges.

PruneJuice [104] uses graph pruning to evaluate subgraph isomorphism. The data graph is pruned based on a set of constraints extracted from the query graph so that every data vertex respecting these constraints will be part of a match. The embedding enumeration can then be performed on the resulting pruned graph having the smallest size possible. These constraints are divided into local constraints such as the vertex label and the neighbor labels and that will be verified by exchanging direct messages between the data vertices. Non-local constraints require walks within the data graph such as path search, cycle checking when a query label is repeated in the pattern or a query edge is part of a cycle respectively. Both types of constraints require a token passing to verify the cycle membership and to verify that two data vertices having the same label have a specific labeled path . Another type of non-local constraints that ensure that the pruned graph contains exactly all the subgraph isomorphism matches is the Template-Driven Search (TDS). The TDS prunes further the data graph by walking through the pruned graph and checking for the union of cycle constraints and the union of repeated label constraints and eliminating every data vertex not satisfying them.

Furthermore, Reza *et al.* [103] use k edit-distance to find approximate matches. Here, a k edit-distance is the subgraphs of the data graph that become isomorphic to the input query upon the deletion of up to k edges from the query. The authors have already defined in [104] a set of local and non-local constraints that help pruning the data graph to keep only exact matches and thus enumerate subgraph isomorphism embeddings. In this contribution, they define a set of prototypes from the query graph (variations of the query graph within distance k) and then, they use the common local and non-local constraints to avoid reevaluating subgraph isomorphism for every single prototype separately. A data vertex that satisfies a non-local constraint for $k = \delta + 1$, will also satisfy this constraint for $k = \delta$. So, every time a data vertex meets specific non-local constraint x, this information is saved in the context of that vertex so that it can be reused in the constraints verification for prototypes of smaller k.

In [119], a vertex-centric approach was presented for performing topology pattern matching (subgraph isomorphism) in a distributed environment composed of a Wireless Sensor Network (WSN). Each sensor in this WSN is analogous to a data vertex having a partial

view of its closest neighbors in the network topology. Different from the previous works, this proposition aims to reduce the disk storage and message overhead in a resource-limited context. There is an initiator node v_0 that starts the matching process and builds a partial match set if possible. Then, it will ask its neighbors to further extend the partial match set with their local views of the graph. When a node receives a request from its neighbor to complete the match set, if it finds a complete match set, it will send back the result to v_0. Otherwise, it will propagate its partial answer to other nodes in the data graph. The authors confirmed that such approach can also be used in contexts other than a WSN. However, the solution was only tested on 400 nodes.

2.4.3 Distributed relaxed graph pattern matching

Since the recent works on graph pattern matching are mostly oriented toward reaching higher performance and scaling up the existing algorithms into larger graphs, we will focus on distributed relaxed GPM. Graph simulation along with its induced models gave an evidence of superiority over its conventional counterpart, and this is due to relaxing the structural constraints imposed by subgraph isomorphism. The works discussed here are divided into four categories. First, we have the master-slave execution model with both synchronous and asynchronous timing. Then, we have BSP-based approaches adopting a vertex-centric or subgraph-centric abstraction.

Synchronous master-slave

We review under this section a work that distributes processing following a master-slave model where the workers communicate only with the master.

Graph simulation: Similar to the work presented in [36], graph simulation is evaluated on top of GRAPE [37] by incrementally updating the match set of a query vertex u in each fragment F_i. Then, the matching information of a vertex v in F_i is needed by their original fragments to complete their matching. So, at the end of partial evaluation (PEval), each processor sends a variable $X(u, v)$, with true or false value to indicate whether $u \in Q$ is matched to $v \in G$ or not, to the coordinator. The coordinator will then propagate these variables into the processors that need to complete their matching information. Upon the reception of these variables (in IncEval), each processor will verify the matching constraints and update its local variables accordingly. If a variable $X(u, v)$ switches to false, it will propagate this update by sending a message to the coordinator, which in turn propagates the change to the appropriate processors until there is no

update issued. Finally, the coordinator will execute the assemble function which is the union of partial matches generated by each processor.

Asynchronous master-slave

We present under this section the recent works adopting an asynchronous programming model where the evaluation of the query is triggered on a master and then executed on the different worker machines.

Graph simulation: The first work ever to address graph simulation in a distributed environment was published in 2012 [85]. The authors adopted a message-passing approach for solving graph simulation in the context of big graphs. Their solution is based on the following findings: they proved that evaluating separately the connected components (CCs) of a data graph allows us to find a correct matching set by just computing the union of different partial matches found for each CC. First, each worker receives the query graph and finds candidate vertices for each query vertex in parallel with other workers. After this filtering step, the graph will contain a set of connected components, some of which reside on the same worker while others are distributed over multiple machines. Each worker executes locally the algorithm localHHK (an updated version of HHK algorithm [57]) on the CCs that reside fully on it. Then, it sends back its partial matching set to a coordinator machine, along with the information about vertices in a CC across several machines. The coordinator executes a scheduling algorithm to assign the remaining CCs to a set of workers while minimizing make-span and data shipment at the same time. The assignment is then sent back to these workers that will ship the CCs to their corresponding workers. The assigned workers will perform local graph simulation on the received CCs and send back their results to the coordinator. At the end, a set of partial matches for each connected component will be available on the coordinator that builds a final matching set from their union. It is important to mention that the proposed solution lies much more on distributing the data while executing the same algorithms in parallel (data-parallel approach) rather than on the notion of distributed processing.

In [36], Fan *et al.* proposed a distributed algorithm for solving GPM via graph simulation with bounded response time and data shipment. Each worker maintains locally a dependency graph for tracking linked vertices that are situated on other workers. To perform local graph simulation correctly, the boundary nodes and crossing edges are duplicated on both the source and destination machine and treated separately. Here, instead of shipping parts of the data graph from one site to another, a Boolean variable

$X_{(v,u)}$ is used to indicate whether the data vertex v is a match to the query vertex u or not. If the two vertices v and u share the same label, then there must be at least one matching relation between the child node u_i of u and a child node v_i of v to add this tuple to the match set. This constraint is formulated as follows: $X_{(v,u)} = \vee(\wedge X_{(v_i,u_i)})$. To evaluate graph simulation, the algorithm simply propagates the values of these variables inside each worker such that each variable will be written in function of only the variables that relate to virtual nodes (nodes copied from other fragments) in this machine. Then, a message passing is conducted where each machine broadcasts the values of its in-node variables (nodes considered as virtual-nodes in other fragments) following the dependency graph. After receiving new values of variables related to its virtual-nodes, each machine propagates this information among its local variables. Next, it updates them and conduct a new message passing phase if one or more of its in-node variables were updated. The algorithm converges when no more messages are exchanged. Adopting asynchronous communication made it possible for this system to achieve better response time compared to the solutions proposed in [39].

Strong simulation: In an earlier paper [84], the outlines of a distributed algorithm were given for computing strong simulation. It was the first algorithm proposed in this category but with no performance guarantees. The algorithm simply ships the contents of each data locality ball that is stored across different machines to the machine with the smallest index, then it evaluates strong simulation locally. After this step, the workers send their partial results to a coordinator machine that merges them into a final match set.

In [133], the process of finding matches of a pattern graph is split into two phases: An off-line redistribution phase is conducted to make the data graph locally determinable, then an on-line evaluation phase is held for evaluating strong simulation. The main challenge brought by strong simulation in the distributed setting is the data shipment incurred. Indeed, we need to ship data vertices from different machines for each locality ball created. Thus, the larger the query graph, the larger its diameter d_q and the larger the locality balls. As a reflection, for reducing the size of these balls and hence the amount of data shipped, we can think of reducing the diameter of query graphs. The idea on which this paper is based is to set a maximal query diameter d_q, and use it to further optimize data shipment by redistributing the data graph offline. The redistribution is made such that all the balls centered at $v \in V$ will reside on the same machine. An online processing algorithm is conducted at the reception of every query graph Q. The online phase consists of partitioning the query graph into partition-trees (a tree

rooted at each query vertex and having as diameter d_q) locally on each worker. Then, p-match-graph is computed for every partition-tree, knowing that p-match here is a revision of graph simulation allowing to find partial results of a query graph locally when using its partition-trees as input. Then the local p-match-graph obtained is sent to a coordinator. This latter merges the local p-match-graphs received into one final p-match-graph, then conducts strong simulation locally to find a final answer to the query. Even though this algorithm achieved controllable data shipment in the online phase, it is not well adapted for highly dynamic graphs. (a redistribution phase has to be executed after each update event). Also, this proposal puts too much effort into reducing the data shipment at the expense of response time, another drawback is that no data shipment was involved between different workers but the algorithm still uses network traffic between workers and master (which may lead to a load-balance problem). Such proposition does not take advantage of the distributed systems and their optimizations.

Synchronous vertex-centric

We find several works adopting this programming model to evaluate graph simulation, strong simulation, strict simulation or tight simulation. All the studied works adopt a BSP-based vertex-centric abstraction.

Graph simulation: We review here three different algorithms of graph simulation and dual simulation since the latter is a simple extension of the first.

In [39], the first algorithm to adopt a BSP-based vertex-centric approach to evaluate graph simulation was proposed. Each vertex in the data graph has a local context composed of two variables: a match flag and a match set. Upon receipt of the query graph, it sets its matching flag to true if its label is the same as one of the query vertices', and the match set will contain identifiers of the similar query vertices. Then, if the match flag is true, it will also ask its children for their match sets. In the second super-step, vertices that receive a request while their match flags are true will respond with their match sets. In a third round, each vertex receiving information from its children will update its match flag and match set according to the received match sets and the query graph, it will then inform its parents of any update made. In the fourth round and beyond, each vertex that receives an updated match set will update its match flag and match set accordingly and inform its parents. This algorithm stops when all the vertices stop sending messages, it takes at minimum 4 super-steps to evaluate graph simulation. The same goes for dual simulation when each vertex will ask for information about its

children and parents. Similarly, In the following super-steps, when a match set is updated, the vertex will inform its children and parents about the updates made.

In [109], S2X was proposed for evaluating dual simulation on RDF data, on top of GraphX. Spark allows for data-parallelism thanks to its *Resilient Distributed Datasets* (RDDs) that apply an efficient partitioning strategy permitting to minimize the cut-edges between worker machines. Moreover, graph-parallel computation is made possible with its vertex-centric programming model called GraphX. The query graph consisting of a set of triplets (edges) is broadcast to every data vertex that will verify dual simulation constraints locally. Then, it will exchange its partial match sets with its neighbors to further validate the matching sets built in the first super-step. At the end and after receiving the partial results obtained by each data vertex, a joining phase is executed to construct the final answer. If we compare the contribution made by this paper with the algorithms in [39], here the authors addressed RDF graphs that are characterized by labeled edges and that may have multiple edges between two nodes.

Another recent work proposed two vertex-centric algorithms to evaluate graph simulation for acyclic queries and cyclic queries separately [77]. The first algorithm uses a Boolean variable local to each data vertex which holds the matching information that can be either True, False or Unknown. The algorithm is executed in three different stages: Initialization, Message Broadcast and Match trial. In the first step, matching information for data vertices verifying the match constraints are set to true (the vertices candidates to a query vertex that has no children in the query graph), the remaining vertices set their values to unknown. Then, a message broadcasting step is carried out to propagate these truth values to parents. The vertices having their local matching variables set to True send their labels to their parents and go inactive. On the other side, vertices with unknown values will wait for messages from their children to complete their matching information. Once a data vertex with unknown value receives the labels from all its children, it will compare this label set with the label set needed for satisfying the graph simulation constraints. If the two sets are equal, then it sets its local variable to true and inform its parents. Otherwise, it will set it to false. But, if it does not receive all the matching information from its children, then it will wait for the next super-step. In this super-step, messages from the previous super-step will be delivered and data vertices that did not yet compute their matching value will evaluate it depending on the set of labels received from children. The algorithm proposed is not general since it sets another constraint on the query graph. Indeed, it does not allow repeated labels while it is almost impossible to find practical queries in real-world applications with a no repeated labels.

Relaxation simulation: Gao *et al.* [47] proposed two BSP-Based vertex-centric algorithms to evaluate their models URS and DRS on distributed graphs. In URS, the authors define the condition set of a query vertex u in Q as the set of labels that should be satisfied if two vertices are matched together. It is composed of the set of labels of the children of u that can be substituted by its grand children's labels. The initial set is composed of the children of u, then new combinations are generated by replacing a child in the initial set with its children set. To evaluate an input query in a distributed way, the query is first broadcast to all the data vertices in the graph. Then, each node sets its match flag true if its label is equal to that of a query vertex. It propagates its label to its children and parents during the next two iterations. Then, in the third super-step, if its match flag is true, it computes its condition set and checks whether there is a satisfied condition set among the labels of its children. If yes, it votes for halt, otherwise it sets its match value to false and informs its parents to update their matching information accordingly. The same goes for DRS, such that both grand-parents and grand-children may substitute the parents and children of a query vertex respectively in the condition set. In this case, a data vertex will compute both parent and child condition sets and exchange its matching information with its parents and children. Data vertices that do not find candidates satisfying the two condition sets will set their match value to false and inform their parents and children so they update their matching information accordingly. It is the only distributed approach found in literature that treats a model as interesting as URS and DRS. Nevertheless, the proposed algorithms only consider patterns with unique labels which is not always the case in real-word applications.

Strong simulation: In [39], the authors proposed an algorithm that, firstly, conducts dual simulation in a distributed manner. Secondly, each vertex having a match flag set to true will execute a BFS traversal to collect the contents of a ball, of which it is the center, with a radius equal to d_q (diameter of the input query). It then evaluates dual simulation locally to each data vertex to get only matches falling inside these locality balls. This approach is not scalable as it showed poor performance guarantees when dealing with average data graphs. Its limit lies in the way data locality are generated, because if we try to duplicate the neighborhood of every single node that has a true match flag, up to d_q hops, this will certainly create a bottleneck as the query graph gets larger. That is why the authors proposed a distributed algorithm for strict simulation which scales better as it uses only the dual match graph. The authors used the same approach to evaluate tight simulation in [38], i.e. the locality balls are collected by a subset of the dual match vertices.

Furthermore, G-thinker [148] can be used for evaluating strong simulation as the locality balls creation will not require the duplication of neighborhood data inside every single subgraph generated leading to an intermediate data graph of exponential size. Since strong simulation does not require communication between the subgraphs whose diameter is well defined, using G-thinker will allow it to scale to very large graphs.

Synchronous subgraph-centric

Generally, when we use the vertex-centric model, it becomes difficult for the system to achieve an optimal make-span since vertices do not know whether their neighbors reside on the same worker or not. Consequently, even the use of information from vertices on the same machine needs to be delayed to the next super-step. This generally leads to a greater response time. Thus, approaches that focus on the multi-hop neighborhood of a data vertex, i.e. subgraph-centric approaches, can be well adapted for distributed GPM.

Graph simulation: In [62], the authors focused on the big data velocity issue, i.e. streaming graphs by adopting a BSP-based subgraph-centric approach on top of GPS. The data graph is partitioned into several subgraphs that are mapped to the GPS vertices. The proposed incremental algorithm evaluates graph simulation in a vertex-centric fashion (an improvement to the proposition in [39]). Each data vertex is analogous to a BSP process and uses two local variables; a match flag and a match set. They take into account different kinds of events that come in to update the data graph, e.g. vertex addition/removal, edge addition/removal and attribute addition/removal. To evaluate the same query while having a graph updated every couple of seconds for example, the algorithm maintains the previous matching information instead of resetting the variables and starting from nothing. The update event will trigger a new iteration of graph simulation. The vertices concerned will update their match flags and match sets accordingly, inform their children that will conduct the basic updating process, and so on. In streaming graphs, the data graph is updated quite often with the reception of new events in a continuous way. Here, the main improvement brought was to avoid computing the maximum perfect match from zero for the same query graph whenever the data graph is modified. Also, the use of subgraph-centric programming model reduced the amounts of network traffic significantly.

Several other incremental algorithms were already proposed, but they do not deal with the problem at high scale (no distributed algorithms were found except this work). Fan *et al.* proposed in [34] three algorithms for incrementally evaluating graph matching via graph simulation, bounded simulation and subgraph isomorphism. A recent work

addressing the same problem was done in [154] where the authors proposed incremental algorithms for performing dual simulation when no more than half of the data graph edges are updated.

2.4.4 Classification of distributed GPM approaches

Table 2.1 summarizes the presented GPM approaches, where every work is cited with the programming paradigm adopted and the GPM model applied. We also give the size of the largest graph tested within each work. The different approaches were not tested in the same context. In addition to the size of graphs, there is also the nature of graphs that differs from one experimentation to another; e.g. the density of the data graph, number of the cycles appearing in it, and the number of present labels. Moreover, these different works use very different system configurations as the number of CPU cores varied from tens to thousands of cores. We cannot compare the response times since the size and nature of patterns used in the experiments change from one to another work. We give such details only to show how far the authors went in their experimentation to validate their approaches. The largest graph that has been tested contains 137 billion vertices of synthetic data. It is common to reach such size in the current big graphs issued from social networks, the World Wide Web and many other applications.

Some of the conclusions drawn out of this comparative study are as follows. First, graph simulation is a promising model that aims to address the challenges of actual GPM applications like network motif discovery, finding communities in social networks, and many other applications. That is why it is important to consider this research area as one of the main current challenges of subgraph matching on big graphs. Second, strong simulation is a challenging model that should be considered in future work. The way its algorithm was distributed is not efficient and needs to be revised. Although, Ref. [133] provided a new distributed algorithm for strong simulation, it does not benefit from the aforementioned programming paradigms. Furthermore, redistributing the data graph after every single update of its contents would arise a performance bottleneck when dealing with highly dynamic graphs. Finally, even though relaxed graph pattern matching is highly common among the recent works addressing GPM for social networks, subgraph isomorphism approaches are also present and they showed high performance guarantees when tested on massive graphs.

In Figure 2.8, we provide a taxonomy for the distributed algorithms we discussed earlier. Our classification is based on three selection criteria: the programming paradigm used,

Table 2.1 Distributed graph pattern matching approaches.

Work	Year	Model	Timing	Execution model	size of graph
[12]	2010	SubIso	Async.	Master-slave	778M edges
[122]	2012	SubIso	Async.	Master-slave	1B nodes
[40]	2014	SubIso	BSP	Vertex-centric	100K nodes
[48]	2014	Inexact SubIso	BSP	Vertex-centric	105M nodes
[113]	2014	SubIso	BSP	Vertex-centric	42M nodes
[96]	2016	SubIso	BSP	Master-slave	1.3B nodes
[102]	2017	Inexact SubIso	Async.	Vertex-centric	68B nodes
[104]	2018	SubIso	Async.	Vertex-centric	137B nodes
[119]	2018	SubIso	Async.	Vertex-centric	400 nodes
[37]	2018	SubIso Graph sim.	BSP	Master-slave	65M nodes
[103]	2020	Inexact SubIso	Async.	Vertex-centric	34B nodes
[84]	2011	Strong sim.	Async.	Master-slave	548K nodes
[85]	2012	Graph sim.	Async.	Master-slave	875K nodes
[39]	2013	Graph sim. Dual sim. Strong sim. Strict sim.	BSP	Vertex-centric	3M nodes
[36]	2014	Graph sim.	Async.	Master-slave	3M nodes
[38]	2014	Tight sim.	BSP	Vertex-centric	5M nodes
[47]	2014	Relaxation sim.	BSP	Vertex-centric	1M nodes
[109]	2015	Dual sim.	BSP	Vertex-centric	10M nodes
[62]	2017	Graph sim.	BSP	Subgraph-centric	9.5 M nodes
[133]	2018	Strong sim.	Async.	Master-slave	1M nodes
[77]	2018	Graph sim.	BSP	Vertex-centric	403K nodes

the pattern matching model evaluated and whether the algorithm proposed is synchronous or asynchronous.

Based on this comparative study, we notice that both synchronous and asynchronous approaches were employed with more works adopting the BSP model. The master-slave paradigm is the oldest one used in distributed graph processing. The vertex-centric approaches form the largest part of these works, and finally subgraph-centric is the least frequent paradigm with only one work addressing graph simulation. Even though Ref. [148] proposed an asynchronous subgraph-centric algorithm to evaluate subgraph isomorphism on G-thinker, they did not give the algorithmic details of their approach. The edge-centric paradigm was not adopted in distributed GPM. GpSM [127] and some join-based algorithms process the data graph edge by edge. Their distributed version

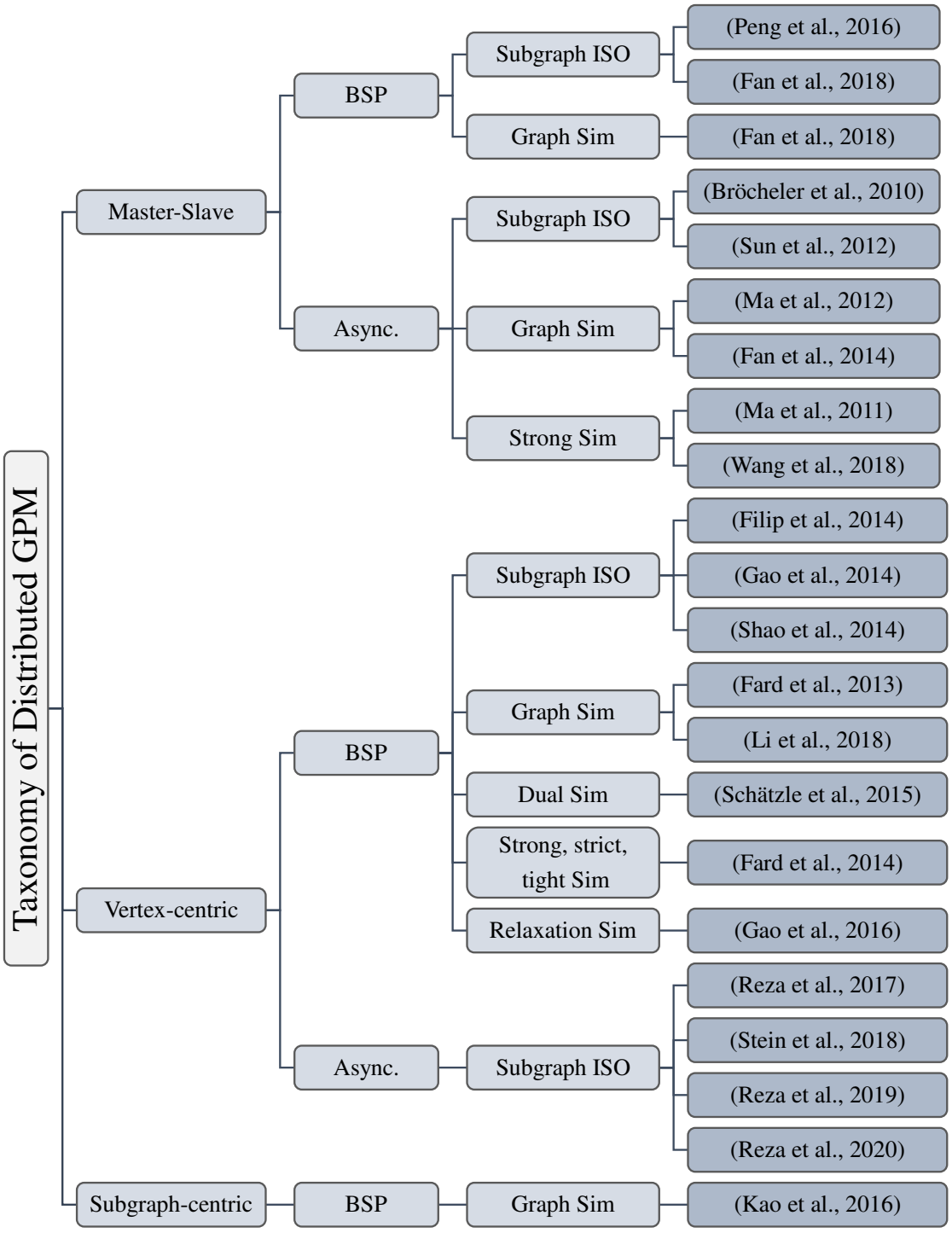


Figure 2.8 Taxonomy of the distributed GPM approaches.

can be implemented in an edge-centric way. Similarly, this paradigm can be applied to implement graph simulation that maps edges of the two graphs. Furthermore, the

edge-centric paradigm naturally supports dynamic graphs as events occur on the graph edges and need to be processed at the edge level. Therefore, it is interesting to explore this area of research by proposing edge-centric GPM approaches. Besides, we note that the synchronous algorithms can be easily designed and they are better understood compared to the asynchronous ones. A synchronous algorithm endures less communication costs. However, an asynchronous one allows a considerable improvement in the response time. Actually, the computing units (vertices, edges or even subgraphs) do not have to wait the termination of another process to send and receive requests or data to and from their neighbors.

The taxonomy provided categorizes only the existing distributed GPM approaches, some combinations were not employed and that is why we do not see them in this taxonomy. An open question would be: Why do all combinations not appear in the literature of distributed GPM? One reason could be that the vertex-centric paradigm, which is the most common one, has been widely studied in the field of networks where machines behave naturally in a vertex-centric way. Moreover, as we stated earlier, the most intuitive way to evaluate subgraph matching is by a subgraph-centric algorithm since the neighborhood of a data vertex is needed to decide whether it is a match or not. This could be a reason that makes the subgraph-centric paradigm more interesting compared to the vertex-centric one. However, subgraph-centric computing needs an additional level of abstraction to make the neighbourhood of multiple hops available locally to every single node in the graph. Such an abstraction layer is responsible for shipping data from one node to its neighbors whenever a local update is made, which may cause additional data shipment increasing the overall response time. Apart from the subgraph mining systems proposed such as Arabesque, Rstream or G-thinker, few works adopt this model even though adding an abstraction layer would certainly reduce the efforts for designing graph algorithms.

2.5 Chapter summary

This chapter presented recent advances in graph processing systems for big graphs. We discussed several programming models that are currently used in designing graph algorithms for the context of distributed graphs. However, less work has been done to take advantage of such paradigms in evaluating graph pattern matching at a high scale. Subgraph-centric programming models are recommended for this kind of problem together with asynchronous timing. Noticeably, studies were recently focused on graph simulation due to its quadratic-time complexity. Furthermore, the recent research works

are either addressing the velocity of data generation or the size of data graphs, seeking to reduce the time and space complexity of previously proposed algorithms. However, it is rare to find both issues handled in the same work.

Moreover, we identified categories of distributed GPM approaches and classified them based on multiple criteria. We gave more attention to the relaxed pattern matching since it is widely used in the current applications of social networks and massive graphs in general. An abundant literature exists discussing subgraph isomorphism queries, but the limitations of this model make it impractical for current applications of graph pattern matching. Hence, benefiting from subgraph isomorphism techniques to build an enhanced approach for strong simulation is one of the current challenges in this domain, especially in the distributed and parallel contexts. That is to say, we can combine the inherent flexibility of the model itself and exploit the strong background behind subgraph isomorphism to achieve better results.

In the second part of this thesis, we address this research direction by proposing an alternative to strong simulation with scalability guarantees. Furthermore, motivated by the shortcomings of the approaches discussed in this chapter, we design scalable distributed algorithms to evaluate strong simulation, graph simulation and dual simulation.

Part II

Parallel and Distributed Algorithms for Relaxed GPM

Chapter 3

Distributed Graph Pattern Matching via Bounded Dual Simulation

3.1 Introduction

Even though graph simulation is considered to be the first model that can be evaluated in quadratic time, its weak part is that it does not capture the complete topological structure of the pattern graph. Indeed, we often end up with answers having a structure different from the input query. This is why dual simulation was proposed, allowing to capture more topological structures via the duality property. The difference between the two models is that graph simulation requires the matching constraints to be satisfied only for the children of the data vertex, while dual simulation brings duality by requiring those constraints to be satisfied for both the children and parents. Such specification captures better the topology of the query; undirected cycles are preserved and connected graphs are matched only to connected graphs. However, it is true that dual simulation improves the quality of the results returned by graph simulation, yet, this model may result in answers of much larger size, because cycles in the query graph can be translated into cycles of unbounded length in the resulting match graph. This limitation motivated the appearance of strong simulation, another model that captures the structure in graph simulation while maintaining a low complexity, hence giving a compromise between flexibility and tractability. Strong simulation is defined as graph simulation reinforced by duality and locality. The locality property limits the accepted answers to only subgraphs

falling within a fixed diameter. Thus, cycles that are longer than the query's diameter are filtered out to keep answers of reasonable size. Although prior works addressed strong simulation for large graphs, they did not give any scalability guarantees when the size of data graphs increases reaching at least millions of vertices.

In this chapter, we propose Bounded Dual Simulation (BDSim) to answer relaxed GPM queries on massive graphs. BDSim is an extension to dual simulation that allows not only to capture duality and locality of the pattern graph, but it also preserves its cyclic structure. As opposed to strong simulation that duplicates data graph vertices in an exponential way when creating data locality balls, BDSim extends dual simulation by eliminating cycles that have unbounded length from the match graph, hence allowing to capture the locality and obtain results of reasonable size that are semantically correct.

The contributions made by this work are improving the response time because no data locality balls are created. The number and size of messages communicated are also reduced (tokens are exchanged instead of the whole vertex matching information). BDSim scales well with the size of the data graph and can handle massive graphs. Finally, our approach is adaptive because it uses a configurable parameter k for limiting the size of cycles in the resulting match graph, thus allowing more flexibility based on the input query and application domain, which ensures avoiding the problem of zero answers.

The remainder of this chapter is organized as follows. Section 3.2 introduces the new GPM model BDSim, whereas distributed vertex-centric algorithms for evaluating BDSim on massive graphs are presented in Section 3.3. Section 3.4 provides a formal validation of the proposed algorithms, and Section 3.5 is dedicated for their experimental evaluation on synthetic and real-world datasets.

3.2 Bounded Dual Simulation (BDSim)

When addressing the problem of subgraph matching, the cyclic structure of the query graph is an important property that captures the locality between query vertices. However, dual simulation preserves only the child and parent relationships, which results into a model that maps cycles in the query to cycles of unbounded length in the resulting match graph. Actually, a cycle in the query graph can be translated to cycles having a multiple length in the match graph. As a result, we end up having answers of large size that do not necessarily resemble the input query.

