Designing Application-Aware Systems for Mining Large Graph Data

by

Kasra Jamshidi

B.Sc., Simon Fraser University, 2019

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2024

Declaration of Committee

Name:	**Kasra Jamshidi**
Degree:	**Doctor of Philosophy**
Thesis title:	**Designing Application-Aware Systems for Mining Large Graph Data**
Committee:	**Chair:** Yuepeng Wang Assistant Professor, Computing Science
	Keval Vora Supervisor Associate Professor, Computing Science
	Nick Sumner Committee Member Associate Professor, Computing Science
	Anders Miltner Examiner Assistant Professor, Computing Science
	Matei Ripeanu External Examiner Professor Electrical and Computer Engineering University of British Columbia

Abstract

Graph mining is a class of graph analytics whose applications depend on solutions to the NP-complete subgraph isomorphism problem, as well as related problems like subgraph counting, to process an input graph's *subgraphs*. As a result, graph mining is typically more computationally expensive than traditional graph processing workloads. Furthermore, efficient algorithms for graph mining applications have extremely diverse structures. While different research communities have studied domain-specific graph mining applications for many years, research on efficiently executing general classes of graph mining problems and systems for easily programming scalable graph mining applications has emerged only recently. Existing work has focused on adapting graph mining applications to existing generic systems techniques. Instead, this thesis argues for ***application-aware*** design, a philosophy that seeks to leverage the semantics of graph mining applications to build faster, more scalable, and more fault tolerant systems.

Specifically, this thesis formalizes a model of graph mining applications and applies it in four scenarios:

(i) extending the traditional definition of subgraph isomorphism with novel graph constructs that can express local constraints on subgraphs;

(ii) developing an application-aware graph mining system which leverages well-known but difficult to integrate techniques for efficiently mining graphs;

(iii) exploiting algebraic properties of subgraphs and aggregations to design a generic middle-end optimization framework that automatically discovers more efficient alternatives to a given set of input graph patterns while preserving application correctness;

(iv) leveraging the gap in cost between computing and verifying the results of subgraph matching (the NP-complete problem underpinning graph mining) in order to develop a novel distributed architecture with stronger fault tolerance guarantees and greater scalability than existing systems.

Keywords: parallel systems; distributed systems; graph mining

Acknowledgements

It takes a village to write a doctoral dissertation, and the completion of the work before you reflects the outstanding character of the people who have supported me up to this point. My greatest influence has been my advisor Prof. Keval Vora, who poured a significant portion of his own time and energy into my work, and his attentiveness were instrumental in sparking my interest in research, building my confidence, and teaching me how to write papers. None of this research would have come about without his persistent efforts and invaluable advice. I was fortunate to work alongside several other talented researchers. Rakesh Mahadasa and I implemented Peregrine together, one of my most exciting professional experiences. Mugilan Mariappan and Joanna Che were always keen, compassionate, and helpful, whether we were bouncing ideas off each other, grading assignments, or collaborating on papers. I am also grateful to much of the computing science faculty at SFU. Insofar as my research blends ideas from different fields and subfields, I was first exposed to these ideas through conversations with kind and enthusiastic professors whose generous guidance often extended beyond technical ideas and feedback on presentations to broader topics of life itself.

My friends and family were perhaps even more crucial to any successes I've had. Putting together my parents, brother, friends, uncles, cousins, grandmothers, and in-laws, I had a personal cheerleading team every step of graduate school. Their faith in me and support for my research never wavered, despite suffering many declined invitations, reschedulings, and cancellations at my hands while I pursued deadlines. My parents provided consistently sage and empathetic advice, deftly balancing immediate struggles with thoughts for my future. My brother Sina always knew how to use his humour to uplift or divert me, as necessary, and his own successes did much to inspire me. Likewise, my friends became adept at redirecting my focus off work, whether through lighthearted conversation with Behbod Negahban, or exercise and excellent music with Stefan Nazarevich, Philip Nienartowicz, and Josh Smith. Finally, my wife Iulia has been my steadfast partner through this and many other projects. I am grateful for her love, care, and understanding even as she was bogged down by the stresses of her own obligations, and for her patient, sympathetic, yet matter-of-fact reassurances that I *would* complete this dissertation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In both industrial and academic settings, the size of available graph data—and the desire to analyze it—is growing rapidly [183]. Small subgraphs provide insight into the broader structure of a graph, and thus constitute key components in data mining algorithms on graphs, across a variety of domains including bioinformatics [220, 156, 158], computer vision [58], cybersecurity [112], program analysis [84, 201], and social network analysis [179, 185, 221]. This important class of algorithms is defined in the systems literature as graph mining, and focuses on structural analysis of graphs through their subgraphs [205].

Graph mining poses two fundamental challenges that makes it interesting to systems researchers. The first challenge is the cost in both time and space required for mining large graphs. Analysis of subgraphs may depend on solutions to the NP-complete subgraph isomorphism problem or related hard problems like subgraph counting, and the number of subgraphs that must be analyzed scales exponentially with the size of the input graph. As a result, graph mining is resource intensive, especially compared to traditional graph processing workloads like PageRank which can be computed in polynomial time [140], to the point where the 50-kilobyte CiteSeer graph [82] containing only 4,700 edges can take minutes to mine [205] despite fitting easily within the L1 cache. Thus, scaling graph mining workloads to real-world graphs with millions or billions of edges requires sophisticated parallel systems.

Second, efficient graph mining algorithms have been developed independently by different scientific communities, each applying their own perspectives to solve narrow problems within their respective domains. The result is that vertical slices of graph mining knowledge are siloed within each field, with large inconsistencies in the structures and methods of state-of-the-art solutions to graph mining problems. These solutions are rigid, requiring effort to adapt to different use cases even in related domains, and leave performance on the table by neglecting techniques learned in other communities.

The response to these challenges has been the rise of programmable graph mining systems, all designed with the same philosophy. Each system begins with a well-known principle from past systems research that addresses one or more of the scalability challenges posed by graph mining, and combines it with a simple unifying model for the diverse set of graph

mining algorithms to develop a parallel graph mining engine. This forms the backend of the system, which is wrapped with an application programming interface (API) on the frontend to allow domain experts to implement their own graph mining applications. These systems are an initial step towards feasibly executing large-scale graph mining workloads, but they oversimplify graph mining applications in order to fit the interfaces of existing system design paradigms. This approach is *application-oblivious*, rendering the user intent opaque to the system and therefore impossible to leverage in efficiently executing graph mining workloads.

Application-Awareness. This thesis provides an alternative view on designing systems that centres formal understanding of the applications being executed. Such a perspective is *application-aware*, and opens avenues for optimizing execution based on application-level insights. In particular, the knowledge silos of domain-specific graph mining communities can be accessed, allowing the many techniques and optimizations to be applied, only given a baseline understanding of application semantics. In other words, the ability of a general-purpose graph mining system to optimize its execution is contingent on a sufficiently sophisticated unifying model for graph mining applications.

The thesis begins by formulating such a model of graph mining applications in Section 2.2, formally defining graph mining applications as an *aggregation* $\oplus$ over the *subgraphs* S of a *data graph* g. The formalisms surrounding each component of this model are then analyzed in order to design novel application-aware systems and frameworks that aggressively optimize their execution and provide greater efficiency, scalability, and fault tolerance than is possible for an application-oblivious system. This is possible due to the transparency afforded by the formal model of graph mining applications. Application-aware design enables integrating seemingly incompatible techniques from different communities, generalizing limited techniques to broader and broader classes of application, and developing brand new constructs for reasoning about graph mining.

Specifically, this thesis applies the formal model in four ways:

(i) following observations on the difficulty specifying S in existing systems, a pattern-based view of graph mining emerges based on the novel concept of *extended subgraph isomorphism*, allowing local constraints on subgraphs to be expressed with well-defined, and transparent semantics (Chapter 4);

(ii) this pattern-based view motivates an application-aware graph mining system which leverages well-known but difficult to integrate techniques for efficiently obtaining S as matches of graph patterns in g (Chapter 5);

(iii) algebraic properties of subgraphs and user-defined aggregations are used to design a generic middle-end optimization framework that automatically discovers more efficient alternatives to a given set of input graph patterns while preserving application correctness (Chapter 6);

(iv) leveraging the gap in cost between computing and verifying the results of subgraph matching (the NP-complete problem underpinning graph mining) in order to develop a novel distributed architecture with stronger fault tolerance guarantees and greater scalability than existing systems (Chapter 7).

Next is a more detailed overview of these four instances of application-aware design.

Pattern Semantics. In instance (i), the thesis formalizes the semantics of local structural filters on subgraphs, as these are implemented in existing systems through opaque user-defined functions (UDFs). A pattern-based approach is adopted, where the desired subgraphs S can be thought of as matches of a set of pattern graphs in g. Since the existing definition of matches (*i.e.*, subgraph isomorphism) can only express the *existence* of edges, and therefore does not obviate UDFs, we develop novel graph constructs *anti-edge* and *anti-vertex* with formal semantics that, when added to patterns, enable expression of nuanced constraints on the local structure of matches. These constructs make structural filters transparent to underlying runtimes, unify the dichotomy between edge-based and vertex-based exploration perpetuated by existing systems, integrate with existing techniques, and are declarative (*i.e.*, they do not rely on implementation details of the underlying system).

Pattern-Aware Graph Mining. In instance (ii), the thesis develops Peregrine, a shared-memory general-purpose graph mining system that can be programmed with a mixture of pattern semantics and user-defined functions. Peregrine leverages pattern semantics to integrate and generalize a large suite of previously incompatible application-specific techniques, delivering up to 737× faster execution time than the state-of-the-art distributed graph mining system [77] with 8× fewer hardware resources. Implementing the state-of-the-art shared-memory application-oblivious system design [146] on top of the Peregrine runtime showed that the domain-specific techniques enabled by application-awareness account for up to 42× speedup in execution time.

Generic Query Optimization. In instance (iii), we analyzed the performance of graph mining systems and developed Subgraph Morphing in response, a middle-end query optimization framework that integrates with any pattern-based graph mining engine. Subgraph Morphing recognizes equivalent programs using the semantics of subgraph isomorphisms and traverses the space of possible programs to select an efficient one, with no changes to the system runtime and one additional application specification required from the user. There is no single bottleneck in any given graph mining system across different workloads, and performance characteristics vary wildly from application to application, so efficiency of a given program is determined using the underlying system's own internal optimizer. Subgraph Morphing was integrated with 4 different systems, and improved execution time by up to 34× when wrapping existing engines without changes to system code.

Byzantine Fault Tolerance Without Task Replication. In instance (iv), we developed OsirisBFT, a new distributed architecture that tolerates Byzantine faults (the most difficult failure model, since faulty machines can act arbitrarily or even maliciously) with stronger resiliency guarantees and greater performance and scalability than the state-of-the-art architecture for distributed graph mining. Traditional approaches to Byzantine fault tolerance replicate expensive graph mining tasks to ensure they are executed correctly, but OsirisBFT uses pattern semantics to guarantee the results of tasks without replication through lightweight verification. Compared to the traditional architecture, the application-aware design yields up to 4× higher mining throughput while tolerating 2× more failures. Furthermore, we demonstrate that the OsirisBFT architecture is applicable beyond graph mining to other workloads such as robot motion planning and video analysis, with similar improvements in performance and fault tolerance.

Contributions. This thesis uses application-aware design to develop novel systems, frameworks, and techniques for more efficient, scalable, and fault tolerant graph mining, and in the process generalizes and repurposes graph mining knowledge across domains and applications. The key contributions include:

1. Introduction of *pattern-awareness* as an application-aware design philosophy for graph mining systems, a key driver in the performance of recent work in graph mining systems [53, 50, 202, 47, 46] (Chapter 4 and Chapter 5).

2. Integration of previously incompatible application-specific techniques like symmetry breaking and graph orientation (*i.e.*, data vertex reordering). The obstacles to integrating these techniques were resolved due to the formal semantics of extended subgraph isomorphism. These and other pattern-aware techniques contribute to orders-of-magnitude execution time speedups over pattern-oblivious systems (Chapter 5).

3. The first system-agnostic query optimization framework, Subgraph Morphing, which enabled previously intractable workloads (*e.g.*, 4-motif counting on billion-scale graphs) without requiring changes to system code. Subgraph Morphing accomplishes this by generalizing combinatorial identities reserved for counting subgraphs to a broad class of user-defined aggregations (Chapter 6).

4. A Byzantine fault tolerant distributed architecture OsirisBFT for processing task-parallel applications (not only graph mining applications) on data streams that leverages application semantics to avoid replicating expensive computations. Executing in a fixed-size cluster, OsirisBFT can tolerate more faults without sacrificing safety than any other existing approach, while making fewer assumptions on the nature of failures in the cluster, and empirically demonstrating higher throughput (Chapter 7).

Roadmap. The remainder of the thesis is structured as follows. Next, Chapter 2 defines the terminology used in this paper and defines a formal model for graph mining applications. Application-oblivious graph mining systems are then analyzed according to this model in Chapter 3. What follows are the four instances of application-aware design. Chapter 4 develops a model for understanding the structural aspects of graph mining applications through the anti-edge and anti-vertex constructs alongside extended subgraph isomorphism. Chapter 5 develops a novel parallel and programmable graph mining system, Peregrine, that leverages extended subgraph isomorphism and sophisticated pattern analysis techniques to efficiently and scalably execute graph mining applications. Chapter 6 carries forward the pattern-centric approach to reason about substructural similarities between subgraphs, giving rise to the Subgraph Morphing framework which automatically optimizes pattern-based graph mining programs. Finally, OsirisBFT is presented in Chapter 7. OsirisBFT's distributed processing architecture exploits the algorithms at play in pattern-based graph mining systems to tolerate more faults with fewer assumptions than traditional approaches to Byzantine fault tolerance, all while delivering higher throughput. The thesis closes with a brief literature review in Chapter 8 covering related work from the systems, databases, and algorithms communities followed by concluding remarks in Chapter 9.

Chapter 2

Preliminaries

This chapter summarizes the prerequisite knowledge for understanding graph mining works and clarifies the terminology used throughout this thesis (summarized in Table 2.1).

2.1 Terminology

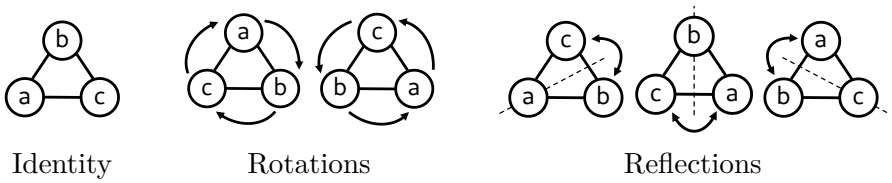
Graphs. Given a graph g, we denote its vertices by $V(g)$ and its edges by $E(g)$. Vertices and edges are uniquely identified by integer ids, and may also have labels, given by functions $L_g(v)$ for $v \in V(g)$ and $W_g(e)$ for $e \in E(g)$. The *neighbourhood* or adjacency list of a vertex $v \in V(g)$ is written $\text{adj}(v) = \{(u, v) \in E(g)\} \cup \{(v, u) \in E(g)\}$. A *clique* is a graph where every pair of vertices is adjacent. For ease of exposition, this thesis uses simple, undirected graphs unless specified otherwise, though most techniques presented apply naturally to more general graph models. When the generalization for a technique is not trivial, it is described in the appendices.

A *subgraph* of a graph g is a graph s such that $E(s) \subseteq E(g)$ (and thus, $V(s) \subseteq V(g)$). Unless specified otherwise, the term term subgraph in this thesis refers to a *connected subgraph*, consisting of a single connected component. A subgraph s is *vertex-induced* if it contains all edges between its vertices, *i.e.*, it satisfies

$$\forall u, v \in V(s), (u, v) \in E(g) \implies (u, v) \in E(s).$$

Otherwise, s is *edge-induced*. The works covered focus on simple, undirected, vertex-labeled graphs, though their contributions apply to directed, edge-labeled, multigraphs with small adjustments. For instance, the anti-vertex presented in Chapter 4 is generalized from simple undirected graphs to the rich property graph model in Appendix A.1.2.

Subgraph Isomorphisms. Central to many graph mining applications is the *subgraph isomorphism* problem. Given a *data graph* g and a *pattern graph* p, a subgraph isomorphism



Identity Rotations Reflections

Figure 2.1: The 6 unique automorphisms of a triangle graph, obtained through the identity transformation, clockwise or counter-clockwise rotation, or reflection around a vertex.

is an injective function $\phi : V(p) \to V(g)$ satisfying the following:

$$\forall (u, v) \in E(p), (\phi(u), \phi(v)) \in E(g),$$
$$\forall v \in V(p), L_p(v) = L_g(\phi(v)),$$
$$\forall (u, v) \in E(p), W_p((u, v)) = W_g((\phi(u), \phi(v))).$$

The range of a subgraph isomorphism describes a subgraph of g that has the same structure as p (*i.e.*, the subgraph is isomorphic to p). For convenience, this subgraph is sometimes written $\phi(p)$. The literature gives many names to $\phi(p)$, including *match*, *embedding*, and even simply *subgraph*. Some works apply the same terms to the subgraph isomorphism mapping instead. This thesis refers to $\phi(p)$ as a *match*, and sometimes abuses the term to also mean the isomorphism (*e.g.*, "computing matches of a pattern p in graph g" instead of "computing subgraph isomorphisms of p into g").

This abuse includes referring to vertices in a match using vertices in its domain. If m is a match of p in g, then for a vertex $v \in V(p)$, $m(v) \in V(g)$ is the vertex mapped by v in the subgraph isomorphism of m. It is also often convenient to write a subset of $V(m)$ in function notation: if $V' \subset V(p)$, then $m(V') \subset V(m) \subseteq V(g)$ is the set of vertices the subgraph isomorphism of m maps to the vertices in V'.

The subgraph isomorphism problem is intractable in general, since for arbitrary inputs there are $O(n^k)$ possible matches, where $k = |V(p)|$ and $n = |V(g)|$. This thesis refers to the set of matches for a pattern p in a graph g as $\mathcal{E}(g, p)$.

Graph Isomorphisms and Automorphisms. When $|V(g)| = |V(p)|$, ϕ is a *graph isomorphism*. Computing graph isomorphisms is thought to be computationally simpler than subgraph isomorphism [189], and in practice it is efficient for small graphs [148, 117].

In the special case $g = p$, ϕ is called an *automorphism*. Automorphisms reassign the ids of vertices and edges in a graph without changing the structure or labeling. Figure 2.1 shows the 6 automorphisms of a triangle graph, corresponding to each of the ways the triangle can be rotated or flipped to obtain a new assignment of ids. Intuitively, automorphisms are different "views" of a graph. The set of automorphisms for a graph g is an equivalence class called the *automorphism group* of g.

Definition	Common Terms
Large graph being mined; one of its edges/vertices	**data graph**, input graph, graph; **data edge/vertex**
Graph in the domain of a subgraph isomorphism; one of its edges/vertices	**pattern**, query graph, template, subgraph, motif, graphlet; **pattern edge/vertex**
Subgraph described by the range of a subgraph isomorphism	**match**, embedding, instance, subgraph, pattern, motif, graphlet
Subgraph that contains all edges between its vertices	**vertex-induced**, induced
Subgraph that may not contain all edges between its vertices	**edge-induced**, non-induced
Measure of frequency in FSM	**frequency**, support
Column in an MNI table	**MNI column**, domain
Graph with k vertices where every pair of vertices is adjacent	**k-clique**, K_k, complete graph on k vertices
Set of all unique graphs (up to isomorphism) with k vertices	**k-motifs**

Table 2.1: Terminology used in the literature.
This thesis uses the bolded terms.

This thesis abuses the term automorphism to refer both to a mapping ϕ as well as to a representative of the automorphism group. The *pattern of a subgraph* s is a canonical representative p for the automorphism group of a graph that is isomorphic to s. Two matches/subgraphs are *duplicates* if they are automorphic. Similarly, a set of *unique* matches/subgraphs is one where no pair of matches/subgraphs are automorphic. The set of all unique matches for a pattern p in a graph g is written $\mathcal{E}^*(g, p)$.

2.2 Application Semantics

In this section we formalize a model for the semantics of graph mining applications considered in this thesis, and use it to describe several common existing workloads.

Definition 2.2.1 (Graph Mining Application Semantics)**.** Let $\mathcal{G}$ be the set of all finite graphs. This thesis defines a graph mining application on a graph $g \in \mathcal{G}$ as an aggregation over a set of subgraphs $S \subseteq S_g$. Let $\mathcal{R}$ be an application-specific set representing graph mining results. Then, an *aggregation* is a commutative monoid[†] $\langle \mathcal{R}, \oplus \rangle$, where $\mathcal{R}$ is the set of *aggregation values*, and $\oplus$ is called the *aggregation operator*. We model graph mining applications as functions $App : \mathcal{G} \to \mathcal{R}$ which take a graph as input and perform an

[†]A commutative monoid $\langle \mathcal{R}, \oplus \rangle$ (abbreviated as simply $\mathcal{R}$ when $\oplus$ is apparent from context) consists of a set $\mathcal{R}$ with an associative and commutative binary operator $\oplus : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$, such that there exists an identity element $e \in \mathcal{R}$ for which $e \oplus v = v$ for all $v \in \mathcal{R}$. *E.g.*, non-negative integers form a commutative monoid under addition.

aggregation over some subset of its subgraphs. This can be expressed formally as

$$\forall g \in \mathcal{G}, \exists S \subseteq S_g, App(g) = \bigoplus_{s \in S} \nu(s) \tag{2.1}$$

where $\nu : S \to \mathcal{R}$ is an application-specific function mapping subgraphs to aggregation values.

This model captures the core workload in graph mining applications which dominates execution time: exploring and aggregating subgraphs [205]. The set S is a set of interesting subgraphs the application wishes to process. The function ν extracts valuable information from each subgraph, such as its pattern or its vertex/edge ids, or simply yields the subgraph unchanged. The aggregation operator combines each subgraph-specific value into one value that can be analyzed by users. On the other hand, the model does not prescribe how an application should be implemented, or how a graph mining system should enable a given application. Eq. 2.1 also purposely excludes computations performed on the input graph before mining, as well as additional analysis of the aggregation values after mining. In this manner, important graph mining workloads can be represented concisely without generalizing their semantics to the point of superficiality or overspecifying their semantics by assuming implementation details.

2.2.1 Examples

Consider the following examples of graph mining applications whose semantics are modeled by Eq. 2.1.

Example 2.2.1 (Subgraph Matching)**.** In the subgraph matching application, the user is interested in all matches for a target pattern p in data graph g. This application is a pure example of the subgraph isomorphism problem, and there are many works focused solely on efficient subgraph matching in various contexts [123, 172, 33, 102, 101, 190, 18, 125, 175, 191, 134]. Most commonly, the user is interested in edge-induced matches, but some works consider vertex-induced matches. Subgraph matching is also called pattern matching, subgraph querying, subgraph listing, or subgraph enumeration in the literature.

Here, S is the set of matches for the target pattern p, $\mathcal{R}$ is the set of all subsets of $\mathcal{S}_g$, ν is the identity function returning each match unchanged, and $\oplus$ is the set union operation. If the set of matches is not necessary to maintain, *e.g.*, the application simply performs some computation on each match that does not produce any value, then ν can return an empty set for every subgraph. Another common variation on subgraph matching is subgraph counting, where only the number of matches is required. In subgraph counting, $\mathcal{R}$ is the set of non-negative integers, ν maps each match to the integer 1, and $\oplus$ simply performs addition.

Key-Value Aggregations. Example 2.2.1 shows simple aggregations that produce a single value or set. Other applications produce a map of results, represented as key-value aggregations by setting $\mathcal{R} = \mathbb{P}(\{(k, r) : k \in K, r \in R\})^\dagger$, where K is an arbitrary set but the set of values R has its own binary operator $\oplus_R$ such that $\langle R, \oplus_R \rangle$ is a commutative monoid. Then the aggregation operator $\oplus$ sums values $A, B \in R$ with matching keys using $\oplus_R$:

$$\begin{aligned} A \oplus B = &\{(k, r_A \oplus_R r_B) : (k, r_A) \in A \wedge (k, r_B) \in B\} \\ &\cup \{(k, r) \in A : \nexists v'(k, r') \in B\} \\ &\cup \{(k, r) \in B : \nexists v'(k, r') \in A\}. \end{aligned}$$

Keys present in both A and B have their values summed using $\oplus_R$, while keys only present in one are unchanged. $\oplus$ and $\oplus_R$ are distinguished as the *outer* and *inner* aggregation operators, respectively. Such aggregations are used in the next three examples.

Example 2.2.2 (k-Motif Counting)**.** In k-motif counting, the user is interested in how many matches in g correspond to each pattern with k vertices. This application is especially important in bioinformatics [156, 220], and has been extensively studied in that community [93, 141, 168, 9, 173, 139, 24, 196, 88]. Some works materialize the matches for the user as well as counting them, some use combinatorial identities to efficiently count matches without allowing the user to access them, and others only approximate the distribution. Motif is a synonym for pattern, and is also sometimes referred to as a graphlet.

In the k-motif counting application, S is the set of all unique subgraphs of size k and $\mathcal{R}$ is the powerset $\mathbb{P}(\{(p, n) : p \text{ is a } k\text{-motif}, n \in \mathbb{Z}^{\geq 0}\})$, with addition as the inner binary operator and $\oplus$ defined as above. Every subgraph s is mapped to $(p, 1)$ by ν, where p is the pattern of s.

Example 2.2.3 (Frequent Subgraph Mining)**.** Frequent subgraph mining (FSM) seeks to detect which patterns occur frequently in a data graph g ("subgraph" in FSM actually refers to patterns, not matches or subgraphs of g) [82]. Classic works on FSM considered the context of a graph database containing many graphs, where frequency referred to the number of graphs in the database that contained a match for a given pattern. More recently, and for the purposes of graph mining systems, FSM is executed on one large graph, where frequency is not measured by the number of matches [82, 203, 182, 12, 104, 237, 31, 2], because match counts are not *anti-monotonic.*. Anti-monotonicity means that if a pattern of size k is infrequent, no pattern of size $l > k$ will be frequent, and is essential to efficiently compute FSM, since it allows pruning all subgraphs that contain a match for a smaller infrequent pattern from the search space.

[†]For a set A, $\mathbb{P}(A) = \{A' \subseteq A\}$ is the powerset of A, *i.e.*, the set of all subsets of A. Note that the empty set is a subset of any set.

The most commonly used frequency measure is Minimum Node Image (MNI) [37]. To compute MNI for a pattern, a table is constructed where each column corresponds to a pattern vertex and contains the set of all data vertices that correspond to it in some match. MNI is the size of the smallest column in this table.

Eq. 2.1 models FSM on a graph g as follows. For a k-vertex pattern p, the set of MNI tables can be formalized as $R_k = \{(V_1, V_2, \ldots, V_k) : V_i \subseteq V(g)\}$. Two elements in R are combined by taking the union of corresponding columns:

$$(V_1, \ldots, V_k) \oplus_R (V_1', \ldots, V_k') = (V_1 \cup V_1', \ldots, V_k \cup V_k').$$

Then finally the result set is $\mathcal{R} = \mathbb{P}((p, r) : p \text{ is a graph} \wedge \exists i \in \mathbb{N}, r \in R_i)$. S is the set of subgraphs that are isomorphic to a frequent pattern, and ν maps a subgraph s to the pair (p, r) where p is the pattern of s and r is the MNI table where each vertex in s forms its own column.

Other common workloads associate data vertices or edges with structural characteristics such as match counts [108].

Example 2.2.4 (Local k-Clique Counting)**.** Local k-clique counting records the number of k-cliques each vertex in a data graph g participates in. In Eq. 2.1 this is formalized by setting $\mathcal{R} = \mathbb{P}(\{(v, c) : v \in V(g) \wedge c \in \mathbb{N}\})$, with addition as the inner binary operator. The relevant subgraphs are $S = \{s \in S_g : s \text{ is a } k\text{-clique}\}$, which can be more concisely stated as $S = \mathcal{E}^*(g, k\text{-clique pattern})$.

These formalisms for graph mining applications enable critical discussion of existing system designs in the following chapter.

Chapter 3

Application-Oblivious Graph Mining Systems

This chapter contextualizes the work of this thesis within the graph mining systems literature.

3.1 Understanding Programmable Graph Mining Systems

Existing programmable graph mining systems were designed by combining well-known systems ideas (*e.g.*, concurrent disk-backed task queues in G-Miner [45], relational joins [4] in RStream [219]), or even entire systems (*e.g.*, Giraph [23] in Arabesque [205], Spark [230] in Fractal [77]), and building a graph mining abstraction to match. These graph mining abstractions are typically thin wrappers around the processing model required by the underlying systems techniques. For instance, Arabesque explores in breadth-first fashion by extending one subgraph at a time due to its bulk-synchronous parallel [211] processing model inherited from Giraph, which in turn inherited it from MapReduce [72].

This section sketches a high-level overview of the foundations of the graph mining systems literature. Three systems are discussed: the first programmable graph mining system Arabesque [205], as well as the most recent two systems Fractal [77] and AutoMine [146].

3.1.1 Arabesque: General-Purpose Graph Mining

The first system to tackle programmable graph mining was Arabesque [205], which identified that existing graph processing systems are ill-suited to operations on patterns and their matches. Specifically, there are two major obstacles to solving graph mining problems on a traditional graph processing system: (a) graph processing systems typically expose APIs that operate on individual vertices or edges at a time (*i.e.*, users write small functions which are applied to each vertex/edge over several iterations), while graph mining applications operate on matches, leading to *awkward and buggy user programs* that must build up matches one vertex/edge at a time in ad-hoc fashion; and (b) since graph processing sys-

```
bool filter(match e) {
  if (e.numVertices() == 3)
    return e.numEdges() == 3;
  else
    return e.numVertices() < 3;
}
void process(match e) {
  if (e.numVertices() == 3)
    map(pattern(e), 1);
}
pair reduce(pattern key, int vals[]) {
  return pair(p, sum(vals));
}
void aggProcess(pattern key, int val) {
  print(key, val);
}
```

(a) Filter-Process

```
int count = 0;
void vertexMap(vertex u) {
  for (v in u.nbrs()) {
    vertex x[] = u.nbrs() ∩ v.nbrs();
    count += x.size();
  }
}
count /= 3;
```

(b) Think Like A Vertex

Figure 3.1: Triangle Counting programs in the Filter-Process [205] and TLV [140] programming models.

tems are optimized for such vertex-centric and edge-centric programs, they cannot handle the *combinatorial explosion of intermediate state* when exploring subgraphs of a graph. In this section, we examine Arabesque's design in detail and discuss its drawbacks.

Programming Model

Arabesque's answer to the first obstacle is the "Think Like an Embedding" (TLE) paradigm (inspired by "Think Like a Vertex" (TLV) in graph processing [140]) and associated *filter-process* model, wherein the system iteratively generates larger matches and invokes user-defined `filter` and `process` functions on them. Matches that pass the filter are processed and then extended to larger matches. The program either operates on vertex-induced or edge-induced matches. The set of initial matches is simply all vertices in the graph (or edges, if edge-induced matches are desired). These user-defined functions (UDFs) also have access to generic map-reduce style aggregations to support global computations, along with filtering and processing callbacks to operate on aggregated data. Arabesque guarantees that matches passed to each function are unique (*i.e.*, no automorphisms of the same match are processed). This programming model makes it simple to implement common graph mining benchmarks in a few simple callbacks.

Example 3.1.1. Figure 3.1 shows how Triangle Counting is implemented in both Arabesque and a TLV graph processing system. In Arabesque's filter-process program (Figure 3.1a), all matches that have greater than 3 vertices are filtered, as are those which have 3 vertices but 2 edges. Thus leaving triangles (*i.e.*, 3 vertices and 3 edges), as well as patterns with 2 or 1 vertices (*i.e.*, single edges and single vertices) that can extend later into triangles. If the user becomes interested in patterns with 4 vertices that contain a triangle, all they

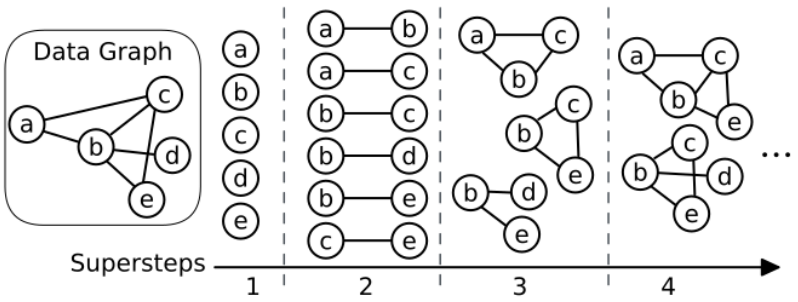


Figure 3.2: Triangle counting in a small graph using the breadth-first processing model from Arabesque [205]. Dashed lines represent barriers between supersteps, where matches from the previous superstep are filtered, processed, and extended into the next superstep.

must do is change Line 5 to accept matches with less than 5 vertices, instead of those with less than 3.

By contrast, in a TLV program (Figure 3.1b), the user-defined function must directly explore the neighbourhood of each vertex and compute an intersection to find triangles. Each triangle match will be encountered from each of its vertices, so the user must divide the final count by 3. Hence, although the TLV program is shorter, it arguably requires more expertise to implement, and must be changed completely if the user is interested in patterns other than a triangle, since simply dividing the count would no longer give correct results.

Processing Model

The second obstacle (handling a combinatorial explosion of matches) is harder to overcome. Arabesque is built on top of a TLV graph processing system, Giraph [23], which adopts a *bulk synchronous parallel* [211] processing model (BSP). The computation proceeds one superstep at a time, where a user-defined `vertexMap` function is applied to *every vertex* in the data graph (or an `edgeMap` function is applied to every edge) before proceeding.

Arabesque builds on top of this abstraction, and uses the `vertexMap` or `edgeMap` functions to iteratively generate larger vertex-induced or edge-induced matches, respectively, and passes them to the filter-process functions. At superstep k, Arabesque filters invalid matches from superstep k and processes the valid matches. Then, for each valid match, all *canonical* (*i.e.*, unique) extensions of the match with a vertex (or edge) are computed for superstep $k + 1$. The computation terminates when there are no more matches. This processing model is known as *breadth-first exploration* in the graph mining literature.

Example 3.1.2. Figure 3.2 shows how triangle counting on a small graph would be executed in Arabesque, using vertex-induced matches. The set of matches at superstep 1 consists of all vertices. These all pass the filter, and are extended into the set of all edges. These also pass the filter, resulting in three size 3 matches. Two matches are for a triangle pattern, while the remaining one is for a wedge pattern. The latter fails the filter because it only

has 2 edges (Line 3 in Figure 3.1a). Hence, the two triangles are counted and extended. At superstep 4, all matches fail the filter, and hence in superstep 5 the computation ends.

Discussion

There are four major challenges arising from the design of Arabesque's programming and processing models. Subsequent general-purpose graph mining systems attempt to address some or all of these topics, and several application-specific solutions have already addressed them.

Challenge: User-Friendliness. While the filter-process model makes expressing aggregations of matches easier compared to the TLV model, it is far from declarative. The main difficulty in writing user programs for the filter-process model is that the user must consider *how the system will arrive at the final set of matches.* In our triangle counting example, despite only being interested in size 3 matches, both the filter and process functions must also consider what to do when the input match is smaller than size 3, since Arabesque invokes these functions at every iteration. This leads to the clunky logic in the example's filter function, where size 3 matches must be checked to ensure they are triangles, size 1 and 2 matches must be passed through, and size 4 matches must be filtered out.

Challenge: Wasted Computation. Another consequence of the filter-process model is that the system is fundamentally unaware of which matches are interesting until *after* they have been explored. Despite only wanting to process triangles in our example, the process function is invoked on every vertex and every edge as well, only for the if-condition to fail. Furthermore, computing the pattern of a match is an expensive graph isomorphism check, even if the user knows explicitly which patterns they are interested in mining.

Challenge: State Explosion. Arabesque struggles with the combinatorial explosion problem in graph mining, which manifests as a memory bottleneck in the breadth-first exploration model since all matches at the current superstep must be stored to be extended at the next superstep. For instance, in the motif counting application, the Orkut graph [225] yields 123 trillion matches at superstep 4, despite the graph containing only 117 million edges.

To cope with such massive state, Arabesque developed the Over-approximating DAG (ODAG), a data structure for aggressively compressing matches, trading space for time. While the ODAG offers impressive compression factors (albeit, without any bounds or guarantees), it suffers from heavy decompression costs that actually consume the majority of CPU utilization in some applications.

Challenge: Load Imbalance. Breadth-first exploration leads to load imbalances between workers. Each worker holds a set of matches which are extended every superstep. However, due to the skewed nature of real-world graphs [49], some matches will contain high-degree vertices and result in far more extensions than most matches which will comprise mainly

low-degree vertices. Hence, workers beginning with matches containing high-degree vertices will have far more work to do than workers with low-degree matches.

Arabesque attempts to address this issue by gathering all matches at a leader node and redistributing matches fairly across workers. However, this load balancing scheme is itself a massive bottleneck, as all workers must synchronize after every superstep, contend for bandwidth to send matches to the leader, and spend long amounts of time compressing and decompressing matches into ODAGs.

Previous Solutions. It is important to note that these challenges were already solved by several application-specific solutions which did not need to support a general programming or processing model. For example, GraMi [82], a framework for frequent subgraph mining (FSM), solved the problems of wasted computation and state explosion in the case of FSM. It aggressively reuses previous results to prune its search space, guides its exploration using information about which patterns were previously frequent, and only exploring a pattern's matches until the pattern can be guaranteed to be frequent. As a result, it explores a fraction of the matches that Arabesque does. Similarly, there have been several works regarding efficient motif counting [9, 141, 108, 173] which do not suffer from state explosion, load imbalance, or wasted per-match computations because they use combinatorial identities to quickly count matches without having to materialize them.

3.1.2 Fractal: Depth-First Graph Mining

Fractal moves away from the bulk synchronous parallel Giraph [23] backend used by Arabesque to a more general Spark backend in order to overcome the state explosion problem in Arabesque's breadth-first exploration. Using Spark on the backend, Fractal is able to explore subgraphs in a depth-first manner, reducing the per-thread memory footprint for t threads from $O(|S|/t)$ in breadth-first exploration to just $O(1)$ by having each thread fully explore one subgraph at a time before moving on to another. In conjunction, Fractal develops the *fractoid* model for defining graph traversals.

Finally, systems emerged that were able to tackle the state explosion problem in memory with a small algorithmic tweak: abandoning breadth-first exploration and exploring in *depth-first* manner instead. In a depth-first exploration, each thread only generates one match at a time, extending it as far as possible before backtracking to a different match, thus completely eliminating the state explosion problem. At the same time, other aspects of the system, like user-friendliness, generality or load balancing are not necessarily sacrificed by this choice.

Fractal [77] is a distributed general-purpose graph mining system from the same group as Arabesque, and makes improvements in addressing all but one of the main challenges of graph mining. Aside from the switch from breadth-first to depth-first, Fractal's processing model is similar to Arabesque. Exploration begins from individual edges or vertices, which are iteratively extended with either edges or vertices. Load imbalances are rectified through work-stealing: workers can steal a portion of the edges or vertices that extend a given match.

```
let computation = g.vfractoid()
    .expand(3) // extend by 3 vertices
    .filter(s -> s.nEdges == 3)
    .aggregate(s -> map("", 1),
      (n1, n2) -> a + b);
```

(a) Triangle Counting

```
let T = FREQUENCY_THRESHOLD;
// using edge fractoid
let computation = g.efractoid()
    .expand(1) // extend by 1 edge
    .aggregate(s -> map(s.pattern(),
                        mniTable(s)),
      (a, b) -> merge(a, b))
    .aggFilter(
      (emb, key, val) -> val > T)
    .explore(k); // loop operator
```

(b) FSM

Figure 3.3: Triangle Counting and FSM applications in Fractal [77].

Fractal has a programming model augments filter-process with the concept of *fractoids*, which represent different methods for iteratively extending matches. Where Arabesque supported vertex-induced or edge-induced matches, and explored one vertex at a time or one edge at a time based on the user's choice, Fractal explicitly represents exploration strategies in terms of vertex fractoids, edge fractoids, and pattern fractoids. Users choose which type of fractoid to use, and chain together a series of primitives that defines the computation. Then, Fractal programs can be conceptually represented by a string corresponding to this chain of primitives (E for "extend", F for "filter", and A for "aggregate"). For example, Figure 3.3 shows how triangle counting and FSM are implemented in Fractal. Triangle counting will be executed as "EEEFA"; three extensions, a filter, then an aggregation.

FSM is more complicated. Because FSM must aggregate all matches of a given size after every extension to take advantage of anti-monotonicity, it seems fundamentally incompatible with depth-first exploration. Fractal resolves this tension by *restarting exploration after an aggregation.* Aggregation values are cached across these restarts, but matches themselves are lost and recomputed in depth-first manner each time. This is represented in the program string by a dash. So FSM until maximum size 4 is executed as "EA-EFEA-EEFEA-EEEFEA". Edge matches are extended by one edge to obtain matches with 2 edges, then aggregated. All matches are lost, so next, edge matches are extended by two edges to obtain matches with 3 edges, but after the first extension matches are filtered based on the previous iteration's aggregation. This continues until the maximum size is reached or no matches can be extended.

While vertex and edge fractoids correspond easily with vertex and edge exploration in Arabesque, the pattern fractoid represents a new form of exploration. A pattern fractoid requires a target pattern as input, and extends matches by one vertex at a time, but automatically filters matches that cannot eventually result in a match for the target pattern. This fractoid is used exclusively for the pattern matching application in Fractal.

In addition to the fractoid API, Fractal also provides access to an *enumerator* class that controls how the system extends matches. The default enumerator looks at all neighbours of a match, but users can override its methods for fine-grain control over the extension process.

For example, the paper implements a specialized algorithm for listing k-cliques [71] within Fractal using the enumerator API, where extension candidates are generated by intersecting adjacency lists of all vertices in the match.

Discussion

Fractal's main contribution is the relatively simple switch to depth-first exploration, which avoids the memory bottlenecks from Arabesque. However, this also leads to more wasted computation than before in applications like FSM, as the matches must be recomputed after every aggregation. In practice, it seems that this recomputation still outweighs the costs of breadth-first exploration, as FSM on Fractal outperforms Arabesque and even the FSM-specific system ScaleMine [2]. As we will see with pattern-based graph mining, however, the wasted computation can be completely eliminated as well. In this sense, Fractal's use of pattern fractoids (or lack thereof) is a missed opportunity.

The other major contribution is the two-level programming model, which allows for more declarative programs using fractoids, as well as greater control using the enumerator. Fractoids remain a strong abstraction for expressing graph mining tasks, but they still are not fully declarative for the same reasons that filter-process is not declarative. Both triangle counting and FSM applications require users to think about how matches will be constructed vertex by vertex or edge by edge.

3.1.3 AutoMine: Pattern-Based Graph Mining

Concurrently to the development of depth-first exploration by [77], AutoMine [146] introduced pattern-based exploration, a radically different processing model that does not naïvely extend matches, but constructs exactly the desired matches according to a static pattern matching plan. The resulting programming and processing models are far simpler (though also more restrictive) than Fractal or previous systems, while performance is much better.

AutoMine is motivated by two observations. First, common graph mining applications involve computing the pattern of a match, and second, specialized pattern matching baselines are orders of magnitude faster than previous graph mining systems at computing the matches for a given pattern. Putting these observations together, AutoMine proposes a pattern-based graph mining model, where user programs consist only of a list of patterns the user is interested in. Then, AutoMine generates efficient native code that matches these input patterns in the data graph and returns their matches to the user.

Conceptually, the compilation process is simple. Given a pattern, AutoMine computes a matching schedule, which dictates in what order vertices in the pattern are matched with vertices in the data graph to obtain a match. Then, AutoMine uses this matching schedule to generate a series of nested loops, one per pattern vertex, that iterate over candidate data vertices for every pattern vertex. This models a depth-first backtracking search through the space of matches for this pattern.

```
1 auto p = generateClique(3);
2 int n = count(g, p);
```

(a) User program.

```
1 int result = 0;
2 for (v in V(g)) {
3   for (u in v.nbrs()) {
4     vertex x[] = v.nbrs() ∩ u.nbrs();
5     x = subtract(x, {u, v});
6     for (w in x) {
7       result += 1;
8     }
9   }
10 }
11 result /= 3;
```

(b) Generated code executed by the backend.

Figure 3.4: Triangle counting application in AutoMine [146].

Figure 3.4b shows how a triangle counting program is compiled. The outermost loop iterates over all vertices in the graph to obtain the first vertex in the triangle. The middle loop iterates over the neighbours of the first vertex to obtain the second vertex, and the innermost loop iterates over all vertices in the neighbourhoods of both the first and second vertex, thereby obtaining the last vertex in the triangle.

Generating such code automatically is straightforward [123, 17]. For each pattern vertex AutoMine checks the neighbours that arrive before it in the schedule. The candidates for the pattern vertex are precisely the intersection of the adjacency lists of whichever data vertices are matched to those neighbours. However, choosing a schedule significantly affects performance. AutoMine uses an abstract probabilistic graph, where all pairs of vertices are adjacent with equal probability, in order to model the efficiency of a given schedule, and explores the space of all schedules to determine the minimal one.

Discussion

Pattern-based graph mining eliminates Arabesque's memory bottlenecks and the wasted computation from per-match filtering, and the compiling pattern programs avoids the overheads of dynamic runtimes in previous systems. But while pattern-based graph mining is a powerful concept, it is difficult to judge it based on its treatment in this work. For instance, "root symmetry" is presented as a novel optimization, but is simply a special case of symmetry breaking, first introduced by the bioinformatics community [93] and widely adopted by pattern matching work in databases [123, 134, 17, 33, 101, 32, 102]. Similarly, AutoMine presents an algorithm for computing the symmetries of a pattern that runs in $O(n!)$ time, whereas the problem is known to be fixed-parameter tractable, and there are available open-source libraries for computing the symmetries, one of which is used by Arabesque for computing the pattern of a match [117].

```
1 val cliques = g.vfractoid().expand(1)
2   .filter(s ->
3     s.nEdgesAdded==s.nVertices-1)
4   .explore(k)
5   .aggregate(s -> // defining ν
6     s.forEachVertex(v -> map(v, 1)),
7     (n1, n2) -> n1+n2) // defining ⊕
```

(a) Fractal

```
1 auto p = generateClique(k);
2 vector<> cliques = match(g, p);
3 for (auto &s : cliques) {
4   if (isDuplicate(s)) break;
5   for (auto v : s.vertices) {
6     aggregationStore[v] += 1;
7   }
8 }
```

(b) AutoMine

Figure 3.5: Local Clique Counting programs in Fractal and AutoMine. System functions are highlighted blue.