Preserving the cyclic structure of the query and hence its locality is an important task in GPM, because it allows to reduce further the size of returned answers, and to improve their quality. Based on this observation, we notice that eliminating such unbounded cycles refines further the returned match graph compared to dual simulation and captures better the topology of the query.

To address this challenge, we introduce BDSim, an adaptive relaxed GPM model that lies between dual simulation and strong simulation. A cycle in the final match graph is said to be bounded (or length-bounded) if its length is k times the length of the cycle to which it is matched in the query, such that k is a fixed parameter. By introducing BDSim, we aim to limit the size of cycles in the final match graph by checking if a data vertex is in a cycle of bounded length or not, and this can be achieved by tracking all the cycles of the query graph. Nevertheless, we will see that the set of short cycles is enough to capture the closeness and proximity between the query vertices.

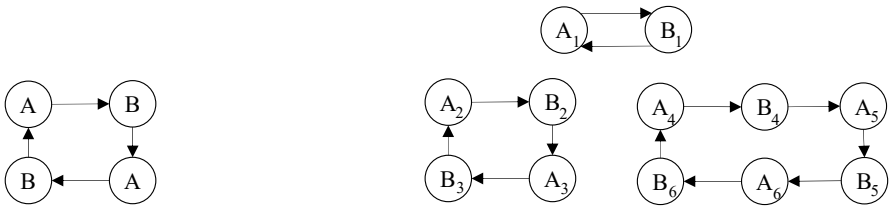
Definition 17 (Short cycle)**.** *Given an unweighted graph $G = (V, E, f)$ and two vertices v and v' such that $e = (v, v') \in E$. We call a short cycle, if it exists, a cycle composed of the edge e and an undirected shortest path between v and v' in $G' = (V, E \setminus \{e\}, f)$.*

Note that, we define an *undirected shortest path* in a directed graph as a shortest path in the underlying undirected graph, i.e. when we ignore edge directions. We define an *undirected cycle* in a directed graph as a cycle in the underlying undirected graph, i.e. we ignore edge directions. Unless expressly stated otherwise, we shortly use *cycle* to refer to an undirected cycle. Furthermore, we omit the repeated vertex of a cycle when giving its elements for simplicity purposes. In a graph, a *long cycle* is a cycle that is not short.

Dual simulation is a flexible model by definition, it allows mapping an edge in query graph Q to one or more edges in the dual match graph G_D. In addition to that, two query edges can be mapped to the same edge in G_D. Consequently, given a cycle $c \in Q$ and one of its matched cycles $c' \in G_D$, c' may have the same length as c, be longer or even shorter, i.e. $|c'| = r \times |c|$ where $r \in \mathbb{Q}_+^*$, with $|c'| \geq 2$. An example illustrating such different possibilities is given in Figure 3.1.

Lemma 1. *Given a query graph Q, a cycle in Q is either a short cycle or a cycle composed of edges from other short cycles.*

Proof. By definition, a short cycle is a cycle for which, the incident vertices of at least one edge have their shortest path in this cycle (when excluding the corresponding edge in computing the shortest path). Consequently, a long cycle c_l is a cycle where all the



(a) A query cycle (b) Different matched cycles.

Figure 3.1 A cyclic query with three matches respecting dual simulation.

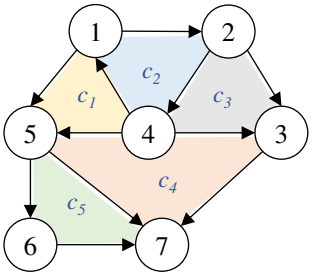


Figure 3.2 A graph with its five short cycles.

shortest paths between the incident vertices of any considered edge are not entirely in c_l, and we refer to such paths as bridges. These bridges decompose c_l into *adjacent* cycles that are all short. Actually, an edge $e \in c_l$ that has its shortest path p outside c_l results into a new cycle $c = e \cup p$. c is effectively a short cycle from the definition of short cycles. The same thing applies for all the other edges in c_l. Therefore, all the edges in c_l belong to at least one short cycle; i.e. c_l—which is a long cycle—is composed of edges from the short cycles of Q. □

Example 3. *In the graph of Figure 3.2, the cycle* $c_1 = \{1, 2, 3, 4, 5, 1\}$ *is composed of five edges whose shortest paths do not belong to* c_1*. Actually, the edge* $(1, 2)$ *has the shortest path* $\{2, 4, 1\}$ *that forms the short cycle* $\{1, 2, 4, 1\}$*. Similarly, the remaining edges* $(2, 3), (4, 3), (4, 5)$ *and* $(1, 5)$ *result in the short cycles* $\{2, 3, 4, 2\}, \{4, 3, 2, 4\}, \{4, 5, 1, 4\}$ *and* $\{1, 5, 4, 1\}$*, respectively. Indeed, enumerating all the simple cycles in the graph returns exactly 21 cycles, whereas the set of short cycles contains only 5 cycles. When using these short cycles to bound the size of a match graph, we reduce the number of tracked cycles by a fourth while still giving the same result. As a result, we minimize the number of computations significantly and reduce the overall evaluation time of BDSim.*

Lemma 2. *Given a query graph* Q *and a data graph* G*. Let* G_D *be the dual match graph of* Q *in* G*. If the matches of the short cycles of* Q *are length-bounded in* G_D*, then all the matched cycles in* G_D *will also be length-bounded.*

Proof. We know that every cycle in Q is matched to a cycle in G_D (from the definition of dual simulation). First, we assume that all the short cycles in Q have bounded matches in G_D. Then, we suppose that there exists a cycle c in Q having a match of an unbounded length in G_D. First, if c is a short cycle, then all its matches are bounded by supposition. Second, if c is long, then it is composed of edges from other short cycles (See Lemma 1). Besides, its matches are composed of the matches of these short cycles. Hence, it must be bounded because all the short cycles are already bounded, and the composition of a set of bounded cycles cannot be unbounded. □

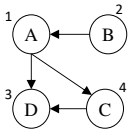
Theorem 1. *In the dual match graph G_D of query graph Q in data graph G, using only the set of short cycles of Q is sufficient to bound the length of the matches of all the cycles of Q in G_D.*

Proof. The proof of this theorem is directly derived from Lemma 1 and Lemma 2. From Lemma 1, every cycle in Q is either a short cycle or composed of other short cycles. According to Lemma 2, bounding the length of the matches of the short cycles of Q in G_D results into bounding the matches of all the cycles in Q. □

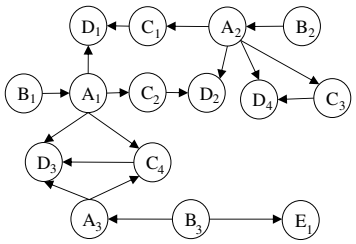
BDSim extends dual simulation with a cycle constraint, such that a data vertex matched with query vertices in a short cycle c_i of Q must be a member of a matched cycle that have the maximum length $k \times l_i$ in the match graph, where l_i is the length of c_i and $k \in \mathbb{N}^*$. Moreover, BDSim is adaptive thanks to the parameter k that allows for more flexibility. Actually, we may have a context where it is required to keep only cycles of the same length, while in other cases, the allowed length would be equal to twice the initial cycle length or even more. Hence, k is defined depending on the semantics of the data graph and application domain.

The example below explains how BDSim works and demonstrates its adaptability based on the application needs.

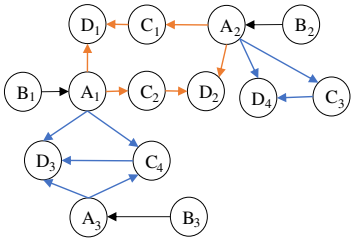
Example 4. *Figure 3.3 shows a query graph Q_3 and a data graph G_3. The query graph contains one undirected cycle $c_1 = \{1, 3, 4\}$ of length $l_1 = 3$. When evaluating dual simulation on the data graph G_3 for query Q_3, we get the dual match graph G_D shown in Figure 3.3c. The cycle c_1 is matched to cycles of different lengths in G_D (three cycles of length $l_1 = 3$ that are colored in blue and one cycle having a double length $l_2 = 6$, which is colored in orange). To restrict the returned answers to only matches having cycles of bounded length, we verify for every data vertex matched to one of the cycle members $\{1, 3, 4\}$ whether it is in a cycle of length $l \leq k \times l_1$ or not. Table 3.1 answers*



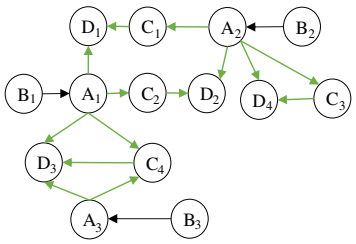
(a) Query graph Q_3



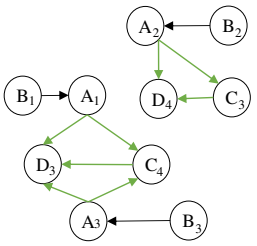
(b) Data graph G_3



(c) Dual match graph G_D



(d) BDSim for $k = 2$



(e) BDSim for $k = 1$

Figure 3.3 Example of a query graph Q_3, a data graph G_3 and the different match graphs resulting from dual simulation G_D and BDSim (for both cases $k = 1, 2$). Q_3 contains one short cycle (c_1 of length 3) that is translated into four cycles of different lengths in G_D, colored in orange or blue. Setting $k = 2$ allows capturing cycles of length up to 6 (Figure 3.3d), while $k = 1$ limits the answer of BDSim to cycles of exactly the same size as c_1, i.e. only cycles colored in blue from G_D (Figure 3.3e).

this question for such data vertices in the cases where $k = 2$ and $k = 1$. For example, the data vertices $\{C_1, C_2, D_1, D_2\}$ are in the dual match graph but they should not be considered as valid matches when seeking a more restricted answer. Hence, setting k to 1 eliminates them because they are members of only cycles with length $l = 6 > k \times l_1$. Figures 3.3d and 3.3e give the desired final match graphs according to BDSim for $k = 2$ and $k = 1$, respectively.

Table 3.1 Different possible BDSim matches in data graph G_3 for cycles in Q_3 when $k = 2$ and $k = 1$.

Data vertex in G_D	A_1	A_2	A_3	C_1	C_2	C_3	C_4	D_1	D_2	D_3	D_4
Length of 1^{st} matched cycle in G_D	6	6	3	6	6	3	3	6	6	3	3
Length of 2^{nd} matched cycle in G_D	3	3	/	/	/	/	3	/	/	3	/
Vertex satisfies BDSim for $k = 2$?	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Vertex satisfies BDSim for $k = 1$?	✓	✓	✓	X	X	✓	✓	X	X	✓	✓

Varying the value of parameter k while answering the same query allows for more flexibility, especially to deal with the problem of zero answers when k is initially set to 1. A formal definition of BDSim is given below.

Definition 18 (BDSim)**.** *Let $G = (V, E, f)$ be a data graph, $Q = (V_q, E_q, f_q)$ a query graph, ζ the set of short cycles related to every single edge in Q and $k \in \mathbb{N}^*$ a fixed parameter. G matches Q via BDSim, if there exists a maximum match relation $R \subseteq V_q \times V$ and a subgraph G_R of G, such that:*

(1) G matches Q via dual simulation with maximum match relation R,

(2) G_R is the match graph w.r.t. R,

(3) $\forall (u, v) \in R$, if $u \in c_i$ and $c_i \in \zeta$, then $\exists c'_i$, a cycle in G_R that matches c_i and verifies:

(a) $v \in c'_i$,

(b) c'_i has a maximal length $(k \times l_i)$ in G_R where l_i is the length of c_i.

Indeed, BDSim aims to keep only matches that preserve locality. If a data vertex v is a match to a query vertex u ($(u, v) \in R$) and $u \in c_i$ such that $c_i \in \zeta$, this mapping is kept only if v belongs to a cycle c'_i that matches c_i and has a bounded length in the match graph.

Propositions 1 and 2 give the relationships that BDSim has with dual simulation and subgraph isomorphism, respectively.

Proposition 1. *If a data graph matches a query graph via BDSim, then it also matches the same query graph via dual simulation.*

Proof. By definition, BDSim is a restriction of dual simulation, i.e. every data vertex that matches a query vertex via BDSim must match that vertex *w.r.t.* dual simulation. Hence, the proposition above is correct. □

Proposition 2. *If a data graph matches a query graph via subgraph isomorphism, then it also matches this query w.r.t. BDSim.*

Proof. First, a data graph that matches a query graph via subgraph isomorphism matches the same query via dual simulation (Proposition 1 in [84]). Moreover, Definition 18 states that a data vertex matches a query vertex *w.r.t.* BDSim, if it satisfies at least dual simulation constraints. Thus, this data graph already satisfies the first two constraints of BDSim.

Next, we prove that subgraph isomorphism already satisfies constraint 3 of BDSim. Let $G_I = (V_I, E_I, f)$ be a match graph of G for Q *w.r.t.* subgraph isomorphism and

$R \subseteq V_q \times V_I$ is the corresponding bijective match relation. According to Theorem 3 in [84], given a cycle $c_i = u_1, u_2, \ldots, u_l$ in ζ, c_i must be matched to a cycle $c'_j = v_1, v_2, \ldots, v_l$ in G_I where every single vertex u in c_i is mapped to exactly one vertex v in c_j. Formally, $\forall m \in \{1, \ldots, l\}, (u_m, v_m) \in R$.

Constraint 3 in Definition 18 of BDSim intends to eliminate the data vertices $v \in V_I$ where $(u, v) \in R$ and $u \in c_i$ in ζ, such that v belongs only to cycles of unbounded length. However, subgraph isomorphism does not allow these cases by definition, hence, G_I already satisfies this constraint. Therefore, a data graph that matches a query graph via subgraph isomorphism necessarily matches the same query graph via BDSim. □

Figure 3.4 illustrates the inclusion relationships between BDSim and the other GPM models. BDSim adaptability, ensured by the parameter k, allows it to be as flexible as dual simulation or as strict as subgraph isomorphism. The smaller the value of k, the closer we get to subgraph isomorphism and the larger it is, the closer we get to dual simulation. Graph simulation, on the other hand, is more flexible than dual simulation.

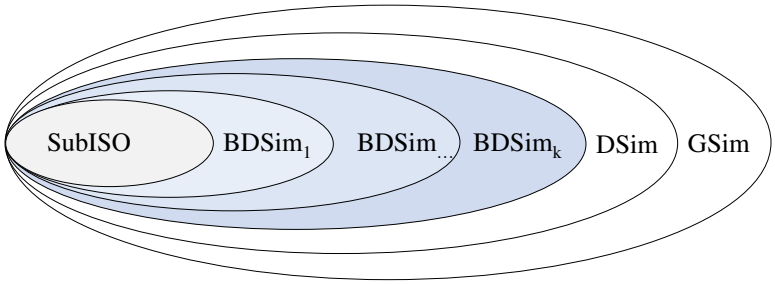


Figure 3.4 The inclusion relationships between BDSim for different values of k (BDSim$_k$) and the other models subgraph isomorphism (SubISO), dual simulation (DSim) and graph simulation (GSim).

We propose an approach composed of the following four phases to evaluate BDSim: (1) a preprocessing phase to extract the set of short cycles from the input query graph, (2) evaluation of dual simulation and extraction of the dual match graph, (3) detection of invalid matches (data vertices that are in cycles of unbounded length), and (4) elimination of these matches to get the final match graph. The next section is dedicated for presenting these phases one by one.

3.3 Distributed evaluation of BDSim

After extracting the set of short cycles from the input query graph using a sequential algorithm, we evaluate dual simulation and extract the dual match graph in a distributed way by adopting a TLAV paradigm and Pregel computation model. After that, we use a distributed token passing algorithm to detect invalid matches that do not satisfy the third constraint of Definition 18. Finally, a distributed filtering phase allows to keep only the correct matches *w.r.t.* BDSim.

In what follows, we present in detail the four phases of BDSim along with the proposed algorithms.

3.3.1 Extracting short cycles from the query graph

We use a simple and yet efficient algorithm to compute the set of all the short cycles of the query graph in quadratic time. Given a query graph $Q = (V_q, E_q)$ and the set of its short cycles ζ. To compute ζ, we verify for every edge in E_q, if it is part of a short cycle c_i and then add c_i to ζ.

For each edge (u, v) in E_q, proceed as follows.

(1) Remove (u, v) from E_q,

(2) Check if u is still reachable from v,

(3) For every shortest path p_i between v and u, let $c_i = p_i \cup (u, v)$ be a short cycle and add c_i to ζ.

To test the reachability of u from v, we execute a BFS traversal from vertex v that returns the set of shortest paths between v and u if they exist. Since Q is an unweighted graph, such traversal runs in $O(|V_q| + |E_q| - 1)$ [28]. This routine will be executed for every edge in the graph. Hence, the time complexity for finding ζ is $O(|E_q|^2)$.

Next, we give the vertex-centric algorithms for evaluating dual simulation, the token passing procedure and final filtering to get BDSim match graph. The notations used by these algorithms are given in Table 3.2.

3.3.2 Vertex-centric algorithm for dual simulation

Algorithm `VC-DSim` depicts the steps of the vertex program used in evaluating dual simulation in a distributed vertex-centric way.

Table 3.2 Notations used in the distributed algorithms of BDSim

Object	**Notation**
Local vertex ID	v
Parent and child constraints	$\mathscr{C}_p, \mathscr{C}_c$
Local, parent and child match sets	M, M_p, M_c
Set of locally removed query vertices	R
Data message, removal message	D_{msg}, R_{msg}
Set of global seeds	S_g
Set of local seeds	S_l
A short cycle in S_g or S_l	c_i
Length of a short cycle c_i	l_i
A seed	s
The short cycle corresponding to s	$s.c$
The matched query vertex of seed s	$s.m$
Position of $s.m$ in $s.c$	$s.p$
Direction of $s.m$ in $s.c$	$s.d$
Next hop after vertex $s.m$ in $s.c$	$s.h$
Received tokens flag for s	$s.f$
A token	t
The next matched vertex in t	$next$
A token expiration value	exp

The match set of a data vertex v is the set of query vertices it matches, whereas a matching constraint of a query vertex u in M includes its parents $\mathscr{C}_p(u)$ and children $\mathscr{C}_c(u)$. The local context of a data vertex v is composed of its local match set M, the match sets of its parents M_p and its children M_c. In addition to that, the matching constraints related to M are kept locally.

We introduce the notion of matching constraints that are used in the evaluation of dual simulation and BDSim. Indeed, instead of duplicating the whole query graph inside every data vertex during the execution of our distributed algorithm, we generate a set of constraints that are maintained locally by each data vertex. A data vertex keeps only the constraints related to the query vertices that match it, therefore reducing the size of replicated data.

Definition 19 (Matching constraint)**.** *Given a query graph $Q(V_q, E_q, f_q)$, a candidate data vertex (a vertex that has a similar label) is said to be a match for the query vertex $u \in V_q$ if and only if it satisfies the parent and child relationship constraints of dual simulation. Let $\mathscr{C}_p : V_q \to 2^{V_q}$ ($\mathscr{C}_c : V_q \to 2^{V_q}$) be the function that maps a query vertex $u \in V_q$ to its parent constraints $\mathscr{C}_p(u) \subseteq V_q$ (child constraints $\mathscr{C}_c(u) \subseteq V_q$), respectively. Each edge $(u', u) \in E_q$ defines a parent matching constraint u' for u. Similarly, every edge $(u, u'') \in E_q$ defines a child matching constraint u'' for u. Consequently, $\mathscr{C}_p$ (or $\mathscr{C}_p$) maps a vertex with zero parents (or children) to an empty set of constraints, respectively.*

The distributed algorithm starts by an initial broadcast message to all the vertices of G. The vertex program is then triggered by every vertex in G to start the evaluation of dual simulation. Then, two types of messages are exchanged between the data graph vertices: data messages D_{msg} and removal messages R_{msg}. When a data vertex v receives the initial message containing the query graph Q, it stores locally in M a copy of the query vertices that have the same label as v, along with the set of constraints for each query vertex (set of parents $\mathscr{C}_p$ and children $\mathscr{C}_c$ of each $u \in M$) (Lines 1–6). After that, if M is not empty, v sends it with its identifier to its neighbors (Lines 7–9). Otherwise, M is empty, v goes inactive (Lines 10–11).

In the next super-step, at the reception of these local match sets by neighbors, every data vertex stores the parents and children match sets in separate variables M_p and M_c, respectively (Lines 12–13). Then, it invokes *Procedure EvalDSim* (Line 14) to verify the matching constrains in $\mathscr{C}_p$ and $\mathscr{C}_c$ using both match sets M_p and M_c, respectively. Invalid matches are removed from M and added to R (a temporary variable that stores the removed matches in the form of pairs (v, u)). It also removes these query vertices from $\mathscr{C}_p$ and $\mathscr{C}_c$ as they will not serve in evaluating the matching constraints anymore. The same routine is repeated until no new changes are made in M. If R is not empty, v sends a removal message with R to all its neighbors, then it goes inactive (Lines 15–18).

In Super-step 3 and upon the reception of a removal message R_{msg}, v removes the mappings in R_{msg} from M_p and M_c (Lines 19–20). Afterwards, it reevaluates the matching constraints again, removes invalid matches from M, adds them to R and updates $\mathscr{C}_p$ and $\mathscr{C}_c$ (call of *Procedure EvalDSim* in Line 21). The set of removed matches is then shared with direct neighbors if it is not empty (Lines 22–25). The same process will be repeated in the next super-steps until the algorithm converges.

At the end of this algorithm, every data vertex will have its local match set *w.r.t.* dual simulation. If every single query vertex has at least one match, then, the union of these local matches is exactly the global match set *w.r.t.* dual simulation. Otherwise, we return an empty match set. Now, to extract the dual match graph, we filter out vertices having an empty match set and also the edges that does not represent a match.

Example 5. *To illustrate the execution steps of Algorithm* `VC-DSim`*, we use the query graph Q_3 of Figure 3.3a and the data graph G_3 of Figure 3.3b. At the end of the first super-step, only data vertex E_1 will have an empty match set, whereas all the remaining vertices share their non empty matches to their direct neighbors. In the second iteration, all the data vertices with a non-empty match set reevaluate dual simulation constraints. However, since all the vertices except E_1 satisfy already dual simulation, no removal*

Algorithm VC-DSim

Super-step 1: *At the reception of* Q*, do*;
foreach *vertex* **u** *in* Q **do**
    if $f(v) = f_q(u)$ **then**
        Add **u** to M;
        Add parents of **u** to $\mathscr{C}_p$;
        Add children of **u** to $\mathscr{C}_c$;
if $M \neq \emptyset$ **then**
    $D_{msg} \leftarrow (v, M)$;
    Send D_{msg} to all neighbors;
else
    Vote to halt;
Super-step 2: *At the reception of* D_{msg} *while* $M \neq \emptyset$*, do* ;
Update M_p and M_c;
$R \leftarrow EvalDSim(M, \mathscr{C}_p, \mathscr{C}_c, M_p, M_c)$;
if $R \neq \emptyset$ **then**
    $R_{msg} \leftarrow (v, R)$;
    Send R_{msg} to all neighbors;
Vote to halt;
Super-step 3 and beyond: *At the reception of an* R_{msg} *while* $M \neq \emptyset$*, do*;
Update M_p and M_c;
$R \leftarrow EvalDSim(M, \mathscr{C}_p, \mathscr{C}_c, M_p, M_c)$;
if $R \neq \emptyset$ **then**
    $R_{msg} \leftarrow (v, R)$;
    Send R_{msg} to all neighbors;
Vote to halt;

Procedure EvalDSim

Input : $M, \mathscr{C}_p, \mathscr{C}_c, M_p, M_c$
Output : R
repeat
    foreach *vertex* u *in* M **do**
        Verify the matching constraints;
        if u *is not a match* **then**
            Remove u from M;
            Update $\mathscr{C}_p$ and $\mathscr{C}_c$;
            Add u to R;
until *there are no changes in* M;

messages will be generated, announcing the end of the distributed algorithm. The dual match graph is therefore the same as G_D *given in Figure 3.3c.*

3.3.3 Vertex-centric algorithm for detecting invalid matches

In this section, we propose the distributed vertex-centric algorithm `VC-TPass` that ensures the detection (and then removal) of cycles of unbounded length from the dual match graph G_D, i.e. not satisfying the closeness property represented by the third constraint in the definition of BDSim. Algorithm `VC-TPass` handles the following cases: (i) multiple query vertices with the same label, (ii) multiple short cycles in the query graph, (iii) the same vertex being part of more than one short cycle and (iv) data vertices that are matched to different query vertices within the same short cycle.

Every data vertex in G_D that is matched to a query vertex in $c_i \in \zeta$ verifies its membership to a similar, yet bounded cycle in G_D. It initiates a token with its identifier and forwards it to its neighbors based on some predefined conditions. Then, every vertex that receives such token will decrement its expiration value and forward it to potential next hops in the cycle. After $k \times l_i$ steps of forwarding the tokens, if an initiator does not receive back its own token, this means that it does not belong to a bounded cycle that is matched to c_i in G_D. Hence, it should be filtered out in the next phase.

The main challenge during this phase is to generate a small number of messages while exploring the whole graph to detect unbounded cycles. Since the cycles are undirected, knowing to which neighbor a token should be forwarded can be tricky. Hence, we guide the token passing procedure with a set of seeds that are assigned to the query vertices of every short cycle. Such data structure ensures that tokens are only forwarded to eligible neighbors, i.e. vertices that are most likely to participate in this cycle detection phase. It also allows the data vertices to explore a given cycle in the same way. We define the set of global seeds S_g computed directly from ζ and a set of local seeds S_l, a subset of S_g that corresponds to only a given data vertex.

Definition 20 (Set of Seeds)**.** *The set of seeds S_g is a data structure maintained by the graph data vertices to guide the token passing procedure. An element of S_g is called a seed, and it represents a matching query vertex m in the short cycle $c_i \in \zeta$. Moreover, a seed is composed of multiple fields that are defined as follows. Given the sequence of ordered vertices $O_i = \{m_1, m_2, \ldots, m_n\}$ in c_i, we keep for each vertex m in c_i.*

(1) The cycle identifier c_i,

(2) The query vertex identifier m ,

(3) The position of m in O_i, given as p,

(4) The direction of m in its edge, given as d,

(5) The next hop after m in O_i, given as h and,

(6) A flag f to indicate if the vertex has received tokens for this seed or not.

A data vertex matching one or more query vertices in c_i keeps the set of their seeds locally in S_l. Then, guided by S_l, it will send or forward tokens only to neighbors that can be part of the currently explored cycle c_i, i.e. only data vertices matched to the next hop vertex in c_i.

Example 6. *In the query graph of Figure 3.3a, there is one short cycle c_1 composed of query edges $(1,3)$, $(4,3)$ and $(1,4)$. The set of global seeds is therefore given as $S_g = \{(c_1, 1, 0, OUT, 3, false), (c_1, 3, 1, IN, 4, false), (c_1, 4, 2, IN, 1, false)\}$ where OUT and IN indicate whether the vertex is considered as a source or destination in this cycle, respectively.*

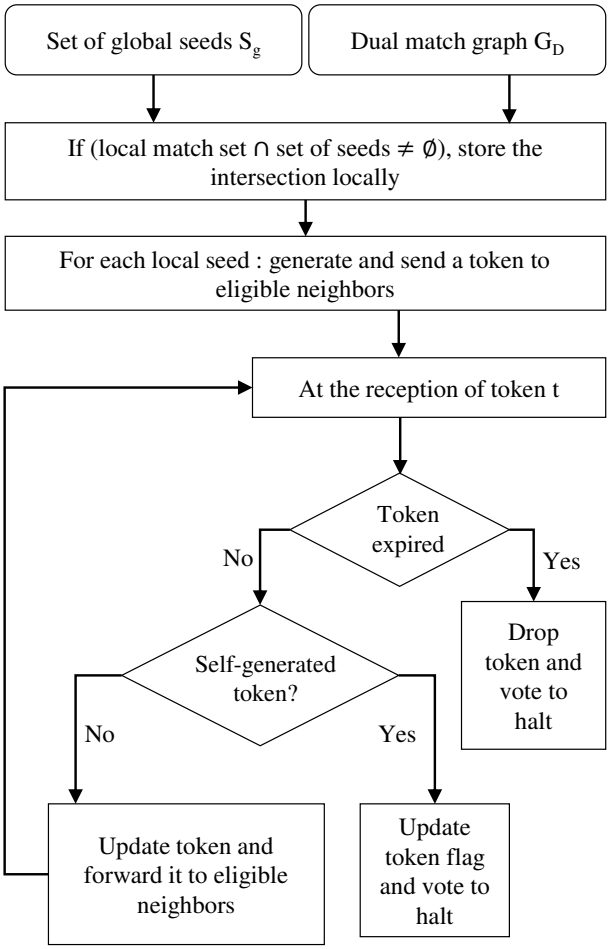


Figure 3.5 Diagram of the distributed token passing procedure for detecting invalid matches.

The diagram given in Figure 3.5 illustrates the different steps of the token passing algorithm for BDSim. After computing the short cycles ζ and set of global seeds S_g, they are broadcast to all the data vertices in G_D. When a data vertex v receives the set of seeds, it checks for each seed s whether its matched query vertex $s.m$ is a member of the local match set. If it is the case, v will store s locally (Lines 1–3). Then, it generates a token with an expiration value equal to $k \times l_i$, such that l_i is the length of the short cycle c_i of this seed. Finally, it sends the token only to eligible neighbors that might be part of a bounded cycle (Lines 4–11). An eligible neighbor is a data vertex matched to the next hop vertex following the direction of the seed (a parent if the direction is IN and a child if the direction is OUT).

During the next super-steps and upon the reception of a token t, the data vertex v decrements the expiration value of t (Lines 13–14). When v receives back its own token, it sets the local token flag to true for that seed and stop forwarding it (Lines 15–17). However, if v is not the initiator and the token has expired already, it will drop it as it is not valid anymore (Lines 18–19). If the token is still valid and v is not the initiator, it verifies among its neighbors for eligible next hops and forward the token to them, then goes inactive (Lines 20–29). The same super-step is executed in the next iterations. The maximum number of super-steps is set to $k \times l_m$, where l_m is the size of the longest short cycle in ζ and $l_m \leq |V_q|$.