3.2 Consequences of Application-Obliviousness

Such systems are *application-oblivious*, since they attempt to adapt graph mining semantics into otherwise generic systems while ignoring nuances between different applications. Programmable graph mining system frontends must allow users to specify S, ν, and $\langle \mathcal{R}, \oplus \rangle$ from Eq. 2.1 in order to support diverse applications. However, because existing systems were not designed with these application semantics in mind, there is considerable variation in the interfaces for specifying these semantics, as well as backend support for efficiently executing them. Specifically, different frontends afford different levels of understanding to the underlying system backend, and backends have limited support for executing these semantics natively.

Performance, scalability, and programmability issues often manifest in application-oblivious systems as mismatches between the system's internal view of an application and the application's true semantics as defined by Eq. 2.1. Thus, drawbacks of application-oblivious design can be illustrated by relating the semantics of existing programmable graph mining systems with the semantics of different graph mining applications. This section analyzes state-of-the-art systems preceding Peregrine (developed in Chapter 5) in this manner, showing how the issues discussed in the previous sections trace back to application-oblivious design choices. Consider Fractal [77] and AutoMine [146], two recent graph mining systems. Figure 3.5 shows programs in Fractal [77] and AutoMine [146] for local k-clique counting.

Fractal. On Line 1 of Figure 3.5a, using `vfractoid` specifies that vertex-induced subgraphs are desired, and `expand(1)` specifies that new subgraphs should be produced by expanding previous subgraphs by one vertex at a time. To avoid generating several automorphic subgraphs from expansions, Fractal calls an internal `isCanonical` function on each expanded subgraph and discards those which are not the canonical representative of their automorphism group. The remaining subgraphs are then filtered on Line 3 with a user-defined function that checks a sufficient property for a subgraph to be a clique. By calling `explore(k)` on Line 4, the user specifies that the preceding traversal and filter should be

repeated k times. Finally, the unique k-clique matches that result can be passed to the user-defined `aggregate` functions. On Line 6, each vertex in the match is mapped to 1, to be later summed by the system into the local k-clique counts, and on Line 7 the user defines the inner aggregation operator as addition.

Thus, the combination of fractoid logic, canonicality checking, the user-defined `filter` function, and the final check in the user-defined `aggregate` function all come together to define $S \subseteq S_g$. The calls to `map` from the user code define ν, and the aggregation operator is defined by adding values.

There are two major points to notice. First is that even conceptually simple applications require additional knowledge of the underlying system to implement. Consider the definition of S as the set of k-cliques in g is split across 3 distinct regions of system and user code (fractoid API, canonicality checks, and filter), because Fractal directly splices user code into simple Spark queries. The user-defined `filter` function, for instance, uses the `nEdgesAdded` property of the subgraph, which is directly tied to Fractal's internal graph traversal implementation. Fractal stores the number of edges added to a subgraph when expanding by a vertex, and in a clique the number of edges added will be equal to the remaining vertices. In order to properly specify S, the user must reason about how Fractal traverses the graph and at what point each user-defined function will be called.

Second, the crux of the application is opaque to the system. The user-defined (and hence opaque) `filter` definition is what specifies cliques as the desired subgraphs, and the check for k-cliques before aggregation also occurs in user code. Fractal's backend is left with only 2 pieces of actionable information: all processed subgraphs are unique and all processed subgraphs have at most k vertices. This leaves few opportunities to optimize execution based on the user application. Fractal does assume that the `filter` function is antimonotonic, *i.e.*, once a subgraph e is filtered out, no subgraph containing e will pass the filter. This assumption allows Fractal executions to terminate without checking every subgraph in S_g while maintaining equivalent semantics (though it renders any non-antimonotonic applications difficult to implement, as non-antimonotonic filters have to be implemented by the user within the `aggregate` UDF).

AutoMine. AutoMine has only one method for directly accessing subgraphs of the graph: the `match` function that returns a set of matches for a pattern. On Line 1, the user specifies a k-clique pattern and on Line 2 the call to `match` yields the k-clique matches in g. The remaining semantics of the application must be specified fully in user code. The user checks each clique to determine whether it is a canonical automorphism or a duplicate in order to narrow down $S = \mathcal{E}^*(g, p)$. The unique matches must then be aggregated by the user, who directly implements ν and $\oplus$.

Most graph mining applications are infeasible to execute on large data graphs in AutoMine without significant developer effort. Unlike Fractal which provides automatic dupli-

cate filtering and parallel aggregation utilities, AutoMine users must correctly and efficiently eliminate duplicates and aggregate matches. Crucially, the user code must also cope with the memory overheads of materializing the billions or trillions of k-clique matches that exist even in medium-sized graphs, and perform aggregations scalably. Requiring such effort undermines the purpose of a general-purpose graph mining system.

Furthermore, AutoMine has no view of the application semantics beyond the pattern passed as input to `match`. Since user programs interact little with system APIs, there is no opportunity for AutoMine to analyze the underlying application and optimize its execution accordingly. As a result, AutoMine is application-oblivious.

Other existing graph mining systems support the semantics of Eq. 2.1 in similar ways (*e.g.*, RStream [219] is designed similarly to Arabesque and Fractal, G-Miner [45] is similar to AutoMine), and are therefore also application-oblivious. In the next chapter, the thesis lays groundwork for application-aware graph mining by developing constructs for specifying S that can unify any structural filters on subgraphs and their neighbourhoods with existing subgraph isomorphism semantics.

Chapter 4

Using Patterns to Specify Interesting Subgraphs

A key component of the graph mining application semantics given by Eq. 2.1 is the set S of interesting subgraphs the application processes. Specifying S can be thought of as filtering S_g based on either *structural* or *non-structural* properties of subgraphs. Structural properties refer to the presence or absence of edges, labels on edges and vertices, as well as the structure of the graph surrounding the subgraph. Non-structural properties are the application-specific meaning given to a given structure. For example, being isomorphic to a given pattern, being vertex-induced, possessing a given set of labels, or not being adjacent to a given vertex are all structural properties of subgraphs, while the sum of a subgraph's integer labels or the frequency of its pattern is a non-structural property.

As illustrated in Chapter 3, previous systems specify S using a mixture of configuration flags, user-defined functions, and system APIs to encode the set of desired subgraphs, resulting in opaque user programs and wasted computation. Instead of forcing users to define S in ad-hoc ways corresponding to the design of the underlying system, this thesis takes a principled approach, proposing graph patterns as the primary abstraction for specifying structural properties, including constraints on the edges, labels, and local neighbourhood of desired subgraphs. Viewing graph mining from a pattern-based perspective offers several benefits when designing application-aware systems.

1. *Transparent.* Patterns clearly describe subgraph structure without the need for opaque user-defined functions. This allows systems that can interpret patterns to directly generate matches instead of exploring unnecessary subgraphs that end up being filtered by user code.

2. *Flexible.* Patterns are a system-agnostic abstraction divorced from the processing model of the underlying graph mining backend. A pattern does not prescribe how the graph must be traversed to find its matches, it only describes the end result.

Thus, adopting patterns as the foundation for a system frontend affords complete freedom in designing an efficient backend to explore matches.

3. *Efficient.* By specifying S with patterns instead of user-defined functions, graph mining systems can exploit the well-established literature on fast subgraph matching given pattern inputs. For instance, in previous graph mining systems like Fractal [77] and AutoMine [146], indicating only subgraphs with a specific labeling are desired user-defined filtering subgraphs based on their patterns, incurring a per-subgraph isomorphism check. On the other hand, the subgraph matching literature uses the labels of the input graph pattern to guide its exploration, completely avoiding any subgraphs with undesirable labels and thereby eliminating the need for a filter [102, 175, 101].

However, patterns currently cannot express many useful structural properties of subgraphs in S. The existing definition of subgraph isomorphism used by graph mining systems (and the broader graph mining and subgraph matching literatures) does not account for other structural properties that are commonly used to distinguish desired subgraphs in graph mining applications. Namely, subgraph isomorphism is only concerned with the existence of edges, and cannot express the absence of edges. A common example of edge absence in graph mining is the vertex-induced constraint on subgraphs, which requires that all edges between the vertices of a subgraph to be present within the subgraph. Viewed from the negated perspective, this means any pair of vertices in the subgraph which are not adjacent must not be adjacent in the data graph: *i.e.*, there is an absence of an edge. Because this absence is not expressible in standard subgraph isomorphism, AutoMine [146] only returns vertex-induced subgraphs, with no option for edge-induced, while other systems choose between edge-induced and vertex-induced based on a configuration flag [205, 77].

To rectify these shortcomings, this chapter develops an extended definition of subgraph isomorphism that augments patterns with two novel constructs, *anti-edge* and *anti-vertex*, which encode the *lack* of an edge or vertex, respectively. Anti-edges and anti-vertices express filters on the structures and neighbourhoods of matches according to well-defined semantics that can be integrated with existing systems to increase application-awareness.

Notation. For clear exposition, we introduce notation to clearly distinguish anti-edges and anti-vertices from standard vertices and edges. Where $\mathcal{G}$ is the set of all finite graphs, let $\mathcal{G}_\star \supset \mathcal{G}$ also contain all finite graphs containing anti-vertices and/or anti-edges. Then consider a pattern $p \in \mathcal{G}_\star$. *Anti-vertices* are a proper subset of vertices written $V^-(p) \subset V(p)$, while the remaining *standard vertices* are written $V^+(p) = V(p) \setminus V^-(p)$. It is also convenient to distinguish edges with anti-vertex endpoints. Note that no edge ever has both endpoints as anti-vertices. Write the set of edges with standard vertex endpoints as

$$E^{V+}(p) = \{(u, v) \in E(p) : u, v \in V^+(p)\},$$

and the set of edges with one anti-vertex endpoint as

$$E^{V^-}(p) = \{(u, v) \in E(p) : u \in V^-(p) \vee v \in V^-(p)\}.$$

Then *anti-edges* are a proper subset of edges with standard vertex endpoints, $E^-(p) \subset E^{V+}(p)$, and the remaining *standard edges* are written $E^+(p) = E^{V+}(p) \setminus E^-(p)$. As a final piece of notation, the *standard subgraph* of p consisting of only standard vertices and standard edges is written p^+.

Overview. Section 4.1 and Section 4.2 introduce formal semantics for anti-edges and anti-vertices, respectively. Then Section 4.3 defines extended subgraph isomorphism. Finally, Section 4.4 concludes with a discussion on the impact of extended subgraph isomorphism on application-awareness. For many of the same reasons that anti-edge and anti-vertex are useful for graph mining, they are useful in graph databases and subgraph matching systems and can be generalized. To demonstrate, Appendix A generalizes anti-vertex to different graph models and subgraph matching definitions.

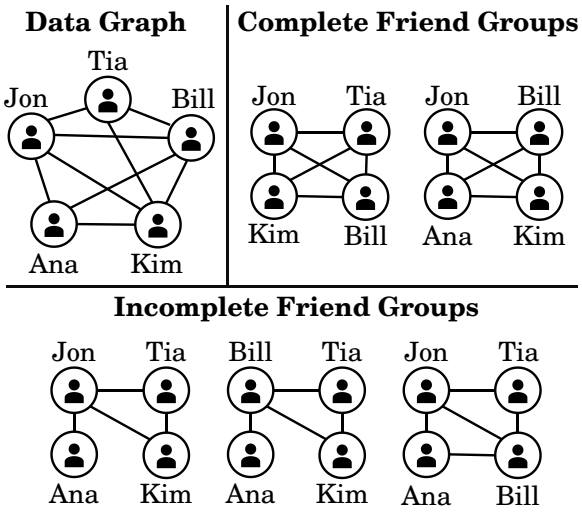
4.1 Anti-Edge: Concept and Semantics

This section develops the concept of the anti-edge. An anti-edge is a special edge indicating *non-adjacency* of a pair of vertices in a match, *i.e.*, expressing the absence of an edge. This allows for local structural filters on subgraphs to be specified transparently using graph patterns instead of configuration flags or user-defined code. Section 4.1.1 provides use cases motivating the anti-edge construct, and Section 4.1.2 formalizes semantics for the anti-edge constraint. Anti-edges in patterns are visualized by a dashed line connecting two vertices.

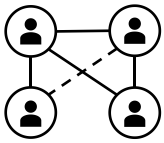
4.1.1 Absence of Edges

In this section, we present a motivating use case for structural filters encoding the absence of edges in subgraphs.

Example 4.1.1. Consider the following Friend Recommendation use case. Link prediction in social networks [137] is a common application of graph analytics. In a friendship graph for a social network service, where vertices represent people and edges represent friendship, it is desirable to recreate real-life associations between users as accurately as possible in order to better target advertisements and other services. To this end, the service recommends users who are *not adjacent* in the friendship graph, but who seem likely to be friends in real life to connect on their platform. One way to measure this likelihood is by observing the overlap between subgraphs representing incomplete friend groups where some pairs of users are not adjacent but have friends in common [138].



(a) A social network service recommends friendship between non-adjacent users who have common friends.



(b) Likely friend group pattern: anti-edge denotes non-friends.

Figure 4.1: Friend recommendation use case.

Figure 4.1 shows a dense friendship graph, where Jon, Bill, and Kim are friends with every other user. Ana and Tia are not friends with each other, but they are both friends with Jon, Bill, and Kim. The pattern shown in Figure 4.1b captures established friend groups with three users where at least one user has an additional friend not currently in the group. Due to the anti-edge, cliques in the friendship graph are excluded from results, as there are no missing links to be predicted. On the other hand, both tailed triangles and chordal cycle patterns are included. In previous systems, these semantics can only be expressed through a user-defined filter that checks subgraphs for the existence of edges.

4.1.2 Formal Anti-Edge Semantics

Anti-edges in a pattern encode constraints on the structure of its matches, specifying which vertices should not be adjacent. Let $g \in \mathcal{G}$ be a graph and $p \in \mathcal{G}_\star$ be a pattern. A match $m \in \mathcal{E}(g, p^+)$ matching the standard edges and vertices of p satisfies the anti-edge constraint if and only if

$$\forall (u, v) \in E^-(p), (m(u), m(v)) \notin E(g)$$

The two vertices connected by an anti-edge are called *anti-adjacent*. These semantics enable complex structural filters to be expressed simply: if a match does not satisfy the anti-edge constraint, it can be pruned from exploration.

Edge-Induced and Vertex-Induced Patterns. As discussed in Section 2.1, depending on the mining use case, desired subgraphs are either *edge-induced* or *vertex-induced*. For

example, Frequent Subgraph Mining (FSM) relies on edge-induced subgraphs, whereas Motif Counting requires vertex-induced subgraphs. Similarly, matches of a pattern can be thought of as edge-induced or vertex-induced, depending on the existence of additional edges in the graph.

Whereas previous systems presented a dichotomy between vertex-induced and edge-induced exploration [205, 77], or only allowed one form of exploration [146], under extended subgraph isomorphism the vertex-induced requirement is expressed directly through anti-edges. Specifically, the following result shows how edge-induced and vertex-induced patterns are related.

Theorem 4.1.1. *Let $g \in \mathcal{G}$ be a graph. Let p^E be a pattern without any anti-edges, and let p^V be a pattern with the same vertices and edges as p^E, such that every pair of vertices in p^E that are not adjacent are anti-adjacent in p^V. Then the vertex-induced matches of p^E in g are equal to the edge-induced matches of p^V in g:*

$$\{s \in \mathcal{E}_\star(g, p^E) : s \text{ is vertex-induced}\} = \{s \in \mathcal{E}_\star(g, p^V) : s \text{ is edge-induced}\}.$$

Proof. To prove equivalence of the two sets, we first show that every edge-induced match of p^V is a vertex-induced match of p^E, and then we show that every vertex-induced match of p^E is an edge-induced match of p^V.

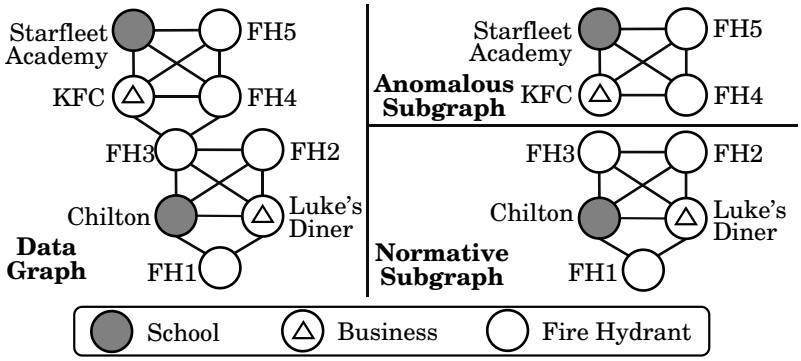
Let m be an edge-induced match for p^V. Observe that as p^E and p^V contain the same edges and vertices, m is also a match for p^E. By definition of p^V, for any edge $(u, v) \notin E(p^E)$, there is an anti-edge constraint between u and v in p^V. Since m is a match for p^V, it satisfies this anti-edge constraint, such that $m(u)$ and $m(v)$ are not adjacent in the data graph. This means there is an edge between $m(u)$ and $m(v)$ if and only if $(u, v) \in E(p^E)$. Therefore, m is a vertex-induced match of p^E.

Conversely, let m be a vertex-induced match for p^E. Since m is isomorphic to p^E, it contains all edges of p^E. Furthermore, m is vertex-induced, so if a pair of pattern vertices u_1, u_2 in p^E are not adjacent, then the corresponding data vertices $m(u_1)$ and $m(u_2)$ are not adjacent either. Hence, m satisfies the anti-edge constraint for u_1, u_2. As this holds for all pairs of non-adjacent vertices in p^E, m is also a match for p^V. □

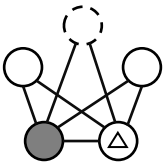
Theorem 4.1.1 unifies edge-induced and vertex-induced exploration, such that no binary choice between the two is necessary. Anti-edges can be added to graph patterns in order to directly specify the desired subgraphs.

4.2 Anti-Vertex: Concept & Semantics

This section develops the concept of the anti-vertex. An anti-vertex is a vertex in the pattern that indicates absence of a vertex in the resulting subgraph. Anti-vertices allow users to express constraints on the neighbourhoods of subgraph vertices declaratively, simply by

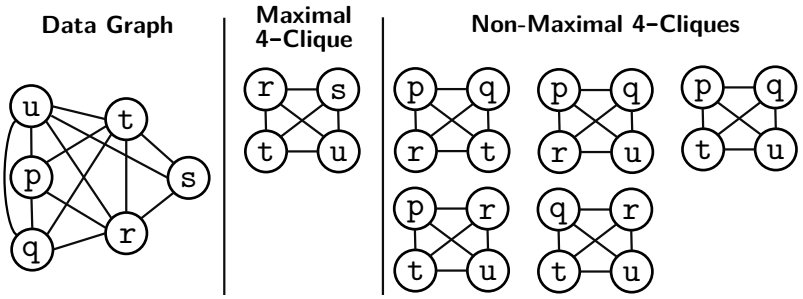


(a) The anomalous subgraph of interest is the one where the school and the business are connected with two fire hydrants and not three.

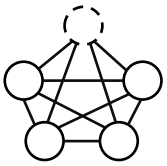


(b) Anomaly pattern: third fire hydrant marked as anti-vertex.

Figure 4.2: Anomaly detection use case.



(a) Maximal cliques use case. Only clique r-s-t-u is a maximal 4-clique since all other 4-cliques are part of larger 5-clique p-q-r-t-u.



(b) Maximal 4-clique pattern: the anti-vertex filters 5-cliques.

Figure 4.3: Maximal cliques use case.

describing which vertices are undesirable. Section 4.2.1 provides use cases to motivate the need for the anti-vertex construct, and Section 4.2 formalizes semantics for the anti-vertex constraint.

4.2.1 Constraints on Match Neighbourhoods

In this section, we present motivating use cases that constrain the subgraphs of interest based on their neighborhoods. These use cases can employ anti-vertices in the pattern to declaratively specify the absence of vertices in a subgraph. To easily visualize an anti-vertex in a pattern, anti-vertices are pictorially represented as vertices with dashed borders as opposed to solid borders used for regular vertices.

Example 4.2.1. Consider the use cases below.

1. *Anomaly Detection.* Identifying anomalies in graph data [164] is crucial across various domains. Certain anomalies are identified as subgraphs that have missing vertices from a reference (normative or non-anomalous) subgraph [79]. Figure 4.2a shows an example of a city planning scenario. One of the planning requirements is if there is a school and

a business close to each other, then there must be at least three fire hydrants nearby that are useful for both locations. For the graph shown in Figure 4.2a, the subgraph with Chilton and Luke's Diner satisfies the allocation requirement because of fire hydrants FH1, FH2 and FH3. However, the subgraph with Starfleet Academy and KFC is anomalous since there are only two fire hydrants FH4 and FH5.

Finding subgraphs with exactly two fire hydrants is not straightforward. In a naïve solution, the user first finds all subgraphs with two fire hydrants, then implements a user-defined filter to check each subgraph for a third fire hydrant. Instead, *the absence of a third fire hydrant neighbor* can be directly expressed using an anti-vertex. To find anomalous subgraphs with two fire hydrants, a normative subgraph can easily be turned into a pattern by marking a fire hydrant as an anti-vertex. Figure 4.2b shows the pattern containing an anti-vertex that exactly returns the anomalous subgraph containing FH4, FH5 without returning the normative subgraph involving FH1, FH2, FH3. The anti-vertex (indicated by the dotted border) requires that any school and business matched by the query do not have a third fire hydrant in their neighborhood. In this case, the anti-vertex provides a declarative way to express a constraint on the shared neighborhood of nearby schools and businesses.

2. *Maximal Cliques.* Finding and enumerating maximal cliques is a popular graph mining problem, with applications in social network analysis, financial analysis, security and biology [54]. In Figure 4.3a, the clique r-s-t-u is maximal, but the other cliques are not maximal since they all can be extended into the larger clique p-q-r-t-u by adding a vertex. Since the maximality constraint simply limits the vertices in cliques to not have a common neighboring vertex, *the absence of this common neighboring vertex* can be directly expressed using an anti-vertex. It suffices to express a clique of size $k+1$ and mark one of the vertices as an anti-vertex. As shown in Figure 4.3b, the pattern contains an anti-vertex connected to all the vertices of a 4-clique. This eliminates all the non-maximal 4-cliques from the result set, while still returning the maximal r-s-t-u clique.

3. *Approximate Subgraph Matching.* Approximate subgraph matching often allows optional and forbidden vertices and edges [232] to provide a loose subgraph template for which subgraphs are matched. As subgraphs get matched for approximate templates, identifying which subgraphs result due to the vertices being optional requires adding a constraint for the vertex to be absent. Such a constraint indicating *absence of vertices* can be achieved using anti-vertices.

4. *Contrasting Quasi-Cliques.* Recent research [13] on mining for multigraphs argues the strength of finding collection of vertices that are dense in one graph but less connected in a second graph. An interesting sub-case is mining contrasting quasi-cliques where the

sparser subgraph is fully imposed on a subset of vertices, i.e., remaining vertices are not connected to the subgraph. Here, *the isolated vertices from the rest of the subgraph* can be represented using anti-vertices.

The above examples showcase the need for easily expressing absence of vertex connections in the neighborhood of explored subgraphs. A well-defined anti-vertex construct would also enable thorough reasoning about correctness as well as methodical exploration of useful optimizations.

4.2.2 Formal Anti-Vertex Semantics

As demonstrated in Example 4.2.1, the anti-vertex is a declarative construct that allows users to express constraints simply in terms of which results are not desired. Next, the formal semantics of anti-vertices are discussed.