Example 7. *The detection of invalid matches for G_D of Figure 3.3c proceeds as follows. The data vertices that will generate tokens are $\{A_1, \ldots, A_3\}$, $\{D_1, \ldots, D_4\}$ and $\{C_1, \ldots, C_4\}$. These sets of vertices are assigned to s_1, s_2, s_3 from S_g, respectively. We recall that $S_g = \{(c_1, 1, 0, OUT, 3, false), (c_1, 3, 1, IN, 4, false), (c_1, 4, 2, IN, 1, false)\}$. Let $k = 1$, hence, the longest matched cycle allowed must be of length $l = 3$. For example, during the first super-step, the data vertex D_1 will generate a token $t_1 = (Token(D_1, c_1, 4, 3))$, then, it will send it to the only parent vertex mapped to $t_1.next = 3$ which is data vertex C_1. At the reception of t_1 in the second super-step, C_1 will decrement t_1 and look for a local seed that corresponds to the short cycle c_1 and query vertex 4, it finds s_3. C_3 will then look for the next eligible neighbor, i.e. a parent mapped to query vertex 1. This neighbor happens to be A_2, hence, C_1 will forward the updated token $t_1 = Token(D_1, c_1, 1, 2)$ to it. In the third super-step, A_2 receives t_1 and decrements its expiring value. It updates the token in the same way which results into $t_1 = Token(D_1, c_1, 3, 1)$, therefore, it will forward it to data vertices D_2 and D_4. In the last super-step, both vertices D_2 and D_4 drop t_1 since they are not its initiators and the token has expired. At the end of the distributed algorithm, data vertices D_1, D_2, C_1, C_2*

Algorithm VC-TPass

```
1  Super-step 1:  At the reception of S_g while M ≠ ∅ , do;
2  foreach s ∈ S_g where M contains s.m do
3      Add s to S_l;
4      if d = IN then
5          foreach (u', v') ∈ M_p where u' = s.h do
6              t ← Token(initiator ← v, cycle ← s.c_i, next ← s.h, exp ← l_i × k) ;
7              Send t to v';
8      else
9          foreach (u', v') ∈ M_c where u' = s.h do
10             t ← Token(initiator ← v, cycle ← s.c_i, next ← s.h, exp ← l_i × k) ;
11             Send t to v';
12 Vote to halt ;
13 Super-step 2 until l_m + 1:  At the reception of t while S_l ≠ ∅, do;
14 t.exp ← t.exp − 1;
15 Find s ∈ S_l where s.m = t.next and s.c = t.c;
16 if v = t.initiator then
17     s.f ← true;
18 else if t.exp = 0 then
19     Drop t ;
20 else
21     if s.d = IN then
22         foreach (u', v') ∈ M_p where u' = s.h do
23             t.next ← u' ;
24             Send t to v' ;
25     else
26         foreach (u', v') ∈ M_c where u' = s.h do
27             t.next ← u' ;
28             Send t to v' ;
29 Vote to halt;
```

will all have their token flags unchanged (set to false) while the other initiators would get back their initiated tokens and set the corresponding flag to true.

3.3.4 Vertex-centric algorithm for filtering invalid matches

The local vertex program for every data vertex is given in Algorithm `VC-FFilter`. After the convergence of Algorithm `VC-TPass`, a new broadcast message is sent to all the graph vertices. Every data vertex that triggered a token passing, i.e. has a non-empty set of local seeds, will verify the reception of its initiated tokens and update its local match set accordingly (Lines 1–6). An initiator is considered no longer a match to a certain query vertex if it does not receive any token for the cycle $c_i \in \zeta$ containing that query vertex.

Any removal from the local match set triggers the creation of a removal message that will be sent to both parents and children (Lines 7–9).

Afterwards, at the reception of a removal message R_{msg}, every data vertex having a non empty match set updates first its neighbors match sets (Lines 11–12), then, verifies the matching constraints and updates its match set and local constraints accordingly (Line 13). Next, it creates a removal message if there are changes in M and sends it to all its neighbors (Lines 14–17), that will execute the same routine until no more removal messages are received announcing the end of the algorithm.

At the end of Algorithm `VC-FFilter`, every data vertex in the dual match graph will have its local matches for BDSim. One last step verifies that every query vertex has at least one match. If it is the case, then, the union of the local match sets is the maximum match set *w.r.t.* BDSim. Otherwise, the returned answer must be empty.

Algorithm VC-FFilter

```
1  Super-step 1: At the reception of an initial message while S_l ≠ ∅, do ;
2  foreach s ∈ S_l do
3      if s.f = false then
4          Remove s.m from M;
5          Update C_p and C_c;
6          Add s.m to R;
7      if R ≠ ∅ then
8          R_msg ← (v, R);
9          Send R_msg to all neighbors ;
10 Vote to halt;
11 Super-step 2 and beyond: At the reception of R_msg while M ≠ ∅, do ;
12 Update M_p and M_c;
13 R ← EvalDSim(M, C_p, C_c, M_p, M_c);
14 if R ≠ ∅ then
15     R_msg ← (v, R);
16     Send R_msg to all neighbors;
17 Vote to halt;
```

Example 8. *After the detection of invalid matches from the dual match graph G_D of Figure 3.3c, the filtering phase proceeds as follows. In the first super-step, the vertices D_1, D_2, C_1, C_2 will update their local match sets resulting all to an empty match set. Consequently, they will send a removal message to their direct neighbors. Nevertheless, the algorithm stops at the second super-step since no removal messages will be generated during this iteration. Indeed, the removal of these data vertices gives us the match graph of Figure 3.3e.*

3.4 Theoretical guarantees

In this section, we prove the convergence and correctness of the proposed algorithms.

3.4.1 Convergence of the distributed algorithms

Given the data graph $G = (V, E, f)$, the query graph $Q = (V_q, E_q, f_q)$ and corresponding dual match graph $G_D = (V_D, E_D, f)$. Let Δ be the maximum vertex degree in the data graph, $\Gamma \leq |V_q|$ the maximum size of a local match set, σ the size of ζ the set of short cycles and $\rho \leq |V_q|$ the length of the longest cycle in ζ. In a distributed environment, let P be the number of processors running in parallel. We are using local data structures that take effective constant time to add, remove, update or find an element. In the following, we prove the convergence of the proposed algorithms. Theorems 2, 3 and 4 give an upper bound time complexity for each distributed algorithm.

Theorem 2. *Algorithm* `VC-DSim` *has a time complexity of* $O\left((|V| \times |E| \times \Delta \times |V_q|^2)/P\right)$

Proof. The first super-step for verifying the label similarity takes $O(|V_q|)$, because each data vertex iterates over the vertices of the query.

In the second super-step, at the reception of the match sets of its neighbors (Δ messages in the worst case where every neighbor sends its local matches) M_c and M_p are updated in constant time. *Procedure EvalDSim* executes the repeat loop for Γ iterations in the worst case when a single member of the local match set is removed in every iteration. The inner loop verifies the matching constraints for each member of the match set *w.r.t.* dual simulation $O(\Gamma)$. This results into a complexity of $O(\Delta \times |V_q|^2)$.

When a data vertex is removed, the removal message can propagate to cross the edges of the data graph in the worst case resulting to $|E|$ additional iterations. When receiving removal messages (Δ messages in the worst case), the data vertex updates the match sets of its neighbors in constant time, and reevaluate the matching constraints in $O(\Gamma^2)$, therefore, the complexity of this super-step is also $O(\Delta \times |V_q|^2)$. Hence, the algorithm complexity in a fully distributed environment is equal to $O(|E| \times \Delta \times |V_q|^2)$. The time complexity in an environment with P processors is $O\left((|V| \times |E| \times \Delta \times |V_q|^2)/P\right)$. □

Theorem 3. *The time complexity for Algorithm* `VC-TPass` *is* $O\left((|V_D| \times \Delta^2 \times |V_q|^3 \times \sigma)/P\right)$.

Proof. The first super-step is composed of a simple operation executed for every seed in S_g, hence, the number of iterations is at most $(\sigma \times \rho)$. The body of the loop iterates

over local match set M to check the membership of a seed, adds it to S_l in constant time. Then, it iterates over the match sets of the neighbors having a size of $\Delta \times \Gamma$ to find eligible next hops. Hence, time complexity of this super-step is $O(\sigma \times \rho \times \Delta \times \Gamma)$.

In the next super-step and beyond, a data vertex can receive up to $\Delta \times \sigma \times \rho$ tokens in the worst case, where Δ is the maximum number of neighbors from which a token is issued and $\sigma \times \rho$ is the maximum size for S_l. One token is processed in $O(\Delta \times \Gamma)$, because we first get the related matched vertex from S_l in constant time, then iterate over the neighbors' match sets to check if one of them is matched to the next hop for the processed token. Therefore, the time complexity of a single iteration is equal to $O(\Delta^2 \times \sigma \times \rho \times \Gamma)$.

The number of the remaining super-steps is equal to $k \times \rho$ in the worst case (k is the parameter of BDSim). Therefore, in a fully distributed environment, the time complexity of this local program is equal to $O(k \times \Delta^2 \times \rho^2 \times \sigma \times \Gamma)$. The algorithm complexity in an environment composed of P processors is equal to $O\left((|V_D| \times \Delta^2 \times |V_q|^3 \times \sigma)/P\right)$. □

Theorem 4. *Algorithm* ***VC-FFilter*** *has a time complexity of* $O\left((|V_D| \times |E| \times \Delta \times |V_q|^2)/P\right)$.

Proof. The first super-step of Algorithm `VC-FFilter` iterates over local seeds S_l and removes a matched vertex from its local match set if it did not receive back any tokens for this matched vertex. The removal operation takes constant time. The maximum size of S_l can be equal to $\sigma \times \rho$, which is the size of S_g. Thus, the time complexity of this super-step is equal to $O(\sigma \times \rho)$. The remaining super-steps are similar to super-step three and beyond in Algorithm `VC-DSim`, hence, the time complexity of this algorithm in a fully distributed system is equal to $O\left(|E| \times \Delta \times |V_q|^2\right)$. The time complexity in an environment having P processors is $O\left((|V_D| \times |E| \times \Delta \times |V_q|^2)/P\right)$. □

3.4.2 Correctness of the distributed algorithms

We give Theorems 5, 6 and 7 that prove the correctness of Algorithms `VC-DSim`, `VC-TPass` and `VC-FFilter`, respectively.

Theorem 5. *Algorithm* ***VC-DSim*** *computes the correct and complete match set w.r.t. dual simulation.*

Proof. The correctness of Algorithm `VC-DSim` is guaranteed by the following properties (1) the algorithm will terminate (Theorem 2) , (2) the algorithm returns a correct match

set *w.r.t.* dual simulation, and (3) the algorithm returns the complete match set *w.r.t.* dual simulation.

The second property is proved as follows. We suppose that at the end of the algorithm, at least one match $(u, v) \in V_q \times V$ that is kept even though (u, v) does not satisfy dual simulation. First, if (u, v) does not respect dual simulation, this means that (i) u and v do not share the same label value, or (ii) the parent or child constraints of u are not satisfied by v. Case (i) cannot happen because we only keep in M the query vertices having the same label as v in the beginning of Super-step 1. Case (ii) also cannot happen because every time M_c or M_p is updated, v receives a removal message and reevaluates the matching constraints according to the new values of M_c and M_p. Therefore, all the matches stored locally are correct matches.

The third property can be verified as follows. We suppose that at the end of the algorithm there exists at least one match $(u, v) \in V_q \times V$ that respects dual simulation while the match set M of v does not contain u. Actually, if (u, v) is a correct match, this means the two vertices share the same label, hence, u has necessarily been added to M during the first super-step. During the remaining super-steps of the algorithm, the only removals made from M are for query vertices that do not satisfy dual simulation constraints anymore (Lines 3–5 in *Procedure EvalDSim*). Therefore, the computed match sets are complete.

Furthermore, every single vertex of the query graph must have at least one match which is guaranteed by the last step of phase one that checks such constraint and extracts the dual match graph, hence, the union of the computed local match sets gives the correct and complete match set *w.r.t.* dual simulation. □

Theorem 6. *At the end of Algorithm* `VC-TPass`*, all data vertices that triggered the token passing procedure will receive back at least one token for every bounded cycle to which they belong.*

Proof. The correctness of Algorithm `VC-TPass` is ensured by the following properties (1) the algorithm will terminate (Theorem 3), (2) a data vertex that is part of a bounded cycle in the dual match graph G_D will receive back its token, and (3) a data vertex that does not belong to a bounded cycle for a given seed will not receive back any tokens.

The second property is guaranteed by the following. We suppose that at the end of the algorithm, there exists at least one vertex v_0 that is member of a bounded cycle which did not receive back its token. If v_0 is part of a matched bounded cycle c' in G_D,

then, it already has the seeds related to c, the related query graph, stored locally in S_l, hence, v_0 has indeed generated at least one token t for this seed. Therefore, if v_0 did not receive back its initiated token t then, either t did not traverse c', or t has been dropped before reaching v_0. First, if t did not traverse c', then there exists v_i in c' that did not forward t to the next hop v_j in c'. However, if v_i and v_j belong to c', then the matches of v_j are already stored in M_p and M_c of v_i. Therefore, v_j cannot be ignored by v_i when forwarding t because v_j is matched to the next hop in c. Consequently, t necessarily traverses the bounded cycle. Furthermore, we know that the initial expiration value of t is set to $l \times k$ where l is the length of c and a token is only dropped if it has expired, meaning t has traversed c' but it did not reach v_0. Hence, c' has a length l' greater than the token's expiration value ($l' > l \times k$), but this contradicts the supposition that c' is a bounded cycle. Therefore, every data vertex that has received back its initiated token is member of a bounded cycle in G_D.

To prove the third property, we suppose that there is at least one data vertex v_0 that receives back its generated token even though v_0 does not belong to a bounded cycle. First, the token expiration value is initialized to $k \times l$, the maximum allowed cycle length, where l is the length of the corresponding short cycle c in the query graph. Additionally, the token expiration value is decremented in every super-step by the data vertex receiving it. Moreover, an expired token that is not received by its initiator is immediately dropped (Lines 18–19), hence, if a token is forwarded back to its initiator, then it must have taken at most $k \times l$ hops (the length of the traversed path) to reach back v_0. Therefore, v_0 belongs to a bounded cycle that has a length $\leq k \times l$. □

Theorem 7. *Algorithm* `VC-FFilter` *computes the correct and complete match set w.r.t. BDSim.*

Proof. Algorithm `VC-FFilter` will terminate (Theorem 4. Moreover, we already proved that every data vertex in the dual match graph satisfies dual simulation constraints. Furthermore, the data vertices not satisfying constraint 3 in Definition 18 are the ones matched to a seed in S_g but that did not receive back any token for those seeds. They must be eliminated, which is what we do in the first super-step of the algorithm. Each initiator having the token flag set to false for a given local seed in S_l will remove its related query vertex from its local match set. At the end of this super-step, all the data vertices not satisfying the cycle constraint have an updated matching information. Hence, their neighbors need an updated version of their match sets, which we ensure by the removal messages sent to all direct neighbors. When a removal message is received by

a data vertex with a non-empty match set, it will update its overview of the matching information for its parents and children accordingly. Then, it reevaluates dual simulation and updates its local match set. If this reevaluation results into any removal, a removal message will be sent to its neighbors that need an updated version of the local match set of their parents and children. The third super-step and the following super-steps guarantee that every data vertex not satisfying dual simulation anymore is filtered out (its local match set is reduced as we proved it in Theorem 5).

Now, we need to ensure that all the vertices satisfying constraint 3 of Definition 18 at the end of the first super-step are still satisfying the same constraint after reevaluating dual simulation. Intuitively, vertices that do not satisfy BDSim are only those not satisfying dual simulation, but these vertices have already been filtered out. Super-step two and beyond cannot result into a data vertex that satisfies dual simulation and not BDSim because if this happens, it simply means that this vertex belongs to only unbounded cycles mapped to $c \in \zeta$. Nevertheless, the only changes made in this stage to the dual match graph are removals, hence, such data vertices cannot exist. This proves that every data vertex contains the correct and complete matches *w.r.t.* BDSim.

Finally, the verification made at the end of this phase ensures that each query vertex has at least one match which, in turn, guarantees that the union of these local matches is the correct and complete match set *w.r.t.* BDSim. □

3.5 Experimental evaluation

We conducted different experiments on synthetic and real datasets with the goal of measuring the performance of the proposed approach and compare it to the three models: graph simulation, dual simulation and strict simulation. Another aim is to identify the bottlenecks of the proposed algorithms.

3.5.1 Experimental data sets

In addition to synthetic graphs generated using the R-MAT model [18], we worked on two real datasets from SNAP Library [74], their characteristics are given in Table 3.3.

Amazon0601: This graph represents the recommendation system of the online store Amazon. The graph vertices are Amazon products, while the links between these products mean that the two are usually ordered together, i.e. *"Users Who Bought This Also Bought This"* relationships.

Table 3.3 Characteristics of the data graphs used in the different experiments

Dataset (G)	$\|V\|$	$\|E\|$	$\|\Sigma\|$
Amazon0601	403,394	3,387,388	10
LiveJournal	4,847,571	68,993,773	100

LiveJournal: It is another social network that allows users to share and write blogs. Vertices represent users while edges represent the friendships among them.

Synthetic graphs: The R-Mat model is used to generate the synthetic data graphs. It takes as input $|V|, |E|$ and $|\Sigma|$ such that $|\Sigma|$ is the number of distinct labels and $|E| = 20 \times |V|$.

3.5.2 Experimental setup

The experiments presented in this thesis were carried on the High Performance Computing Platform IBNBADIS. This platform is composed of 32 compute nodes, each composed of two processors Intel(R) Xeon(R) CPU E5-2650 2.00GHz. Each processor has 8 cores, which gives a total of 512 cores to the cluster. The theoretical power of IBNBADIS is around 8 TFLOPS. Moreover, each node has 32GB memory with 100GB of disk capacity.

We deployed a 20-node cluster of Apache Spark [151] on top of IBNBADIS, where one node plays the role of the master while the remaining nodes are considered workers.

All the algorithms proposed in this thesis were implemented using GraphX [143] and SCALA programming language. GraphX is a module provided by Apache Spark that allows vertex-centric graph processing through the exposed API of Pregel [86].

3.5.3 Pattern generation

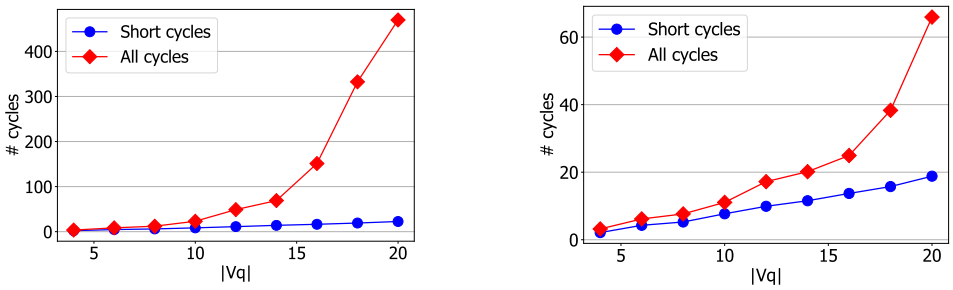
We have tested the proposed algorithms of BDSim on real-world graphs by varying the size of the pattern graph such that for each value $|V_q|$, we extract 50 different patterns randomly.

For this purpose, we implemented an algorithm that extracts patterns of small size randomly from a data graph. The algorithm takes two input parameters that are $|V_q|$, the size of the subgraph and its density α such that $|E_q| = |V_q|^{\alpha}$. The default value of α is set to $\alpha = 1.2$ in all our experiments. This algorithm executes a BFS starting from a random vertex in the graph, collects vertices randomly based on a fixed value of average

out degree d_O computed such that $\alpha = 1.2$, then we extract the subgraph connecting these vertices and add up edges to match the input parameters.

3.5.4 Experimental results

In Figure 3.6, we compare the number of short cycles in a given query graph to the number of all simple cycles (Amazon0601 in Figure 3.6a and LiveJournal in Figure 3.6b). Bounding the length of the cycles in the resulting match graph becomes intractable if we decide to track all the simple cycles of the query graph. The reason is because we have to deal with an exponential number of cycles. Nevertheless, the number of short cycles increases linearly *w.r.t.* the size of query graphs. Therefore, using the set of short cycles allows us to reduce significantly the number of processed cycles, resulting to a controlled number of tokens and messages, which allows us to bound the length of all the existing cycles in the dual match graph in a reasonable time.



(a) Amazon0601 dataset (b) LiveJournal dataset

Figure 3.6 The number of cycles *w.r.t.* the size of query graph.

Exp-1: In this set of experiments, we evaluate our approach by varying the size of the query graph and compare the obtained results with graph simulation (GSim), dual simulation (DSim) and strict simulation (STSim). The results are shown in Figures 3.7 and 3.8.

We can see in Figures 3.7a and 3.7e that for *Amazon0601* and *LiveJournal*, BDSim gives a response time better than strict simulation and it yields to better quality answers since the number of returned matches is reduced in comparison to the other models (Figures 3.7b and 3.7f).

The number of super-steps for BDSim is almost always the highest (Figures 3.7c and 3.7g). However, it does not affect negatively the final response time, because as we can see in the the four charts of Figure 3.8, the number of active vertices during every super-step, for different values of $|V_q|$ and for the two datasets, is always lower than that

of strict simulation, without forgetting that for strict simulation the last iteration will involve all the vertices present in the dual match graph.

The average response time increases with the query size because more vertices in the query graph will most likely lead to larger intermediate results. Hence, having more active vertices which will increase the size of match sets processed in every super-step resulting into higher response time.

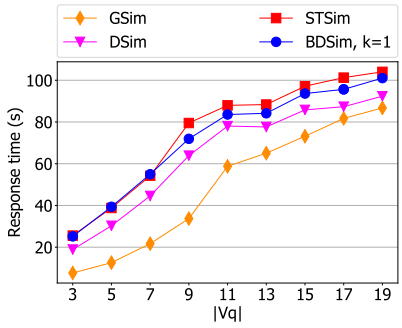
The number of exchanged messages shown in Figure 3.7d and Figure 3.7h for Amazon0601 and LiveJournal, respectively, is sensitive to the size of the query graph for the same reason explained above. Actually, large intermediate results lead to more exchanges between the graph vertices. We notice that the number of messages for BDSim is very close to that of dual simulation. This is possible thanks to the token passing approach that limits the search space into only eligible neighbors of a data vertex.

Exp-2: In this set of experiments, we evaluate our algorithms by varying the size of the data graph. We use the R-Mat graph generator implementation provided by GraphX, we give as an input the graph scale such as $|V| = 2^{scale}$, $|E| = 20 \times |V|$ and $\Sigma = 100$. We vary the graph scale from 13 to 21, and for each generated graph, we extract 50 different patterns of the same size $|V_q| = 9$ with $\alpha = 1.2$. After that, we report the average response time, number of super-steps, number of exchanged messages and number of returned matches in Figure 3.9.

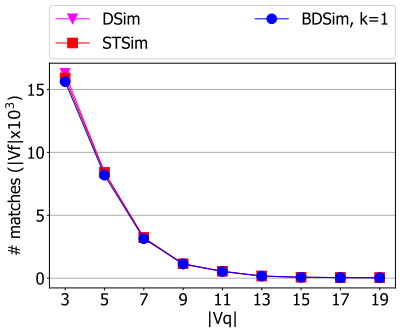
BDSim scales well with the size of the data graph. Actually, our algorithm runs in a cubic time in function of the data graph size as it is shown in Figure 3.9a. Moreover, Figure 3.9c gives the number of super-steps following logarithmic curve with an average difference of 6 additional super-steps compared to dual simulation when $|V_q| = 9$. We also notice that our approach performs better in terms of quality of results when the data graph is larger where the number of returned matches is reduced by 2/3 times compared to dual simulation. Furthermore, the number of exchanged messages follows the same pace as dual simulation (Figure 3.9d).

Exp-3: In this set of experiments, we evaluate our approach by varying the number of distinct labels using the same R-Mat generated data graph ($|V| = 0.45M$, $|V_q| = 9$). We vary $|\Sigma|$ from 100 to 500. After that, we report the average response time, number of super-steps, number of exchanged messages and number of returned matches. The obtained results are given in Figure 3.10.

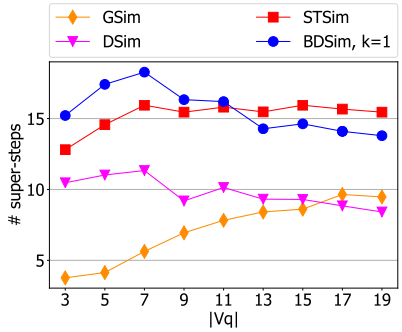
The response time is also sensitive to $|\Sigma|$ (Figure 3.10a). It decreases when increasing $|\Sigma|$ for both BDSim and dual simulation. Actually, having more distinct labels in the data



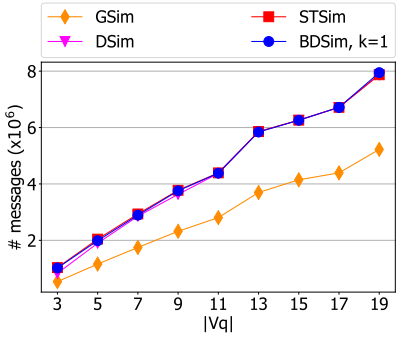
(a) Response time (*Amazon0601*)



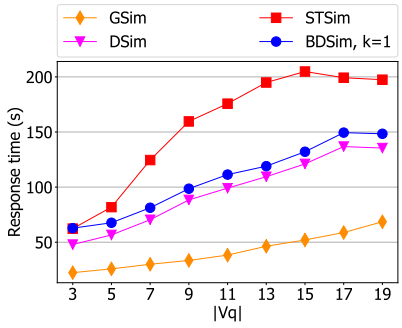
(b) Matches (*Amazon0601*)



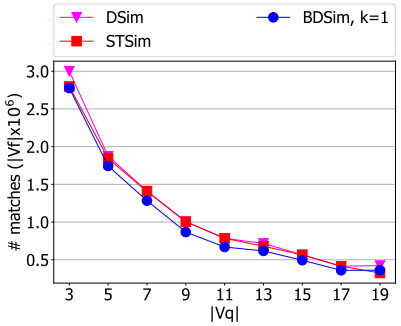
(c) Super-steps (*Amazon0601*)



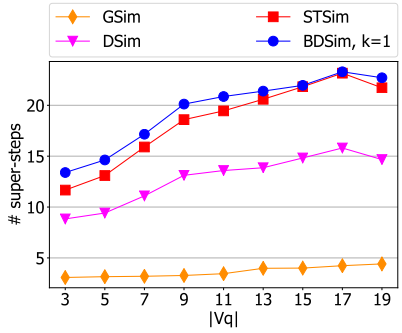
(d) Messages (*Amazon0601*)



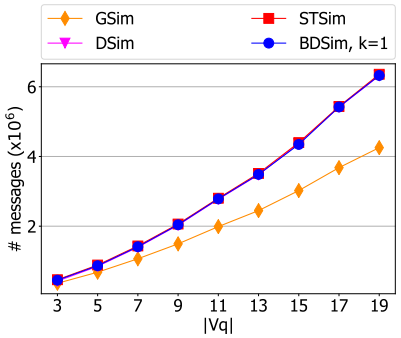
(e) Response time (*LiveJournal*)



(f) Matches (*LiveJournal*)

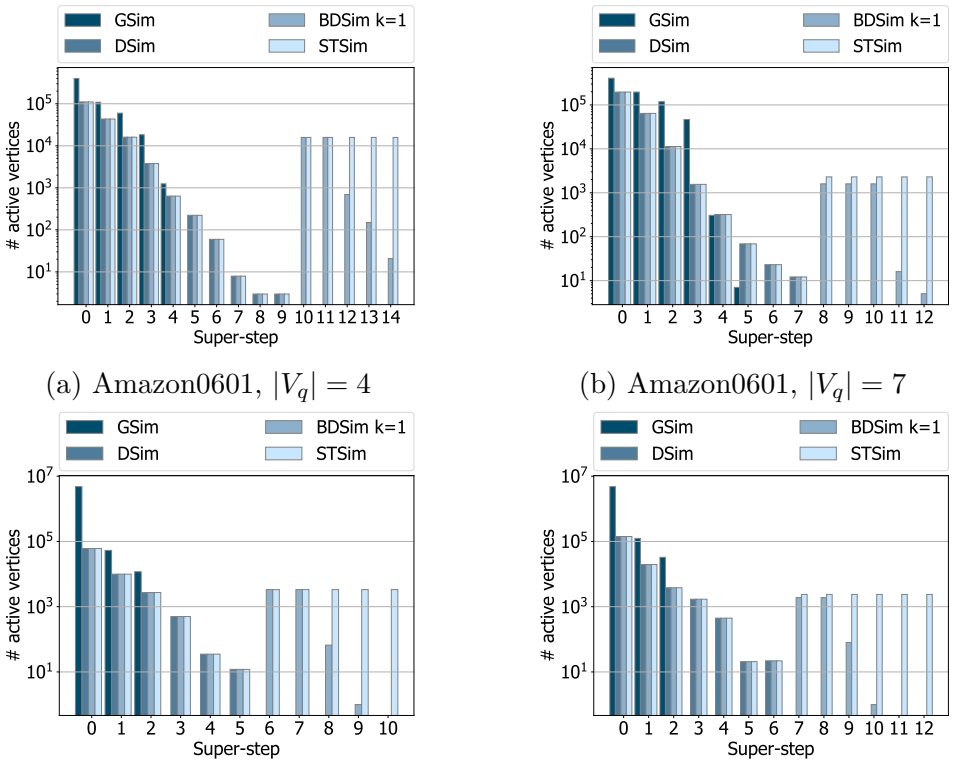


(g) Super-steps (*LiveJournal*)



(h) Messages (*LiveJournal*)

Figure 3.7 Varying the pattern graph size $|V_q|$ for real datasets *Amazon0601* and *LiveJournal*.