Denoting Vertex Neighbourhoods. To specify the anti-vertex constraint concisely, it is convenient to use a shorthand for the vertices in the neighbourhood of a match vertex. Let g and h be graphs. For $v \in V(g),\ e \in E(h),\ u \in V(h)$, the (e, u)-*neighbourhood* of v is the set of vertices in $V(g)$ which contain the labels of u and are adjacent to v via edges like e. Formally, we write the (e, u)-neighbourhood of v as

$$\begin{aligned} N(v, e, u) = \{v' \in V(g) : (v, v') \in E(g) \\ \wedge W_g(e) = W_g((v, v')) \\ \wedge L_h(u) \subseteq L_g(v')\}. \end{aligned}$$

Semantics. Anti-vertices encode an additional requirement on matches, namely that an anti-vertex should not be possible to match. Suppose m is a subgraph matching all edges and standard vertices of a pattern $p \in \mathcal{G}_\star$. Let $C : V^-(p) \to \mathbb{P}(V(g))$ be a function which returns the set of data vertices that can be mapped to by m from a query anti-vertex. The match m is valid only if

$$\forall \overline{u} \in V^-(p), C(\overline{u}) = \varnothing.$$

Hence, an anti-vertex will invalidate a match if there are vertices in the data graph which can be mapped to it.

The definition of C depends on the underlying subgraph matching semantics (homomorphism, no-repeated-edge, isomorphism). To ensure the semantics of anti-vertices is consistent with the matching semantics, we define C by adhering to the requirements on m which allow or disallow multiple different pattern vertices/edges to be mapped to the same vertices/edges. Here, we use isomorphism semantics as they are most pertinent for graph mining.

Let p be a pattern, g be the data graph, and m be a match for p in g. In isomorphism semantics, m must be injective with respect to both edges and vertices, with no repetition. Hence, data vertices already mapped to by m (and implicitly, the edges incident on those vertices) cannot fulfill the anti-vertex requirement to invalidate the match. C is defined as follows.

$$C(\overline{u}) \quad = \bigcap_{v:(\overline{u},v)\in E^+(p)} N(m(v),(\overline{u},v),\overline{u}) \setminus m(V(p))$$

Applicability. These semantics are system-agnostic, and can be adapted to different matching semantics, graph models, and even workloads. Appendix A.1.1 defines C for homomorphism and no-repeated-edge matching semantics, and generalization of anti-vertex to the property graph model can be found in Appendix A.1.2. Finally, Appendix A.2 integrates anti-vertex with the popular Cypher [86] query language commonly used in graph database management systems [161].

4.3 Extended Subgraph Isomorphism

With formal semantics for anti-edge and anti-vertex, this section extends the notion of subgraph isomorphism to encompass structural filters and neighbourhood constraints.

For graphs $p \in \mathcal{G}_\star, g \in \mathcal{G}$, an extended subgraph isomorphism $\phi_\star : V(p) \to V(g)$ is a subgraph isomorphism that preserves anti-edge and anti-vertex constraints. Formally, an extended subgraph isomorphism $\phi_\star$ satisfies the following properties:

$$\begin{aligned}
&\phi_\star(p^+) \in \mathcal{E}(g^+, p^+),\\
&\forall (u,v) \in E^-(p), (\phi_\star(u), \phi_\star(v)) \notin E(g),\\
&\forall \overline{u} \in V^-(p), C(\overline{u}) = \emptyset.
\end{aligned}$$

The first property states that every extended subgraph isomorphism is also a standard subgraph isomorphism from p^+ to g^+, *i.e.*, it produces a match of p^+ in g^+. The latter properties enforce that the match generated by $\phi_\star$ meets the anti-vertex and anti-edge constraints developed earlier. The set of matches obtained through extended subgraph isomorphisms is $\mathcal{E}_\star(g,p) \subseteq \mathcal{E}(g,p^+)$.

Automorphisms. Since a match $m \in \mathcal{E}_\star(g,p)$ is a subgraph of g, which has no anti-edges or anti-vertices, its automorphisms are as defined in Section 2.1. On the other hand, it is important to note that not every automorphism of m is necessarily a match of p under extended subgraph isomorphism. The set of unique matches for p in g under extended subgraph isomorphism is written $\mathcal{E}^*_\star(g,p) \subseteq \mathcal{E}_\star(g,p)$ and consists of one representative per automorphism group contained in $\mathcal{E}_\star(g,p)$.

4.4 Conclusion

This chapter proposed formal semantics for using patterns to encode constraints on the local structure and neighbourhood of subgraphs, in the form of the *anti-edge* and *anti-vertex* constructs. Anti-edge and anti-vertex are *declarative*, *i.e.*, they are agnostic to any given system or algorithm, describing the behaviour of filters without prescribing how the filters must be enforced by a system backend. Anti-edges resolve the tension between edge-induced and vertex-induced exploration as separate execution modes in previous systems, because vertex-induced subgraphs are matches of patterns where every pair of vertices is either adjacent or anti-adjacent. Therefore, a system that is aware of anti-edges only needs one execution mode to explore edge-induced and vertex-induced subgraphs. Meanwhile, anti-vertices enable reasoning about the neighbourhoods of matches without requiring user code to manually inspect adjacency lists of match vertices.

By supporting the extended definition of subgraph isomorphism incorporating anti-edges and anti-vertices, system backends become more application-aware than previously possible. Application-specific structural properties of desired subgraphs hitherto implemented in opaque user-defined functions become transparent, facilitating the development of novel optimizations as well as reuse of existing research on subgraph matching. Developing constructs and semantics for expressing non-structural properties or non-local structural properties of subgraphs, such as constraints on matches beyond their immediate neighbourhoods, are left for future work.

In the next chapter, the thesis applies the application-aware design philosophy, including extended subgraph isomorphism semantics, to develop an efficient graph mining system, Peregrine.

Chapter 5

Peregrine: Application Semantics in Pattern-Based Systems

As discussed in Chapter 3, at the heart of existing graph mining systems is an exploration engine that exhaustively searches subgraphs of the graph, and a series of filters that prune the search space to continue exploration for only those subgraphs that are of interest (e.g., ones that match a specific pattern) and are unique (to avoid redundancies coming from structural symmetries). The exploration happens in a step-by-step fashion where small subgraphs are iteratively extended based on their connections in the graph. As these subgraphs are explored, they are verified via *canonicality checks* to guarantee uniqueness, and are analyzed via *isomorphism computations* to understand their structure (or pattern). After that, the subgraphs are either pruned out because they don't match the pattern of interest, or forwarded down the pipeline where their information is aggregated at the pattern level.

While such an exploration process is general enough to compute different mining use cases including Frequent Subgraph Mining and Motif Counting, we observe that it remains largely oblivious to the patterns that are being mined. Hence, state-of-the-art graph mining systems face three main issues, as described next: (1) These systems perform a large number of unnecessary computations; specifically, every subgraph explored from the graph, even in intermediate steps, is processed to ensure canonicality, and is analyzed to either extract its pattern or to verify whether it is isomorphic to another pattern. Since the exploration space for graph mining use cases is very large, performing those computations on every explored subgraph severely limits the performance of these systems. (2) The exhaustive exploration in these systems ends up generating a large amount of intermediate subgraphs that need to be held (either in memory or on disk) so that they can be extended. While systems based on breadth-first exploration [205, 219] demand high memory capacity, other systems like Fractal [77] and AutoMine [146] use guided exploration strategies to reduce this impact; however, because they are not fully pattern-aware, they process a large number of intermediate subgraphs which severely limits their scalability as graphs grow large. (3) The programming model in these systems is strongly tied to the underlying exploration

	Arabesque	Fractal	G-Miner	RStream	PRG-U
Peregrine	2-1317×	1.1-737×	3-131×	2-2016×	2-42×

Table 5.1: Peregrine performance summary. PRG-U indicates Peregrine without symmetry breaking, to model systems that are not fully pattern-aware (e.g., AutoMine).

strategy, which makes it difficult for domain experts to express complex mining use cases. For example, subgraphs containing certain pairs of strictly disconnected vertices (i.e., absence of edges) are useful for providing recommendations based on missing edges; mining such subgraphs with constraints on their substructure cannot be directly expressed in any of the existing systems.

In this chapter, we take a 'pattern-first' approach towards building an efficient, application-aware graph mining system. We develop Peregrine[†], a pattern-aware graph mining system that directly explores the subgraphs of interest while avoiding exploration of unnecessary subgraphs, and simultaneously bypassing expensive computations (isomorphism and canonicality checks) throughout the mining process. Peregrine incorporates a pattern-based programming model that enables easier expression of complex graph mining use cases, and reveals patterns of interest to the underlying system. Using the pattern information, Peregrine efficiently mines relevant subgraphs by performing two key steps. First, it analyzes the patterns to be mined in order to understand their substructures and to generate an exploration plan describing how to efficiently find those patterns. And then, it explores the data graph using the exploration plan to guide its search and extract the subgraphs back to user space.

Our pattern-based programming model treats *graph patterns* as first class constructs: it provides basic mechanisms to load, generate and modify patterns along with interfaces to query patterns in the data graph. Furthermore, Peregrine supports the extended subgraph isomorphism semantics introduced in Chapter 4 to express advanced structural constraints on patterns to be matched. This allows users to directly operate on patterns and express their analysis as 'pattern programs' on Peregrine. Moreover, it enables Peregrine to extract the semantics of patterns which it uses to generate efficient exploration plans for its pattern-aware processing model.

We rely on theoretical foundations from existing subgraph matching research [93, 33] to generate our exploration plans. Since Peregrine directly finds the subgraphs of interest, it does not incur additional processing over those subgraphs throughout its exploration process; this directly results in much lesser computation compared to the state-of-the-art graph mining systems. Moreover, Peregrine does not maintain intermediate partial subgraphs in memory, resulting in much lesser memory consumption compared to other systems.

[†]Peregrine source code: https://github.com/pdclab/peregrine

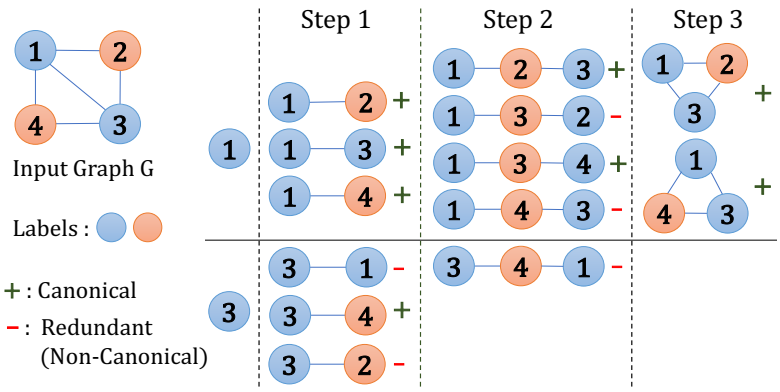


Figure 5.1: Step-by-step exploration in graph mining systems starting at vertex 1 and vertex 3. In total, 13 partial matches get explored and 13 canonicality checks are performed that prune out 5 partial matches. Isomorphism checks are performed on the remaining 8 matches for applications like FSM.

Peregrine runs on a single machine and is highly concurrent. We demonstrate the efficacy of Peregrine by evaluating it on several graph mining use cases including frequent subgraph mining, motif counting, clique finding, pattern matching (with and without structural constraints), and existence queries. Our evaluation on real-world graphs shows that Peregrine running on a single 16-core machine outperforms state-of-the-art distributed graph mining systems including Arabesque [205], Fractal [77] and G-Miner [45] running on a cluster with eight 16-core machines; and significantly outperforms RStream [219] running on the same machine. Furthermore, Peregrine could easily scale to large graphs and complex mining tasks which could not be handled by other systems. Table 5.1 summarizes Peregrine's performance.

5.1 Issues with Graph Mining Systems

While several graph mining systems have been developed [205, 77, 45, 219, 146], they are not pattern-aware. Hence, they demand high computation power and require large memory (or storage) capacity, while also lacking the ability to easily express mining programs at a high level.

5.1.1 Performance

(A) High Computation Demand. Graph mining systems explore subgraphs in a step-by-step fashion by starting with an edge and iteratively extending it depending on the structure of the data graph. Since they do not analyze the structure of the pattern to guide their exploration, they perform a large number of: (a) unnecessary explorations; (b) canonicality checks; and, (c) isomorphism checks.

Figure 5.1 shows an example of step-by-step exploration starting from vertex 1 and vertex 3. In step 1, both the vertices get extended generating 6 partial matches each of size 1 (edges). These are tested for canonicality which prunes out $(3, 1)$ and $(3, 2)$ (non-canonical matches are marked with $-$). For applications like FSM, isomorphism checks are performed on each of the canonical matches to identify their structure and compute metrics. Then, the remaining 4 matches progress to the next step and the entire process repeats. While explorations get pruned via both canonicality and isomorphism checks, every valid partial match is extended to multiple matches which may no longer be valid; generation of intermediate matches which do not result into valid final matches is unnecessary. Furthermore, all intermediate partial matches (unnecessary and valid matches) are operated upon to identify their structure (i.e., isomorphism check) and to verify their uniqueness (i.e., canonicality check). In our example, 13 intermediate matches are generated, 5 of which are unnecessary; 13 canonicality checks and 8 isomorphism checks are performed. If these checks are not performed at every step (as done in Fractal [77] by delaying its `filter` step), a massive amount of partial and complete matches that do not contribute to final result would get generated.

We verified the above behavior by profiling graph mining systems on clique counting and motif counting applications. As shown in Figure 5.2a and Figure 5.2b, on Patents [100] (a real-world graph dataset), RStream [219] and Arabesque [205] generate over a billion partial matches for clique counting while the total number of cliques is only ∼3.5M (∼99.7% matches were unnecessary); similarly for motif counting, RStream generates over 40 billion partial matches (∼99.2% unnecessary) and Arabesque generates over 685 million partial matches (∼52% unnecessary). They also perform a large number (hundreds of millions to billions) of canonicality checks and isomorphism checks. Since Fractal [77] explores in depth-first fashion, its numbers are better than RStream and Arabesque; however, they are still very high.

(B) High Memory Demand. Graph mining systems often hold massive amounts of (partial and complete) matches in memory and/or in external storage. Systems based on step-by-step exploration require valid partial matches so that they can be extended in subsequent steps; the total size (in bytes) required by all matches (partial and complete) quickly grows (often beyond main memory capacity) as the size of the pattern or data graph increases. Such a memory demand is lower in DFS-based exploration (as done in Fractal [77]). For clique and motif counting in Figure 5.2, Arabesque consumes ∼101GB main memory while Fractal requires ∼32GB memory.

5.1.2 Programmability

Programming in graph mining systems is done at vertex and edge level, with semantics of constructing the required matches defined explicitly by user's mining program. This

System	Total Matches	Canonicality Computations	Isomorphism Computations
RStream	1.2B (342×)	33.0M	0
Arabesque	1.4B (400×)	1.4B	3.5M
Fractal	659.0M (188×)	599.6M	0

(a) Profiling results for 4-Clique Counting on Patents [100] which contains ∼3.5M cliques of size 4. Isomorphism counts are 0 for RStream and Fractal due to their native support for clique computation.

System	Total Matches	Canonicality Computations	Isomorphism Computations
RStream	40.1B (125×)	40.1B	343.3M
Arabesque	685.8M (2.1×)	685.8M	320.7M
Fractal	665.6M (2.1×)	649.1M	320.7M

(b) Profiling results for 3-Motif Counting on Patents [100] which contains ∼320M 3-sized motifs.

Figure 5.2: Number of matches explored (partial and full), canonicality checks performed, and isomorphism checks performed by RStream [219], Arabesque [205] and Fractal [77]. Numbers in brackets indicate the magnitude of matches explored relative to result size.

means, mining programs expressed in those systems contain the logic for: (a) validating partial and complete matches; (b) extending matches via edges and/or vertices; and, (c) processing the final valid matches. As the size of subgraph structure to be mined grows, the complexity of validating partial matches increases, making mining programs difficult to write. For example, the multiplicity algorithm to avoid over-counting in AutoMine [146] cannot be used if the user wants to enumerate patterns, which leaves the responsibility of identifying unique matches to the user. Furthermore, complicated structural constraints beyond the presence of vertices, edges and labels cannot be easily expressed in any of the existing systems.

5.2 Overview of Peregrine

We develop a pattern-aware graph mining system that directly finds subgraphs of interest without exploring unnecessary matches while simultaneously avoiding expensive isomorphism and canonicality checks throughout the mining process. We do so by designing a pattern-based programming model that treats *graph patterns* as first class constructs, and by developing a processing model that uses the pattern's substructure to guide the exploration process.

Pattern-based Programming. In Peregrine, graph mining tasks are directly expressed in terms of subgraph structures (i.e., graph patterns). Our pattern-aware programming model allows declaring (statically and dynamically generated) patterns, modifying patterns, and performing user-defined operations over matches explored by the runtime. This allows concisely expressing mining programs by abstracting out the underlying run-

time details, and focusing only on the substructures to be explored. Moreover, we introduce two novel abstractions, *anti-edges* and *anti-vertices*: an anti-edge enforces strict disconnection between two vertices in the match whereas an anti-vertex captures strict absence of a common neighbour among vertices in the match. These abstractions allow users to easily express advanced structural constraints on patterns to be mined.

Automatic Generation of Exploration Plan. With patterns of interest directly expressed, Peregrine analyzes the patterns and computes an *exploration plan* which is later used to guide the exploration in the data graph. Specifically, the pattern is first analyzed to eliminate symmetries within itself so that expensive canonicality checks during exploration can be avoided. Then the pattern is reduced to its *core substructure* that enables identifying matches using simple graph traversals and adjacency list intersection operations without performing explicit isomorphism checks.

Guided Pattern Exploration. After the exploration plan is generated, Peregrine starts the exploration process using our pattern-aware processing model. The exploration process matches the core substructure of the pattern to generate partial matches using recursive graph traversals in the data graph. As partial matches are generated, they are extended to form final complete matches by intersecting the adjacency lists of vertices in the partial matches. Since the entire exploration is guided by the plan generated from the pattern of interest, the exploration does not require intermediate isomorphism and canonicality checks for any of the partial and complete matches that it generates. This reduces the amount of computation done in Peregrine compared to state-of-the-art graph mining systems. Moreover, since matches are recursively explored and instantly extended to generate complete final results, partial state is not maintained in memory throughout the exploration process which significantly reduces the memory requirement for Peregrine.

Finally, we reduce load imbalance in Peregrine by enforcing a strict matching order based on vertex degrees. Furthermore, we incorporate on-the-fly aggregation and early termination features to provide global updates as mining progresses so that exploration can be stopped once the conditions required to compute final results are met.

5.3 Peregrine Programming Model

Since graph mining fundamentally involves finding subgraphs that satisfy certain structural properties, we design our programming model around *graph patterns* as first class constructs. This allows users to easily express the subgraph structures of interest, without worrying about the underlying mechanisms of how to explore the graph and find those structures. With such a declarative style of expressing patterns, Peregrine enables users to program complex mining queries as operations over the matches. The clear separation of *what to*

```
[L1] Set<Pattern> loadPatterns(string filename);
[G1] Set<Pattern> generateAllEdgeInduced(int size);
[G2] Set<Pattern> generateAllVertexInduced(int size);
[S1] Pattern generateClique(int size);
[S2] Pattern generateStar(int size);
[S3] Pattern generateChain(int size);
[C1] Set<Pattern> extendByEdge(Set<Pattern> patterns);
[C2] Set<Pattern> extendByVertex(Set<Pattern> patterns);
```

```
class Pattern {
  Set<Vertex> getNeighbours(Vertex u);
  Label getLabel(Vertex u);
  bool areAdjacent(Vertex src, Vertex dst);
  bool areAntiAdjacent(Vertex src, Vertex dst);
  void addEdge(Vertex src, Vertex dst);
  void addAntiEdge(Vertex src, Vertex dst);
  void removeEdge(Vertex src, Vertex dst);
  void markAntiVertex(vertex u);
  void addLabel(Vertex u, Label l);
  . . .
};
```

Figure 5.3: Peregrine Pattern Interface.

find and *what to do with the results* helps users to quickly reason about correctness of their mining logic, and develop advanced mining-based analytics.

We first present how patterns are directly expressed in Peregrine, and then show how common graph mining use cases can be programmed with patterns in Peregrine.

5.3.1 Peregrine Patterns

Figure 5.3 shows our API to directly express, construct and modify connected graph patterns. Patterns can be constructed statically and loaded using `[L1]`, or can be constructed dynamically `[G1-G2, C1-C2, S1-S3]`. `[G1]` and `[G2]` generate all unique patterns that can be induced by certain number of edges and vertices respectively. `[S1-S3]` generate special well-known patterns. `[C1-C2]` take a group of patterns as input, and extend one of them by an edge or a vertex, to return all of the unique new patterns that result from these extensions. This allows constructing patterns step-by-step which is useful to perform guided exploration. The `Pattern` class provides a standard interface to access and modify the pattern graph structure.

In most common applications, the edges and vertices in the pattern graph are sufficient to specify S. For advanced mining use cases that require structural constraints within the pattern, Peregrine supports adding anti-edges and anti-vertices to patterns for extended subgraph isomorphism semantics from Chapter 4. Consequently, due to Theorem 4.1.1, our pattern-based programming doesn't need to separately define edge-induced and vertex-induced exploration strategies, as done in pattern-unaware systems [205, 77, 219].

The next section discusses non-structural constraints and operations on matches, which are handled outside of the pattern graph in a user-defined function.

```
void updateSupport(Match m) { mapPattern(m.getDomain()); }
bool isFrequent(Pattern p, Domain d) {
  return (d[p].support() >= threshold);
}
DataGraph g = loadGraph("labeledInput.graph");
Set<Pattern> patterns = generateAllEdgeInduced(2);
while (patterns not empty) {
  Map<Pattern, Domain> results = match(g, patterns, updateSupport);
  Set<Pattern> frequentPatterns = results.filter(isFrequent).keys();
  patterns = extendByEdge(frequentPatterns);
}
```

(a) Frequent Subgraph Mining

```
int numTriplets = 0;
void countAndCheck(Match m) {
  int numTriangles = loadAggregatedValue(m);
  if (numTriangles*3/numTriplets > bound)
    stopExploration();
  else
    mapPattern(m, 1);
}

DataGraph g = loadGraph("input.graph");
Pattern wedge = generateStar(3);
numTriplets = 2*count(g, wedge);

Pattern triangle = generateClique(3);
Map<Pattern, int> result = match(g, triangle, countAndCheck);
```

(b) Global Clustering Coefficient Bound

```
DataGraph g = loadGraph("input.graph");
void output(Match m) { write(m); }
Pattern p = loadPattern("pattern.txt");
match(g, p, output);
```

(c) Subgraph Matching

```
DataGraph g = loadGraph("input.graph");
Pattern p =
    generateClique(desiredSize);
int result = count(g, p);
```

(d) Clique Counting

```
DataGraph g = loadGraph("input.graph");
Set<Pattern> patterns = generateAllVertexInduced(size);
Map<Pattern, int> result = count(g, patterns);
```

(e) Motif Counting

```
void found(Match m) {
  mapPattern(m, True);
  stopExploration();
}
DataGraph g = loadGraph("input.graph");
Pattern p = generateClique(desiredSize);
Map<Pattern, bool> result = match(g, p, found);
```

(f) Clique Existence

Figure 5.4: Graph mining use cases in Peregrine's pattern-aware programming model.

5.3.2 Pattern-Aware Mining Programs in Peregrine

Figure 5.4 shows Peregrine programs for motif counting, frequent subgraph mining (FSM), clique counting, pattern matching, an existence query for global clustering coefficient bound, and an existence query for k-sized clique. All the programs first express patterns by dynamically generating them or by loading them from external source. Then they invoke Peregrine engine to find (`match()`) and process matches of those patterns. For every match for the pattern, user-defined function (e.g., `updateSupport()`, `countAndCheck()`, `found()`, etc.) gets invoked to perform desired analysis. The `count()` function is a syntactic sugar and is equivalent to `match()` with a function that increments a counter. Most of the programs are straightforward; we discuss FSM and existence queries in more detail.

FSM: Anti-Monotonicity & Label Discovery

FSM leverages anti-monotonicity in support measures (discussed in Section 2.2.1). Peregrine natively provides MNI support computation where it internally constructs the *domain* of patterns, *i.e.*, a table mapping vertices in g to those in p (similar to [2]). After exploration ends for a single iteration, the support measure maintained by Peregrine can be directly used to prune infrequent patterns using a threshold, as shown in Figure 5.4a, and only the remaining frequent patterns are then programmatically extended to be explored.

Before finding the first small frequent labeled patterns, the FSM program has no information about which labelings are frequent. Peregrine provides dynamic label discovery by starting with unlabeled (or partially labeled) patterns as input and returning labeled matches. Hence, the FSM program in Figure 5.4a first starts with unlabeled patterns of size 2, and discovers frequent labeled patterns. It then iteratively extends the frequent labeled patterns with unlabeled vertices to discover frequent labeled patterns of larger sizes.

Existence Queries

Existence queries allow quickly verifying whether certain structural properties hold within a given data graph. Peregrine allows dynamically stopping exploration when the required conditions get satisfied.

Figure 5.4b shows a Peregrine program to verify if the global clustering coefficient [128] of graph g is above a certain bound. The global clustering coefficient is the ratio of three times the number of triangles and the number of triplets (all connected subgraphs with three vertices, including duplicates) in g. The number of triplets is equal to twice the number of edge-induced 3-star matches since the endpoints of a 3-star are symmetric. Hence, the program quickly computes the number of 3-stars, and then starts counting triangles. During exploration, if the number of triangles reaches the requisite number to exceed the bound, exploration stops immediately.

```
ExplorationPlan generatePlan(Pattern p) {
  partialOrders = breakSymmetries(p);
  vc = minConnectedVertexCover(p);
  p_C = vertexInducedSubgraph(vc, p);
  matchingOrders = computeMatchingOrders(p_C, partialOrders);
  return (p_C, partialOrders, matchingOrders);
}
```

Figure 5.5: Computing exploration plan.

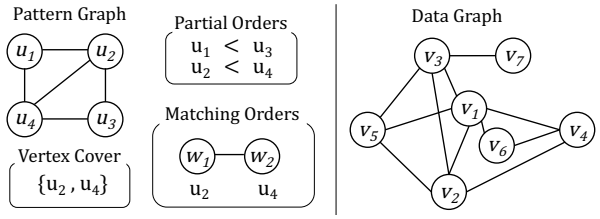


Figure 5.6: Example of a pattern graph and a data graph.

Figure 5.4f shows PEREGRINE program to check whether a clique of a certain size is present in g. As soon as the exploration finds at least one match, it stops and returns `True`.

5.4 Pattern-Aware Matching Engine

PEREGRINE is pattern-aware, and hence, it directly finds patterns in any given data graph. In this section, we discuss our core pattern matching engine that directly finds canonical subgraphs from a given vertex in the data graph. In Section 5.5, we will use this engine to build PEREGRINE. For simplicity, we assume the data graph and the pattern are unlabeled.

5.4.1 Directly Matching A Given Pattern

To avoid the overheads of a straightforward exhaustive search, we develop our pattern matching solution based on well-established techniques [93, 33, 123]. Since patterns are much smaller than the data graph, we analyze the given pattern to develop an exploration plan. This plan guides the data graph exploration to ensure generated matches are unique up to automorphism.

Figure 5.5 shows how the exploration plan is computed from a given pattern p. First, to avoid non-canonical matches we break the symmetries of p by enforcing a partial ordering on matched vertices [93]. For our example pattern in Figure 5.6, we obtain the partial ordering $u_1 < u_3$ and $u_2 < u_4$.

In the next step, we compute the core of p (called p_C) as the subgraph induced by its minimum connected vertex cover[†]. Given a match m for p_C, all matches of p which contain

[†]A connected vertex cover is a subset of connected vertices that covers all edges.

m can be computed from the adjacency lists of vertices in m. In our example, p_C is the subgraph induced by u_2 and u_4.

To simplify the problem of matching p_C, we generate matching orders to direct our exploration in the data graph. A matching order is a graph representing an ordered view of p_C. The vertices of the matching order are totally-ordered such that the partial ordering of $V(p)$ restricted to $V(p_C)$ is maintained. This allows matching p_C by traversing vertices with increasing vertex ids without canonicality checks.

We compute matching orders by enumerating all sequences of vertices in p_C that meet the partial ordering, and for each sequence we create a copy of p_C where the id of each vertex is remapped to its position in the sequence. Then, we discard duplicate matching orders. For our example pattern (Figure 5.6), its core substructure has only one valid vertex sequence, $\{u_2, u_4\}$, so we obtain only one matching order. Note that there can be multiple matching orders for a given p_C depending on the partial orders. We call the i^{th} matching order p_{Mi}.

Thus, to match p_C it suffices to match its matching orders p_{Mi}. A match for p_{Mi} results in 1 match for p_C per valid vertex sequence. In our example, a match for p_{M1}, say $\{v_2, v_3\}$, is converted to a single match for p_C, $v_2 \to w_1 \to u_1, v_3 \to w_2 \to u_2$.

It is important to note that the exploration plan is generated by analyzing the pattern graph only, *i.e.*, all the computations explained above are applied on p (and its derivatives). Hence, exploration plans are computed quickly (often in less than half a millisecond).

5.4.2 Matching Under Extended Subgraph Isomorphism

In this section, we describe how anti-edge and anti-vertex constraints described in Chapter 4 are handled by the Peregrine matching engine, how each construct interacts affects the pattern core, and finally how symmetries are broken in their presence.

Matching Anti-Edges. To enforce an anti-edge constraint, we perform a set difference between the adjacency lists of its endpoints. Ex. 1 in Figure 5.7 shows an example pattern p with an anti-edge and a data graph g. If x matches a and w matches b, the candidates for d are the elements of adj(w) $\setminus$ adj(x).

To perform the set difference, we need to ensure that one of the vertices of the anti-edge is already matched so that its adjacency list is available. Hence, when computing the vertex cover we also cover the anti-edge by including one of its endpoints.

Matching Anti-Vertices. The anti-vertex constraint can only be verified after the common neighbours of an anti-vertex's neighbour have been matched. Thus, we perform the check after all standard vertices are already matched.

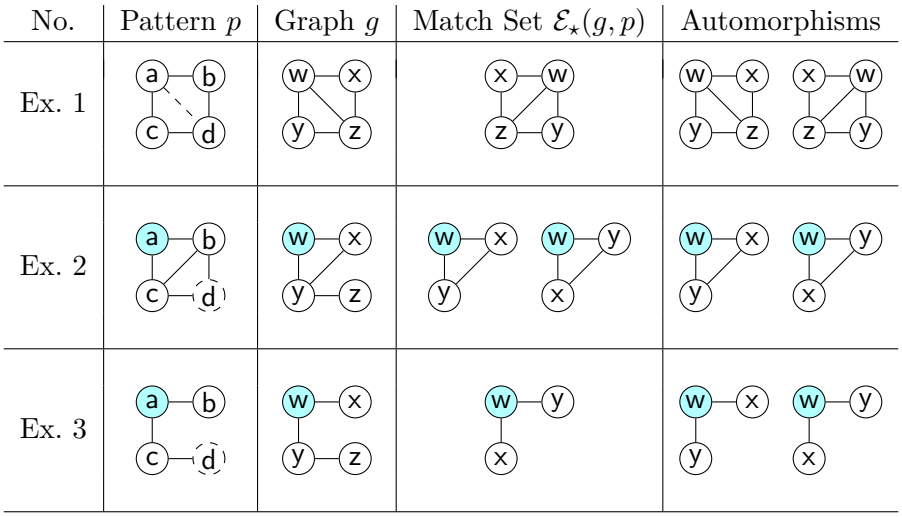


Figure 5.7: Anti-Edge and Anti-Vertex Examples.

For example, consider Ex. 2 in Figure 5.7, with anti-vertex d. If w, x, y in g match a, b, c respectively, then we verify the anti-vertex constraint for d as follows:

$$(\text{adj(x)} \setminus \{\text{y}\}) \cap (\text{adj(y)} \setminus \{\text{x}\}) = \varnothing$$

In this case, since x and y do not have a common neighbour, w-x-y and w-y-x are valid matches. On the other hand, in Ex. 3, the anti-vertex d disallows matching y in g with c, since adj(y) contains a valid candidate z. Since anti-vertices do not participate while matching regular vertices and edges, we keep the pattern core p_C the same as the core pattern when anti-vertices are removed.

Symmetry-Breaking with Anti-Edges & Anti-Vertices. Given the definition of automorphism in Section 4.3, it may seem that anti-edges and anti-vertices should be ignored when generating partial orders on standard vertices in a pattern p. However, ignoring anti-edges and anti-vertices can lead to the situation where only part of a given match's automorphism group is a valid match under extended subgraph isomorphism. This is undesirable because the partial orders generated without knowledge of anti-edges or anti-vertices may prune the only automorphisms that are valid matches.

Example 5.4.1. Consider the following examples where ignoring anti-edges or anti-vertices, respectively, results in missing the desired match.

1. *Anti-edges.* Consider Ex. 1 in Figure 5.7 and suppose anti-edges are ignored for the purposes of symmetry breaking. Suppose the partial ordering on p is $\text{a} < \text{b} < \text{d}$, and $\text{a} < \text{c}$. Then note that the only match in $\mathcal{E}_\star(g,p)$ violates this ordering, as $\text{x} > \text{w}$. The other automorphism of that match satisfies the partial ordering but fails the anti-edge constraint since w and z are adjacent.

2. *Anti-vertices.* Consider Ex. 3 in Figure 5.7 and suppose anti-vertices are ignored for the purposes of symmetry breaking. If the partial ordering on p is chosen to be $\mathsf{b} < \mathsf{c}$, then the sole match violates it, since $\mathsf{y} > \mathsf{x}$. Similar to the anti-edge example, the other automorphism satisfies the partial ordering but violates the anti-vertex constraint since y and z are adjacent.

Therefore, to avoid the above situations, PEREGRINE treats anti-edges and anti-vertices as standard edges and vertices for the purposes of symmetry breaking, but with a placeholder label to distinguish them from the original standard edges/vertices. In this manner, anti-edges and anti-vertices disrupt the symmetries between standard edges and vertices and thereby relax the partial orders. Continuing the previous examples, this means in Ex. 1 the partial ordering on p becomes $\mathsf{a} < \mathsf{d}$ and $\mathsf{b} < \mathsf{c}$, which the match satisfies. Likewise, in Ex. 3, the partial ordering on p vanishes, since b and c are no longer symmetric since d is only adjacent to c, preventing the match from being pruned.

Crucially, including anti-edges when generating partial orders does not affect vertex-induced patterns. As the following result shows, despite containing anti-edges, vertex-induced patterns end up with the same partial ordering as their edge-induced counterparts.

Theorem 5.4.1. *Let p^E be an edge-induced pattern with no anti-edges, and let p^V be the vertex-induced pattern obtained by adding anti-edges to every pair of vertices in p^E which are not adjacent. Then p^E and p^V have the same partial ordering.*

Proof. Partial orders are generated by breaking the symmetries of a pattern: adding constraints between pattern vertices until only one automorphism of the pattern remains. By treating anti-edges in p^V as standard edges with a different labeling, the symmetry breaking algorithm considers the automorphisms of p^V to preserve anti-edge relationships as well as edge relationships. We prove the theorem by showing that the symmetry breaking algorithm views p^E and p^V as having the same automorphisms.

Remark that due to the difference in labeling, anti-edges can never be mapped to standard edges by one of these automorphisms, and vice versa. Thus, the automorphisms of p^V according to the symmetry breaking algorithm consist of the automorphisms of the pattern formed by the standard edges of p^V, *i.e.*, p^E, intersected with the automorphisms of the pattern formed by the anti-edges of p^V, *i.e.*, the complement of p^E[†]. But it is well-known that a graph and its complement have the same automorphism group [26]. Therefore, the symmetry breaking algorithm will view the automorphisms of p^V as the automorphisms of p^E. □

[†]The complement of a graph g is a graph h such that two distinct vertices of h are adjacent if and only if they are not adjacent in g. To generate the complement of g, edges are added between every pair of non-adjacent vertices, and all edges that previously existed are removed.

5.4.3 Neighbourhood Groups

We observe that sets of non-core vertices with identical neighbourhoods exhibit useful properties that further enable Peregrine to avoid redundant computation and reduce the match enumeration depth. Peregrine collects such vertices into *neighbourhood groups*, which it leverages for several important optimizations. For example, p_a has one neighbourhood group $\{u_1, u_3\}$, while in p_b there are two neighbourhood groups $\{u_4\}$ and $\{u_5\}$, as the non-core vertices are adjacent to different core vertices.

Candidate Sets per Neighbourhood Group

Since the vertices in a neighbourhood group have the same core neighbours, they also have the same candidate matches. In p_a, the non-core vertices vertices u_1 and u_3 are both adjacent to u_2 and u_4, and hence have the same candidate set. Peregrine computes candidate sets per neighbourhood group instead of per non-core vertex to avoid performing duplicate operations for each member of a neighbourhood group.

Reducing Match Enumeration Depth

We make an important observation about the vertices within a neighbourhood group. Namely, the vertices in a neighbourhood group are symmetric to each other. We exploit these symmetries to efficiently enumerate matches instead of the traditional DFS enumeration process that iteratively maps the candidates for each non-core vertex and backtracks.

In unlabeled patterns, the partial ordering of the pattern restricted to a neighbourhood group is a total ordering. For example, the neighbourhood group $\{u_1, u_3\}$ is totally ordered due to the condition $u_1 < u_3$. In labeled patterns, each subset of the neighbourhood group vertices with the same labels will be totally ordered. For ease of explanation, we use the term *match group* to refer to a totally ordered subset of a neighbourhood group in labeled patterns, and an entire neighbourhood group in unlabeled patterns.

This allows us to break up enumeration into several mostly independent stages. As the elements of a match group are ordered and there are no orderings between match groups, we can map the elements of an individual match group quickly in tight loops with few branches. Importantly, vertices in separate neighbourhood groups are not symmetric, so there are no checks for partial orders across groups. The only dependence outside a match group is to avoid re-mapping data vertices that have already been matched in m. We proceed to map one match group at a time in depth-first manner. By using match groups, the depth of exploration is reduced from the number of non-core vertices to the number of match groups.

5.4.4 Match Groups & Fast Paths

With match groups enabled in Peregrine, we taxonomize patterns into different classes based on the number of match groups they contain. By doing so, fast paths can be devel-

oped for different classes to skip certain depth-first exploration steps. PEREGRINE currently incorporates two fast paths, one for the common case of a single match group, and another for the common case of two match groups. From our example patterns, p_a follows the former fast path, and p_b follows the latter.

Single Match Group

When there is a single match group containing k vertices, there is no need for any further depth-first exploration. It remains only to enumerate all unique k-tuples from a single vector. We can also skip checking whether vertices in m are present in the candidate set for the match group, since all core vertices must be adjacent to the members of the match group, otherwise there would be more than one.

When the graph mining use case only requires the number of pattern instances, the count can be computed in constant time as $\binom{|A|}{k}$ where A is the candidate set for the match group.

Two Match Groups

When there are two match groups, enumeration requires a Cartesian product of the sets of unique tuples representing matches for vertices in each group, subtracting any overlap. For example, consider p_b. Each of its non-core vertices represents a separate match group. Suppose u_4 and u_5 have candidate sets A and B, respectively. Then the matches for the non-core vertices are precisely the pairs:

$$A \times B \setminus \{(v, v) : v \in A \cap B\}$$

Even though this requires an additional set intersection and an additional set difference to account for overlaps, directly computing this set is much faster than a general depth-first exploration.

To count the matches instead of enumerating, PEREGRINE simply computes the cardinality of the set directly. For example, the cardinality of the above set is simply $|A| \cdot |B| - |A \cap B|$.

Three+ Match Groups

The approach for two match groups generalizes to larger numbers of match groups as well. However, the number of additional set operations required to remove overlaps grows combinatorially with the number of match groups. In a pattern with k match groups, it requires $\sum_{i=2}^{k} \binom{k}{i}$ additional set intersections and just as many set differences. This leads to diminishing returns after two match groups, and therefore when there are three or more match groups PEREGRINE simply traverses them in depth-first fashion as described previously.

```
void matchFrom(Match m, Pattern p, Func f, MatchingOrder mo, PartialOrder
    po, int i) {
  if (i > |V(p_C)|) {
    // remaps m as in Section 5.4.1, before completing it
    // and invoking the user's callback f()
    completeMatch(m, p, f, po, 1);
  } else {
    for (v in getExtensionCandidates(mo, po, i)) {
      matchFrom(m+v, p, f, mo, po, i+1);
    }
  }
}

AggregationVal match(Graph g, Pattern p, Func f) {
  Aggregator a;
  (p_C, partialOrder, matchingOrders) = generatePlan(p);
  parallel for (v in g) {
    for (matchingOrder in matchingOrders) {
      matchFrom({v}, p, f, matchingOrder,
                partialOrders, 1);
    }
  }
  return a.result();
}
```

Figure 5.8: Pattern-Aware Processing Model.

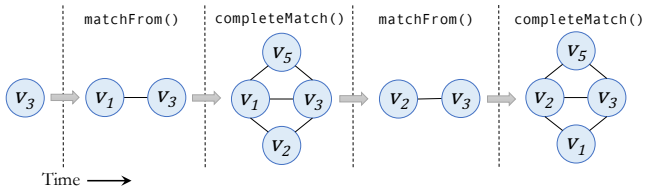


Figure 5.9: Pattern-guided exploration in Peregrine for pattern and data graph in Figure 5.6 with matching order high-to-low.

5.5 Peregrine: Pattern-Aware Mining

We will now discuss how Peregrine performs pattern-aware mining using the matching engine presented in Section 5.4.

5.5.1 Pattern-Aware Processing Model

Mining in Peregrine is achieved by matching patterns starting from each vertex and invoking the user function to process those patterns. Hence, a task in Peregrine is defined as the data vertex where the matching process begins. As shown in Figure 5.8, each mining task takes a start vertex and the exploration plan generated in Section 5.4 (matching orders, partial orders, pattern core p_C). From the starting vertex, we recursively match vertices in the matching order. At each recursion level, a data vertex is matched to a matching order

vertex. To avoid non-canonical matches, we maintain sorted adjacency lists and use binary search to generate candidate sets comprised only of vertices that meet the total ordering.

Once a matching order is fully matched, it is converted to matches for p_C. Matches for p_C are then completed by performing set intersections (for standard edges) and set differences (for anti-edges) on sections of adjacency lists that satisfy the partial orders. Each completed match is passed to a user-defined callback for further processing. Figure 5.9 shows a complete exploration example.

Note that our processing model doesn't incur expensive isomorphism and canonicality checks for every match in the data graph, while simultaneously avoiding mis-matches and only exploring subgraphs that match the given pattern. Furthermore, tasks in our processing model are independent of each other since explorations starting from two different vertices do not require any coordination. Threads dynamically pick up new tasks when they finish their current ones.

5.5.2 Early Pruning for Dynamic Load Balancing

While a matching order enforces a total ordering on the data vertices matching p_C, there is flexibility in the order in which its vertices are matched. To reduce the load imbalance across our matching tasks, we: (a) follow matching orders high-to-low, e.g. in our example in Figure 5.6 we match w_2 before w_1; and, (b) order vertices by their degree such that $v_i < v_j$ in the data graph if and only if $degree(v_i) \leq degree(v_j)$.

High-degree vertices have fewer neighbours with degrees higher than or equal to their own, so the degree-based ordering ensures that when a high-degree vertex is matched to w_2, only those few neighbours can be matched to w_1. Thus, explorations of neighbours with lower degrees are pruned. Note that the total number of matches generated remains the same; the high-to-low matching order traversal, along with degree-based vertex ordering, reduces the workload imbalance of matching across high-degree and low-degree vertices by dynamically pruning more explorations from high-degree tasks while enabling those explorations in low-degree tasks.

Finally, it is important to note that this process does not 'eliminate' workload imbalance simply because the mining workload is dynamic and depends on the pattern and data graphs. Hence, to avoid stragglers and maximize parallelism, we process tasks in the order defined by the degree of the starting vertex, beginning with the highest-degree vertices.

5.5.3 Early Termination for Existence Queries

For existence queries, Peregrine allows actively monitoring the required conditions so that the exploration process terminates as quickly as possible. When the matching thread observes the required conditions, the user function calls `stopExploration()` to notify other matching threads. Threads monitor their notifications periodically while matching, and

when a notification is observed, the thread-local values computed up to that point are aggregated and returned to the user.

5.5.4 On-the-fly Aggregation

Peregrine performs on-the-fly aggregation to provide global updates as mining progresses. This is useful for early termination and for use cases like FSM where patterns that meet the support threshold can be deemed frequent while matching continues for other patterns.

We achieve this using an asynchronous aggregator thread that periodically performs aggregation as values arrive from threads. The matching threads swap the global aggregation value with the local aggregation value and set a flag to indicate that new thread-local aggregation values are available for aggregation. The aggregator thread blocks until all thread-local aggregation values become available, after which it performs the aggregation and resets the flag to indicate that the global aggregation value is available. With this design, our matching threads remain non-blocking to retain high matching throughput.

5.5.5 Implementation Details

Peregrine is implemented in C++ where concurrent threads operate on exploration tasks, each starting at a different vertex in the data graph. The data graph is represented using adjacency lists, and the tasks are distributed dynamically using a shared atomic counter indicating the next vertex to be processed. To minimize coordination, threads maintain information regarding their exploration tasks, including candidate sets for each pattern vertex as exploration proceeds.

Peregrine provides native computation of support values for frequency-based mining tasks like FSM. Domains are implemented as a vector of bitmaps representing the data vertices that can be mapped to each pattern vertex. They are aggregated by merging their contents via logical-or. To scale to large datasets, we use compressed Roaring bitmaps [43], which are more memory efficient than dense bitmaps.

5.6 Evaluation

We evaluate the performance of Peregrine on a wide variety of graph mining applications and compare the results with the state-of-the-art general purpose graph mining systems [†]: Fractal [77], Arabesque [205], RStream [219] and G-Miner [45].

5.6.1 Experimental Setup

System. All experiments were conducted on `c5.4xlarge` and `c5.metal` Amazon EC2 instances. Most experiments use `c5.4xlarge`, with an Intel Xeon Platinum 8124M CPU

[†]We could not evaluate AutoMine [146] directly since its source code is not available.

G	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	$\lvert L(G)\rvert$	Max. Deg.	Avg. Deg.
(MI) MiCo [82]	100K	1M	29	96637	21.6
(PA) Patents [100]					
└— Unlabeled	3.7M	16M	—	793	10
└— Labeled	2.7M	13M	37	789	10
(YT) YouTube [55]	6.9M	44M	38	4039	12
(OK) Orkut [225]	3M	117M	—	33133	76
(FR) Friendster [225]	65M	1.8B	—	5214	55

Table 5.2: Real-world graphs used in evaluation.
'—' indicates unlabeled graph.