(a) Amazon0601, $|V_q| = 4$

(b) Amazon0601, $|V_q| = 7$

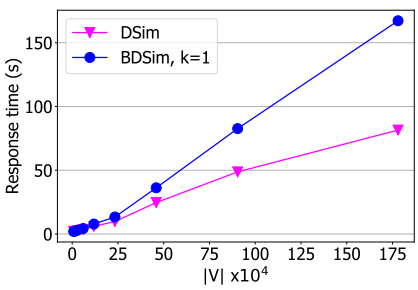
(c) LiveJournal, $|V_q| = 4$

(d) LiveJournal, $|V_q| = 7$

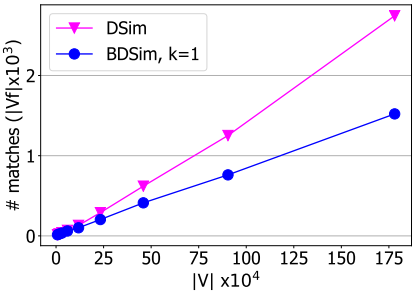
Figure 3.8 Active vertices during the execution of the distributed algorithms on real-world datasets

graph will lead to decreasing largely the size of intermediate results and this happens right from super-step one, hence, less data vertices are likely to share their match sets, resulting into a decrease in the number of returned matches and number of exchanged messages (Figures 3.10b and 3.10d respectively). The number of super-steps also decreases (Figure 3.10c) since more data vertices are filtered during the first few iterations of the algorithm unlike when $|\Sigma|$ is high where more data vertices are considered as candidates, which requires more iterations to filter out incorrect matches, thus resulting into higher response time.

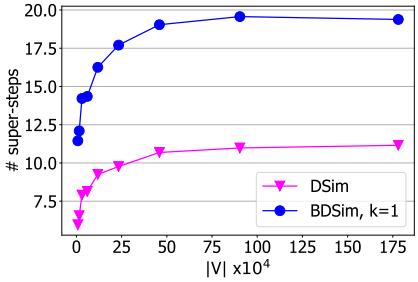
The obtained results confirm that BDSim improves the quality of the returned matches compared to the existing models. Furthermore, the approach used to detect and filter out unbounded cycles ensures an effective and efficient elimination of invalid matches without requiring large amounts of data shipment. BDSim can handle the problem of failing query thanks to parameter k. This adaptability of BDSim guarantees that only results of good quality are returned first when k is initialized to a small value. Nevertheless, k can be increased by one step in the case of zero answers to allow returning more flexible



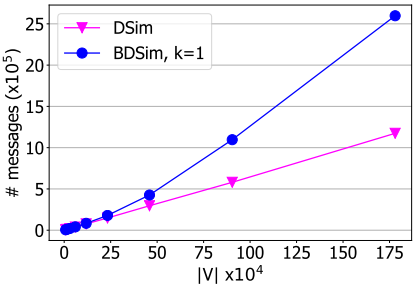
(a) Response time *w.r.t.* $|V|$



(b) Matches *w.r.t.* $|V|$

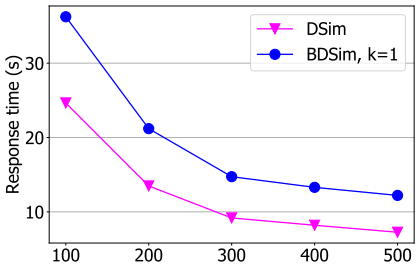


(c) Super-steps *w.r.t.* $|V|$

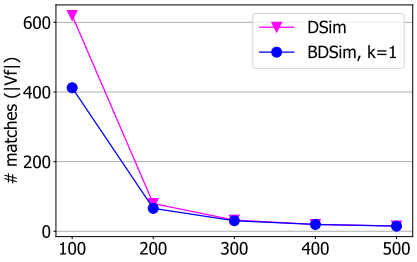


(d) Messages *w.r.t.* $|V|$

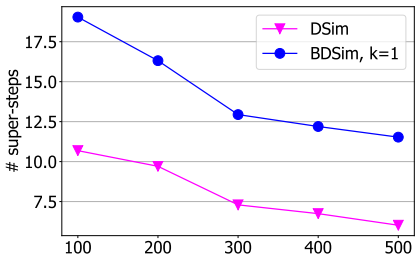
Figure 3.9 Varying the size of data graph $|V|$ for synthetic graphs, $|V_q| = 9$.



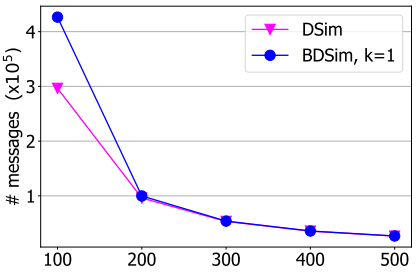
(a) Response time *w.r.t.* $|\Sigma|$



(b) Matches *w.r.t.* $|\Sigma|$



(c) Super-steps *w.r.t.* $|\Sigma|$



(d) Messages *w.r.t.* $|\Sigma|$

Figure 3.10 Varying the number of distinct labels Σ for synthetic graphs, $|V_q| = 9$.

results. Finally, even though BDSim returns answers of better quality, it remains a scalable model that requires a processing time comparable to that of dual simulation.

3.6 Chapter summary

Despite the abundant work on graph pattern matching, it is only recently that research works started focusing on the challenges brought about by current real-world applications, mainly scalability. In this chapter, we proposed BDSim, an extension of dual simulation that captures the cyclic structure of the input query graph. Our main contribution is preserving proximity between the query vertices by tracking and filtering out cycles of unbounded length from the final match graph. Indeed, we proved that the set of short cycles of a query graph is sufficient to bound the length of all the matched cycles of the match graph. Besides, we proposed distributed vertex-centric algorithms to evaluate BDSim and demonstrated their effectiveness and efficiency through theoretical guarantees and extensive experiments on real-world and synthetic data sets. The obtained results showed that BDSim significantly improves the quality of the returned answers while requiring a response time comparable to dual simulation.

BDSim and strong simulation are both relaxed GPM models that preserve duality and locality. However, even though it is not intuitive to scale up strong simulation, this model is interesting since it returns separate subgraphs as an answer to the input query graph, while BDSim returns a single match graph. Therefore, we give particular attention to strong simulation in the next chapter by proposing fully distributed and scalable algorithms that evaluate strong simulation and strict simulation with performance guarantees.

Chapter 4

A Distributed and Scalable Approach for Strong Simulation

4.1 Introduction

Strong simulation was introduced to improve the quality of graph simulation by imposing two properties, namely duality and locality. The duality ensures preserving the connectivity of the answers, while the locality property guarantees that the returned subgraphs have bounded size. Strong simulation is particularly interesting as it is close to subgraph isomorphism in terms of quality while it can still be evaluated in cubic time. Prior distributed algorithms of strong simulation [84, 39, 35, 133] did not provide any guarantees when it comes to scalability. Indeed, these approaches require moving large parts of the data graph across the cluster workers in order to make the locality balls available inside each machine, therefore resulting in large amounts of data shipment that may create a bottleneck in the case of data graphs not having a uniform degree distribution.

Motivated by these limitations, we propose D3S, a distributed and scalable algorithm that evaluates strong simulation with performance guarantees. First, D3S does not duplicate any parts of the data graph which ensures reduced communication costs. Moreover, D3S addresses the natural skewness in real graphs by processing the locality balls in a vertex-centric fashion, i.e. each member of a given locality ball participates in computing the answer related to it. Furthermore, based on D3S, we design a distributed algorithm for the more restricted model; strict simulation that reduces the size of the generated locality balls, resulting in answers of better quality compared to strong simulation. We

refer to the algorithm of strict simulation as D3S$^+$. Finally, we provide theoretical guarantees in addition to extensive experiments on synthetic and real-world datasets to validate the different algorithms proposed. The test results confirm that our approach can be five times faster than the state-of-the-art work.

The remainder of this chapter is structured as follows. Section 4.2 details the distributed vertex-centric algorithms proposed in addition to their formal validation. Strict simulation is discussed in Section 4.3. Finally, the performance evaluation of D3S and D3S$^+$ is presented in Section 4.4.

4.2 D3S: A distributed and scalable approach for strong simulation

Strong simulation strikes a balance between flexibility and tractability. It guarantees a number of properties such as (1) preserving the child relationships, i.e., if a query vertex u matches a data vertex v, then each child of u matches a child of v, (2) similarly, it preserves the parent relationships, (3) strong simulation preserves directed and undirected cycles, i.e., each undirected (directed) cycle in the query graph matches an undirected (respectively directed) cycle in the match graph, (4) locality is also preserved by strong simulation such that subgraph matches are searched in a locality ball having a diameter equal to $2 \times d$ where d is the query diameter, and finally (5) the number of subgraph matches returned by strong simulation is bounded by the number of vertices in the input data graph. Indeed, each data vertex can result to a locality ball that may return a match subgraph.

The above properties guarantee for strong simulation to be the best fit model for modern GPM applications. In a spectrum of several models varying in flexibility and tractability, strong simulation stands in the middle between subgraph isomorphism—being NP-Complete while having rigid constraints—and graph simulation, which is considered too loose for capturing the structural information of a query graph. Nevertheless, massive graphs of today are generally distributed over clusters of machines, therefore requiring the sequential graph algorithms, including strong simulation, to be revisited for such complex environments. The challenges we aim to address are the following. First, designing a scalable algorithm for evaluating strong simulation in massive graphs. Second, how to make this algorithm fully distributed so it can benefit from all the computing resources available, and hence gain in terms of performance? Next, how can we avoid duplicating the graph data, therefore allowing a scalability of the approach? In fact, avoiding the

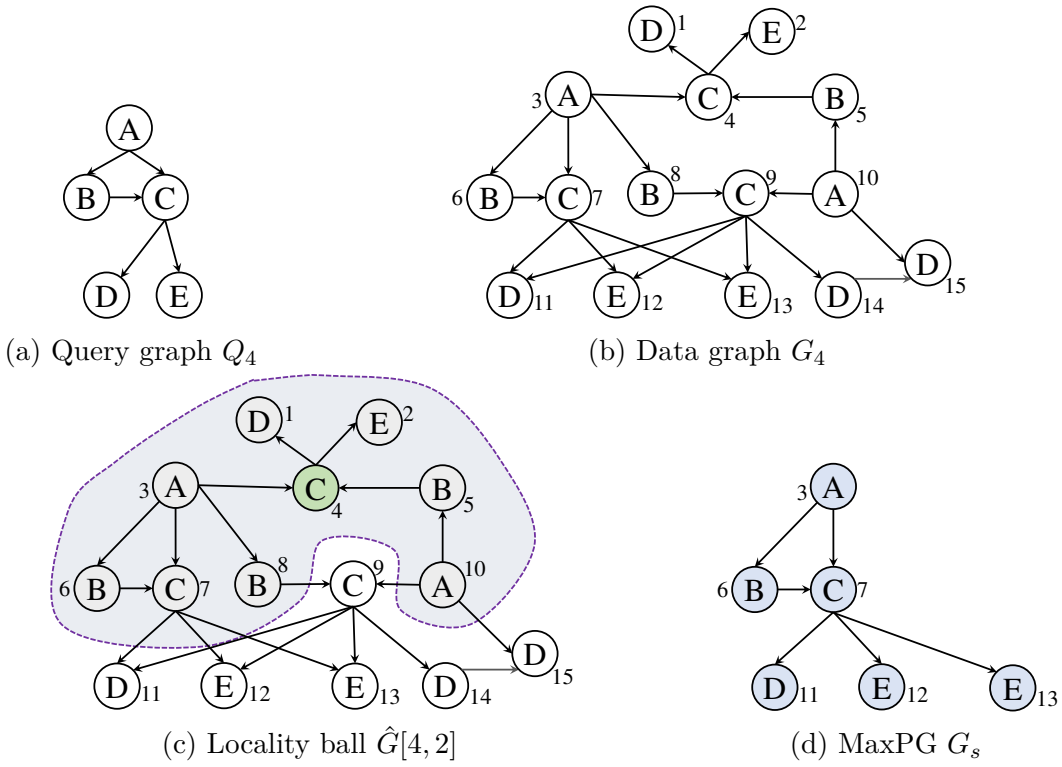


Figure 4.1 Strong simulation

replication of graph data is crucial, especially in the case of massive and dynamic graphs where update events need to be reflected while making minimum changes to the data graph.

To address these challenges, our approach for a distributed and scalable evaluation of strong simulation (D3S) is designed to work in three stages. First, a vertex-centric algorithm enumerates the dual simulation matches in a distributed way. After that, we propose a novel algorithm that performs a distributed neighborhood discovering. This step allows every data vertex being part of some locality ball to receive the identifier of the center of this ball. Finally, strong simulation is evaluated in a fully distributed way among all the data graph vertices to get the final maxPGs.

4.2.1 Overview of D3S

In this section, we give an overview of the distributed algorithm `D3S`. After that, we proceed with the design of each phase apart. However, it is important to first recall the notion of strong simulation. Strong simulation uses the notion of locality balls that are subgraphs of the data graph in which a match graph is computed. Looking for a match

graph inside such locality balls guarantees preserving distances between the match graph vertices. It is formally defined as follows.

Definition 21 (Locality ball)**.** *Given a data graph $G = (V, E, f)$, vertex $v \in V$ and positive integer d. The induced subgraph of G that contains v and all the vertices in G within a maximum distance d from v is referred to as a locality ball. This ball is denoted as $\hat{G}[v, d]$ such as v is its center and d is its radius.*

In Figure 4.1, we give query graph Q_4 and data graph G_4. Strong simulation evaluates a graph pattern matching query by first, creating locality balls centered at every vertex in G_4 and having a radius = 2 (diameter of Q_4). The locality ball $\hat{G}[4, 2]$ centered at data vertex 4 is illustrated in Figure 4.1c. When evaluating dual simulation inside this ball, the maximum dual match set returned is $\emptyset$. On the other hand, locality ball $\hat{G}[7, 2]$ contains one maxPG denoted G_s (illustrated in Figure 4.1d) that will be returned as an answer by strong simulation.

We also use the notion of matching constraints introduced in Definition 19 in evaluating dual simulation and strong simulation. For query graph Q_4, the set of parent constraints $\mathscr{C}_p$ for query vertex A is $\mathscr{C}_p(A) = \emptyset$, while the child constraints $\mathscr{C}_c$ is $\mathscr{C}_c(A) = \{B, C\}$. Similarly, query vertex C has $\{A, B\}$ as parent constraints and $\{D, E\}$ as child constraints. Given the data graph G_4 of Figure 4.1b, data vertex 4 is labeled C. Hence, it will only keep constraints related to query vertices labeled C, i.e., $\mathscr{C}_p(C) = \{A, B\}$ and $\mathscr{C}_c(C) = \{D, E\}$ should be maintained locally by vertex 4.

Algorithm D3S

```
1 Input: G, Q;
2 Output: G_s;
3 Broadcast(Q);
4 G_1 ← D3S_A(G, Q);
5 G_2 ← InducedSubgraph(G_1) // filter out vertices having empty M in parallel
6 G_3 ← D3S_B(G_2);
7 G_4 ← D3S_C(G_3);
8 G_s ← ExtractMaxPGs(G_4);
9 return G_s;
```

Algorithm `D3S` takes as input a data graph G and a query graph Q. The first step of the algorithm is broadcasting Q to the data graph vertices that will run in a vertex-centric way. The reception of this message triggers $\texttt{D3S}_A$, a distributed and vertex-centric algorithm that evaluates dual simulation locally to each data vertex (Lines 1–4). At the end of Algorithm $\texttt{D3S}_A$, every data vertex will have its correct matching information computed locally. We note that the union of these local matches gives the maximum dual

match set R_d. Nevertheless, we use directly the current state of the data graph to filter out vertices that will not participate in evaluating strong simulation (call of *Procedure InducedSubgraph* in Line 5). A detailed discussion of the importance of this filtering phase is provided under Section 4.2.3. On a relatively smaller data graph, where each data vertex has its matching information stored locally, we execute the vertex-centric neighborhood discovery algorithm $\texttt{D3S}_B$. This algorithm allows the remaining vertices to get the information of their locality balls (Line 6). Afterward, each vertex will execute a local vertex-centric algorithm, referred to as $\texttt{D3S}_C$, to evaluate dual simulation for the locality balls where it is considered a member (Line 7). Indeed, each data vertex will have different matching information stored locally depending on which locality ball is considered as the main one. We note that these locality balls are identified by their centers. Finally, Algorithm D3S calls *Procedure ExtractMaxPGs* (Lines 8) to extract the maximum perfect subgraphs from the resulting match sets related to each locality ball.

In what follows, we present the different stages of Algorithm D3S separately with examples and give their respective vertex-centric algorithms in addition to the proofs of correctness and convergence.

4.2.2 Vertex-centric dual simulation

The distributed and vertex-centric algorithm for evaluating dual simulation is denoted $\texttt{D3S}_A$. It is presented and discussed in details in Section 3.3.2. $\texttt{D3S}_A$ is a constraint-based algorithm that generates the matching constraints related to each query vertex, then, it broadcasts them among the data graph vertices. After that, each data vertex maintains a local context composed of its match set M and the set of constraints for every member in this match set. At the reception of the query graph, every data vertex v initializes M with the set of query vertices that has a similar label. Moreover, it keeps locally the set of constraints related to these query vertices for verifying dual simulation. If M is not empty, then, vertex v will share it with its direct neighbors at the end of this super-step.

Algorithm $\texttt{D3S}_A$ takes as input $\mathscr{C}_c$ and $\mathscr{C}_p$. Each data vertex v maintains a local context composed of its match set M and the set of constraints for every member in this match set. The data vertices can exchange two types of messages: data messages D_{msg} and removal messages R_{msg}. The data messages are used in propagating the matching information of a vertex to its neighbors, while the removal messages carry an update event of the local matching information to the neighbors.

At the reception of the first broadcast message containing Q, data vertex v initializes M with the set of query vertices in Q that have a similar label. It also keeps the constraints related to them locally (Lines 1–6). If M is not empty, then, data vertex v will share M with its direct neighbors in the form of a data message D_{msg} (Lines 7–9). Otherwise, v becomes inactive because it does not belong to the global match set *w.r.t.* dual simulation (Lines 10–11).

In the second super-step of $\texttt{D3S}_A$, every data vertex v having a non empty match set will be activated. If it receives a data message D_{msg}, then, it stores the received match sets of its parents and children locally in separate variables M_p and M_c, respectively (Lines 12–13). Next, even if it does not receive any data, it must evaluate the constraints of dual simulation using its local information, i.e., $M, \mathscr{C}_p, \mathscr{C}_c, M_p$ and M_c (call of *Procedure EvalDSim* in Line 14). If the evaluation leads to having one or more non satisfied constraints for a given query vertex u, then u will be removed from M and its related constraints $\mathscr{C}_p(u)$ and $\mathscr{C}_c(u)$ will also be removed. A removal message R_{msg} containing the removed matches R_m will be generated and broadcast to the neighbors of v (Lines 15–18). Every time a removal of a query vertex from M happens, it will be followed by the removal of its related constraints in local $\mathscr{C}_p$ and $\mathscr{C}_c$.

In the next super-steps, the reception of a removal message will trigger two operations. First, each data vertex receiving this removal message will update the match sets of its parents M_p and children M_c accordingly (Lines 19–20). Then, it will reevaluate the matching constraints *w.r.t.* dual simulation (call of *Procedure EvalDSim* in Line 21). If the matching constraints are not satisfied for a given query vertex u in M, then M is updated, u is added to R_m, and a removal message R_{msg} containing R_m is generated and broadcast to the direct neighbors of v (Lines 22–25). The same process is repeated in the following super-steps until the local match sets converge and no new removal messages are generated anymore. At the end of the distributed algorithm, every data vertex will have its local match set *w.r.t.* dual simulation. If every single query vertex has at least one match, then, the union of these local matches is exactly the global match set *w.r.t.* dual simulation. Otherwise, the global match set is $\emptyset$.

Example 9. *In data graph G_4 and query graph Q_4, if we notice data vertex* 15 *that has label D, its parent constraints are $\mathscr{C}_p(D) = \{C\}$ and it has an empty child constraints $\mathscr{C}_c(D) = \emptyset$. On the other hand, its parents in G_4 are $\{10, 14\}$ and their respective mappings from the first iteration are $\{(10, A), (14, D)\}$. Since vertex* 15 *does not have a parent that satisfies the parent constraints (a parent matched to query vertex C), then it will remove C from its local match set (along with the related constraints). After that,*

it will create a removal message R_{msg} *with the mapping* $(15, D)$*. Then, it will share it with neighbors* 10 *and* 14*. since the removal of vertex* 15 *will not affect the match sets of vertices* 10 *and* 14*, the next super-step is the last one in Algorithm* $\texttt{D3S}_A$*.*

4.2.3 Vertex-centric neighborhood discovery

Let $G = (V, E, f)$ be a data graph, $Q = (V_q, E_q, f_q)$ a query graph and d the diameter of Q. Strong simulation improves the quality of dual simulation by imposing an additional constraint which is the locality. This property requires dual simulation matches to be located inside a locality ball of radius d and centered at data vertices that are already part of the maximum dual match set R_d.

Moreover, given G' the induced subgraph of G *w.r.t.* V' (set of vertices in R_d), it has been shown in [84] that strong simulation can be directly evaluated on G' instead of G. Indeed, for every $v \in V$, the maxPG resulting from the locality ball b centered at v with radius d in G is equivalent to the maxPG extracted from locality ball b' centered at v with radius d in G'. Therefore, we use G' in this phase to reduce the size and number of locality balls, hence, avoiding unnecessary computations. Working on the induced subgraph G' eliminates locality balls centered at vertices with an empty match set *w.r.t.* dual simulation. It also ensures that locality balls are composed of only valid dual simulation matches.

We propose discovering the *d-neighborhood* of each data vertex in R_d, then evaluating dual simulation for the resulting locality balls in a distributed way. To do so, we give the vertex-centric algorithm $\texttt{D3S}_B$ that allows each vertex in G' to propagate its identifier in a ball of radius d. This propagation allows other vertices that share the same locality to discover it.

$\texttt{D3S}_B$ is triggered by an initial broadcast message I_{msg}. At the reception of I_{msg}, the centers of locality will generate and send a data message with their identifiers to their locality members (Lines 1–3). Upon the reception of a locality ball center c, the receiving vertex will add it to a local set named L_b. It will also map it in a *[key, value]* form with its local match set $M(c)$ and its parent and child match set $M_p(c)$ and $M_c(c)$, respectively (Line 4–5). An illustration of partial local context of data vertex 4 is given in Figure 4.2. After that, a data vertex will propagate the received centers to its neighbors (Lines 6–7). The propagation will follow in the next super-steps until Super-step $d + 1$ where every data vertex will have the list of centers of all the locality balls to which it belongs (Lines 8–10).

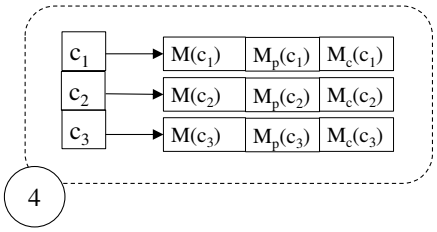


Figure 4.2 Local context of vertex 4

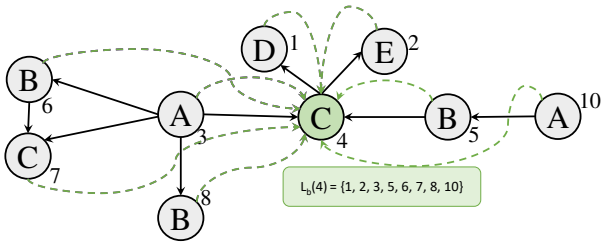


Figure 4.3 Construction of L_b for vertex 4

At the end of Super-step $d+1$, each data vertex with a non empty list of centers will share this list with its direct neighbors (Lines 11–13). When a data vertex receives such list, it will store it locally in a variable named N_b (Lines 14–16). At this point, every data vertex has a list of locality balls centers attached to their corresponding local match sets. The parent and child match sets are also kept locally as they help in identifying the matches to be removed in the next phase. The variable N_b is used to refine M_p and M_c for each center $c \in L_b$.

Algorithm D3S$_B$

1 **Super-step 1:** At the reception of I_{msg} while $M \neq \emptyset$, do;
2 Send v to direct neighbors;
3 Vote to halt;
4 **Super-step 2 to** d**:** at the reception of S_b while $M \neq \emptyset$, do;
5 Add new S_b to L_b;
6 Send new S_b to direct neighbors;
7 Vote to halt;
8 **Super-step** $d+1$**:** if $M \neq \emptyset$, then do;
9 **if** *Received* $S_b \neq \emptyset$ **then**
10 Add new S_b to L_b;
11 **if** $L_b \neq \emptyset$ **then**
12 $N_b \leftarrow (v, L_b)$ Send N_b to direct neighbors;
13 Vote to halt;
14 **Super-step** $d+2$**:** at the reception of N_b while $M \neq \emptyset$, do;
15 Store N_b locally;
16 Vote to halt;

Example 10. *In Example query graph Q_4 and data graph G_4, we have the query diameter $d = 2$, hence the locality balls will have a radius equal to 2. Moreover, since data vertex 4 belongs to the maximum dual match set, it must propagate its identifier in a 2-neighborhood. In the next super-step, vertex 4 will also discover the locality balls to which it belongs and that are at one hop, i.e. it will get the set $\{1, 2, 3, 5\}$. Vertex 4 will store these centers locally in variable L_b, then it will receive centers $\{6, 7, 8, 10\}$ at the beginning of the third super-step. It will also share this list with its direct neighbors*

$\{1, 2, 3, 5\}$. *Finally, it will receive the locality centers of these neighbors and store them in* N_b. *Figure 4.3 illustrates the execution of this phase.*

Lemma 3. *At the end of Algorithm* $\texttt{D3S}_B$, *every data vertex having a non-empty match set will have the set of locality balls centers to which it belongs.*

Proof. The correctness of Algorithm $\texttt{D3S}_B$ is guaranteed by the following properties. (1) the algorithm will terminate (Lemma 4), (2) every vertex v has the correct and complete set of centers for each locality ball in which it is a member.

First, we suppose that the algorithm collects an invalid center, and then prove the impossibility of this case. The set L_b contains vertices that are in a *d-neighborhood* from v and that have a non empty match set, hence, an incorrect center is either a vertex with a non empty match set or a vertex that is located in a distance $> d$ from v. In every super-step of the algorithm, a data vertex is activated only if it has a non empty match set, hence, a vertex that communicates data with its neighbor must have a non empty match set. Furthermore, L_b is constructed in an iterative way between the super-steps 1 and $d+1$ where a message generated in Super-step i arrives at the next hop in Super-step $i+1$. Consequently, a message triggered by a vertex v' reaches hop v which has a distance $d' > d$ from v after exactly d' super-steps. However, the construction of L_b takes exactly $d+1$ super-steps. Therefore, v cannot receive a center of a locality ball b such that $v \notin b$.

Second, we suppose that the algorithm misses a center c for a vertex v and we prove that this cannot happen. A center c is added to L_b for data vertex v if and only if: v has a non empty match set, c has a non empty match set, the undirected distance between v and c is $\leq d$. We distinguish two scenario cases, c is a direct neighbor of v, it is directly add to L_b in Super-step 2 (Line 5) where the new S_b is exactly the received S_b because L_b is empty at the beginning of this super-step. The other case is where c has a distance $d' > 1$ from v. If $' = 2$, then c and v share a common neighbor v', thus, c has already been communicated to v' in the first two super-steps. At the beginning of Super-step 2, v' has c in its new S_b, hence, it will share it with its direct neighbors including vertex v. In the same way, we prove that c will reach v in d' super-steps at most $(2 < d' \leq d)$ and will be added to L_b. □ □

Lemma 4. *Algorithm* $\texttt{D3S}_B$ *for discovering the d-neighborhood of a data vertex will terminate.*

Proof. We know that the maximum number of super-steps of $\texttt{D3S}_B$ is finite ($d + 2$ such that d is the diameter of the query graph). Moreover, each super-step will terminate.

Actually, the first super-step is an atomic operation that will terminate. The second super-step will also terminate because it iterates over a list of finite size (number of received messages from neighbors is finite). The same thing applies to the super-steps from 3 to $d + 1$ that will terminate. The last super-step involves an atomic operation for storing the received data locally, hence, the algorithm will terminate. □ □

4.2.4 Vertex-centric strong simulation

The last phase of our approach consists of evaluating strong simulation according to every locality ball in a distributed way. Each data vertex in G' uses the N_b as a guide to update $M_p(c)$ and $M_c(c)$ for each center c in L_b (lines 1-4). Actually, if a neighbor n does not belong to a locality ball centered at $c \in L_b$, then, its entries in $M_p(c)$ and $M_c(c)$ will be filtered out. Indeed, the membership of n is retrieved from N_b.

Next, each data vertex v reevaluates the matching constraints *w.r.t.* dual simulation according to each center in $\{L_b\}$ (the call of *Procedure EvalDSim* in Line 5). If the dual simulation constraints are not satisfied anymore for some specific center c, this means that v does not belong to a maxPG resulting from the locality ball centered at c, hence, a removal message R_{msg} must be generated to inform its neighbors. The removal message is composed of the related locality ball center, the data vertex identifier and removed query vertex (Lines 6–8). Furthermore, v will also remove itself from a locality ball if its corresponding match set becomes empty, i.e. it removes the center entry from variable L_b (Lines 9–11). Afterward, v continues processing all the locality centers while generating removal messages if there are any non respected constraints.

Upon the reception of a removal message R_{msg} in the second super-step, each data vertex checks first if it belongs to the related locality ball, if not, the message is ignored (Line 14). Otherwise, the vertex updates the parents and children local match sets for that center c (Line 15). After that, it reevaluates again the matching constraints *w.r.t.* dual simulation (call of *Procedure EvalDSim* in Line 16). This call returns the updated match set $M(c)$ in addition to the set of removed matches R_m. If R_m is not empty, a removal message R_{msg} is generated and sent to all the direct neighbors (Lines 17–19). Similar to the previous super-step, if the local match set *w.r.t.* center c becomes empty, then v removes itself from the corresponding locality ball, i.e., it removes c from L_b (Lines 20–22).