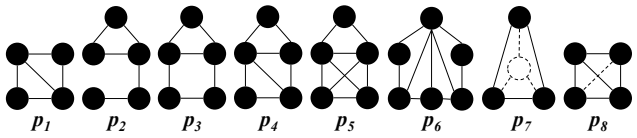


Figure 5.10: Patterns used in evaluation.

containing 8 physical cores (16 logical cores with hyper-threading), 32GB RAM, and 30GB SSD. Fractal (FCL), Arabesque (ABQ) and G-Miner (GM) were evaluated using both a cluster of 8 nodes (denoted by the suffix '-8'), as well as in single node configuration (denoted by the suffix '-1').

RStream was evaluated on a `c5.4xlarge` (RS-16) as well as a `c5.metal` (RS-96) equipped with an Intel Xeon Scalable Processor containing 48 physical cores (96 logical cores with hyper-threading), and 192GB RAM. Both instances were provisioned with a 500GB SSD.

In all performance comparisons, we ran Peregrine on a `c5.4xlarge`, and we used `c5.metal` to study Peregrine's scalability and resource utilization.

Datasets. Table 5.2 lists the data graphs used in our evaluation. MiCo (MI) is a co-authorship graph labeled with each author's research field. Patents (PA) is a patent citation graph. In the labeled version, each patent is labeled with the year it was granted. Youtube (YT) consists of videos crawled by [55] from 2007–2008, with edges between related videos. Videos are labeled according to their ratings, as in [77]. Orkut (OK) and Friendster (FR) are unlabeled social network graphs where edges represent friendships between users. MiCo and labeled Patents have been used by previous systems [205, 77, 219] to evaluate FSM while Orkut and Friendster were used by [45]. Except for FSM and labeled pattern matching, all experiments on Patents use its larger, unlabeled version.

Applications. We evaluated Peregrine on a wide array of applications: counting motifs with 3 and 4 vertices, labeled 3- and 4-motifs; counting k-cliques, for k ranging from 3 to 5; FSM with patterns of 3 edges on labeled datasets using various supports; matching the

patterns shown in Figure 5.10; and checking the existence of 14-cliques. We selected the patterns in Figure 5.10 to cover all the patterns used in [77] and [45]; note that patterns like triangles and empty squares are covered via applications like cliques and motifs. Since G-Miner's pattern matching program is specific to labeled p_2 (in Figure 5.10), we used labels on p_2 for all the systems to enable direct comparison. To match it on Orkut and Friendster graphs, which are unlabeled, we added synthetic labels (integers 1-6 as done in [45]) with uniform probability.

5.6.2 Comparison with Breadth-First Enumeration

Table 5.3 compares Peregrine's performance with Arabesque and RStream on motif-counting, clique-counting, and FSM (these systems do not support pattern matching). As we can see, Peregrine outperforms the breadth-first systems by at least an order of magnitude on every application except FSM. RStream, despite being an out-of-core system, runs out of memory during FSM computations because of the massive amounts of aggregation information, as well as during 4- and 5-cliques on MiCo where it could not handle the size of a single expansion step.

It was interesting to observe that Arabesque performed better in single-node mode compared to 8-node configuration across all experiments except FSM on Patents, where it ran out of memory. This is because its breadth-first exploration generates large amounts of partial matches which must be synchronized across the entire cluster between supersteps, incurring high communication costs that impact its scalability.

When support thresholds are high, Arabesque on 8 nodes computes FSM more quickly than Peregrine. This is because its breadth-first strategy leverages high parallelism when there are few frequent patterns to explore and aggregate. However, this approach is sensitive to the support threshold, which stops Arabesque from scaling to lower threshold values where there are more frequent patterns. In these scenarios Arabesque simply fails due to the memory burden of maintaining the vast amount of intermediate matches and aggregation values. We suspect that even with more main memory per node, the intermediate computations (canonicality, isomorphism, etc.) for each individual match in Arabesque would significantly limit its performance. Since Peregrine is pattern-aware, it only needs to maintain aggregation values for the patterns it is currently matching, allowing it to scale to inputs that yield many frequent patterns.

5.6.3 Comparison with Depth-First Enumeration

Table 5.4 compares Peregrine's performance with Fractal on motif-counting, clique-counting, FSM, and pattern matching. As we can see, Peregrine is faster than Fractal by at least an order of magnitude across most of the applications. For instance, 4-cliques on Patents finished in less than a second on Peregrine whereas Fractal took over 200 seconds in both cluster and single-node configurations.

App	G	Peregrine	Arabesque		RStream	
			ABQ-8	ABQ-1	RS-96	RS-16
3-Motifs	MI	**0.12**	158.05	39.05	51.83	252.74
	PA	**3.10**	870.70	525.49	2685.45	2186.93
	OK	**17.90**	—	—	/	/
	FR	**370.64**	—	—	/	/
4-Motifs	MI	**6.74**	—	—	/	/
	PA	**12.04**	—	—	/	×
	OK	**6156.10**	—	—	/	/
2K-FSM	MI	**380.81**	3418.25	821.60	×	—
3K-FSM	MI	**279.74**	3520.82	784.27	×	—
4K-FSM	MI	**250.68**	3514.97	779.75	×	—
20K-FSM	PA	**859.41**	—	—	1757.69	—
21K-FSM	PA	**647.97**	—	—	1711.87	—
22K-FSM	PA	507.56	**342.63**	—	1626.53	—
23K-FSM	PA	402.57	**299.12**	—	1936.92	—
3-Cliques	MI	**0.05**	18.62	5.98	7.34	11.32
	PA	**0.59**	155.55	87.26	8.40	11.97
	OK	**13.75**	—	—	986.20	1643.10
	FR	**296.99**	—	—	/	/
4-Cliques	MI	**2.02**	1598.09	353.37	266.61	—
	PA	**0.90**	249.38	107.02	105.00	181.30
	OK	**281.47**	—	—	/	/
	FR	**1337.77**	—	—	/	/
5-Cliques	MI	**89.60**	×	—	—	—
	PA	**1.12**	352.64	122.09	145.00	237.90
	OK	**3182.56**	—	—	/	/
	FR	**4214.72**	—	—	/	/

Table 5.3: Execution times (in seconds) for Peregrine, Arabesque [205] and RStream [219].
'×' indicates the execution did not finish within 5 hours.
'—' indicates the system ran out of memory.
'/' indicates the system ran out of disk space.

App	G	Peregrine	Fractal	
			FCL-8	FCL-1
3-Motifs	MI	**0.12**	22.13	17.11
	PA	**3.10**	231.95	214.34
	OK	**17.90**	—	—
	FR	**370.64**	—	—
4-Motifs	MI	**6.74**	78.66	420.67
	PA	**12.04**	362.19	742.35
	OK	**6156.10**	—	—
2K-FSM	MI	380.81	**154.47**	675.98
3K-FSM	MI	279.74	**154.74**	680.33
4K-FSM	MI	250.68	**144.34**	663.26
20K-FSM	PA	**851.41**	×	—
21K-FSM	PA	**647.97**	×	—
22K-FSM	PA	**507.56**	×	—
23K-FSM	PA	**402.57**	451.18	—
3-Cliques	MI	**0.05**	18.71	17.21
	PA	**0.59**	232.60	216.76
	OK	**13.75**	—	—
	FR	**296.99**	—	—
4-Cliques	MI	**2.02**	25.77	34.79
	PA	**0.90**	237.64	224.50
	OK	**281.47**	—	—
	FR	**1337.77**	—	—
5-Cliques	MI	**89.60**	181.30	904.65
	PA	**1.12**	266.88	217.30
	OK	**3182.56**	—	—
	FR	**4214.72**	—	—
Match p_1	MI	**0.12**	24.76	36.02
	PA	**0.84**	235.72	189.03
	OK	**38.97**	—	—
	FR	**824.62**	—	—
Match p_2	MI	**0.03**	22.11	16.85
	PA	**1.07**	260.15	202.23
	OK	**474.09**	—	—
	FR	**18.09**	—	—
Match p_3	MI	**19.93**	181.76	1288.94
	PA	**13.41**	30.18	69.33
	OK	**13292.77**	—	—
Match p_4	MI	**12.29**	120.99	789.81
	PA	**2.23**	25.58	21.63
	OK	**1569.73**	—	—
	FR	**7057.40**	—	—
Match p_5	MI	**14.94**	56.51	345.35
	PA	**1.89**	25.30	17.39
	OK	**1381.03**	—	—
	FR	**6726.51**	—	—
Match p_6	MI	**65.26**	×	×
	PA	**27.94**	210.04	205.39

Table 5.4: Execution times (in seconds) for Peregrine and Fractal [77].
'—' indicates the system ran out of memory.
'×' indicates the execution did not finish within 5 hours.

Given equal resources (i.e., on a single node), FSM on MiCo is up to 2.6× faster on Peregrine compared to that on Fractal. Furthermore, Peregrine scales to the larger dataset while Fractal does not. Even with 8 nodes, Fractal only outperforms Peregrine on the small MiCo graph, and cannot handle the Patents workload except for very high support thresholds, where there is less work to be done; there too, Peregrine executes faster than Fractal.

Similar to Arabesque, Fractal's pattern-unawareness requires it to maintain global aggregation values throughout its computation. In FSM, the aggregation values consume $O(|V|)$ memory per vertex in each pattern in the worst case, and thus quickly become a scalability bottleneck. On the other hand, Peregrine only needs to maintain aggregation values for the current patterns being matched, which allows it to achieve comparable performance and superior scalability while matching up to 15,817 patterns on MiCo and 6,739 patterns on Patents.

5.6.4 Comparison with Purpose-Built Algorithms

G-Miner is a general-purpose subgraph-centric system that targets expert users to implement the mining algorithms using a low-level subgraph data structure. Since expressing common mining algorithms requires domain expertise, we only evaluated the applications that are already implemented in G-Miner: 3-clique counting and pattern matching on p_2 (pattern matching for other patterns is not supported). This experiment serves to showcase how Peregrine compares to custom algorithms for matching specific patterns.

Table 5.5 compares Peregrine's performance with G-Miner. As we can see, Peregrine is 3× to 77× faster than G-Miner when counting 3-cliques even though G-Miner implements an algorithm designed specifically to count 3-cliques. When matching p_2, Peregrine is 6× to 131× faster on MiCo and Patents. On Orkut, however, G-Miner performs better on finding p_2; this is because G-Miner indexes vertices by labels when preprocessing the data graph, whereas Peregrine discovers labels dynamically. Due to these indexes, G-Miner could not handle Friendster even with 240GB disk space on the cluster.

App	G	Peregrine	G-Miner	
			GM-8	GM-1
3-Cliques	MI	**0.05**	3.79	3.86
	PA	**0.59**	7.91	8.93
	OK	**13.75**	44.26	62.65
	FR	**296.99**	/	/
Match p_2	MI	**0.03**	3.67	3.95
	PA	**1.07**	6.84	9.80
	OK	474.09	**145.00**	396.72
	FR	**18.09**	/	/

Table 5.5: Execution times (in seconds) for Peregrine and G-Miner [45]. '/' indicates the system ran out of disk space.

G	Existence 14-Clique	Anti-Vertex p_7	Anti-Edge p_8
MI	0.07	0.65	6.92
PA	3.95	0.67	1.69
OK	4.08	56.06	879.01
FR	50.39	470.21	4017.15

Table 5.6: Peregrine execution times (in seconds) for matching with an anti-vertex (p_7), matching with an anti-edge (p_8), and 14-clique existence query.

5.6.5 Mining with Constraints in Peregrine

We evaluate Peregrine on mining tasks with structural constraints. We match a pattern containing an anti-vertex (p_7), one containing an anti-edge (p_8), and perform an existence query of a 14-clique. The results are show in Table 5.6.

Mining with Anti-Vertices. Pattern p_7 expresses a maximal clique of size 3 (triangle) using a fully-connected anti-vertex, i.e., it matches all triangles that are not contained in a 4-clique. While satisfying the anti-vertex constraint requires computing set-intersections across all vertices of the triangle, Peregrine takes less than a minute on Orkut, and under eight minutes on the billion scale Friendster graph.

Mining with Anti-Edges. Pattern p_8 represents a vertex-induced chordal square using an anti-edge constraint. Satisfying the anti-edge constraint is computationally demanding, since it requires computing set differences of adjacency lists, which is twice as many operations as the sum of the adjacency list sizes. Nevertheless, Peregrine still easily completes it on all the datasets.

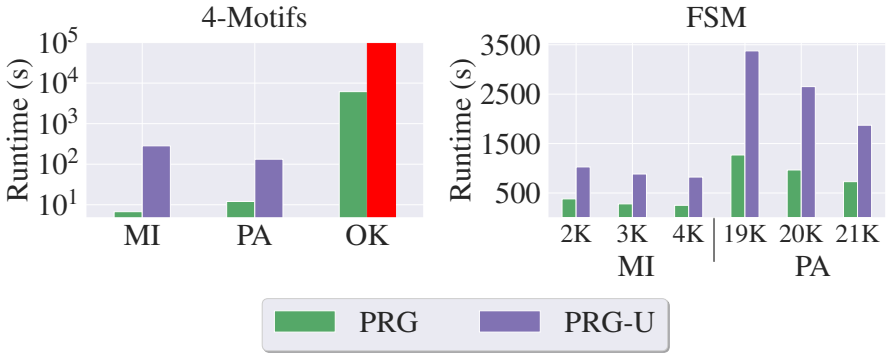


Figure 5.11: Execution times (in seconds) for Peregrine with (PRG) and without (PRG-U) symmetry breaking. PRG-U could not finish matching any of the 4-motif patterns on Orkut within 5 hours.

Existence Query. The goal of this query is to determine whether a 14-clique exists in the data graph. Peregrine stops exploration immediately after finding an instance of 14-clique. We observe that Patents and Orkut performed similarly; this is because the rarer the target pattern for an existence query, the longer it takes to find it. Patents does not contain a 14-clique, so the entire graph was searched, but in the much larger and denser Orkut graph, a 14-clique gets found quickly during exploration. Friendster is both large and sparse, and hence, 14-cliques are rare. Furthermore, since 14-clique is a large pattern, several partial explorations do not lead to a complete 14-clique.

5.6.6 Peregrine's Pattern-Aware Runtime

We evaluate the pattern-aware techniques in Peregrine as evidence for the impact of application-aware design. The following experiments use the patterns in Figure 5.12.

Benefits of Symmetry Breaking Symmetry breaking is a well-studied technique for subgraph matching that Peregrine uses to guide its graph exploration. However, recent systems like Fractal [77] and AutoMine [146] are not fully pattern-aware and do not leverage symmetry breaking for common graph mining use cases. We showcase the importance of symmetry breaking in Peregrine by disabling it and running 4-motifs and FSM with low support thresholds. These are expensive subgraph matching workloads: 4-motifs contains complex patterns with many matches and FSM involves a large number of patterns to match. Figure 5.11 summarizes the results.

We observe that symmetry breaking improves performance by an order of magnitude for 4-motifs on MiCo and Patents. Orkut 4-motifs without symmetry breaking did not even finish matching even a single size 4 pattern within 5 hours. This shows the importance of symmetry breaking when scaling to large patterns and large datasets. For instance, Orkut contains over 22 trillion *unique* vertex-induced 4-stars, and so without symmetry breaking,

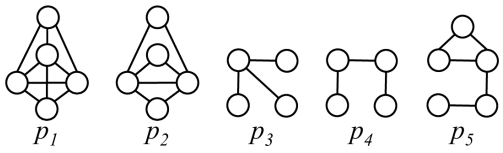


Figure 5.12: Patterns used in micro-benchmarks.

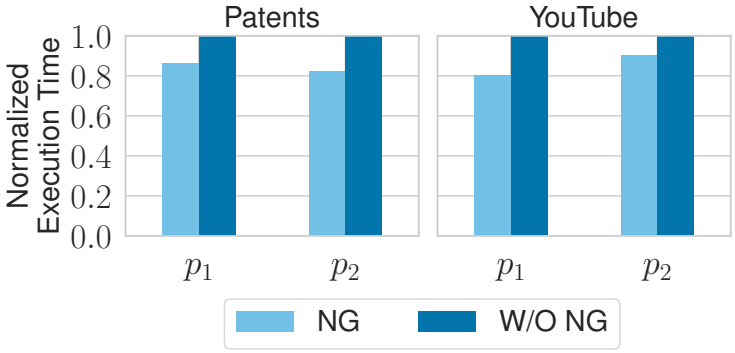


Figure 5.13: Execution times for pattern matching queries with neighbourhood groups (NG) and without neighbourhood groups (W/O NG). All times are normalized w.r.t. W/O NG.

the system must process six times that many matches (a 4-star's automorphisms are the permutations of its 3 endpoints: resulting in $3! = 6$ automorphic subgraphs).

FSM achieves 3× performance improvement through symmetry breaking. This is because with symmetry breaking, FSM's expensive aggregation values are only written to once per unique match in the data graph, whereas the naive approach without symmetry breaking would incur dozens of redundant write (and read) accesses per unique match.

Neighbourhood Groups. We measure the performance benefits achieved by neighbourhood groups. We run pattern matching queries with patterns p_1 and p_2 from Figure 5.10, because in both patterns all the non-core vertices are organized in a single neighbourhood group. Patterns without multiple vertices in a neighbourhood group remain unaffected. We run the queries on the Patents [100] and YouTube [55] graphs. Patents is a sparse graph with 2.7M vertices and 13M edges where each edge represents a citation between US patents. YouTube has 6.9M vertices and 44M edges, where each vertex represents a video and edges link related videos together. The experiments are performed on an Intel Xeon Gold 3.10GHz processor, using 16 physical cores with hyperthreading enabled and 32GB RAM.

Figure 5.13 shows the execution time for Peregrine with and without neighbourhood groups. We observe that Peregrine achieves 11-25% better performance when using neighbourhood groups. As expected, with neighbourhood groups enabled Peregrine performs 1.5-3× fewer set intersections to match the input patterns. This translates to huge savings:

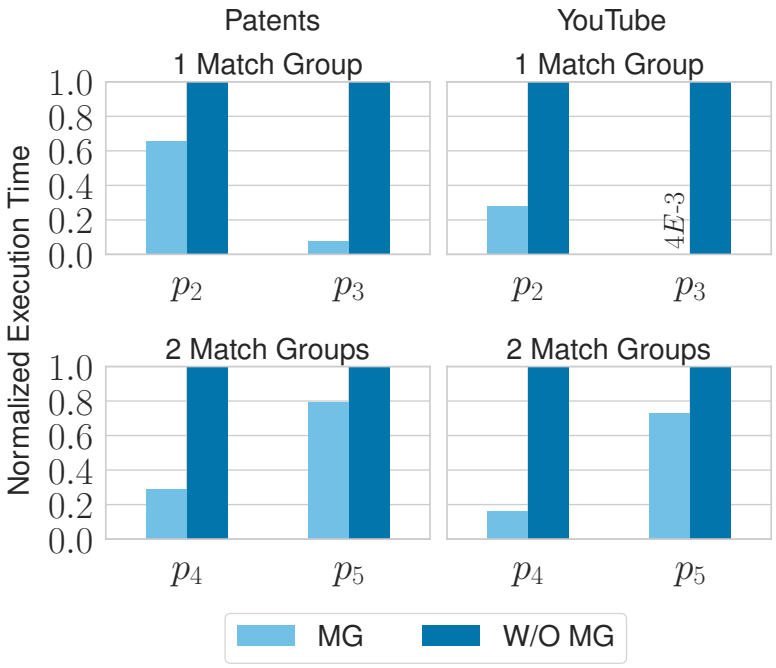


Figure 5.14: Execution time for pattern matching queries with match group fast paths (MG) and without match group fast paths (W/O MG). All times are normalized w.r.t. W/O MG.

for instance, PEREGRINE performs 152M fewer intersections when matching p_1 on YouTube when using neighbourhood groups.

Match Groups. To measure the impact of the fast paths for the two pattern types, we run pattern matching and 4-motif counting with and without the fast paths enabled. For the pattern matching queries we use two patterns p_2, p_3 which have a single match group, and two patterns p_4, p_5 with two match groups. Figure 5.14 shows the execution times. For a single match group, the fast path leads to a 1.5-236× speedup for p_2 and p_3, while the fast path for two match groups leads to a $1.25 - 6\times$ speedup for p_4 and p_5. On YouTube, the two match group fast path improves performance for p_4 by 6× and p_5 by 1.37× despite requiring 50M and 4B more set intersections for p_4 and p_5 respectively.

We also observe performance benefits for 4-motif counting, where 4 out of 6 patterns benefit from fast paths. Overall, 4-motif counting speeds up by $1.6 - 2.7\times$.

We profile the executions using `perf` to measure the reduction in branches due to the fast paths. Figure 5.15 shows the results. On the YouTube graph, all of p_2, p_3, p_4, p_5 incur on average 175× fewer branches during matching with the fast paths enabled, culminating in 34× fewer branch misses on average. Even though the 4-motif counting query contains patterns which do not benefit from the optimizations, it still performs $2 - 4.8\times$ fewer branches and 2.9B fewer mispredictions.

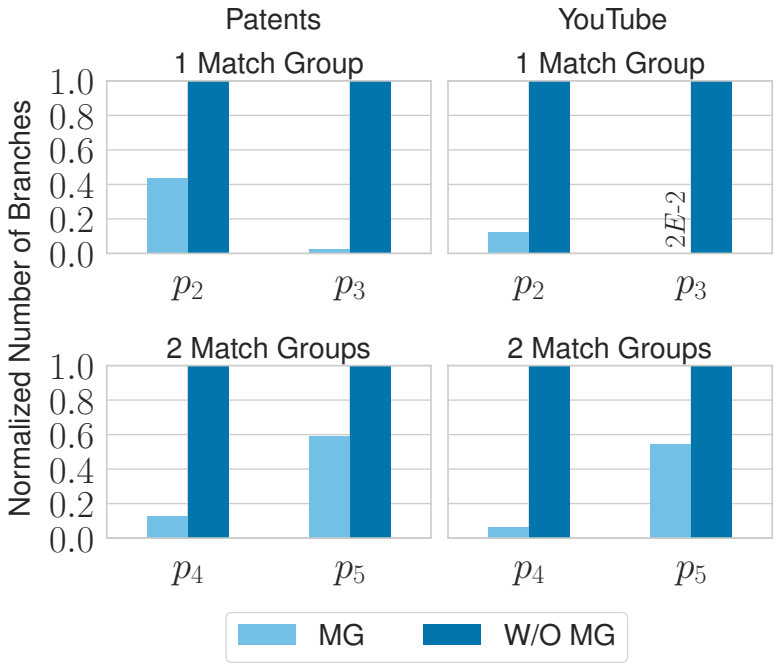


Figure 5.15: Number of branches for pattern matching queries with match group fast paths (MG) and without match group fast paths (W/O MG). All numbers are normalized w.r.t. W/O MG.

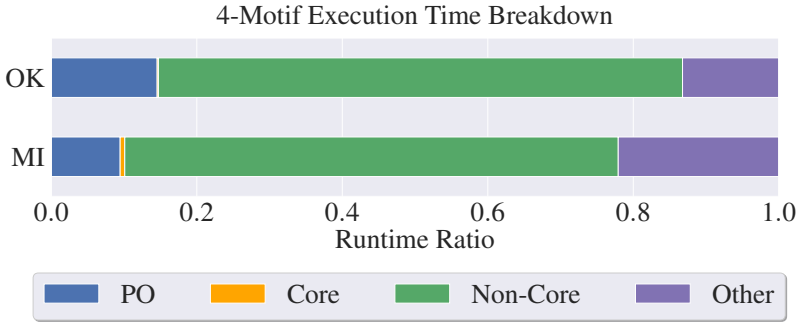


Figure 5.16: Peregrine 4-motif execution time breakdown on Orkut and MiCo.