In the following super-steps, every removal message will trigger the reevaluation of dual simulation constraints with possible updates of the local match sets and generation of new removal messages. The same process is triggered in each super-step until no messages

are exchanged anymore announcing the convergence of the distributed algorithm. At this point, every data vertex has locally the set of locality balls in which it has matches. The union of matches for each locality ball forms the maxPG of that center. Finally, *Procedure ExtractMaxPGs* will check for each returned maxPG if its center has a non-empty local match set and that every query vertex has at least one match. Otherwise, this maxPG is filtered out.

Algorithm D3S$_C$

```
1  Super-step 1: At the reception of I_msg while M ≠ ∅, do;
2  foreach c in L_b do
3      M_p(c) ← updated M_p based on N_b ;
4      M_c(c) ← updated M_c based on N_b ;
5      (M(c), R_m) ← EvalDSim(C, M, M_p(c),M_c(c)) ;
6      if R_m ≠ ∅ then
7          R_msg ← (c, v, R_m);
8          Send R_msg to neighbors ;
9      if M(c) = ∅ then
10         Add c to R_c ;
11 Remove R_c from L_b ;
12 Vote to halt ;
13 Super-step 2 and beyond: At the reception of R_msg;
14 if L_b contains R_msg.c then
15     Update M_p(c) and M_c(c);
16     (M(c), R_m) ← EvalDSim(C, M, M_p(c),M_c(c)) ;
17     if R_m ≠ ∅ then
18         R_msg ← (c, v, R_m);
19         Send R_msg to neighbors ;
20     if M(c) = ∅ then
21         Remove c from L_b ;
22 Vote to halt;
```

Example 11. *In the data graph G_4, vertex* 5 *has the same child match set* $M_c = \{(1, D), (2, E)\}$ *and the parent match set* $M_p = \{(3, A), (5, B)\}$ *for every center* $c \in L_b$. *To evaluate dual simulation for any locality ball, it must first update the parent and child match sets related to it according to the contents of* N_b. *If we take the locality ball* b_6 *centered at* $6 \in L_b$, *the neighbors* $\{1, 2, 5\}$ *do not belong to it, hence, their entries are removed from* $M_p(6)$ *and* $M_c(6)$. *We end up with only one entry in the neighbors match sets of center* 6, *i.e.,* $(3, A)$. *The local dual simulation constraints are not satisfied for* b_6, *hence, vertex* 4 *will remove itself from* b_6, *i.e., remove* 6 *from* L_b *and generate a removal message with this information. This removal message is sent to the direct neighbors that will update their match sets for* b_6. *This data vertex will do the same thing for the other centers in* L_b. *The remaining vertices in* G' *will execute the same routine resulting in*

new removal messages that will be propagated among the graph vertices. At the end of the algorithm, only the centers $\{3, 6, 7, 11, 12, 13\}$ *will remain. If we take center* 7*, the five vertices* $\{3, 6, 7, 11, 12, 13\}$ *all satisfy dual simulation for its locality ball. These vertices form the maxPG that will be returned by the algorithm.*

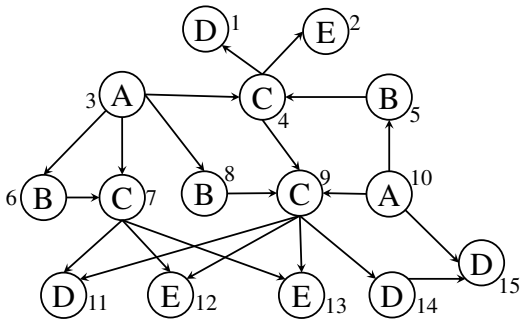
Lemma 5. *Algorithm* $\texttt{D3S}_C$ *returns the correct and complete set of maxPGs w.r.t. strong simulation.*

Proof. In the same way we proved the correctness of $\texttt{D3S}_A$, one can see that, considered separately, each data locality center represents a locality ball in which we evaluate dual simulation in a distributed vertex-centric way (using the same routines in algorithm $\texttt{D3S}_A$). Each locality ball b_c centered at vertex c has its own local match set $M(c)$, parent match set $M_p(c)$ and child match set $M_c(c)$. Furthermore, we evaluate the matching constraints separately and every removal in $M(c)$ triggers a removal message that is propagated among the the members of b_c. □ □

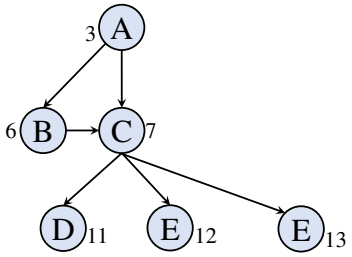
Lemma 6. *Algorithm* $\texttt{D3S}_C$ *for evaluating strong simulation will terminate.*

Proof. The first super-step of the algorithm simply iterates over the local set L_b that has a finite size. Inside this loop, we update the two sets $M_p(c)$ and $M_c(c)$ by iterating over a finite set N_b and updating one element at a time. Procedure *EvalDSim* iterates over finite sets. Therefore, it will terminate. The remaining operations in the loop body are atomic, hence the loop body will terminate. At the end of this iteration L_b is updated in constant time. Therefore, the first super-step will terminate. In the next super-steps, every data vertex updates its local match sets based on the received removal messages. Removing invalid matches from $M_p(c)$ and $M_c(c)$ takes constant time. Reevaluating dual simulation for the locality center c also takes finite time. The remaining operations in this super-step are atomic operations of message creation, and removal of invalid elements elements from a finite set L_b. Hence, the second super-step will terminate. The maximum number of the remaining super-steps is $|E|$ in the worst case where a locality ball is composed of the whole match graph G' and a removal message is propagated across all the locality ball's edges. Consequently, Algorithm $\texttt{D3S}_C$ will terminate. □ □

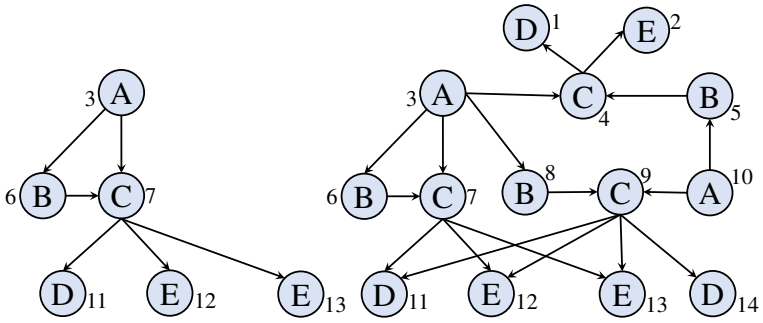
Given a data graph $G = (V, E, f)$ and a query graph $Q = (V_q, E_q, f_q)$, to find the maxPGs *w.r.t.* strong simulation we use the distributed vertex-centric algorithm *D3S* which is given as the successive execution of $\texttt{D3S}_A$, $\texttt{D3S}_B$ and $\texttt{D3S}_C$.



(a) Data graph G'_4



(b) MaxPG returned by strict simulation



(c) MaxPGs returned by strong simulation

Figure 4.4 Difference between maxPGs returned by strict simulation and the one returned by strong simulation in the case of query graph Q_4 and data graph G'_4.

Theorem 8. *Algorithm D3S returns the correct and complete set of maxPGs w.r.t. strong simulation.*

Proof. The correctness of *D3S* results directly from Theorem 5, Lemma 3 and Lemma 5. □ □

4.3 D3S$^+$: Distributed and scalable evaluation of strict simulation

Strict simulation is an extension of strong simulation where the data locality balls are generated from the dual match graph instead of the induced subgraph. This optimization reduces the size of locality balls and restricts further the size of returned matches. Consequently, the data shipment is reduced and the returned maxPGs are of better quality. We name the distributed vertex-centric algorithm for strict simulation `D3S`$^+$. This variant of Algorithm `D3S` uses a dual match graph instead of the induced subgraph (call of *Procedure DualMatchGraph* in Line 7 of *Algorithm D3S*$^+$).

Algorithm D3S$^+$

```
1 Input: G, Q;
2 Output: G_s;
3 Broadcast(Q);
4 G_1 ← D3S_A(G, Q);
5 G_2 ← DualMatchGraph(G_1) // Filter out vertices and edges not having a match in
      Q
6 G_3 ← D3S_B(G_2);
7 G_4 ← D3S_C(G_3);
8 G_s ← ExtractMaxPGs(G_4);
9 return G_s;
```

To show how strict simulation reduces the size of locality balls, we use data graph G'_4 given in Figure 4.4 and the query graph Q_4. Strong simulation returns two maxPGs composed of the whole data graph except data vertex 15 (Figure 4.4c). This is due to the additional edge $(4, 9)$ that allowed data vertices 4 and 9 to find the same large maxPG inside their locality balls. However, since strict simulation is based on the match graph resulting from dual simulation, this particular edge will be removed, because no edges labeled (C, C) are present in Q_4. Now, we build the data locality balls on top of this filtered graph, which results into only one maxPG given in Figure 4.4b. The locality balls collected by vertex 4 in strong simulation and strict simulation are $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$ and $\{1, 2, 3, 4, 5, 6, 7, 8, 10\}$, respectively. Evaluating strong simulation on the first locality ball gives the maxPG composed of data vertices $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$, while strict simulation does not find any matches inside the second ball. This example proves that not only the locality balls are smaller in size but also the quality of results is always similar or better.

4.4 Experimental evaluation

We conducted our experiments on real and synthetic data sets. The goal is to measure the performance of `D3S` and `D3S`$^+$, identify their bottlenecks and compare them to an algorithm from the state-of-the-art.

4.4.1 Distributed implementation

We implemented the two algorithms `D3S` and `D3S`$^+$ on top of GraphX using the programming language `SCALA`. GraphX [143] is a module provided by Apache Spark that allows vertex-centric graph processing through its exposed API of Pregel [86].

The implementation provided by GraphX for Pregel executes a distributed algorithm as a series of super-steps where each vertex that receives a message at the end of the previous super-step is activated to execute its local program and prepare messages for the next super-step. These super-steps are separated by synchronization barriers following the BSP model. GraphX offers an additional feature for exchanging messages. A data vertex can access its neighbors' local context to decide whether it needs to exchange messages with them or not. We use the exposed information from each data vertex to decide whether or not to send a message in three scenarios. The first scenario is related to the evaluation of dual simulation, a data vertex shares its local match set only with data vertices that can benefit from it in building their local match sets. Actually, a neighbor is interested in receiving a match set if the vertex sending it matches their parent and/or child constraints. Hence, if there is no intersection between the vertex constraints and this neighbor's match set, the match set will not be shared with it, which eliminates useless messages and avoids unnecessary computations. The second scenario appears in the phase of neighborhood discovery when a data vertex decides to propagate its list of centers, it only sends the ones not already known by each neighbor. This optimization reduces the size and number of exchanged messages. Finally, the removal messages are only shared with data vertices having a non empty match set, because there is no need to activate data vertices that are already eliminated from the global match set. These improvements reduce the number and size of exchanged messages which optimizes significantly the response time of `D3S` and `D3S`$^+$.

4.4.2 Experimental environment

We worked on four real-world graphs from SNAP Datasets [74]. The characteristics of these real graphs are given in Table 4.1. Unless expressly stated otherwise, the number of distinct labels in each data graph is fixed to $|\Sigma| = 500$.

Furthermore, the R-Mat model [18] is used to generate synthetic data graphs. It takes as an input $|V|, |E|$ and $|\Sigma|$ such that $|\Sigma|$ is the number of distinct labels and $|E| = 20 \times |V|$.

Table 4.1 Characteristics of the data graphs used in the different experiments

Dataset (G)	$\lvert V\rvert$	$\lvert E\rvert$
Epinions	75,879	508,837
Amazon0601	403,394	3,387,388
WebGoogle	875,713	5,105,039
LiveJournal	4,847,571	68,993,773

When varying the values of $|V_q|$, we use the algorithm given in Section 3.5.3 to extract 50 different patterns randomly for each value.

We executed our experiments on a Spark cluster composed of 10 nodes with 32GB memory and 16 cores for each; one node plays the role of the master while the remaining nodes are considered as workers.

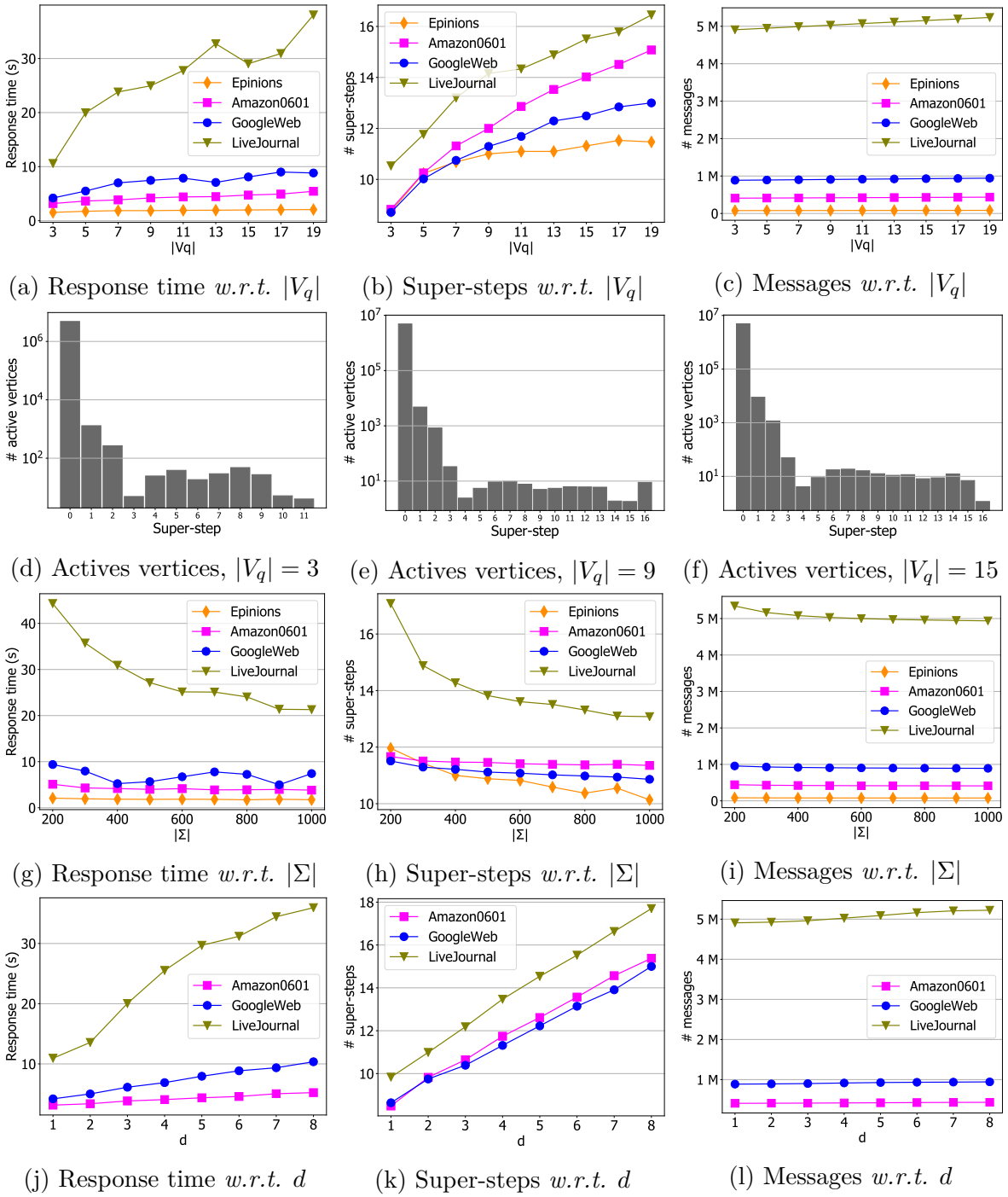
4.4.3 Experimental results

The results of performance evaluation for `D3S` and `D3S`$^+$ and comparison with the state-of-the-art distributed algorithm for evaluating strict simulation `Strict13` from [39] are presented and discussed in this section.

Exp-1: In the first set of experiments, we evaluate the performance of `D3S` by varying the following graph parameters: The size of query graph $|V_q|$, the query diameter d and the number of distinct labels $|\Sigma|$. The obtained results are discussed below.shown in Figure 4.5. In Figure 4.5a, we notice that the size of the query graph does not affect greatly the overall running time of `D3S`, especially for small and average data graphs. The number of super-steps (Figure 4.5b), on the other hand, depends largely on the size of the query graph and follows a logarithmic growth in function of the query size. In Figure 4.5c, we see that the number of exchanged messages is almost constant and does not seem to be dependant of the size of query graphs. In Figures 4.5d, 4.5e and 4.5f, we see that the highest number of active vertices is reached mainly in the first few iterations where data vertices not satisfying dual simulation are being eliminated. After that, the number of active vertices remains stable until the end which confirms that the two algorithms `D3S`$_B$ and `D3S`$_C$ converge with a bounded number of active vertices.

Since the number of iterations of the distributed algorithm is given in function of the diameter of the query graph, we vary this parameter. We report the average response time in Figure 4.5j, number of super-steps in Figure 4.5k and number of exchanged messages in Figure 4.5l. The results obtained show that, indeed, the number of super-steps grows linearly when d increases, reaching at most 20 iterations for queries having a diameter equal to 8. The response time is also sensitive to the change in query diameters. It increases faster for larger data graphs, namely *LiveJournal*. Moreover, the data shipment, represented by the total number of messages, remains constant when varying d, which proves that `D3S` is insensitive to the query diameter in terms of data shipment.

Finally, since the number of distinct labels plays in important role in the filtering phase where only the candidate vertices are kept active, we evaluate the performance of `D3S`



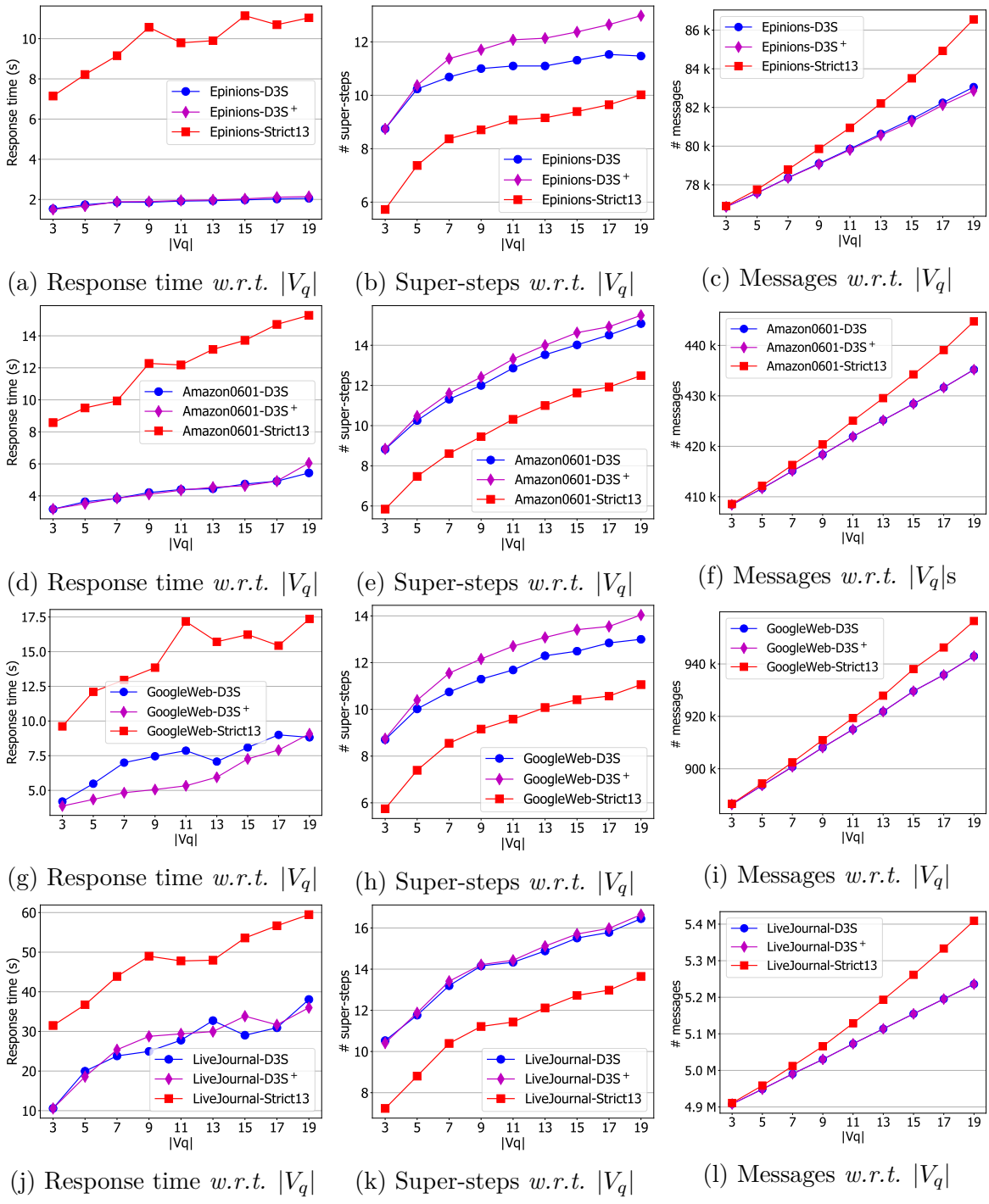
(a) Response time *w.r.t.* $|V_q|$ (b) Super-steps *w.r.t.* $|V_q|$ (c) Messages *w.r.t.* $|V_q|$

(d) Actives vertices, $|V_q| = 3$ (e) Actives vertices, $|V_q| = 9$ (f) Actives vertices, $|V_q| = 15$

(g) Response time *w.r.t.* $|\Sigma|$ (h) Super-steps *w.r.t.* $|\Sigma|$ (i) Messages *w.r.t.* $|\Sigma|$

(j) Response time *w.r.t.* d (k) Super-steps *w.r.t.* d (l) Messages *w.r.t.* d

Figure 4.5 Performance evaluation of *D3S*



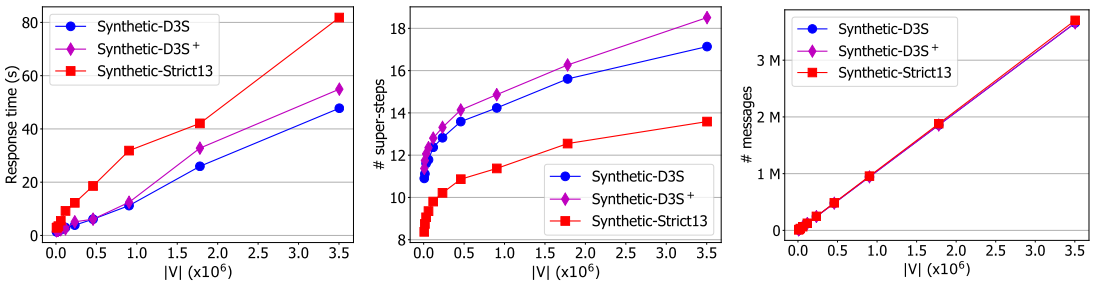
(a) Response time *w.r.t.* $|V_q|$ (b) Super-steps *w.r.t.* $|V_q|$ (c) Messages *w.r.t.* $|V_q|$

(d) Response time *w.r.t.* $|V_q|$ (e) Super-steps *w.r.t.* $|V_q|$ (f) Messages *w.r.t.* $|V_q|$s

(g) Response time *w.r.t.* $|V_q|$ (h) Super-steps *w.r.t.* $|V_q|$ (i) Messages *w.r.t.* $|V_q|$

(j) Response time *w.r.t.* $|V_q|$ (k) Super-steps *w.r.t.* $|V_q|$ (l) Messages *w.r.t.* $|V_q|$

Figure 4.6 Comparing *D3S* and *D3S*$^+$ to *Strict13* on different datasets when varying the query graph size $|V_q|$



(a) Response time *w.r.t.* $|V|$ (b) Super-steps *w.r.t.* $|V|$ (c) Messages *w.r.t.* $|V|$

Figure 4.7 Comparing D3S and D3S$^+$ to Strict13 on synthetic graphs when varying the data graph size $|V|$.

when varying $|\Sigma|$ in the data graph. We report the average response time in Figure 4.5g, average number of super-steps in Figure 4.5h and average number of exchanged messages in Figure 4.5i. The number of labels also has a little effect on the performance of D3S. Actually, the less labels are in the data graph, the more computations are triggered to eliminate invalid matches among the remaining candidates. This is explained by the fact that the number of initial candidates gets larger when the number of distinct labels decreases. The same thing applies to the number of super-steps. However, the number of messages is almost the same for all values of $|\Sigma|$ which proves that D3S is insensitive to this parameter.

Exp-2: In this set of experiments, we compare D3S and D3S$^+$ to Strict13 when varying the size of query graph. The average response time, the average number of super-steps the average number of exchanged messages with respect to $|V_q|$ are presented in Figure 4.6. We conducted the comparison on the real datasets *Epinions*, *Amazon0601*, *GoogleWeb* and *LiveJournal*.

These experiments show that D3S and D3S$^+$ are faster than Strict13 by up to five times. The number of super-steps is, in turn, larger for D3S and D3S$^+$ because all the computations are conducted in a vertex-centric way while in Strict13, the last iteration is the longest as it evaluates strict simulation for every data locality ball in a sequential way. This explains why D3S and D3S$^+$ are faster even though they require additional super-steps. D3S$^+$ runs more super-steps than D3S to get the final maxPGs because strict simulation is more stringent than strong simulation, hence, D3S$^+$ involves the removal of extra invalid matches that were accepted by D3S. The data shipment is larger in Strict13 which slows down the computations compared to D3S and D3S$^+$ for the four datasets. In Figures 4.6c, 4.6f, 4.6i and 4.6l, we see that the larger the query graph, the

larger becomes the gap between the number of messages exchanged in `D3S` (or `D3S`$^+$) and `Strict13`.

Exp-3: In these experiments, we test the weak scalability of our approach in comparison to `Strict13` by increasing the size of the data graph. We varied the scale of R-Mat graphs from 13 to 22 (size of data graphs from 2^{13} to 2^{22}) and compared `D3S` and `D3S`$^+$ to `Strict13` ($|V_q|$ is set to 9). The results are given in Figure 4.7.

In Figure 4.7c, the data shipment is slightly larger for `Strict13`. Even though the number of super-steps (Figure 4.7b) is larger for both `D3S` and `D3S`$^+$, the response time in Figure 4.7a, on the other hand, is shorter by two thirds compared to `Strict13`.

4.5 Chapter summary

In this chapter, we studied strong simulation, an interesting model for answering GPM queries. It allows capturing the topology of the query graph compared to graph simulation while still feasible in cubic time. We proposed the distributed approaches D3S and D3S$^+$ for evaluating strong simulation and strict simulation, respectively. To achieve higher levels of scalability, D3S and D3S$^+$ were designed to be fully distributed such that the data graph parts are never shipped between workers or between vertices residing on the same worker. Consequently, strong simulation (and strict simulation) can be extended to cases where the semantic similarity is defined based on multiple attributes. We validated the effectiveness and efficiency of our algorithms through theoretical guarantees in addition to extensive experiments on synthetic graphs and real-world datasets. Furthermore, the test results show that D3S and D3S$^+$ run faster than the state-of-the-art work, which makes them suitable for massive graphs and the current applications of GPM in social networks.

Notably, the acceleration achieved by D3S and D3S$^+$ mostly concerns the neighborhood discovery and strong simulation evaluation. Although we used a distributed vertex-centric algorithm to evaluate dual simulation, an important portion of the processing time of D3S and D3S$^+$ is spent in dual simulation. Moreover, the two GPM models dual simulation and graph simulation are particular as they do not require locality during the matching process, which raises a question on the need for a vertex-centric processing to find answers *w.r.t.* these two models. We attempt to answer this question in the next chapter by proposing two parallel and edge-centric algorithms that outperform significantly the vertex-centric versions. Accordingly, we can use the parallel dual simulation to accelerate further the two algorithms D3S and D3S$^+$.

Chapter 5

An Efficient Parallel Edge-Centric Approach for Relaxed GPM

5.1 Introduction

Even though many works have addressed the problem of GPM for large graphs, the actual size of data graphs is still challenging. In fact, social networks are generating huge amounts of data continuously, e.g., Facebook was the largest network with 2.4 billion monthly active users in June 2019 [31]. These networks cannot fit on the memory of a single machine due to their large size; hence, they need distributed storage and processing. Among the challenges we encounter in such distributed environments, there is linear scalability, a key property for distributed graph algorithms. Yet, prior works addressing relaxed GPM in this context are limited due to the bounded level of parallelism that can be reached [85, 39, 36, 109, 62, 77, 37].

In this chapter, we show that graph simulation and dual simulation can be both efficiently evaluated by parallel algorithms that achieve linear scalability. Actually, the different algorithms given by [39, 109, 62, 77] adopt the vertex-centric paradigm of Pregel [86]. In Pregel, the data graph is seen as a distributed system where vertices are analogous to computing units that can execute their local programs in parallel and exchange messages with their neighbouring vertices. The algorithm runs in a series of computations separated by a synchronization barrier and message exchanges. An initial message consisting of the query graph is sent to all the data vertices that will store their matching information locally. In each iteration the graph vertices share their matching information or removal messages with their neighbors that update their local matches accordingly.

The distributed algorithm converges when no further messages are exchanged anymore. Nevertheless, the vertex-centric paradigm is limited because it generally exhibits a heavy message passing. Additionally, the natural skew present in real-world graphs does not allow for a higher degree of parallelism. Finally, the vertex-centric paradigm is more suitable for graph algorithms that require data locality. Indeed, data locality is important for graph problems where a large neighbourhood of a vertex is required to evaluate the GPM model, e.g. subgraph isomorphism and strong simulation.

Furthermore, the algorithms proposed in [36, 77, 37] adopt partial evaluation in answering graph simulation queries. In partial evaluation, each worker of the distributed system evaluates graph simulation based only on the data stored locally, then, it communicates with a coordinator machine that propagates the newly computed matching information among the workers. At the reception of the matching information of other workers, a worker will update its local matching information accordingly. If there are any updates, another round of message exchange and computations are carried out until the algorithm convergence. Depending on the partitioning strategy adopted, some workers may remain inactive for most of the processing time. Moreover, these works do not provide mechanisms for parallelizing computations on the same machine, hence, limiting the degree of parallelism that can be achieved.

These shortcomings motivated our work to propose *PGSim* and *PDSim*, two parallel edge-based algorithms to answer GPM queries in parallel for the two models, respectively. Our main contributions are summarized as follows.

(1) We introduce *ST*, a novel distributed data structure for storing the data graph edges which ensures a high degree of parallelism. *ST* is composed of basic computing units called *STwigs*; a set of edges having the same source or the same destination vertex. *ST* ensures high scalability due to the independence between its elements that can be processed in parallel.

(2) We propose *PGSim*, an edge-centric algorithm for evaluating graph simulation in parallel on distributed data graphs. To the best of our knowledge, this is the first work on graph simulation that adopts an edge-centric programming model.

(3) Based on *ST* and *PGSim*, we propose *PDSim*, a fast parallel algorithm for evaluating dual simulation in the same context of massive graphs.

(4) In addition to that, we prove the effectiveness of our approach by giving theoretical guarantees on the correctness of the different algorithms proposed in this chapter.

(5) Furthermore, we propose an implementation of the parallel algorithms on top of the in-memory distributed system Apache Spark [151].

(6) Finally, we prove the efficiency of *PGSim* and *PDSim* through extensive experiments and compare them to the state-of-the-art vertex-centric approach on different real-world and synthetic data graphs. The obtained results verified that *PGSim* and *PDSim* can be ten times faster than their vertex-centric counterparts.

The remainder of this chapter is organized as follows. First, we introduce *PGSim* in Section 5.2 and give a formal validation of the proposed algorithm. In Section 5.3, we propose the parallel edge-centric algorithm for dual simulation with theoretical guarantees. Finally, Section 5.4 gives the implementation details in addition to the results of experimental evaluation of *PGSim* and *PDSim*.

5.2 Parallel edge-centric graph simulation

A particularity of graph simulation is that it requires availability of only the matching information from the children of a data vertex to decide whether it is a correct match or not. To illustrate this, a query graph Q_5, a data graph G_5 and their match graph G_s *w.r.t.* graph simulation are given in Figures 5.1, 5.2 and 5.3, respectively. We use query labels as identifiers only for simplicity purpose. For example, the query vertex B is mapped to data vertex 6 in the resulting match graph. Moreover, vertex B has a child vertex C, hence, at least one of the children of vertex 6 should be mapped to C. Notice that B has also a parent relationship with vertex C, however, this relation is not reflected in G_s. Such scenario is possible because we only care about the child relationships when answering queries via graph simulation.

To design an efficient and scalable algorithm, we consider the fact that only children of a data vertex are required in graph simulation, which allows us to avoid a sequential traversal of the data graph. Exploiting this property will decouple the different parts of the data graph and the computations performed on them, hence, increasing the degree of parallelism. Moreover, the data locality preserved by the vertex-centric programming model is more important for graph problems where a large neighbourhood of a vertex is required to evaluate the GPM model, e.g. subgraph isomorphism and strong simulation. Therefore, this paradigm is more suitable for such problems, but is less accurate for the case of graph simulation because the message passing communication adopted causes a very low speed of computations. Moreover, the vertex-centric paradigm requires message passing even for vertices residing on the same physical machine even though the exchanged

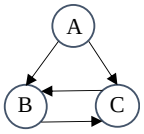


Figure 5.1 Query graph Q_5

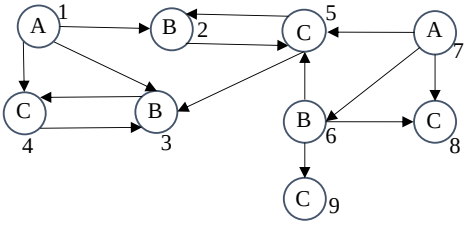


Figure 5.2 Data graph G_5

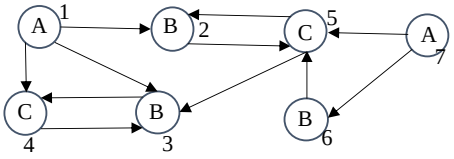


Figure 5.3 Match graph *w.r.t.* graph simulation (G_s)

information between any pair of vertices is directly available if we adopt an edge view. These observations motivated our proposition of *PGSim* a parallel, edge-based algorithm that adopts a shared memory abstraction instead of message passing to evaluate graph simulation on distributed graphs.

In what follows, we introduce the different data structures used by our approach. Then, we present the steps of the parallel algorithm *PGSim* and give its formal validation.

5.2.1 Data structures

Graph simulation defines a set of matching constraints that must be respected by each data vertex in order to be considered as a correct match for a given query vertex. We define a matching constraint as follows.

Definition 22 (Matching constraint of a vertex)**.** *Given a query graph $Q = (V_q, E_q, f_q)$, the matching constraints of a query vertex $u \in V_q$ are given as the set of its children in Q. Let $\mathscr{C} : V_q \to 2^{V_q}$ be the function that maps a query vertex $u \in V_q$ to its constraints $\mathscr{C}(u) \subseteq V_q$. Each edge $(u, u') \in E_q$ defines a matching constraint u' for u. Consequently, $\mathscr{C}$ maps a vertex with zero children to an empty set of constraints.*

As an example, the set of constraints of query graph Q_5 are given as $\{\mathscr{C}(A) = \{B, C\}, \mathscr{C}(B) = \{C\}, \mathscr{C}(C) = \{B\}\}$.

Now, let $G = (V, E, f)$ be a data graph, a data vertex $v \in V$ is matched to a query vertex $u \in V_q$, if and only if, $f(v) = f_q(v)$ and v respects the matching constraints $\mathscr{C}(u)$ of u. We say that constraints of u are also constraints of its candidates such that a

candidate is a data vertex having the same label as u. Therefore, a data vertex can have multiple constraints that should be met by its children, such that each constraint must be met by one or more of its outgoing edges. These edges sharing the same source vertex are grouped together to form a data structure that we call *STwig*, which will be defined next. Moreover, we call *ST*, the set of all *STwigs* extracted from a data graph.

Definition 23 (STwig)**.** *Given a data graph, each vertex in this graph generates an* STwig *structure. Every* STwig *is composed of the set of outgoing edges from this vertex, that we refer to as root. The destination vertex of each edge is considered a child of this* STwig*.*

An *STwig* has exactly one root but it can have zero or more children. An empty *STwig* is an *STwig* that has a root but no children, it represents an isolated data vertex or a data vertex with no children.

The set of *STwigs* related to our data graph G_5 is illustrated in Figure 5.4. *PGSim* processes these *STwigs* in parallel to evaluate graph simulation.

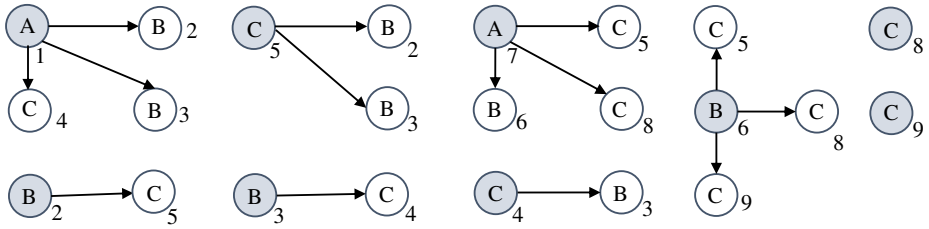


Figure 5.4 The set of *STwigs* extracted from the data graph G_5. The different *STwig* roots are colored in gray

Each *STwig* in the initial *ST*, which is extracted from the data graph, has its own local context. A local context is composed of two types of match sets; a *local match set* and *child match set*.

Definition 24 (Local match set)**.** *Given the* STwig *t, we define a local match set* $M(t)$ *as a set of matches* (u, v, b) *of the root vertex such that each pair* $(u, v) \in V_q \times V$ *is mapped to a Boolean value b indicating whether the matching is correct or not.*

Definition 25 (Child match set)**.** *Given the* STwig *t, the child match set, noted* $M_c(t)$*, is simply the union of the local matches of the children of t. However,* M_c *does not include a match flag b, because we only keep the correct matches.*

An *STwig* with a non empty match set has its root mapped to at least one query vertex. Consequently, an *STwig* that has an empty match set is not a correct match and therefore

can be eliminated from *ST*. Intuitively, an empty *STwig* can only be mapped to query vertices having an empty set of constraints. Hence, its local match set is initialized but never updated afterwards.

Definition 26 (ST)**.** *The set of all* STwigs *of a given data graph are stored in the distributed data structure* ST*. Each* STwig *in* ST *is mapped to a local context composed of a match set M and a child match set M_c.*

ST is constructed off-line from the initial data graph and then processed in parallel. Furthermore, the global match set for graph simulation is defined as follows.

Definition 27 (Global match set)**.** *Given a query graph Q, and* ST*, the distributed set of* STwigs *extracted from data graph G. We compute the union of all the local match sets in* ST *w.r.t. $\mathscr{C}$, the set of constraints of Q. If there is at least one constraint in $\mathscr{C}$ that is not present in this union, the global match set M_g is $\emptyset$. Otherwise, it is given as this union.*

Since we defined the basic concepts and data structures used by *ST*, we are now ready to introduce *PGSim* in the following section.

5.2.2 Parallel graph simulation via PGSim

Our approach for evaluating graph simulation in parallel consists of three points. First, a preliminary phase builds *ST* off-line by simply grouping the data graph edges that share the same source vertex together. Moreover, vertices without outgoing edges result into empty *STwigs*. Next, at the reception of an input query graph Q, we extract the set of constraints $\mathscr{C}$ in a straightforward way, as they are directly retrieved from the edges of Q. Finally, we compute the global match set *w.r.t.* graph simulation based on *ST* and $\mathscr{C}$.

We give Algorithm `PGSim` that takes as input *ST* and $\mathscr{C}$. It is an iterative algorithm that applies a set of transformations on the initial *ST*, resulting each time to a new refined data structure that is closer to the final match graph. Each iteration of *PGSim* transforms a given *STwig* by updating its local match set M in addition to its child match set M_c in parallel. The parallel algorithm executes two categories of steps (iterations); the initialization step and the computation steps. After its convergence, *PGSim* extracts the global match set M_g from the refined *ST*, also in parallel.

Algorithm PGSim

```
1  Input: ST, 𝒞;
2  Output: M_g;
3  I ← ∅ ;
4  foreach STwig t in ST do
5      I_t ← InitializeSTwig(t, 𝒞);      // Initialize the local match set and
         child match set
6      I ← I ∪ I_t;
7  ST ← Refine(ST) ;                    // Filter out STwigs with empty match sets
8  while I ≠ ∅ do
9      I_tmp ← ∅;
10     foreach STwig t in ST do
11         I_t ← ComputeSTwig(t, I) ;   // Evaluate graph simulation locally
12         I_tmp ← I_tmp ∪ I_t;
13     I ← I_tmp;
14     ST ← Refine(ST) ;                // Filter out STwigs with empty match sets
15 M_g ← ExtractM_g(ST, 𝒞) ;            // Extract the global match set
16 return M_g;
```

Initialization step

During the initialization (Algorithm `InitializeSTwig`), we evaluate for each *STwig* t its local match set based on the root label and the different children labels only. Each *STwig* runs a local program in parallel where it initializes the children match set M_c (Call of Procedure `InitMatch` in Lines 3–5) and its local match set M (Call of Procedure `InitMatch` in Line 6). Initially, M is composed of matches based only on the label constraint. Then, t verifies for each match in M (a pair (u, v)) whether the child constraints are respected or not by assigning a true or false value (Call of Procedure `GraphSim` in Line 7). The Boolean flag b attached to each pair (u, v) indicates whether this pair passes the child constraints or not. The matches that do not pass child constraints are collected in I and returned by the parallel algorithm (Lines 6–12). For now, an *STwig* root does not know whether its children satisfy the child constraint themselves or not, so here, I allows us to update the local child match set M_c. After that, the local match set M must be recomputed based on the new M_c. After the initialization, Algorithm *PGSim* reduces the size of ST by invoking Procedure `Refine` (Line 7) that removes from ST all the *STwigs* having an empty local match set.

Procedure EXTRACTM$_g$

```
1 Input: ST, 𝒞;
2 Output: M_g;
3 M_g ← ∅;
4 foreach STwig t in ST do
5 |   M_g ← M_g ∪ M(t) ;     // M_g is initialized by the union of STwigs
  |    match sets
6 foreach u in 𝒞 do
7 |   if u ∉ M_g then
8 |   |   return ∅ ;     // Every single query vertex must appear at least
  |   |    once in M_g
```

Procedure `InitMatch` takes the set of constraints $\mathscr{C}$ and a vertex v and computes the match set of v based only on the label similarity constraint. The initialization program of an *STwig* t invokes this procedure for each local edge to initialize $M(t)$ and $M_c(t)$.

Procedure `GraphSim` takes both $M(t)$ and $M_c(t)$ in addition to the set of constraints $\mathscr{C}$ as input parameters and verifies for each pair (u, v) in $M(t)$ whether there exist children of t that are matched to $\mathscr{C}(u)$. If at least one constraint in $\mathscr{C}(u)$ remains unmatched, (u, v) is mapped to $b = false$, otherwise the match flag b is set to $true$.

In the example of query graph Q_5 and data graph G_5, *PGSim* starts by building the initial ST data structure from G_5 where each vertex in G_5 results into an *STwig*. The initial ST contains the two types of *STwig* described earlier, i.e. seven non empty *STwig* and two empty *STwig*. In the initialization step, each *STwig* computes its local match set M and children match set M_c. Then, it updates M based on M_c which gives the local matches shown in Figure 5.5. The two empty *STwigs* rooted at 8 and 9 are initially mapped to query vertex C during the initialization phase. However, C has a child constraint that requires a vertex to have at least one of its children matched to query vertex B, but both *STwigs* are empty (they do not have any children), hence their initial matches become invalid (they are colored in red) and therefore will be broadcast and added to I, the global set of invalid matches for this first iteration of the parallel algorithm.

Computation steps

A computation step starts if the set of invalid matches I is not empty. I is composed of the pairs (u, v) in every local match set M that did not pass the child constraints (Lines 8–13 in Algorithm `InitializeSTwig`). Consequently, the *STwigs* having a non

Algorithm InitializeSTwig

```
1  Input: t, 𝒞 ;
2  Output: I_t ;
3  M_c ← ∅;
4  foreach v ∈ t.children do
5      M_c ← M_c ∪ InitMatch(𝒞, v) ;        // Initialize the child match set
6  M_r ← InitMatch(𝒞, t.root);              // Initialize the STwig match set
7  M ← GraphSim(𝒞, M_r, M_c) ;          // Evaluate graph sim for this STwig
8  I_t ← ∅;
9  foreach (u, v, b) in M do
10     if b = false then
11         M ← M \ (u, v, b) ;     // Filter out invalid matches ((u,v) that
                                    satisfy only label similarity)
12         I_t ← I_t ∪ (u, v) ;      // Collect the set of removed matches in I_t
13 return I_t;
```

Procedure InitMatch

```
1  Input: 𝒞, v;
2  Output: M;
3  M ← ∅ ;                                  // Initialize M with an empty set
4  foreach u ∈ 𝒞 do
       /* Iterate over the set of vertices in the query graph,
          available also in 𝒞                                          */
5      if f_q(u) = f(v) then
6          M ← M ∪ (u, v) ;     // Add (u, v) to M only if it respects the
                                    label similarity constraint
7  return M;
```

empty match set run Algorithm `ComputeSTwig` in parallel to update their child match sets according to I (Line 3) and reevaluate their local constraints for graph simulation based on the new value of M_c (Call of Procedure `GraphSim` in Line 4). We also return the set of new invalid matches that should be propagated to other *STwigs* in ST (Lines 5–10). If the set of invalid matches I is not empty, another computation step (Call of Algorithm `ComputeSTwig`) is triggered to propagate these removals and update the local match sets accordingly. Afterwards, the new invalid matches I are reevaluated (Lines 9–13 in Algorithm `PGSim`) and the *STwigs* having only invalid matches are filtered out (Call of Procedure Refine in Line 14). If there are new invalid matches, we repeat the computations again until yielding to an empty I. Then, Algorithm `PGSim` converges, and

Procedure GraphSim

```
1  Input: 𝒞, M, M_c;
2  Output: M';
3  M' ← ∅ ;                                   // Initialize the new match set
4  foreach (u, v) ∈ M do
5      b ← true ;                             // Initialize the match flag to true
6      foreach c ∈ 𝒞(u) do
7          if ¬(M_c contains c) then
8              b ← false ;  // Change the match flag to false if v does not
                satisfy 𝒞(u) anymore
9      M' ← M' ∪ (u, v, b) ;                  // Add the updated matching info to M'
10 return M' ;
```

Procedure Refine

```
1  Input: ST;
2  Output: ST;
3  foreach STwig t in ST do
4      if M(t) = ∅ then
5          ST ← ST \ t ;          // Filter out every member of ST having an
            empty match set
6  return ST;
```

the global match set M_g is computed based on the final match set of each remaining *STwig* in ST (Lines 15–16).

Back to the example of query graph Q_5 and data graph G_5. At the end of the first iteration, the two *STwigs* rooted at 8 and 9 have both empty match sets, consequently, they are eliminated during the refinement phase of the ST before the next iteration. Since I is not yet empty, a second iteration of computation starts where each *STwig* in the refined ST updates its child match set based on the invalid matches of I. For example, both *STwigs* rooted at data vertices 6 and 7 are affected by the previous iteration. Actually, the child constraint for the *STwig* rooted at 6 is $\{C\}$, but since the two children $\{8, 9\}$ were filtered out, it eliminates them from its local M_c. Nevertheless, the third child 5 has a valid match, therefore, the local match set remains the same. The same thing applies for the *STwig* rooted at vertex 7 that updates only its local M_c without changing its local match set (see the updated ST in Figure 5.5). At the end of this iteration, I remains empty which announces the end of the parallel algorithm in only two iterations. The global match set is the union of the local match sets for the

Algorithm ComputeSTwig

```
1  Input: t, I ;
2  Output: I_t ;
3  M_c ← M_c \ I ;                          // Filter out invalid matches from M_c
4  M ← GraphSim(𝒞, M, M_c) ;    // Reevaluate graph simulation based on
     the new M_c
5  I_t ← ∅;
6  foreach (u, v, b) in M do
7      if b = false then
           /* GraphSim sets match flag b to false when (u, v) is not a
              correct match anymore                                    */
8          M ← M \ (u, v, b) ;                  // Filter out the invalid match
9          I_t ← I_t ∪ (u, v) ;   // Keep track of the invalidated matches in I_t
10 return I_t;
```

remaining *STwigs* in ST. Moreover, the resulting match graph composed of the remaining *STwigs* in ST, is exactly the same as G_s given in Figure 5.3.

5.2.3 Convergence and Correctness of PGSim

First, we prove the convergence of the proposed algorithm through the following Lemmas.

Lemma 7. *The maximum number of super-steps to run* PGSim *is* $|E|$.

Proof. In the worst case, we have a successive elimination of matches one by one, such that only one match is filtered out in each iteration. A removed match in a given *STwig* is only propagated to its children's *STwigs* and such removal information can propagate over a path from the first removed *STwig* to cross all the edges of G. Therefore, the maximum number of iterations is equal to $|E|$. □ □

Let Δ_O be the maximum out degree of G and Γ be the maximum size of a local match set. We know that $\Gamma \leq l_q$ where $l_q \leq |V_q|$ is the highest label frequency in Q.

Lemma 8. *The time complexity of Algorithm* `InitializeSTwig` *is equal to* $O(\Delta_O \times |V_q|)$.

Proof. The *Initialize* program of a given *STwig* t initializes the match set of every child in t in $O(\Delta_O \times |V_q|)$, then it initializes the match set of its root in $O(|V_q|)$. After that, t evaluates graph simulation in $O(|V_q| \times \Gamma)$. Finally, it takes at most $O(\Gamma)$ to update

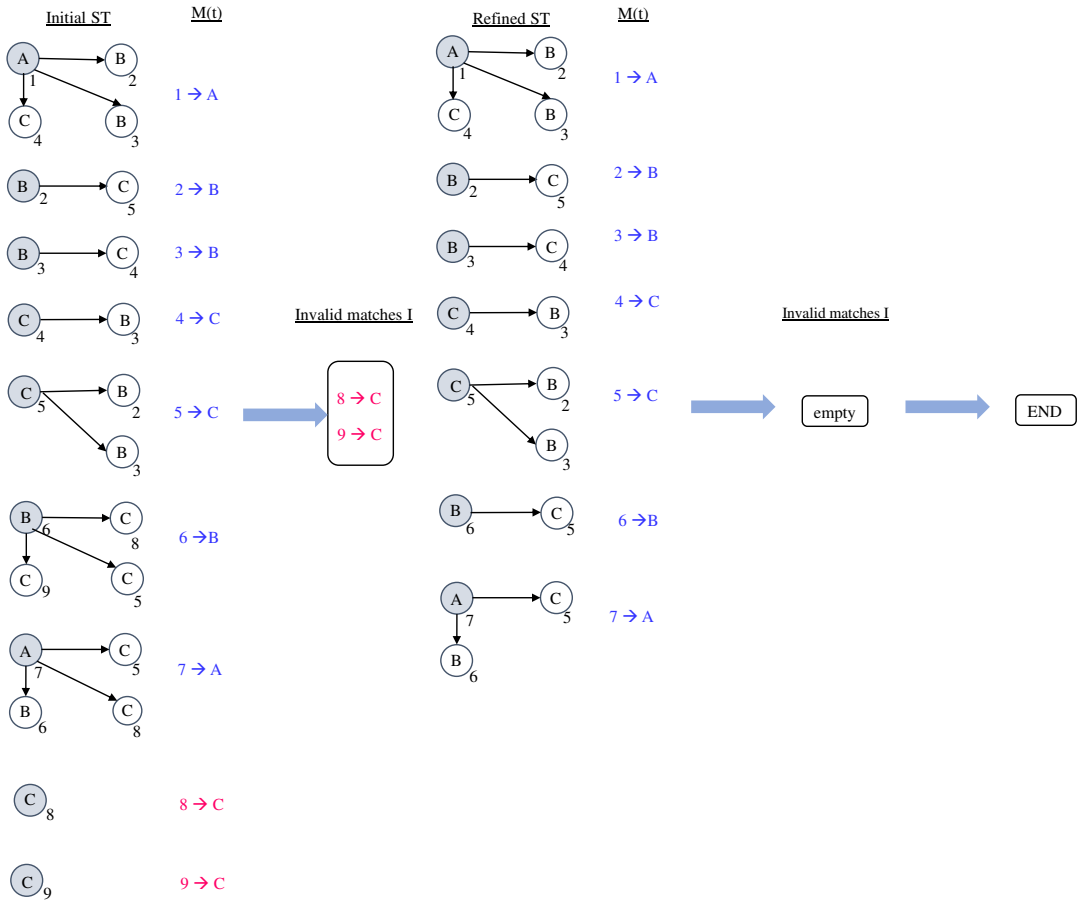


Figure 5.5 A running example of *PGSim* on pattern graph Q_1 and data graph G_1. We illustrate the initial *ST* and the different transformations applied by *PGSim* to get the final match graph *w.r.t.* graph simulation. Values of the flag b of each match in $M(t)$ are illustrated by two colors; blue for true and red for false

M and populate I_t. Therefore, the time complexity of Algorithm `InitializeSTwig` is $O(\Delta_O \times |V_q|)$. □ □

Lemma 9. *The time complexity of Algorithm* `ComputeSTwig` *is equal to* $O(\Gamma \times |V_q|)$.

Proof. The *Update* program of an *STwig* t updates M_c in at most $O(\Gamma)$, then it reevaluates graph simulation in $O(\Gamma \times |V_q|)$. Finally, it updates M and I_t in $O(\Gamma)$. Hence, the time complexity of Algorithm `ComputeSTwig` is $O(\Gamma \times |V_q|)$. □ □

Theorem 9. *Algorithm PGSim for evaluating graph simulation in parallel will terminate with time complexity* $O(|V|/P \times |V_q|^2 \times |E|)$.

Proof. The data structure ST has a maximum length equal to $|V|$. After its construction, it contains exactly $|V|$ *STwigs*. After the first iteration it gets smaller in size because the refine procedure can only remove elements from ST. Algorithm `PGSim` starts by iterating over ST, the set of *STwigs* and updating each *STwig* in parallel. Given P, the number of processors running in parallel, it will take $O(|V|/P \times |V_q| \times \Delta_O)$ to finish the loop. After that, the second loop runs for at most $|E|$ iterations (from Lemma 7) such that in each iteration, it takes $O(|V|/P)$ to refine ST in parallel and $O(|V|/P \times |V_q| \times \Gamma)$ to update the *STwigs* in parallel. The global match set is extracted in $O(|V|/P)$. Therefore, the time complexity of Algorithm `PGSim` in an environment with P processors is $O(|V|/P \times |E| \times |V_q|^2)$. □ □

Theorem 10. *Algorithm PGSim computes the correct and complete match set w.r.t. graph simulation.*

Proof. The correctness of *PGSim* can be verified by the following. (1) the parallel algorithm terminates (Theorem 9), (2) the parallel algorithm computes the correct matches *w.r.t.* graph simulation and (3) the parallel algorithm returns the complete set of matches *w.r.t.* graph simulation.

To prove the second property, we suppose that at the end of the parallel algorithm, there exists a match $(u, v) \in M_g$ not satisfying the constraints of graph simulation for $u \in V_q$. If $(u, v) \in M_g$, this means that at the end of *PGSim*, the *STwig* rooted at v has u in its local match set $M(t)$. If (u, v) does not satisfy graph simulation, then, either u and v do not have the same label or v does not satisfy the set of constraints $\mathscr{C}(u)$. However, the first case is impossible because $M(t)$ is initialized in Procedure `InitMatch` with only query vertices that share the same label. Moreover, if v does not satisfy $\mathscr{C}(u)$, it should have been already filtered out in the initialization program of t where we evaluate graph simulation through Procedure `GraphSim`, or during the following iterations where the matching constraints are reevaluated at every child's update. Furthermore, the only operation performed on the initial match set is removal, thus, $M(t)$ contains correct matches at the end of *PGSim*.

One can verify that M_g contains the complete matches *w.r.t.* graph simulation as follows. Actually, we are sure that any correct match necessarily belongs to the initial match set of a given t that was generated by Procedure `InitMatch` for every single data vertex v in the data graph. Next, we suppose that *PGSim* filters out incorrectly a correct match (u, v) from local $M(t)$. For any *STwig* $t \in ST$, $M(t)$ is only updated inside Procedure `GraphSim` and the only removal of (u, v) made happens when $\mathscr{C}(u)$ are not

satisfied anymore. Therefore, *PGSim* returns the complete match set M_g for graph simulation. □ □

5.3 Parallel edge-centric dual simulation

In this section, we introduce our split-and-combine approach to evaluate dual simulation. We also introduce the parallel algorithm used by this approach, dubbed *PDSim*, and give theoretical guarantees on its correctness. Dual simulation requires the availability of both the child and parent information to decide on the matching of any data vertex. In addition, the existing algorithms for dual simulation (whether centralized or vertex-centric) all process the matching information for incoming and outgoing edges of a data vertex sequentially. However, in the case of high degree vertices, this strategy can lead to a load imbalance. Therefore, to address this problem, we run the computations related to the parent constraints and those related to the child constraints in parallel.

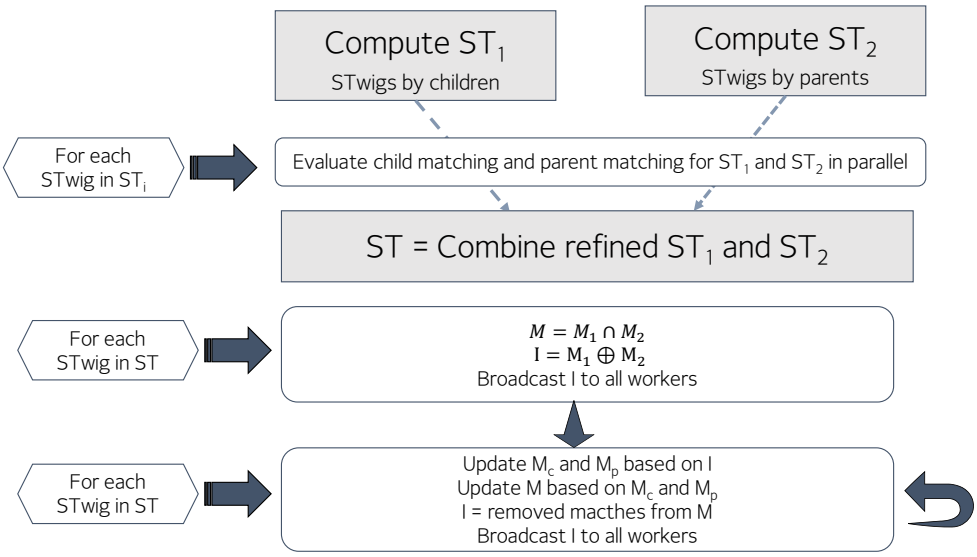


Figure 5.6 The split-and-combine approach for parallel dual simulation

5.3.1 A split-and-combine approach for parallel dual simulation

The split-and-combine approach considers dual simulation as a set of constraints of two types: parent-based and child-based constraints. This separation allows generating two types of *STwigs*, the first type is the same as the *STwig* we have seen in *PGSim*, while

the second type is based on the parents of a data vertex v, i.e. each data vertex results into an *STwig* that groups the edges having v as their destination. In the same way, the empty parent based *STwigs* have only a root and represent vertices with zero parents in the data graph. This phase is referred to as the Split phase. Figure 5.6 illustrates the different steps of computation followed by Algorithm `PDSim` for parallel dual simulation.