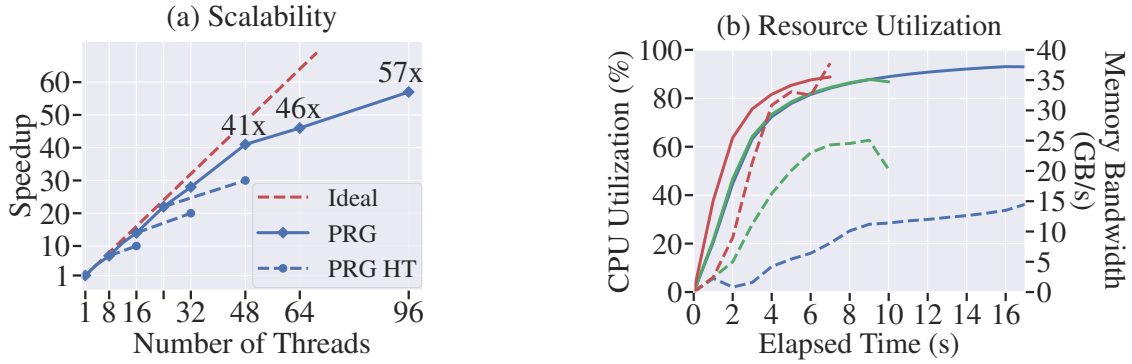


Figure 5.17: (a) Scalability (PRG HT = hyper-threaded). (b) CPU utilization (solid) and memory bandwidth (dashed) for 24 cores (blue), 47 cores (green) and 94 cores (red).

Breakdown on Mining Time. Figure 5.16 shows the ratio of time spent in each stage of matching during 4-motif execution: finding the range of sorted candidate sets that meet the pattern's partial order (PO), performing adjacency list intersections and differences to match the pattern core (Core) and finally, intersecting the adjacency lists of the pattern core to complete the match (Non-Core). Some time is also spent on the other requirements of matching, for example, fetching adjacency lists and mapping vertices (Other).

We observe that the majority of execution is spent intersecting adjacency lists of candidate vertices to complete matches. In comparison to the overall execution time, matching the core pattern is insignificant. This is because the core pattern is matched according to all valid total orderings of its vertices, and hence, the traversal is fully guided. In contrast, the non-core vertices may or may not be ordered with respect to each other, and with respect to the core vertices; so the runtime usually has less guidance when exploring the graph. Furthermore, in most patterns the core is small and involves fewer intersections than the non-core component.

5.6.7 System Characteristics

Scalability. We study Peregrine's scalability by matching pattern p_1 on Orkut using `c5.metal` instance. Note that we do not perform a COST analysis [149] with this experiment since we already compared Peregrine with optimized algorithms in Section 5.6.4, and state-of-the-art serial pattern matching solutions like [210, 102] performed much slower than our single threaded execution.

Figure 5.17a shows how Peregrine scales as number of threads increase from 1 to 96. As we can see, Peregrine scales linearly until 48 threads, after which speedups increase gradually. This is mainly because `c5.metal` has 48 physical cores, and scheduling beyond 48 threads happens with hyper-threading. We verified this effect by alternating how threads get scheduled across different cores; the dashed lines in Figure 5.17a show speedups when every pair of Peregrine threads is pinned to two logical CPUs on one physical CPU. As we can

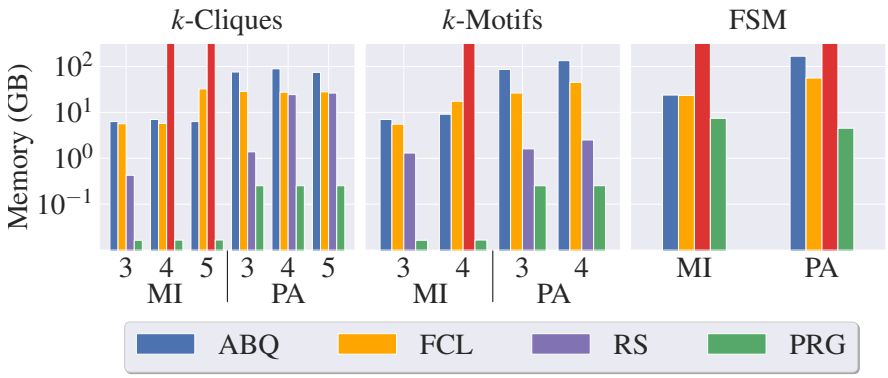


Figure 5.18: Peak memory usage of different systems across various applications. Tall red bars represent RStream out of memory errors.

see, with 48 threads but only 24 physical cores, Peregrine only achieves a 30× speedup, whereas with 48 physical cores it achieves a 41× speedup. Since pattern exploration involves continuous random memory accesses throughout execution, hyper-threading helps in hiding memory latencies only up to an extent. Figure 5.17b verifies this, as memory bandwidth grows considerably higher when using more cores, though CPU utilization remains similar.

We observe that speedups also decline slightly between 24 cores and 48 cores. This is because `c5.metal` has two NUMA nodes, each allocated to 24 physical cores. We measured remote memory accesses to observe the NUMA effects: when running on 48 cores, cross-numa memory traffic was 86GB as opposed to only 4.9MB when running on 24 cores.

Resource Utilization. Figure 5.17b shows CPU utilization and memory bandwidth consumed by Peregrine while matching p_1 on Orkut on `c5.metal` with 24, 47, and 94 threads. We reserve a core for profiling to avoid its overhead. We observe that Peregrine maintains high CPU utilization throughout its execution. The memory bandwidth curve increases over time; as high degree vertices finish processing, low degree vertices do less computation and incur more memory accesses as they get processed.

Figure 5.18 compares the peak memory usage for Peregrine and other systems. For distributed systems we report the sum of all nodes' peak memory. Peregrine consistently uses less memory than all the systems, mainly because of its direct pattern-aware exploration strategy. It is interesting to note that changing the pattern size in cliques and motifs does not impact Peregrine's memory usage. The usage is high for FSM compared to other applications due to large domain maps for support calculation.

Load Balancing. Since Peregrine threads dynamically pick up tasks as they become free, we observe near-zero load imbalance while matching p_1 across all our datasets. The difference between times taken by threads to finish all of their work was only up to 71 ms.

5.7 Conclusion

We presented Peregrine, a pattern-aware graph mining system that efficiently explores subgraph structures of interest, and scales to complex graph mining tasks on large graphs. Peregrine uses extended subgraph isomorphism to enable 'pattern-based programming' that treats *patterns* as first class constructs, including support for Anti-Edge and Anti-Vertex that express advanced structural constraints on patterns to be matched. This allows users to directly operate on patterns and easily express complex mining use cases as 'pattern programs' on Peregrine.

Peregrine's runtime integrates seemingly incompatible application-specific techniques thanks to its application-aware design philosophy. Our extensive evaluation showed that this runtime enables Peregrine to outperform the existing state-of-the-art by several orders of magnitude, even when using 8× fewer CPU cores. Furthermore, Peregrine successfully handles resource-intensive graph mining tasks on billion-scale graphs on a single machine, while the state-of-the-art fails even with a cluster of 8 such machines or access to large SSDs.

Chapter 6

Subgraph Morphing: Application Semantics in a System-Agnostic Framework

Whereas the previous chapter designed a fully-fledged graph mining system Peregrine using application-aware principles, this chapter takes a fundamentally different approach and develops a middle-end framework that fits into pattern-based graph mining systems, using knowledge of application semantics to automatically optimize execution.

We observe that the performance of graph mining applications is not only dependent on the patterns queried by the application, but is also sensitive to *system-level* nuances (*e.g.*, subgraph matching strategies and optimizations employed) as well as *application-level* characteristics (*e.g.*, application UDF and aggregation functions). We aim to take advantage of the performance difference when mining seemingly similar patterns by exploiting the *structural similarities* across different patterns. In general, dramatic performance improvements can be achieved if we could devise a general technique that can infer the results of a pattern for which it is expensive to find matches directly (*i.e.*, hard pattern) from those of other patterns where matching is less expensive (*i.e.*, easy pattern).

We propose Subgraph Morphing for graph mining systems – a generic technique that enables structure-aware algebra over patterns to *morph* the queried patterns into a set of alternative patterns, which can then be used to quickly compute accurate results for the original patterns. We make the following key contributions in this chapter.

- We expose key factors that impact the performance of graph mining (Section 6.1). Our observations from benchmarking various graph mining workloads across multiple graph mining systems show that the nature of input workloads (input patterns and data graph from the application), the pattern matching engines in graph mining systems, and the processing requirements of mining applications, all together contribute to the final performance. We envision these observations will be useful in building intuition for future research and development on graph mining systems.

- We develop the Subgraph Morphing algebra that shows how patterns can be morphed with guaranteed correct results (Section 6.2). Our algebra is general as it natively incorporates morphing with arbitrary aggregation operations from graph mining applications, and it generates multiple alternative solutions (which we call *alternative pattern sets*) to exploit different performance characteristics. With such generality, the system-level and application-level nuances can be incorporated to improve the overall performance, which is a key strength of our technique.

- We develop efficient strategies to enable Subgraph Morphing in practice. A major challenge is the exponential search space of alternative pattern sets with different benefits. We generate and navigate through different alternatives methodically using a novel data structure called S-DAG and a greedy algorithm to select *efficient* alternative pattern sets (Section 6.3). Our approach is backed by cost models that incorporate all the factors discussed above while estimating the performance of alternative patterns.

  Another challenge is that the results for queried patterns must be inferred from the results of alternative patterns, which can be tedious for users to perform in application logic. We develop strategies to seamlessly convert the results by operating on patterns and their matches only, hence removing the need to modify the application logic (Section 6.4). Our conversion strategies operate efficiently across both the common output modes in graph mining systems: batched mode where aggregation results are output at the end, and streaming mode where matching subgraphs are returned as a continuous output stream.

- We demonstrate the generality and effectiveness of Subgraph Morphing by integrating it into four state-of-the-art graph mining and subgraph matching systems: Peregrine, AutoMine/GraphZero [146, 145], GraphPi [192], and BigJoin [17]. Our extensive evaluation on a variety of graph datasets and graph mining applications demonstrates that Subgraph Morphing accelerates these systems by up to 34× on Peregrine, 10× on AutoMine/GraphZero, 18× on GraphPi, as well as 13× on BigJoin (Section 7.6).

Notation. This chapter deals heavily with the relationships between edge-induced and vertex-induced patterns. As a convenient notation, vertex-induced patterns are denoted with a superscript ^{V} (*e.g.*, p^V always refers to a vertex-induced pattern) while edge-induced patterns are denoted with a superscript ^{E} (*e.g.*, p^E always refers to an edge-induced pattern). Note that cliques are simultaneously edge- and vertex-induced since there exists an edge between any pair of vertices (and hence no anti-edge). For any given pattern p, we refer to p^E and p^V as variants of each other. Throughout the chapter, we omit the superscript on patterns when the discussion applies to both vertex-induced and edge-induced patterns. Several patterns can be easily described by their names (*e.g.*, triangle, clique, *w.r.t.*). Figure 6.1 summarizes the common pattern names used in this chapter.

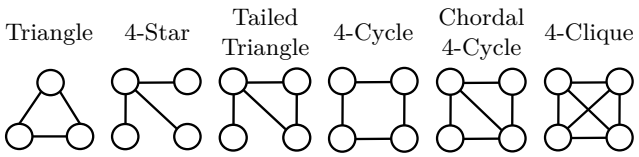


Figure 6.1: Common pattern names.

6.1 Performance Analysis

This section identifies key factors that impact the performance of graph mining workloads in order to motivate the need for a generic technique to accelerate arbitrary graph mining workloads across different graph mining systems. To understand the performance bottlenecks in existing systems, we profiled various graph mining workloads on different open-source systems: Peregrine, GraphPi [192], and BigJoin [17] for subgraph matching. Figure 6.2 shows the profiling results, and we summarize our observations below.

6.1.1 Graph Mining Applications

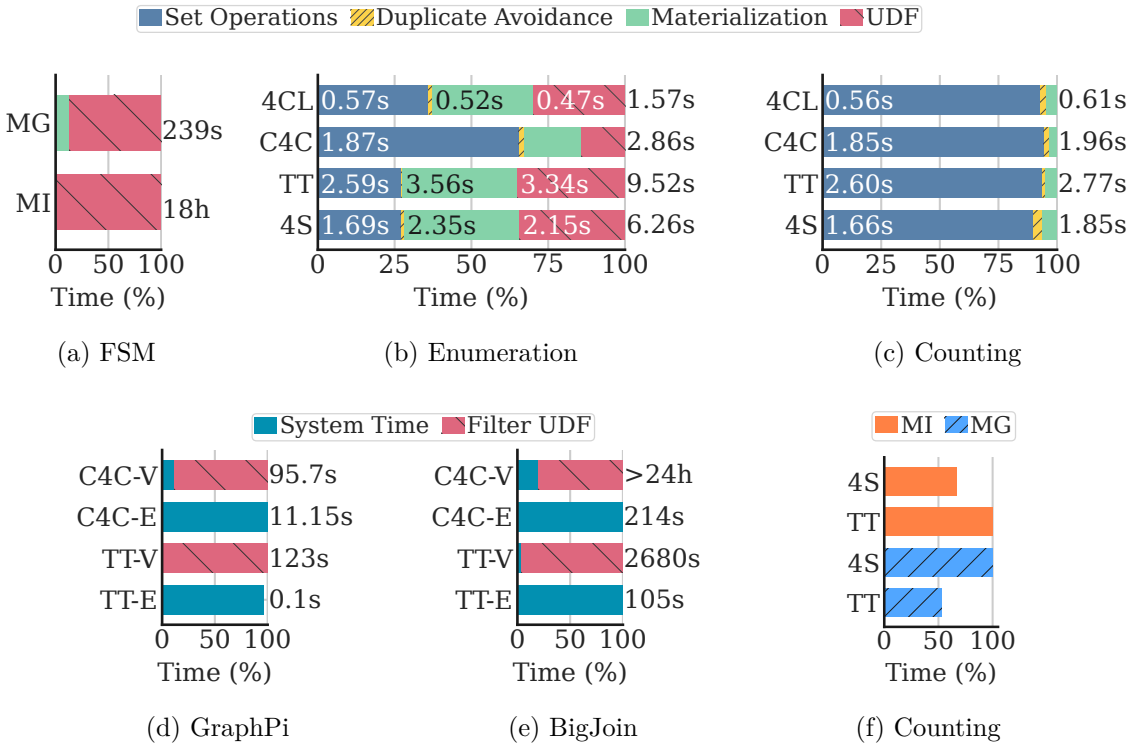
Figure 6.2a, Figure 6.2b and Figure 6.2c show the performance of three graph mining applications on Peregrine: Frequent Subgraph Mining (FSM), Subgraph Counting (SC) and Subgraph Matching (SM). These applications differ widely from each other. FSM invokes a user-defined function (UDF) on each match to compute MNI of patterns, whereas SC does not invoke any UDF since the system natively performs counting using set optimizations. SM is between FSM and SC where it lists out the matches using a UDF, but the UDF is simpler than in FSM.

Observation 1. *Since the number of matches grows exponentially with graph size, invoking UDF on each match impacts the end-to-end processing time. UDFs become the main performance bottleneck when they are expensive (as seen for FSM), while simpler UDFs also influence the processing time by non-trivial amount (as seen for SM).*

The above observation is also valid when mining vertex-induced subgraphs using systems like GraphPi and BigJoin that only perform edge-induced exploration. For these cases, the edge-induced matches mined by the system are processed using a Filter UDF to prune out invalid matches (*i.e.*, matches that do not contain all edges induced by their vertices). As shown in Figure 6.2[d-e], the Filter UDFs are the main performance bottlenecks, and they significantly slow down the overall processing compared to when matching edge-induced subgraphs (which does not require any UDF).

6.1.2 Structure of Patterns

Next, we study the performance of SC and SM (Figure 6.2[b-c]). As expected, the set operations on adjacency lists (set intersections and differences) take most of the time for



(a) FSM (b) Enumeration (c) Counting

(d) GraphPi (e) BigJoin (f) Counting

Figure 6.2: Profiling graph mining systems. Figures (a-c) show performance breakdown of FSM, Subgraph Matching and Subgraph Counting on Peregrine; (d-e) show performance breakdown of enumerating matches in GraphPi [192] and BigJoin [17]; (f) shows the relative performance of mining patterns on different data graphs in Peregrine (relative *w.r.t.* longer execution for each data graph). MG and MI are MAG and MiCo data graphs (see Figure 6.10). 4CL, C4C, TT and 4S are patterns 4-clique, chordal 4-cycle, tailed triangle and 4-star respectively (see Figure 6.1). The suffixes "-V" and "-E" indicate vertex-induced and edge-induced patterns (*e.g.*, TT-V is vertex-induced tailed triangle).

SC, while SM also spends time on materializing matches by merging sets of candidate vertices.

Since graph mining systems analyze the pattern structure to generate customized pattern-specific matching plans, the structure of the pattern dictates the effectiveness of the sub-techniques involved in the matching plan (*e.g.*, pruning strategies to account for pattern symmetries, or different join fast-paths as in Section 5.4.4). Hence, different patterns incur different amount of set operation and materialization time. While one would expect similar looking patterns (*e.g.*, same number of vertices) to have similar performance trends, or denser patterns to be more expensive than sparser patterns with same number of vertices, no such performance trends are guaranteed. For instance, in Figure 6.2[b-c] we can see:

- A 4-star takes more set operation and materialization time than a 4-clique, even though latter has twice the number of edges than the former (see pattern structures in Figure 6.1).
- A chordal 4-cycle has only one additional edge over a tailed triangle, but the latter consumes more time for set operation and materialization compared to the former.

> ***Observation 2.*** *As graph mining systems generate pattern-specific matching plans, even similar-looking patterns incur different amounts of set operations and materialization time, which results in unexpected performance trends across those patterns that are difficult to justify.*

6.1.3 Structure of Data Graphs

Next, we study the performance of mining different patterns on different data graphs. While mining in larger graphs takes more time (which is expected), the structure of the data graph impacts the relative performance of mining different patterns. This is visible in Figure 6.2f where mining 4-stars is 25% faster than tailed triangles in MiCo graph, but it is 125% slower in MAG graph. This is because the graph structure influences how different explorations proceed or get pruned (*e.g.*, matching order violation), which in turn impacts the amount of work performed to mine all matches.

> ***Observation 3.*** *The structure of the data graph also influences the mining performance since it dictates which explorations proceed while others get pruned out.*

6.1.4 Graph Mining Systems

Finally, we study the relative performance between graph mining workloads across different graph mining systems. Since graph mining systems employ different kinds of pattern matching techniques (*e.g.*, matching algorithms) and are implemented and optimized in different manner (*e.g.*, parallelization strategies), the performance relationships across workloads

varies across the systems as well. This is observed when comparing the performance numbers in Figure 6.2[b-d] for Peregrine and GraphPi: while the chordal 4-cycle is faster than a tailed triangle in Peregrine, the performance relation is reverse for GraphPi where tailed triangle is faster.

Observation 4. *The design and implementation choices incorporated in graph mining systems impacts the relative performance between different graph mining workloads.*

6.1.5 Motivation Summary

In summary, the end-to-end processing time is influenced by: (a) the structure of patterns and the data graph, (b) the matching strategies and optimizations employed by the mining system, and (c) the processing requirements of the graph mining application (*i.e.*, UDF calls). More importantly, none of these factors are a clear single variable that should be optimized over the other, making it difficult to improve the performance of mining systems.

This motivates the need for a general technique that graph mining systems can employ across various graph mining workloads. Subgraph Morphing is our general technique. It first methodically exposes the space of performance opportunities (Section 6.2) and then considers the system-level and application-specific nuances to deliver high performance across different scenarios (Section 6.3 and Section 6.4).

6.2 Subgraph Morphing

This section describes principles of subgraph morphing with illustrative examples, and develops its semantics.

6.2.1 Overview

Subgraph Morphing primarily exploits the structural similarities across different patterns. The key idea is to morph the input patterns from graph mining applications into *alternative patterns* that are less expensive to compute, and then convert the results for those alternative patterns into results for the original input patterns. When Subgraph Morphing is integrated in graph mining systems, their workflow gets enhanced with two new steps (shown in Figure 6.3): pattern transformation and result transformation. Instead of being directly passed to the execution planner, the input patterns first undergo a pattern transformation step resulting in alternative patterns. Then, matching plans are computed and followed to explore matches for the alternative patterns from the input graph. Finally, the results for alternative patterns are sent through the result transformation step to compute the results for the original input patterns. Details of pattern transformation and result transformation will be explained in Section 6.3 and Section 6.4 respectively, and Appendix B illustrates the key steps in Subgraph Morphing using two graph mining applications. In this section, we will focus on the semantics of Subgraph Morphing.

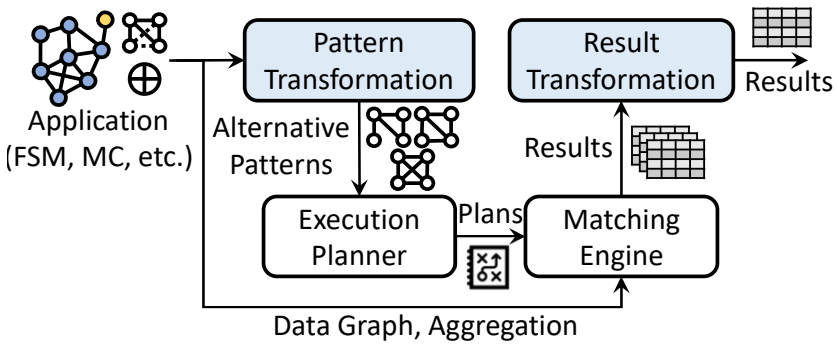


Figure 6.3: Graph mining with Subgraph Morphing.

6.2.2 Intuition & Example

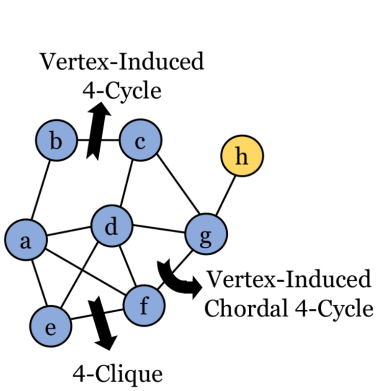
The intuition behind Subgraph Morphing can be summarized with the following two key observations.

[P1] A match for an edge-induced pattern p^E on n vertices is also a match for all of its subpatterns on these vertices. For example, a match for a 4-clique is also a match for an edge-induced 4-cycle, since the 4-clique contains all the edges of the 4-cycle. Note that this observation does *not* apply to vertex-induced patterns—although a vertex-induced 4-cycle contains the same four (regular) edges, the additional two anti-edges exclude matches for 4-cliques.

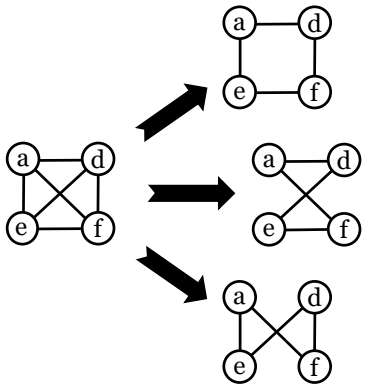
[P2] A match for a vertex-induced pattern p^V is always a match for the corresponding edge-induced pattern p^E, but *not* vice versa—p^V matches exactly the edges in p^E but a subgraph that matches p^E may contain additional edges that are against the anti-edges in p^V.