Algorithm `PDSim` takes as input the two distributed data structures ST_1 and ST_2, that were constructed before the Split phase (performed only once for each data graph off-line). The first ST contains the *STwigs* based on child relationship and the second one is built based on the parent relationship.

During the Split phase, we execute Algorithm `PGSim` on ST_1 using $\mathscr{C}_1$, the child constraints of the input query (Line 3), while a modified version of *PGSim* that checks the parent constraints $\mathscr{C}_2$ is executed on ST_2 (Line 4). We note that, here *PGSim* returns the refined ST without computing the global match set. At the end of this phase, the two refined data structures will have only edges of the data graph respecting the child constraints and parent constraints separately. However, dual simulation requires that the two types of constraints are met by these edges at the same time, hence, we call Algorithm `Combine` that takes as input ST_1 and ST_2 to find the match set M of each *STwig w.r.t.* dual simulation (Line 5).

Algorithm `Combine` groups the matching information of children M_c and parents M_p of a given *STwig*. It also computes a next local match set M by eliminating matches that do not satisfy dual simulation (Line 3). The match set M of a given *STwig* t is computed as the intersection of the match sets M_1 and M_2 for t in ST_1 and ST_2, respectively (Lines 4–8). Indeed, a data vertex is said to be a match of a given query vertex u *w.r.t.* dual simulation if and only if it satisfies the label constraint and both child constraints $\mathscr{C}_1(u)$ and parent constraints $\mathscr{C}_2(u)$. In addition to that, the filtered out matches are stored in I and returned by the parallel algorithm.

Nevertheless, the evaluation of dual simulation does not stop here. Actually, the same removal process that we have seen in *PGSim* must be triggered after each update made to a local match set M in the combined ST to propagate the removed matches across the other *STwigs*. Therefore, the same routine will be executed such that the set of removed matches I is broadcast at the end of every iteration and dual simulation is reevaluated again (based on both M_c and M_p) for each *STwig* affected by this removal until I becomes empty (Lines 6–13 of Algorithm `PDSim`). Consequently, Algorithm `PDSim` converges with the correct final match graph. The global match set M_g is then computed in the same way as the match set of graph simulation (Lines 14–15).

Algorithm PDSim

```
1  Input: ST_1, ST_2, C_1, C_2;
2  Output: M_g;
3  ST_1 ← PGSim(ST_1, C_1) ;          // Evaluate child-based constraints C_1
4  ST_2 ← PGSim(ST_2, C_2) ;          // Evaluate parent-based constraints C_2
5  (ST, I) ← Combine(ST_1, ST_2, C_1, C_2 ) ;     // For each STwig, combine its
     two match sets
6  ST ← Refine(ST) ; // Filter out STwigs in ST having empty match sets
7  while I ≠ ∅ do
8      I_tmp ← ∅;
9      foreach STwig t in ST do
10         I_t ← ComputeSTwig(t, I) ;        // Dual simulation is evaluated
             instead of graph simulation
11         I_tmp ← I_tmp ∪ I_t;
12     I ← I_tmp;
13     ST ← Refine(ST) ;   // Filter out STwigs in ST having empty match
         sets
14 M_g ← ExtractM_g(ST, C_1 ∪ C_2) ;   // Get the global match set M_g w.r.t.
     dual simulation
15 return M_g;
```

Algorithm Combine

```
1  Input: ST_1, ST_2, C_1, C_2 ;
2  Output: I, ST ;
3  ST ← fullOuterJoin(ST_1, ST_2) ;      // Combine the two STs based on the
     STwig root vertex
4  I ← ∅ ;                          // Initialize the set of invalid matches
5  foreach t in ST do
6      M(t) ← M_1(t) ∩ M_2(t) ;          // Combine the two match sets using
         intersection
7      I ← I ∪ (M_1(t) ⊕ M_2(t)) ; // Invalid matches are the ones appearing
         in only one match set
8  return (ST, I)
```

The Split-and-Combine approach increases the degree of parallelism because first, we process the two distributed data structures ST_1 and ST_2 in parallel, which prunes out invalid matches as early as possible. After that, the remaining computations are also performed in parallel on a much smaller ST.

To illustrate the execution steps of *PDSim*, we use the same data graph G_5 and query graph Q_5. The two graphs G_{s1} of Figure 5.7 and G_{s2} of Figure 5.8 represent ST_1 and ST_2, respectively. Let us take the *STwig* rooted at 6, it has a non-empty local match set $M_1 = \{(B, 6)\}$ in ST_1 but that match set M_2 is empty in ST_2. Intuitively, the local match set of this *STwig*, named M, should be empty after calling Algorithm `Combine` because it does not respect the constraints of dual simulation. Indeed, M is given as the intersection between M_1 and M_2. We should inform the other *STwigs* of the combined ST about the removal of the match $(B, 6)$. The set of removed matches I contains elements that do not appear in both M_1 and M_2, i.e., $M_1 \oplus M_2$. Indeed, the local match set of *STwig* rooted at 6 becomes empty. The remaining iterations of Algorithm `PDSim` eliminates the *STwig* rooted at vertex 7. Next, the *STwigs* rooted at data vertices 5 and 2 will be filtered out one after another. The remaining *STwigs* form the final match graph given in Figure 5.9.

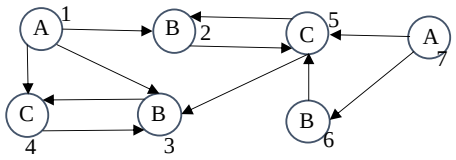


Figure 5.7 G_{s1} resulting from ST_1

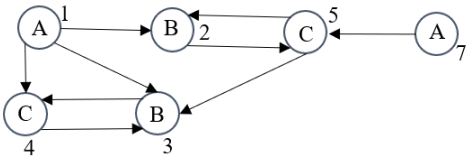


Figure 5.8 G_{s2} resulting from ST_2

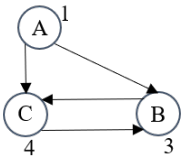


Figure 5.9 Match graph *w.r.t.* dual simulation (G_d)

5.3.2 Convergence and Correctness of PDSim

We give Theorem 11 and Theorem 12 that prove the convergence and correctness of Algorithm `PDSim`, respectively.

Theorem 11. *Algorithm `PDSim` will terminate with a time complexity* $O(|V|/P \times |V_q|^2 \times |E|)$.

Proof. *PDSim* takes the same time complexity to process ST_1 and ST_2, which results into $O(|V|/P \times |E| \times |V_q|^2)$. After that the combine method takes $O(|V|/P)$ to combine ST_1 and ST_2, the remaining steps for dual simulation take at most $O(|V|/P \times |E| \times |V_q|^2)$ to converge (similar to the evaluation of graph simulation). Moreover, the global match set

is extracted in at most $O(V/P)$. Consequently, the time complexity of Algorithm `PDSim` is $O(|V|/P \times |E| \times |V_q|^2)$. □ □

Theorem 12. *Algorithm `PDSim` returns the correct and complete match set w.r.t. dual simulation.*

Proof. The correctness of PDSim is ensured by the following properties. (1) The parallel algorithm will terminate (Theorem 11), (2) the parallel algorithm returns the correct match set of dual simulation, (3) the algorithm returns the complete set of matches of dual simulation.

First, we suppose that at the end of the algorithm, there exists an incorrect match (u, v) in the returned match set M_g. If (u, v) is an incorrect match, then either the two vertices have different labels or v does not satisfy some of the constraints in $\mathscr{C}_1(u)$ or $\mathscr{C}_2(u)$. The first case cannot occur because every local match set is initialized with query vertices having similar labels and M_g is formed by the union of these local matches. Moreover, the local match set M of the *STwig* rooted at v in ST is initialized by the intersection of M_1 (only matches satisfying $\mathscr{C}_1$) and M_2 (only matches satisfying $\mathscr{C}_2$). Moreover, the next iterations of the parallel algorithm always remove the members of M not satisfying dual simulation. Furthermore, the only operations performed on M are removal operations, which ensures that invalid matches cannot be added to M during this stage, hence, the initial supposition is not valid. Therefore, Algorithm `PGSim` returns the correct matches *w.r.t.* dual simulation.

Next, we suppose that there exists a correct match (u, v) such that (u, v) is not part of the returned match set M_g. A match (u, v) is considered correct if and only if $f_q(u) = f(u)$, and v satisfies the child constraints $\mathscr{C}_1(u)$ and parent constraints $\mathscr{C}_2(u)$. Since we proved the correctness of Algorithm `PGSim` in Theorem 10, then a correct match is necessarily part of both $M_1(v)$ and $M_2(v)$. Therefore, if such match exists, then it has been filtered out during the combine phase or removal iterations following it in *PDSim*. However, the only removals made from the local match set $M(v)$ happen after evaluating dual simulation based on the updated child and parent match sets. Hence, Algorithm `PGSim` returns the complete matches *w.r.t.* dual simulation. □ □

5.4 Experimental evaluation

In this section, we evaluate the performance of the proposed parallel edge-centric algorithms and compare them to the vertex-centric one. First, we give details on the

distributed implementation of *ST*, *PGSim* and *PDSim*. Then, we present the data sets used during the different experiments and give their characteristics. Next, we give the cluster configuration used and the environment in which these experiments were carried out. Finally, we present the different sets of experiments and discuss their results.

5.4.1 Distributed implementation of PGSim and PDSim

We implemented the distributed data structure *ST* on top of Apache Spark [151]. Apache Spark is an in-memory data processing framework that offers a distributed computation model based on Resilient Distributed Datasets (RDDs). An RDD is a distributed data structure that can be processed in parallel by applying a transformation on each element of the RDD. The immutability of Spark RDDs guarantees their resilience. If an RDD is lost while the distributed algorithm is running, it will be directly recomputed based on the set of operations that generated it first, hence allowing Spark to be fault-tolerant while avoiding costly I/O operations except for loading the initial RDD. *ST* inherits the properties of Spark RDDs, which makes it a distributed data structure for in-memory processing on a computing cluster. *PGSim* is implemented as a series of successive RDD transformations applied to the initial *ST*. Moreover, we implemented the Split-and-Combine approach of *PDSim* on top of Spark, using RDDs.

5.4.2 Experimental environment

We use four real-world datasets from SNAP Library [74]. Characteristics of these graphs are given in Table 5.1. Unless expressly stated otherwise, the number of distinct labels is fixed to $|\Sigma| = 500$.

Furthermore, the R-Mat model [18] is used to generate synthetic data graphs. The generator takes as an input $|V|$, $|E|$ and $|\Sigma|$ such that $|\Sigma|$ is the number of distinct labels. We fix $|E| = 20 \times |V|$ for all the synthetic graphs generated in these experiments.

Table 5.1 Characteristics of the data graphs used in the different experiments

Dataset (G)	$\lvert V\rvert$	$\lvert E\rvert$
Epinions	75,879	508,837
Amazon0601	403,394	3,387,388
WebGoogle	875,713	5,105,039
LiveJournal	4,847,571	68,993,773
Synthetic	up to 3,504,383	up to 83,886,080

To extract patterns of different sizes, we use the algorithm given in Section 3.5.3 to extract 50 different patterns randomly for each value of $|V_q|$.

We executed our experiments on a Spark cluster composed of 10 nodes with 32GB memory and 16 cores for each; one node plays the role of the master while the remaining nodes are considered as workers.

5.4.3 Experimental results

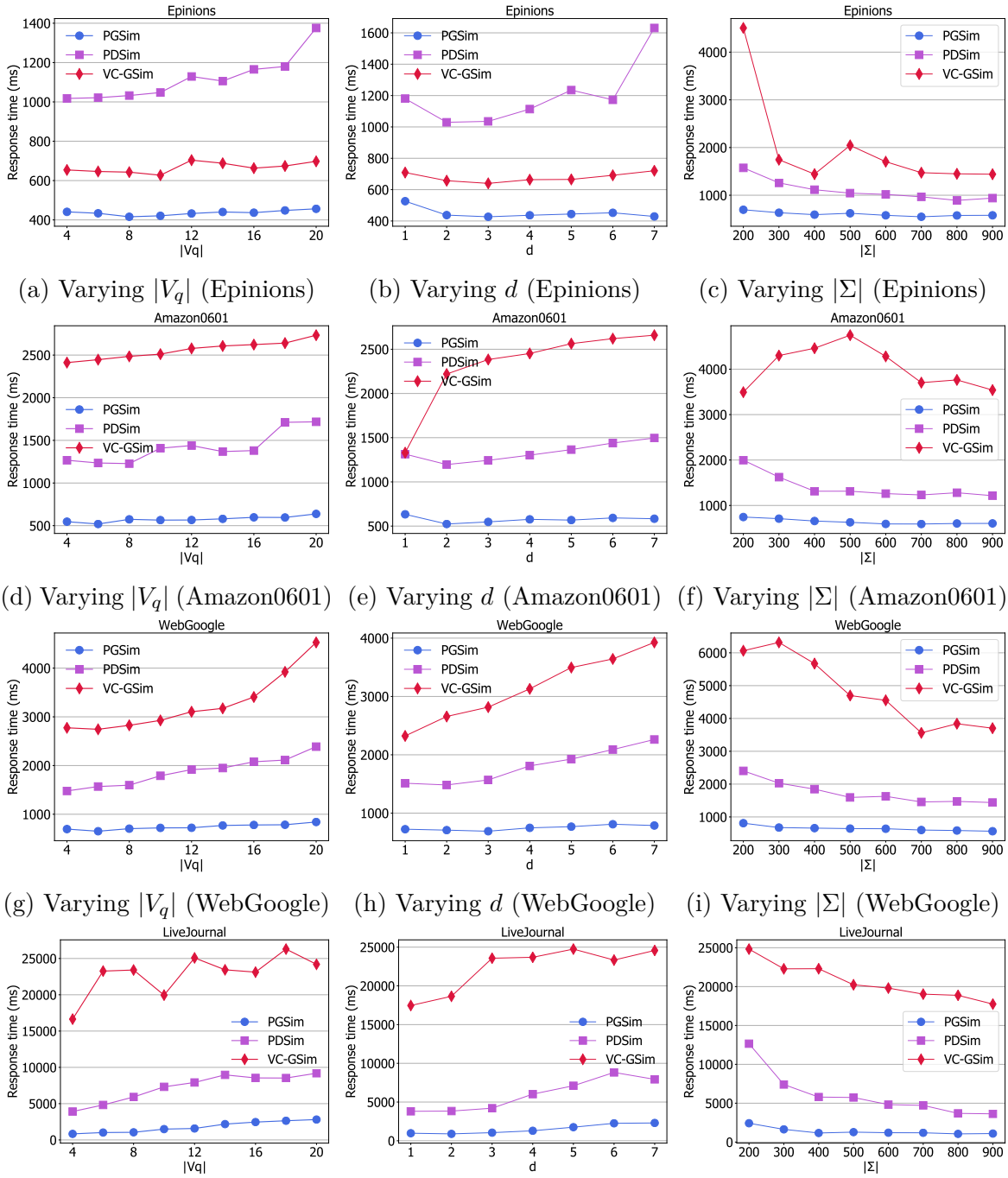
In this section, we present and discuss the results of performance evaluation for *PGSim* and *PDSim* in comparison to the state-of-the-art vertex-centric algorithm (*VC-GSim*) proposed in [39]. We implemented *VC-GSim* on top of GraphX [143], a graph processing system that offers an implementation of Pregel [86] on top of Apache Spark.

Varying the graph parameters

In this set of experiments, we evaluate the impact of three different parameters, $|V_q|$; the size of query graph, d; the diameter of the query graph and $|\Sigma|$; the number of distinct labels in the data graph, on the performance of *PGSim* and *PDSim*. The results are given in Figure 5.10.

First, we discuss the average response time when varying $|V_q|$. As we can see, the obtained results prove the superiority of our approach. *PGSim* is faster than *VC-GSim* for the four real datasets *Epinions*, *Amazon0601*, *WebGoogle* and *LiveJournal*. The improvement in response time achieved becomes more significant when the size of the data graph increases. Indeed, *PGSim* is ten times faster than the vertex-centric version for *LiveJournal*. Moreover, *PGSim* is sensitive to the size of the query graph as it increases in a linear curve with respect to $|V_q|$. On the other hand, the parallel algorithm *PDSim* for evaluating dual simulation behaves slower than *VC-GSim* for *Epinions*, which can be explained by the fact that dual simulation requires more computations to prune out invalid matches compared to graph simulation. However, it takes a shorter time to process queries from the other data sets. Actually, *PDSim* is very scalable when varying the size of query graphs or when varying the size of datasets, for a massive graph like *LiveJournal*, *PDSim*'s response time is closer to that of *PGSim*.

Next, we evaluate the behavior of *PGSim* and *PDSim* compared to *VC-GSim* with respect to the diameter of pattern graphs d, using the same data sets. For small to average graphs, we notice that the response time is not affected by d. However, for *LiveJournal*, the response time increases slightly *w.r.t.* this parameter, since the larger d



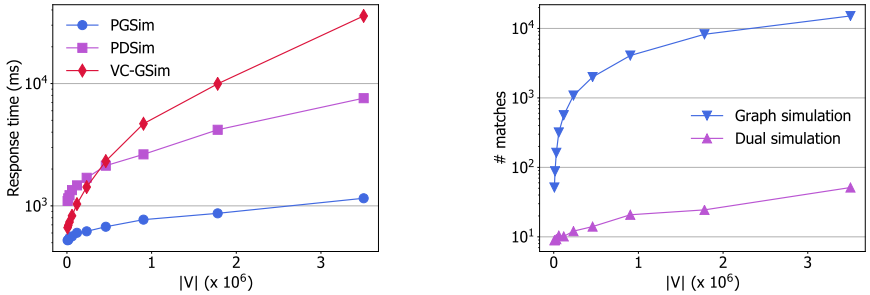
(a) Varying $|V_q|$ (Epinions) (b) Varying d (Epinions) (c) Varying $|\Sigma|$ (Epinions)

(d) Varying $|V_q|$ (Amazon0601) (e) Varying d (Amazon0601) (f) Varying $|\Sigma|$ (Amazon0601)

(g) Varying $|V_q|$ (WebGoogle) (h) Varying d (WebGoogle) (i) Varying $|\Sigma|$ (WebGoogle)

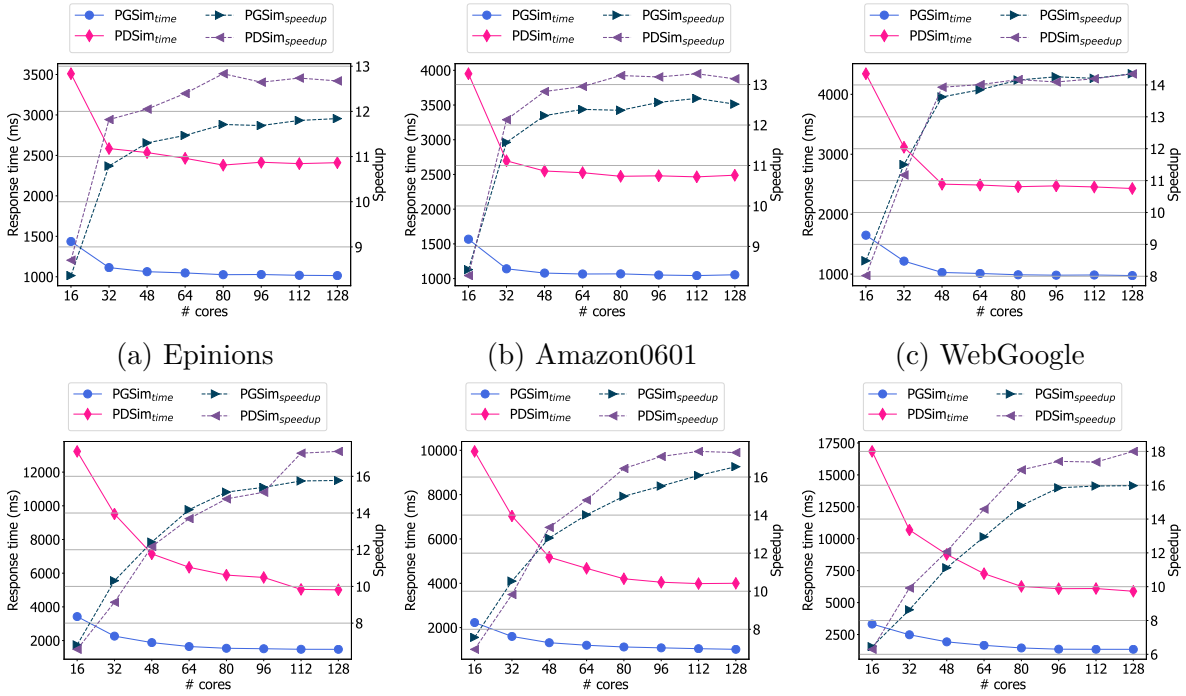
(j) Varying $|V_q|$ (LiveJournal) (k) Varying d (LiveJournal) (l) Varying $|\Sigma|$ (LiveJournal)

Figure 5.10 Performance evaluation of the parallel algorithms *PGSim* and *PDSim* in comparison to the vertex-centric one *VC-GSim* when varying the graph parameters: query graph size $|V_q|$, query graph diameter d and the number of distinct labels $|\Sigma|$



(a) Response time (ms) *w.r.t.* $|V|$ (b) Number of matches *w.r.t.* $|V|$

Figure 5.11 Weak scaling of the parallel algorithms. Note: the Y axis is on log scale



(a) Epinions (b) Amazon0601 (c) WebGoogle

(d) LiveJournal (e) Synthetic graph, scale = 21 (f) Synthetic graph, scale = 22

Figure 5.12 Strong scaling of the parallel algorithms *PGSim* and *PDSim*

gets, the more iterations are required to filter out all the invalid matches. *PGSim* is the least sensitive to the variations in d.

Finally, the number of distinct labels of the data graph $|\Sigma|$ is an important parameter that affects generally the size of the initial candidates set. We vary $|\Sigma|$ from 200 to 900 for the four real-world data sets. Here, the same query graphs are used with $|V_q| = 9$. Moreover, we use the same generated Σ on both data graphs and query graphs for each experiment. We evaluate graph simulation with *PGSim* and *VC-GSim*, dual simulation with *PDSim* and note the average response time. The response time decreases when increasing $|\Sigma|$, this is a normally expected behavior for *PGSim* and *PDSim* as we use a refinement right after the first iteration of the algorithm to prune out all the data vertices not having a label that exists in the query graph. The number of invalid matches increases when the data graph has a large set of distinct labels and inversely. On the other hand, the change in response time for *VC-GSim* is slow and the algorithm takes longer time to evaluate graph simulation in the different data sets.

Weak scaling experiment

In this set of experiments, we evaluate the weak scalability of *PGSim* and *PDSim*. This type of experiments consists of fixing the number of workers/cores of the cluster and increasing the size of the problem, which is the data graph G in our case. We generate synthetic data graphs of different sizes by varying the scale of the R-Mat model from 13 to 22 (resulting to $|V|$ that varies from 2^{13} to 2^{22}). We report the average response time for running graph simulation and dual simulation over 50 instances of queries having the same size ($|V_q| = 9$). The results of comparing *PGSim* and *PDSim* to *VC-GSim* are given in Figure 5.11a.

PGSim outperforms *VC-GSim* by one order of magnitude. We can see how the difference in response time between the two algorithms gets larger when the size of the data graph increases. For the first data graph having only $7.7k$ vertices, the two algorithms are very close with 600 *ms* while *PDSim* evaluates dual simulation for the same data graph in one second. Nevertheless, when increasing the size of data graphs, *PGSim* takes only 1.4 seconds to process the largest synthetic graph containing $3.5M$ vertices and $83.9M$ edges, while *VC-GSim* takes 42.6 seconds to get the same result.

Moreover, we can see in Figure 5.11b, the difference between the number of matches returned by graph simulation and dual simulation. *PDSim* prunes out a big part of the matches returned by graph simulation, which happens during the combine phase along with the remaining iterations that allow the parallel algorithm to converge to the correct

match set. Even with these additional computations performed by *PDSim*, our approach for evaluating dual simulation outperforms significantly *VC-GSim* when increasing $|V|$. Indeed, it only takes 7.5 seconds to process the largest synthetic graph. Consequently, this set of experiments proves the scalability of our parallel algorithms.

Strong scaling experiment

We test the strong scalability of *PGSim* and *PDSim* by fixing the problem size (fixing the data graph) and varying the number of cores in the cluster. We report the average response time and speedup when evaluating graph simulation and dual simulation, respectively, on 50 instances of query graphs having the same size ($|V_q| = 9$). We run this set of experiments on the four real-world data sets in addition to synthetic graphs of different size. Figure 5.12 presents the obtained results.

PGSim scales very well with the number of cores. Indeed, the algorithm speedup increases linearly when increasing the number of cores for all the data graphs. We notice that the larger the data graph size, the higher speedups can be achieved. Indeed, the highest speedups of 16 and 18 are achieved by *PGSim* and *PDSim*, respectively, on the synthetic graph having the largest size ($|V| = 3.5M$ and $|E| = 83.9M$). Nevertheless, for remaining data graphs, the speedup curve starts flattening after some point due to reaching a maximum level of parallelism. For example, for the three data sets Epinions (Figure 5.12a), Amazon0601 (Figure 5.12a) and WebGoogle (Figure 5.12a), the speedup of the two algorithms increases very fast when adding up 16 to 32 cores. After that, the speedup increases very slowly even when doubling the number of cores used.

The different experiments presented in this section prove the strong scalability of the distributed data structure *ST* and the two parallel algorithms *PGSim* and *PDSim*.

5.5 Chapter summary

In this chapter, we proposed *PGSim*, an efficient parallel edge-centric approach for evaluating graph simulation on distributed graphs. *PGSim* relies on the distributed data structure *ST* that groups the edges of the data graph and allows reaching higher degrees of parallelism while avoiding locality issues generally present in the vertex-centric graph algorithms. Moreover, we proposed *PDSim*, a split-and-combine approach for evaluating dual simulation based on *ST* and *PGSim*. We provided theoretical guarantees on the correctness and the convergence of *PGSim* and *PDSim*. Moreover, the experimental results proved that the two propositions outperform the vertex-centric graph simulation

by one order of magnitude. Therefore, we can use *PDSim* directly in evaluating BDSim, strong simulation and strict simulation which will guarantee a noticeable improvement in their response time.

Conclusions and Perspectives

Conclusion

In this thesis, we addressed the problem of scalability of relaxed GPM models. Graph pattern matching is an important problem in graph theory and it has been extensively studied for the past 50 years with applications almost everywhere. This problem has been widely used for biological identification through fingerprint recognition, retina recognition and face recognition. In addition, intrusion detection is an important application domain where malicious behaviours of programs that are modeled as a pattern are queried against the program dependence graph. Moreover, GPM is being highly used nowadays in social networks for recommendation, expert finding and social community detection.

As a first part of this manuscript, we studied the different models proposed for formalizing this problem, from subgraph isomorphism, the most stringent model to the most flexible ones such as graph simulation and bounded simulation that belong to the category of relaxed GPM. All these GPM models differ both in terms of flexibility and tractability, which makes them more suitable for specific application domains than other ones. In the second part, we investigated the use of TLAV programming models and systems in scaling up the existing GPM approaches. Our main contributions are summarized as follows.

First, we proposed a new taxonomy of distributed GPM approaches based on the GPM model and programming paradigm applied. Among the drawn up conclusions, we suggest that the vertex-centric and subgraph-centric programming models are best suited for evaluating GPM models that impose a locality property, since the neighborhood of a single vertex is required in evaluating its matching information. Additionally, strong simulation is considered among the top GPM models that strike a balance between flexibility and tractability, which makes it suitable for a wide range of applications.

The second contribution of this thesis is BDSim, a new relaxed GPM model that captures better the topological structure of the query graph while being feasible in cubic time. We showed that BDSim is more stringent than dual simulation allowing it to return accurate answers. It is more flexible than subgraph isomorphism, which makes it scalable to massive data graphs. Our model preserves the proximity between query vertices by eliminating cycles of unbounded length from the resulting match graph. Besides, the experimental evaluation of the proposed vertex-centric algorithms for evaluating BDSim allowed us to validate the effectiveness and efficiency of this approach.

As a third contribution, we designed the first fully distributed vertex-centric algorithms for a scalable evaluation of strong simulation and strict simulation in massive graphs (named *D3S* and *D3S*$^+$, respectively). Our approach ensures that the locality ball members perform strong simulation in a distributed way, hence avoiding duplication of the graph data while increasing the algorithm scalability. Moreover, even though our approach still requires exchanging the matching information between the data graph vertices, *D3S* outperforms prior work significantly, notably in applications where the vertex similarity depends on multiple attributes. Therefore, unlike previous work that would ship and duplicate all the vertex information, *D3S* will only require shipping the matching information. Consequently, *D3S* allows strong simulation to support further extensions on the semantic similarity adopted. Based on the same approach, we designed *D3S*$^+$ as an extension of *D3S* which inherits its different properties for the evaluation of strict simulation.

Finally, we designed and implemented the first parallel edge-centric approach for evaluating graph simulation and dual simulation. We proposed the distributed data structure *ST* that ensures a fine-grain parallelism. *PGSim*, which is our parallel algorithm for evaluating graph simulation, relies on *ST* that allows it to reach linear scalability and handle the problem of skewed degree distribution present in real-world graphs. Moreover, the vertex-centric programming algorithms would incur heavy message passing even for vertices residing on the same machine, which slows down the processing of a distributed graph and results in many rounds of computation. In contrast, *PGSim* uses shared memory abstraction to propagate updates along the different rounds of the algorithm. Furthermore, the parallel algorithm *PDSim* uses a split-and-combine approach based on *ST* and *PGSim* to increase the degree of parallelism when evaluating dual simulation. This approach is faster even than the vertex-centric graph simulation. *PDSim* can be then used as a building block in the evaluation of BDSim and strong simulation.

Some of the conclusions of this research work are the following. The new adaptive GPM model BDSim strikes a balance between tractability and flexibility while offering guarantees on the preservation of connectivity, duality, closeness and locality of the input pattern graph. Thanks to its configurable parameter "k", BDSim can be used for different application needs depending on the required level of flexibility. Furthermore, we confirmed in this thesis that among the TLAV programming models, no one can be considered the best for designing distributed GPM algorithms for all existing GPM models. As shown by the conducted experiments, our edge-based algorithm for evaluating graph simulation outperforms the vertex-centric version by up to one order of magnitude, hence validating our findings about the importance of all the TLAV paradigms. On the other side, many studies have shown that subgraph isomorphism and strong simulation algorithms are better designed with a subgraph-centric programming model. In fact, the model itself imposes a locality constraint which requires a data vertex to have access to a multi-level neighborhood in order to compute its matching information. Moreover, we proved, through the proposal of a fully distributed vertex-centric algorithm of strong simulation, that the Pregel-based vertex-centric paradigm also gives interesting results with strong simulation when employed properly.

Future directions

Among the perspectives of this thesis, we suggest addressing the following three points.

First, we intend to work on the proposal of parallel algorithms for relaxed GPM in attributed and weighted graphs. Indeed, these graphs carry more information that is currently present in real-world graphs and combining all the available vertex and edge attributes in the matching process guarantees a better reliability in the returned results.

In another research direction, we plan to address another important challenge; relaxed GPM in the context of highly dynamic graphs. We want to answer the following questions regarding the contributions made in this thesis. *"In BDSim, will the proposed vertex-centric algorithms remain efficient in highly dynamic graphs?"*. The next question is *"How good can D3S and D3S$^+$ perform on frequent graph updates?"*. A last important question that is worth answering is *"Whether or not the distributed data structure ST will maintain the same performance when update events arrive at a high frequency"*.

Furthermore, learning graph matching has been the subject of several recent studies whether it is addressing subgraph matching and graph matching directly as in [92, 81, 72, 56] or solving problems that are useful in graph pattern matching such as finding

the maximum common subgraph between two input graphs as in [6]. However, these works only focused on subgraph isomorphism that can be rigid for many applications. Therefore, a challenging task in learning subgraph matching is to learn a relaxed model of subgraph matching that will be based on the semantics of the available data. With these considerations, different flexibility levels will be permitted depending on the query purpose and the application domain with its own definition of similarity between graphs.

Bibliography

[1] Abuhaiba, I. S. (2007). Offline signature verification using graph matching. *Turkish Journal of Electrical Engineering & Computer Sciences*, 15(1):89–104.

[2] Acosta-Mendoza, N., Gago-Alonso, A., and Medina-Pagola, J. E. (2012). Frequent approximate subgraphs as features for graph-based image classification. *Knowledge-Based Systems*, 27:381–392.

[3] Afrati, F. N., Fotakis, D., and Ullman, J. D. (2013). Enumerating subgraph instances using map-reduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 62–73, Brisbane, QLD. IEEE.

[4] Ammar, K., McSherry, F., Salihoglu, S., and Joglekar, M. (2018). Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows. *Proc. VLDB Endow.*, 11(6):691–704.

[5] Anwar, A. and Mahmood, A. N. (2016). Anomaly detection in electric network database of smart grid: Graph matching approach. *Electric Power Systems Research*, 133:51–62.

[6] Bai, Y., Xu, D., Wang, A., Gu, K., Wu, X., Marinovic, A., Ro, C., Sun, Y., and Wang, W. (2020). Fast detection of maximum common subgraph via deep q-learning. *arXiv preprint arXiv:2002.03129*.

[7] Bhattarai, B., Liu, H., and Huang, H. H. (2019). Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1447–1462, Amsterdam, Netherlands. ACM.

[8] Bi, F., Chang, L., Lin, X., Qin, L., and Zhang, W. (2016). Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1199–1214, San Francisco, California, USA. ACM.

[9] Bodaghi, A. and Teimourpour, B. (2018). Automobile insurance fraud detection using social network analysis. In *Applications of Data Management and Analysis*, pages 11–16. Springer.

[10] Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., and Ferro, A. (2013). A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14(S7):S13.

[11] Borthakur, D. (2007). The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21.

[12] Bröcheler, M., Pugliese, A., and Subrahmanian, V. S. (2010). Cosi: Cloud oriented subgraph identification in massive social networks. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 248–255, Odense, Denmark. IEEE, IEEE.

[13] Brynielsson, J., Högberg, J., Kaati, L., Mårtenson, C., and Svenson, P. (2010). Detecting social positions using simulation. In *2010 International Conference on Advances in Social Networks Analysis and Mining*, pages 48–55. IEEE.

[14] Carletti, V., Foggia, P., Greco, A., Saggese, A., and Vento, M. (2018). Comparing performance of graph matching algorithms on huge graphs. *Pattern Recognition Letters*, 134:58–67.

[15] Carletti, V., Foggia, P., Greco, A., Vento, M., and Vigilante, V. (2019). Vf3-light: A lightweight subgraph isomorphism algorithm and its experimental evaluation. *Pattern Recognition Letters*, 125:591–596.

[16] Carletti, V., Foggia, P., Saggese, A., and Vento, M. (2017). Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with vf3. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):804–818.

[17] Carletti, V., Foggia, P., and Vento, M. (2015). Vf2 plus: An improved version of vf2 for biological graphs. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 168–177, Beijing, China. Springer.

[18] Chakrabarti, D., Zhan, Y., and Faloutsos, C. (2004). R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM.

[19] Chaudhuri, B., Demir, B., Bruzzone, L., and Chaudhuri, S. (2016). Region-based retrieval of remote sensing images using an unsupervised graph-theoretic approach. *IEEE Geoscience and Remote Sensing Letters*, 13(7):987–991.

[20] Chen, H., Liu, M., Zhao, Y., Yan, X., Yan, D., and Cheng, J. (2018). G-miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, Porto, Portugal. ACM.

[21] Chen, J., Gu, Y., Wang, Q., Li, C., and Yu, G. (2020). Partition-oriented subgraph matching on gpu. In Wang, X., Zhang, R., Lee, Y.-K., Sun, L., and Moon, Y.-S., editors, *Web and Big Data*, pages 53–68, Cham. Springer International Publishing.

[22] Chikkerur, S., Cartwright, A. N., and Govindaraju, V. (2006). K-plet and coupled bfs: a graph based fingerprint representation and matching algorithm. In *International Conference on Biometrics*, pages 309–315. Springer.

[23] Conte, D., Foggia, P., Sansone, C., and Vento, M. (2004). Thirty Years of Graph Matching in Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(03):265–298.

[24] Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. (2001). An improved algorithm for matching large graphs. *Proceedings of the 3rd IAPR Workshop on Graph-Based Representations in Pattern Recognition*, 219(2):149–159.

[25] Csun, S. and Luo, Q. (2018). Parallelizing recursive backtracking based subgraph matching on a single machine. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–9, Singapore, Singapore. IEEE.

[26] Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

[27] Dias, V., Teixeira, C. H., Guedes, D., Meira, W., and Parthasarathy, S. (2019). Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1357–1374, Amsterdam Netherlands. ACM.

[28] Diestel, R. (2005). Graph theory. 2005. *Grad. Texts in Math*, 101.

[29] Du, R., Yang, J., Cao, Y., and Wang, H. (2018). Personalized graph pattern matching via limited simulation. *Knowledge-Based Systems*, 141:31–43.

[30] Duchenne, O., Joulin, A., and Ponce, J. (2011). A graph-matching kernel for object categorization. In *2011 International Conference on Computer Vision*, pages 1792–1799. IEEE.

[31] Dustin, W. S. (2019). Social media statistics 2020: Top networks by the numbers. https://dustinstout.com/social-media-statistics/. Accessed: 2020-03-01.

[32] Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., and Wu, Y. (2010). Graph Pattern Matching: From Intractable to Polynomial Time. *Proceedings of the VLDB Endowment*, 3(1-2):264–275.

[33] Fan, W., Wang, X., and Wu, Y. (2013a). Expfinder: Finding experts by graph pattern matching. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1316–1319. IEEE.

[34] Fan, W., Wang, X., and Wu, Y. (2013b). Incremental graph pattern matching. *ACM Trans. Database Syst.*, 38(3).

[35] Fan, W., Wang, X., and Wu, Y. (2014a). Querying big graphs within bounded resources. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 301–312.

[36] Fan, W., Wang, X., Wu, Y., and Deng, D. (2014b). Distributed graph simulation: Impossibility and possibility. *Proceedings of the VLDB Endowment*, 7(12):1083–1094.

[37] Fan, W., Yu, W., Xu, J., Zhou, J., Luo, X., Yin, Q., Lu, P., Cao, Y., and Xu, R. (2018). Parallelizing sequential graph computations. *ACM Transactions on Database Systems (TODS)*, 43(4):1–39.

[38] Fard, A., Nisar, M. U., Miller, J. A., and Ramaswamy, L. (2014). Distributed and Scalable Graph Pattern Matching: Models and Algorithms. *International Journal of Big Data (ISSN 2326-442X)*, 1(1):1–14.

[39] Fard, A., Nisar, M. U., Ramaswamy, L., Miller, J. A., and Saltz, M. (2013). A distributed vertex-centric approach for pattern matching in massive graphs. In *2013 IEEE International Conference on Big Data*, pages 403–411, Santa Clara, CA, USA. IEEE.

[40] Filip, S. M. (2014). *A scalable graph pattern matching engine on top of Apache Giraph*. PhD thesis, VU University Amsterdam.

[41] Fischer, A., Suen, C. Y., Frinken, V., Riesen, K., and Bunke, H. (2013). A fast matching algorithm for graph-based handwriting recognition. In *International Workshop on Graph-Based Representations in Pattern Recognition*, pages 194–203. Springer.

[42] Flake, G. W., Lawrence, S., Giles, C. L., and Coetzee, F. M. (2002). Self-organization and identification of web communities. *Computer*, 35(3):66–70.

[43] Foggia, P., Percannella, G., and Vento, M. (2014). Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(01):1450001.

[44] Foundation, T. A. S. (2011). Apache giraph. https://giraph.apache.org/. Accessed: 2021-01-20.

[45] Fuchs, P., Boncz, P., and Ghit, B. (2020). Edgeframe: Worst-case optimal joins for graph-pattern matching in spark. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–11, Portland OR USA. ACM.

[46] Gallagher, B. (2006). Matching Structure and Semantics : A Survey on Graph-Based Pattern Matching. *Technical Report FS-06-02*, 6:45–53.

[47] Gao, J., Liu, P., Kang, X., Zhang, L., and Wang, J. (2016). Prs: parallel relaxation simulation for massive graphs. *The Computer Journal*, 59(6):848–860.

[48] Gao, J., Zhou, C., Zhou, J., and Yu, J. X. (2014). Continuous pattern detection over billion-edge graph using distributed framework. In *2014 IEEE 30th International Conference on Data Engineering*, pages 556–567, Chicago, IL, USA. IEEE.

[49] Garey, M. R. and Johnson, D. S. (1979). Computers and intractability: a guide to np-completeness.

[50] Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, Bolton Landing, NY, USA. Association for Computing Machinery.

[51] Gurajada, S., Seufert, S., Miliaraki, I., and Theobald, M. (2014). Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 289–300, Utah USA. ACM.

[52] Han, M., Kim, H., Gu, G., Park, K., and Han, W.-S. (2019). Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1429–1446, Amsterdam Netherlands. ACM.

[53] Han, W.-S., Lee, J., and Lee, J.-H. (2013). Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 337–348, New York, New York, USA. Association for Computing Machinery.

[54] Harish, P. and Narayanan, P. J. (2007). Accelerating large graph algorithms on the gpu using cuda. In *International conference on high-performance computing*, pages 197–208. Springer.

[55] He, H. and Singh, A. K. (2008). Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 405–418, Vancouver, Canada. Association for Computing Machinery.

[56] He, J., Huang, Z., Wang, N., and Zhang, Z. (2021). Learnable graph matching: Incorporating graph partitioning with deep feature learning for multiple object tracking. *arXiv preprint arXiv:2103.16178*.

[57] Henzinger, M. R., Henzinger, T. A., and Kopke, P. W. (1995). Computing simulations on finite and infinite graphs. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 453–462, USA. IEEE.

[58] Hung, B. W. and Jayasumana, A. P. (2016). Investigative simulation: Towards utilizing graph pattern matching for investigative search. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 825–832. IEEE.

[59] Hunger, M. (2014). *Neo4j 2.0: Eine Graphdatenbank für alle.* entwickler. Press, Eltville, Germany.

[60] Jüttner, A. and Madarasi, P. (2018). Vf2++-an improved subgraph isomorphism algorithm. *Discrete Applied Mathematics*, 242:69–81.

[61] Kalavri, V., Vlassov, V., and Haridi, S. (2017). High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering*, 30(2):305–324.

[62] Kao, J.-S. and Chou, J. (2016). Distributed Incremental Pattern Matching on Streaming Graphs. In *Proceedings of the ACM Workshop on High Performance Graph Processing - HPGP '16*, volume 1828, pages 43–50, New York, New York, USA. ACM Press.

[63] Katz, G. J. and Kider, J. T. (2008). All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, pages 47–55.

[64] Khan, S., Nawaz, M., Guoxia, X., and Yan, H. (2019). Image correspondence with cur decomposition-based graph completion and matching. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(9):3054–3067.

[65] Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P. (2013). Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, Prague, Czech Republic. Association for Computing Machinery.

[66] Kim, K., Seo, I., Han, W.-S., Lee, J.-H., Hong, S., Chafi, H., Shin, H., and Jeong, G. (2018). Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data*, pages 411–426, Houston TX USA. ACM.

[67] Kumar, S. and Spafford, E. H. (1994). A pattern matching model for misuse intrusion detection.

[68] Lai, L., Qin, L., Lin, X., and Chang, L. (2015). Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment*, 8(10):974–985.

[69] Lai, L., Qin, L., Lin, X., Zhang, Y., Chang, L., and Yang, S. (2016). Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228.

[70] Lai, L., Qing, Z., Yang, Z., Jin, X., Lai, Z., Wang, R., Hao, K., Lin, X., Qin, L., Zhang, W., et al. (2019). Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment*, 12(10):1099–1112.

[71] Lajevardi, S. M., Arakala, A., Davis, S. A., and Horadam, K. J. (2013). Retina verification system based on biometric graph matching. *IEEE transactions on image processing*, 22(9):3625–3635.

[72] Lan, Z., Yu, L., Yuan, L., Wu, Z., and Ma, F. (2021). Sub-gmn: The subgraph matching network model. *arXiv preprint arXiv:2104.00186*.

[73] Lee, J., Han, W.-S., Kasperovics, R., and Lee, J.-H. (2012). An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment*, 6(2):133–144.

[74] Leskovec, J. and Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection.

[75] Levchuk, G., Colonna-Romano, J., and Eslami, M. (2017). Application of graph-based semi-supervised learning for development of cyber cop and network intrusion detection. In *Disruptive Technologies in Sensors and Sensor Systems*, volume 10206, page 102060D. International Society for Optics and Photonics.

[76] Li, J., Cao, Y., and Ma, S. (2017). Relaxing graph pattern matching with explanations. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 1677–1686, Singapore Singapore. ACM.

[77] Li, J., Li, J., and Wang, X. (2018). A vertex-centric graph simulation algorithm for large graphs. In Xu, Z., Gao, X., Miao, Q., Zhang, Y., and Bu, J., editors, *Big Data*, pages 238–254, Singapore. Springer.

[78] Lin, W., Xiao, X., Xie, X., and Li, X.-L. (2016). Network motif discovery: A gpu approach. *IEEE transactions on knowledge and data engineering*, 29(3):513–528.

[79] Liu, C., Chen, C., Han, J., and Yu, P. S. (2006). Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, New York, NY, USA. Association for Computing Machinery.

[80] Lladós, J. and Sanchez, G. (2004). Graph matching versus graph parsing in graphics recognition—a combined approach. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(03):455–473.

[81] Lou, Z., You, J., Wen, C., Canedo, A., Leskovec, J., et al. (2020). Neural subgraph matching. *arXiv preprint arXiv:2007.03092*.

[82] Lu, S. W., Ren, Y., and Suen, C. Y. (1991). Hierarchical attributed graph representation and recognition of handwritten chinese characters. *Pattern Recognition*, 24(7):617–632.

[83] Luo, L., Wong, M., and Hwu, W.-m. (2010). An effective gpu implementation of breadth-first search. In *Design Automation Conference*, pages 52–55. IEEE.

[84] Ma, S., Cao, Y., Fan, W., Huai, J., and Wo, T. (2011). Capturing topology in graph pattern matching. *Proceedings of the VLDB Endowment*, 5(4):310–321.

[85] Ma, S., Cao, Y., Huai, J., and Wo, T. (2012). Distributed graph pattern matching. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 949–958, Lyon, France. Association for Computing Machinery.

[86] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, Indianapolis, Indiana, USA. Association for Computing Machinery.

[87] Mekouar, S., Zrira, N., and Bouyakhf, E. H. (2018). Community outlier detection in social networks based on graph matching. *International Journal of Autonomous and Adaptive Communications Systems*, 11(3):209–231.

[88] Menelet, A. and Bichot, C.-E. (2021). Characterization of android malware based on subgraph isomorphism.

[89] Mhedhbi, A. and Salihoglu, S. (2019). Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.*, 12(11):1692–1704.

[90] Newman, M. E. (2004). Detecting community structure in networks. *The European physical journal B*, 38(2):321–330.

[91] Ngo, H. Q., Ré, C., and Rudra, A. (2014). Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16.

[92] Nie, W., Ding, H., Liu, A., Deng, Z., and Su, Y. (2020). Subgraph learning for graph matching. *Pattern Recognition Letters*, 130:362–369.

[93] Ogaard, K., Roy, H., Kase, S., Nagi, R., Sambhoos, K., and Sudit, M. (2013). Discovering patterns in social networks with graph matching algorithms. In Greenberg, A. M., Kennedy, W. G., and Bos, N. D., editors, *Social Computing, Behavioral-Cultural Modeling and Prediction*, pages 341–349, Berlin, Heidelberg. Springer.

[94] Pearce, R. (2012). Highly asynchronous visitor queue graph toolkit.

[95] Pei, W.-Y., Yang, C., Meng, L.-Y., Hou, J.-B., Tian, S., and Yin, X.-C. (2018). Scene video text tracking with graph matching. *IEEE Access*, 6:19419–19426.

[96] Peng, P., Zou, L., Özsu, M. T., Chen, L., and Zhao, D. (2016). Processing sparql queries over distributed rdf graphs. *The VLDB Journal*, 25(2):243–268.

[97] Plantenga, T. (2013). Inexact subgraph isomorphism in mapreduce. *Journal of Parallel and Distributed Computing*, 73(2):164–175.

[98] Qiao, M., Zhang, H., and Cheng, H. (2017). Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment*, 11(2):176–188.

[99] Qiu, X., Cen, W., Qian, Z., Peng, Y., Zhang, Y., Lin, X., and Zhou, J. (2018). Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888.

[100] Ren, X. and Wang, J. (2015). Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment*, 8(5):617–628.

[101] Ren, X. and Wang, J. (2016). Multi-query optimization for subgraph isomorphism search. *Proceedings of the VLDB Endowment*, 10(3):121–132.

[102] Reza, T., Klymko, C., Ripeanu, M., Sanders, G., and Pearce, R. (2017). Towards practical and robust labeled pattern matching in trillion-edge graphs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, Honolulu, HI. IEEE.

[103] Reza, T., Ripeanu, M., Sanders, G., and Pearce, R. (2020). Approximate pattern matching in massive graphs with precision and recall guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1115–1131, Portland OR USA. ACM.

[104] Reza, T., Ripeanu, M., Tripoul, N., Sanders, G., and Pearce, R. (2018). Prunejuice: Pruning trillion-edge graphs to a precise pattern-matching solution. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 265–281, Dallas, Texas, USA. IEEE.

[105] Rostrup, S., Srivastava, S., and Singhal, K. (2013). Fast and memory-efficient minimum spanning tree on the gpu. *International Journal of Computational Science and Engineering*, 8(1):21–33.

[106] Roy, A., Mihailovic, I., and Zwaenepoel, W. (2013). X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, Farminton, Pennsylvania. Association for Computing Machinery.

[107] Sadowski, G. and Rathle, P. (2014). Fraud detection: Discovering connections with graph databases. *White Paper-Neo Technology-Graphs are Everywhere*, 13.

[108] Salihoglu, S. and Widom, J. (2013). Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 1–12, Baltimore, Maryland, USA. Association for Computing Machinery.

[109] Schätzle, A., Przyjaciel-Zablocki, M., Berberich, T., and Lausen, G. (2016). S2x: Graph-parallel querying of rdf with graphx. In Wang, F., Luo, G., Weng, C., Khan, A., Mitra, P., and Yu, C., editors, *Biomedical Data Management and Graph Online Querying*, pages 155–168, Cham. Springer International Publishing.

[110] Serafini, M., De Francisci Morales, G., and Siganos, G. (2017). Qfrag: Distributed graph search via subgraph isomorphism. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 214–228, Santa Clara, CA. ACM.

[111] Shang, H., Zhang, Y., Lin, X., and Yu, J. X. (2008). Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment*, 1(1):364–375.

[112] Shao, B., Wang, H., and Li, Y. (2013). Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, New York, USA. Association for Computing Machinery.

[113] Shao, Y., Cui, B., Chen, L., Ma, L., Yao, J., and Xu, N. (2014). Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 625–636, Utah USA. ACM.

[114] Shemshadi, A., Sheng, Q. Z., and Qin, Y. (2016). Efficient pattern matching for graphs with multi-labeled nodes. *Knowledge-Based Systems*, 109:256–265.

[115] Shi, Q., Liu, G., Zheng, K., Liu, A., Li, Z., Zhao, L., and Zhou, X. (2017). Multi-Constrained Top-K Graph Pattern Matching in Contextual Social Graphs. *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017*, 6(1):588–595.

[116] Shi, X., Zheng, Z., Zhou, Y., Jin, H., He, L., Liu, B., and Hua, Q.-S. (2018). Graph processing on gpus: A survey. *ACM Computing Surveys (CSUR)*, 50(6):1–35.

[117] Simmhan, Y., Kumbhare, A., Wickramaarachchi, C., Nagarkar, S., Ravi, S., Raghavendra, C., and Prasanna, V. (2014). Goffish: A sub-graph centric framework for large-scale graph analytics. In Silva, F., Dutra, I., and Santos Costa, V., editors, *Euro-Par 2014 Parallel Processing*, pages 451–462, Cham. Springer International Publishing.

[118] Stauffer, M., Fischer, A., and Riesen, K. (2018). Keyword spotting in historical handwritten documents based on graph matching. *Pattern Recognition*, 81:240–253.

[119] Stein, M., Frömmgen, A., Kluge, R., Wang, L., Wilberg, A., Koldehofe, B., and Mühlhäuser, M. (2018). Scaling topology pattern matching: A distributed approach. *self*, 1:n2.

[120] Sun, S. and Luo, Q. (2020a). In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 1083 – 1098, Portland, OR, USA. Association for Computing Machinery.

[121] Sun, S. and Luo, Q. (2020b). Subgraph matching with effective matching order and indexing. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1.

[122] Sun, Z., Wang, H., Wang, H., Shao, B., and Li, J. (2012). Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799.

[123] Ta, A.-P., Wolf, C., Lavoue, G., and Baskurt, A. (2010). Recognizing and localizing individual activities through graph matching. In *2010 7th IEEE International Conference on Advanced Video and Signal Based Surveillance*, pages 196–203. IEEE.

[124] Teixeira, C. H., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. (2015). Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 425–440, Monterey California. ACM.

[125] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. (2013). From "think like a vertex" to" think like a graph". *Proceedings of the VLDB Endowment*, 7(3):193–204.

[126] Tran, H.-N., Cambria, E., and Hussain, A. (2016). Towards gpu-based commonsense reasoning: Using fast subgraph matching. *Cognitive Computation*, 8(6):1074–1086.

[127] Tran, H.-N., Kim, J.-j., and He, B. (2015). Fast subgraph matching on large graphs using graphics processors. In *Proceedings of the 20th International Conference on Database Systems for Advanced Applications*, pages 299–315, Hanoi, Vietnam. Springer, Cham.

[128] Ullmann, J. R. (1976). An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42.

[129] Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

[130] Veldhuizen, T. L. (2012). Leapfrog triejoin: a worst-case optimal join algorithm.

[131] Vento, M. (2015). A long trip in the charming world of graphs for Pattern Recognition. *Pattern Recognition*, 48(2):291–301.

[132] Vineet, V., Harish, P., Patidar, S., and Narayanan, P. (2009). Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 167–171.

[133] Wang, H., Li, N., Li, J., and Gao, H. (2018a). Parallel algorithms for flexible pattern matching on big graphs. *Information Sciences*, 436-437:418–440.

[134] Wang, J., Ren, X., Anirban, S., and Wu, X.-W. (2019a). Correct filtering for subgraph isomorphism search in compressed vertex-labeled graphs. *Information Sciences*, 482:363–373.

[135] Wang, K., Zuo, Z., Thorpe, J., Nguyen, T. Q., and Xu, G. H. (2018b). Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 763–782, Carlsbad, CA, USA. USENIX association.

[136] Wang, L., Wang, Y., Yang, C., and Owens, J. D. (2016a). A comparative study on exact triangle counting algorithms on the gpu. In *Proceedings of the ACM Workshop on High Performance Graph Processing*, pages 1–8.

[137] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. (2016b). Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12.

[138] Wang, Z., Gu, R., Hu, W., Yuan, C., and Huang, Y. (2019b). Benu: Distributed subgraph enumeration with backtracking-based framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 136–147, Macao, Macao. IEEE.

[139] Winans, M., Faupel, D., Armstrong, A., Henderson, J., Valentine, E., McDonald, L., Walters, D., Waite, J., Trapani, M., and Magill, E. (2016). 10 Key Marketing Trends for 2017 and Ideas for Exceeding Customer Expectations. ftp://ftp.www.ibm.com/software/in/pdf/10_Key_Marketing_Trends_for_2017.pdf.

[140] Wiskott, L., Krüger, N., Kuiger, N., and Von Der Malsburg, C. (1997). Face recognition by elastic bunch graph matching. *IEEE Transactions on pattern analysis and machine intelligence*, 19(7):775–779.

[141] Wu, X., Theodoratos, D., Skoutas, D., and Lan, M. (2020). Leveraging double simulation to efficiently evaluate hybrid patterns on data graphs. In Huang, Z., Beek, W., Wang, H., Zhou, R., and Zhang, Y., editors, *Web Information Systems Engineering – WISE 2020*, pages 255–269, Cham. Springer International Publishing.

[142] Xiaogang, X., Zhengxing, S., Binbin, P., Xiangyu, J., and Wenyin, L. (2004). An online composite graphics recognition approach based on matching of spatial relation graphs. *Document Analysis and Recognition*, 7(1):44–55.

[143] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013). Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*, pages 1–6, New York, USA. ACM.

[144] Yan, D., Bu, Y., Tian, Y., and Deshpande, A. (2017). Big Graph Analytics Platforms. *Foundations and Trends® in Databases*, 7(1-2):1–195.

[145] Yan, D., Cheng, J., Lu, Y., and Ng, W. (2014). Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992.

[146] Yan, D., Cheng, J., Lu, Y., and Ng, W. (2015). Effective techniques for message reduction and load balancing in distributed graph computation. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1307–1317, Republic and Canton of Geneva, CHE. International World Wide Web Conferences Steering Committee.

[147] Yan, D., Cheng, J., Özsu, M. T., Yang, F., Lu, Y., Lui, J. C., Zhang, Q., and Ng, W. (2016a). A general-purpose query-centric framework for querying big graphs. *Proceedings of the VLDB Endowment*, 9(7):564–575.

[148] Yan, D., Guo, G., Chowdhury, M. M. R., Özsu, M. T., Ku, W.-S., and Lui, J. C. (2020). G-thinker: A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1369–1380, Dallas, TX, USA, USA. IEEE.

[149] Yan, D., Huang, Y., Cheng, J., and Wu, H. (2016b). Efficient processing of very large graphs in a small cluster.

[150] Ye, M., Li, J., Ma, A. J., Zheng, L., and Yuen, P. C. (2019). Dynamic graph co-matching for unsupervised video-based person re-identification. *IEEE Transactions on Image Processing*, 28(6):2976–2990.

[151] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.

[152] Zeng, K., Yang, J., Wang, H., Shao, B., and Wang, Z. (2013). A distributed graph engine for web scale rdf data. *Proceedings of the VLDB Endowment*, 6(4):265–276.

[153] Zeng, L., Zou, L., Özsu, M. T., Hu, L., and Zhang, F. (2020). Gsi: Gpu-friendly subgraph isomorphism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1249–1260.

[154] Zhang, L. and Gao, J. (2018). Incremental Graph Pattern Matching Algorithm for Big Graph Data. *Scientific Programming*, 2018:1–8.

[155] Zhang, L., Wang, H., Li, C., Shao, Y., and Ye, Q. (2017). Unsupervised anomaly detection algorithm of graph data based on graph kernel. In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)*, pages 58–63. IEEE.

[156] Zhang, P., Holk, E., Matty, J., Misurda, S., Zalewski, M., Chu, J., McMillan, S., and Lumsdaine, A. (2015). Dynamic parallelism for simple and efficient gpu graph algorithms. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, pages 1–4.

[157] Zhang, S., Li, S., and Yang, J. (2009). Gaddi: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 192–203, Saint Petersburg Russia. ACM.

[158] Zhang, T., Gao, Y., Qiu, L., Chen, L., Linghu, Q., and Pu, S. (2020). Distributed time-respecting flow graph pattern matching on temporal graphs. *World Wide Web*, 23(1):609–630.

[159] Zhao, J., Han, R., Gan, Y., Wan, L., Feng, W., and Wang, S. (2020). Human identification and interaction detection in cross-view multi-person videos with wearable cameras. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 2608–2616.

[160] Zhao, P. and Han, J. (2010). On graph query optimization in large networks. *Proceedings of the VLDB Endowment*, 3(1-2):340–351.

[161] Zhong, D., Shao, H., and Du, X. (2019). A hand-based multi-biometrics via deep hashing network and biometric graph matching. *IEEE Transactions on Information Forensics and Security*, 14(12):3140–3150.

[162] Zhou, C., Gao, J., Sun, B., and Yu, J. X. (2014). MOCgraph : Scalable Distributed Graph Processing Using Message Online Computing. *Vldb*, 8(4):377–388.