Example. These observations indicate that we can logically *partition* the matches for an edge-induced pattern into disjoint sets of matches for vertex-induced patterns. For example, consider a match for the edge-induced 4-cycle. The vertices in this match may have edges that are in the graph but not present in the pattern. If these edges do not exist, this is also a match for the vertex-induced 4-cycle (*e.g.*, a-b-c-d in Figure 6.4a). If there exists exactly one extra edge, it is a match for the vertex-induced chordal 4-cycle (*e.g.*, d-c-g-f in Figure 6.4a). Finally, if there are two extra edges in the match, it is a match for the 4-clique (*e.g.*, a-d-f-e in Figure 6.4a). These situations are *mutually exclusive* since they depend on specific edges being present or absent.

While the above partitioning enables converting matches, a match for a given pattern can potentially contain multiple matches for another pattern. For example, a match for the 4-clique contains 3 unique matches for the edge-induced 4-cycle, as shown in Figure 6.4b. Hence, in order to convert a match for the 4-clique into a match for the 4-cycle, we must correctly map the 4-clique vertices, using the subgraph isomorphisms, to those of the 4-cycle.



(a) Edge-induced 4-cycle contains 4-clique, vertex-induced chordal 4-cycle, or vertex-induced 4-cycle.



(b) 4-clique contains three unique edge-induced 4-cycles.

Figure 6.4: Identifying matches for different patterns.

6.2.3 Semantics

Subgraph Morphing performs structure-aware algebra over patterns (and hence, their matches) to capture all matches for a given pattern in the input graph by converting the matches of different (alternative) patterns. One way to find alternative patterns for a given pattern is to consider its *superpatterns* because matches of a pattern are guaranteed to contain valid matches for its subpatterns. Hence, our first idea is to derive the matches of the pattern using its superpatterns.

Let $g \in \mathcal{G}$ be a graph and let $p \in \mathcal{G}_\star$ be a pattern. Recall that $\mathcal{E}^*_\star(g, p)$ is the set of all unique matches for p in g under extended subgraph isomorphism. Then,

$$\mathcal{E}^*_\star(g, p^E) \;=\; \mathcal{E}^*_\star(g, p^V) \;\cup \bigcup_{q^E \supset_n p^E} \mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E) \tag{6.1}$$

where $q^E \supset_n p^E$ indicates the superpatterns q^E of pattern p^E containing same number of vertices (n), $\phi(p^E, q^E)$ captures the set of all subgraph isomorphisms from p^E to q^E, and $\mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)$ permutes the vertices in each match $m \in \mathcal{E}^*_\star(g, q^V)$ according to subgraph isomorphism from p^E to q^E.

In simple words, given an edge-induced pattern p^E, we start with using the matches of its vertex-induced variant p^V (observation [P2]). However, since p^V contains anti-edges that eliminate some valid matches for p^E, we compensate by using matches for additional superpatterns, each obtained by replacing anti-edges in p^V one-by-one with true edges (observation [P1]). This ends up generating an ***alternative pattern set*** for p^E that contains all of its vertex-induced superpatterns with the same number of vertices. Since a match for a superpattern can contain multiple matches for a subpattern (*e.g.*, 4-clique contains three edge-induced 4-cycles in Figure 6.4b), we use ***permutation functions*** to generate all the matches of the subpattern. A permutation function converts matches of a superpattern

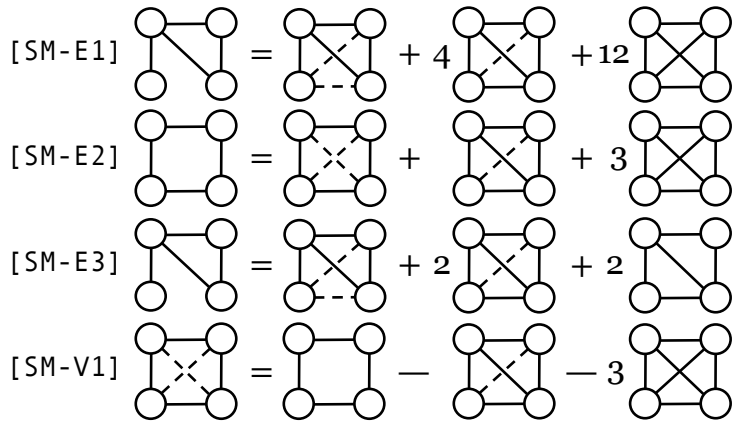


Figure 6.5: Sample equations resulting from subgraph morphing. [SM-V1] morphs vertex-induced pattern (left) whereas other equations morph edge-induced patterns. [SM-E1] and [SM-E2] are directly obtained from Eq. 6.1, [SM-E3] by recursively substituting in [SM-E1], and [SM-V1] by adjusting [SM-E2]. The coefficients indicate the numbers of unique matches resulting from subgraph isomorphism.

into matches for the subpattern based on isomorphic mappings from the subpattern to the superpattern.

To maintain the flow of exposition, the proof for Eq. 6.1 is deferred to Section 6.2.5. Next, we focus on the two key aspects that make our Subgraph Morphing strategy generic: directly converting arbitrary aggregations results and multiple alternatives for converting matches.

Converting Aggregation Results. Subgraph Morphing can be applied directly on aggregation results instead of converting the individual matches. By doing so, we can prevent materialization of a significant number of matches, and reduce UDF overheads while computing aggregations by invoking them on fewer matches.

Let $a = \langle \mathcal{R}, \oplus \rangle$ denote the aggregation, with ν as the map from matches to aggregation values. For a set of matches $\mathcal{E}^*_\star(g,p)$ in graph $g \in \mathcal{G}$, write $a(\mathcal{E}^*_\star(g,p))$ as a shorthand for $\bigoplus_{m \in \mathcal{E}^*_\star(g,p)} \nu(m)$. The aggregation results can be directly converted as follows (correctness proven in Section 6.2.5):

$$a(\mathcal{E}^*_\star(g,p^E)) = a(\mathcal{E}^*_\star(g,p^V)) \oplus \Big(\bigoplus_{q^E \supset_n p^E} \bigoplus_{\substack{f \in \\ \phi(p^E,q^E)}} a(\mathcal{E}^*_\star(g,q^V)) \circ_* f \Big) \tag{6.2}$$

where $\circ_*$ is a *permute* operator for aggregation values to account for the permutations according to $\phi(p,q)$ (similar to $\circ$ defined on matches in Eq. 6.1). The $\circ_*$ operator adjusts the aggregation value based on a given permutation f in $\phi(p,q)$. The definition of the permutation function depends on the aggregation operation performed on the matches. For instance, the permutation function for *counting* accounts for all unique isomorphic mappings, which results in multiplying the number of matches of a superpattern by the

number of unique isomorphic mappings. In FSM, the permutation function permutes the columns of the MNI table of the match, in a similar manner, based on subgraph isomorphism.

Multiple Alternative Pattern Sets. While Eq. 6.1 identifies alternative patterns for edge-induced patterns, we can move in the other direction as well to compute results for vertex-induced patterns (achieved by rearranging the terms in Eq. 6.1 to bring $\mathcal{E}^*_\star(g, p^V)$ on left-hand side). More importantly, the patterns in the alternative pattern set can be iteratively substituted with their conversion equations to obtain different alternative pattern sets. By recursively substituting the patterns with their alternative patterns sets, we can generate a system of equations representing different alternative pattern sets that can be used to compute the results for a given query pattern. Additionally, the alternative pattern sets can contain a combination of vertex-induced and edge-induced patterns by converting the intermediate aggregations through recursive substitution.

Figure 6.5 shows a few samples of how patterns can be morphed to a given pattern. The coefficient associated with a pattern indicates the number of unique matches resulting from subgraph isomorphism (*e.g.*, 4-clique has three 4-cycles, and hence the 4-clique in Equation [SM-E2] has a coefficient 3). As we can see, [SM-E1] and [SM-E3] are two different equations to compute results for the tailed triangle. With different choices for alternative pattern sets, the pattern query can be optimized by selecting the alternative set for which matches can be efficiently computed (discussed in Section 6.3).

6.2.4 Significance of Generic Subgraph Morphing

The generic nature of Subgraph Morphing enables accelerating arbitrary graph mining applications while also incorporating system-level nuances and application-level characteristics. For example, system-level nuances were shown to impact the mining workloads differently in Section 6.1, causing certain patterns to be faster than others on one system but slower on another system. In such a situation, alternative patterns sets can be optimized differently for each individual system by accounting for their relative performance across different patterns; in our example from observation 4, this would mean choosing tailed triangle over 4-cycle for GraphPi but not for Peregrine. Similarly, application-level characteristics like application UDF and the structure of the data graph were shown to impact the query performance in Section 6.1. Since Subgraph Morphing enables direct conversion of aggregation results, the impact of these application-level characteristics can be directly accounted in choosing the right set of alternative patterns. For example, expensive UDF calls like filtering each match or computing MNI tables for FSM can be reduced by using alternative patterns that are expected to generate fewer matches. But on the other hand, applications that employ inexpensive aggregation operations like counting (system-native) can benefit from alternative patterns that are expected to incur fewer set operations.

In comparison, counting techniques developed in prior works like [141, 108, 168, 151, 234] are inapplicable for general-purpose graph mining systems since they focus on count conver-

sion rules that are customized for specific types of equations on specific patterns. Hence, they cannot systematically generate multiple alternative sets for a given query pattern, which renders them useless as they cannot account for various system-level and application-level nuances. For instance, blindly converting results from vertex-induced to edge-induced (or vice-versa) would often be slower than the original query, depending on the system and the application.

6.2.5 Proofs of Equivalence

This section proves that Subgraph Morphing preserves equivalent results after transforming input patterns.

Match Conversion. Recall Eq. 6.1 from Section 6.2.3. Let $g \in \mathcal{G}$ be a graph and let $p \in \mathcal{G}_\star$ be a pattern. Then,

$$\mathcal{E}^*_\star(g, p^E) \;=\; \mathcal{E}^*_\star(g, p^V) \;\cup \bigcup_{q^E \supset_n p^E} \mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E) \tag{6.1}$$

where $q^E \supset_n p^E$ indicates the superpatterns q^E of pattern p^E containing same number of vertices (n), $\phi(p^E, q^E)$ captures the set of all subgraph isomorphisms from p^E to q^E, and $\mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)$ permutes the vertices in each match $m \in \mathcal{E}^*_\star(g, q^V)$ according to subgraph isomorphism from p^E to q^E.

Proof. Since we are proving set equality, we will first show how every match in $\mathcal{E}^*_\star(g, p^E)$ is contained in the set on the right-hand side of the equation, and then show that $\mathcal{E}^*_\star(g, p^E)$ contains every match from the right-hand side.

Let m be a match in $\mathcal{E}^*_\star(g, p^E)$, and q^V be the pattern of the subgraph induced by the image of m. q^V must have at least as many edges as p^E, since it was induced by a match for p^E.

– *Case 1:* If q^V has the same number of edges as p^E, q^V is isomorphic to p^V and hence $m \in \mathcal{E}^*_\star(g, p^V)$.

– *Case 2:* Otherwise, q^V has more edges than p^E. Consider the edge-induced pattern q^E corresponding to q^V. q^E must be a superpattern of p^E, since q^V has more edges than p^E but contains all the edges of p^E. This means $\phi(p^E, q^E)$ is non-empty. For any $f \in \phi(p^E, q^E)$, observe that $m \circ f^{-1} : V(q^E) \to V(p^E) \to G$ is a match for q^V, since $V(q^E) = V(q^V)$ and q^E was obtained from the induced pattern q^V. This means $m \circ f^{-1} \in \mathcal{E}^*_\star(g, q^V)$, and hence $m \circ f^{-1} \circ f = m \in \mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)$.

Therefore, we showed $\mathcal{E}^*_\star(g, p^E)$ is a subset of the right-hand side of the equation. Next, we will prove that the right-hand side is contained within $\mathcal{E}^*_\star(g, p^E)$.

First, note that a match for p^V is trivially a match for p^E, since p^E and p^V differ only in anti-edges, so $\mathcal{E}^*_\star(g, p^V) \subseteq \mathcal{E}^*_\star(g, p^E)$.

Next, take any match $m \in \mathcal{E}^*_\star(g, q^V)$ where $q^E \supset_n p^E$. m is also in $\mathcal{E}^*_\star(g, q^E)$ by the same reasoning as above. But then for any $f \in \phi(p^E, q^E)$, $m \circ f : V(p^E) \to V(q^E) \to G$ is a match for p^E since any match for q^E contains all edges required for matching p^E. Hence, $\mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E) \subseteq \mathcal{E}^*_\star(g, p^E)$.

Taking the union of $\mathcal{E}^*_\star(g, p^V)$ and $\mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)$ for each $q^E \supset_n p^E$ gives the set of matches that are contained in $\mathcal{E}^*_\star(g, p^E)$. □

Eq. 6.1 reflects a useful relationship between edge-induced and vertex-induced patterns. In fact, although the theorem is stated in terms of converting from vertex-induced to edge-induced, we can move in the other direction as well:

Corollary 6.2.1. *Let p^E be an edge-induced pattern with n vertices, and p^V be its vertex-induced variant. Then,*

$$\mathcal{E}^*_\star(g, p^V) = \mathcal{E}^*_\star(g, p^E) \setminus \bigcup_{q^E \supset_n p^E} \mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)$$

Proof. Let B denote the set

$$\bigcup_{q^E \supset_n p^E} \mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E).$$

From Eq. 6.1, we have $\mathcal{E}^*_\star(g, p^E) = \mathcal{E}^*_\star(g, p^V) \cup B$. Notice that if $\mathcal{E}^*_\star(g, p^V)$ is disjoint from all $\mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)$ where $q^E \supset_n p^E$, then we could take the set difference on both sides of the equation from Eq. 6.1 with B to prove the corollary, simply because no element of $\mathcal{E}^*_\star(g, p^V)$ would be removed by the set difference operation.

It remains to prove that $\mathcal{E}^*_\star(g, p^V)$ is disjoint from the various $\mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)$ where $q^E \supset_n p^E$. We prove this by contradiction.

Let m be a match in $\mathcal{E}^*_\star(g, p^V) \cap \mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)$ for any p^V and some q^V where $q^E \supset_n p^E$. First note that if p^V is a clique, there is no $q^E \supset_n p^E$, and m cannot exist, so we are done. Instead, suppose p^V is not a clique, and thus has at least one anti-edge. Since $q^E \supset_n p^E$, for each $f \in \phi(p^E, q^E)$ there is an edge $(f(u), f(v)) \in E(q^E)$ such that $(u, v) \notin E(p^E)$. This means (u, v) forms an anti-edge constraint in p^V, and $(f(u), f(v))$ forms an edge constraint in q^V. As $m \in \mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)$, it can be written in the form $m = m' \circ f$ where $m' \in \mathcal{E}^*_\star(g, q^V)$ and $f \in \phi(p^E, q^E)$. But then $((m' \circ f)(u), (m' \circ f)(v))$ must be an edge in G, since $(f(u), f(v)) \in E(q^V)$ and m' is a match for q^V. This directly contradicts the anti-edge constraint in p^V that we established above.

Hence $\mathcal{E}^*_\star(g, p^V) \cap \mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E) = \emptyset$ for all $q^E \supset_n p^E$, as desired. □

Aggregation Conversion. Let $g \in \mathcal{G}$ be a graph and let $p \in \mathcal{G}_\star$ be a pattern. Recall Eq. 6.2:

$$a(\mathcal{E}^*_\star(g, p^E)) = a(\mathcal{E}^*_\star(g, p^V)) \oplus \Big(\bigoplus_{q^E \supset_n p^E} \bigoplus_{\substack{f \in \\ \phi(p^E, q^E)}} a(\mathcal{E}^*_\star(g, q^V)) \circ_* f \Big) \tag{6.2}$$

Proof. This equation follows immediately from Eq. 6.1.

$$\begin{aligned}
& a(\mathcal{E}^*_\star(g, p^E)) \\
&= a\left(\mathcal{E}^*_\star(g, p^V) \;\cup \bigcup_{q^E \supset_n p^E} \mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E) \right) \\
&= a(\mathcal{E}^*_\star(g, p^V)) \oplus \left(\bigoplus_{q^E \supset_n p^E} a(\mathcal{E}^*_\star(g, q^V) \circ \phi(p^E, q^E)) \right) \\
&= a(\mathcal{E}^*_\star(g, p^V)) \oplus \left(\bigoplus_{q^E \supset_n p^E} \bigoplus_{f \in \phi(p^E, q^E)} a(\mathcal{E}^*_\star(g, q^V)) \circ_* f \right)
\end{aligned}$$

□

6.3 Generating Alternative Pattern Sets

This section describes the pattern transformation step (see Figure 6.3) to compute alternative patterns.

To fully exploit Subgraph Morphing, our goal is to generate alternative pattern sets that would potentially compute the final results efficiently compared to the original query patterns. This cannot be achieved statically because of two main reasons. First, the query patterns in graph mining applications can change dynamically during runtime. For instance in FSM, only those patterns that have enough matches in the data graph (*i.e.*, cross a support threshold) are extended to generate the new set of patterns to be explored in the next step. And second, the input data graph itself impacts the performance of matching (observation 3 in Section 6.1). Hence, we dynamically generate the alternative patterns for the query patterns as they become available at runtime.

Since the possible alternative pattern sets grow recursively, the search space for identifying *efficient* alternative pattern sets grows exponentially. For a single query pattern, the choice may appear simple. However, when the input contains multiple query patterns, the number of choices increases exponentially as the alternative sets for different patterns overlap, making it hard to estimate benefits. For instance, two patterns may have a lower cost (*i.e.*, faster to compute) compared to the cost of their individual alternative pattern sets; however, the cost of the combined alternative pattern sets can be lower (due to overlapping patterns) than that of the two patterns.

Exhaustively searching all possible combinations of alternative pattern sets is impractical. Instead, we develop a greedy exploration strategy backed by a cost model that actively prunes the search space. Our approach is to first generate a single alternative set, and then use a cost-based selection strategy to iteratively improve the alternative set by substituting better (low cost) alternatives.

6.3.1 Initial Alternative Patterns

Given a set of input patterns, we generate the initial alternative pattern set for each pattern in the input set using Eq. 6.1. This primarily involves generating superpatterns of the input pattern (second term in Eq. 6.1). While the final alternative pattern set contains a mix of vertex-induced and edge-induced patterns, the choice of each individual pattern being either vertex-induced or edge-induced is independent of the other patterns in the set. Hence, we generate edge-induced superpatterns, and later optimize the choice for each pattern when constructing the efficient alternative pattern set. By doing so, we maximize the overlap between the alternative patterns generated across different patterns in the input set, which is beneficial since the same superpattern (or its alternatives) covers multiple input patterns.

The superpatterns of the input pattern are generated by extending them recursively, adding edges between disconnected vertices. Naïvely extending the input patterns can end up generating duplicate patterns due to two reasons. First, adding edges between automorphic (or symmetric) sub-components of a given pattern can result in the same superpatterns; for instance, adding an edge between any pair of disconnected vertices in a 4-cycle would result in the same chordal 4-cycle pattern. Second, different patterns with the same number of vertices have a common subset of superpatterns; for instance, a 4-cycle and a tailed triangle both have the 4-clique and chordal 4-cycle as their superpatterns.

We avoid generating redundant superpatterns by maintaining them in the S-DAG data structure, as described next.

S-DAG for Superpattern Sets. As superpatterns get generated, we memoize them and their superpattern-subpattern relationships in form of a directed acyclic graph, called S-DAG. The S-DAG is queried each time before recursively extending any pattern and adding new superpatterns in order to avoid redundant pattern alternatives.

Each vertex of the S-DAG represents a pattern (either one of the input patterns or one of their superpatterns), and we draw directed edges from each pattern with k edges to its superpatterns with $k+1$ edges. Figure 6.6 shows two S-DAG examples: one where patterns are unlabeled, and other where patterns are labeled. Depending on the number of labels in the pattern, the number of possible patterns increases compared to the number of unlabeled patterns, and each labeled pattern can have many more superpatterns than an unlabeled one.

For fast comparison and lookup operations on S-DAG, we represent the patterns using 64-bit pattern IDs which uniquely correspond to the pattern structures. Each pattern is first

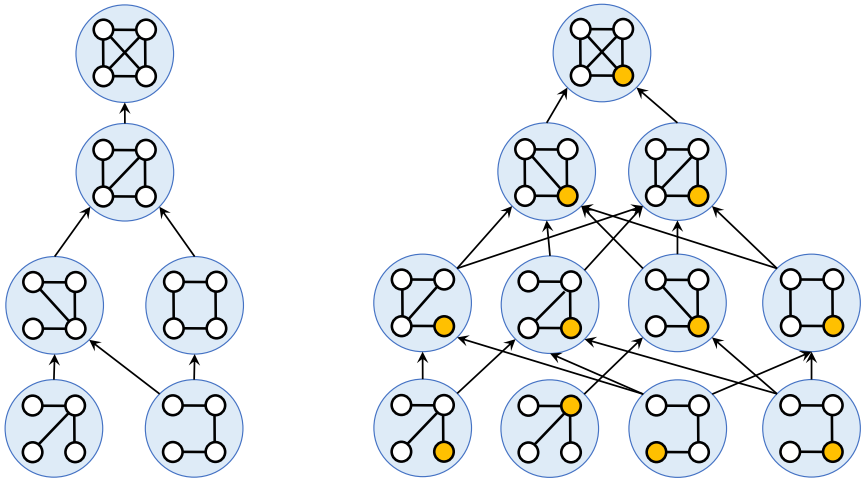


Figure 6.6: S-DAG for unlabeled patterns (on left), and for patterns with one yellow labeled vertex (on right).

canonicalized (using Bliss library [117]) so that its vertices have consistent vertex IDs. Then, the edges of the canonicalized pattern are hashed consistently to compute its pattern ID that uniquely identifies the pattern structure. While pattern IDs can be computed quickly (in milliseconds) as patterns get generated, they can also be computed offline.

6.3.2 Selecting Efficient Alternative Patterns

Once the S-DAG is generated, the final alternative pattern set is constructed by carefully selecting the set of patterns based on the potential performance benefits they can bring. To avoid evaluating every possible alternative set in the exponential search space, we develop a greedy algorithm that iteratively finds better alternatives using the S-DAG. Instead of searching for the optimal alternative pattern set, our goal is to construct an *efficient* alternative set that promises faster execution compared to the original query patterns.

Algorithm 1 shows the selection algorithm. Initially, each node in S-DAG is assigned a cost that estimates the time taken to match that pattern. Since either variant of super-patterns can be used in the alternative pattern set, the nodes are assigned the minimum cost between the two variants of the patterns they represent. Then the algorithm proceeds iteratively to replace patterns with lower cost alternatives.

For each pattern we check whether any subsets of the pattern's children in the S-DAG benefit from morphing. If the combined cost of the children is more than the cost of their combined superpatterns, then the superpatterns are selected for the alternative pattern sets. When the alternative patterns are selected, the S-DAG is re-weighted to reflect that those patterns are free, *i.e.*, their cost is set to 0, and then the patterns are traversed again. The algorithm incrementally reduces the cost of the alternative pattern set until it can no longer be improved. By considering only the subsets of each pattern's children, we reduce the exponential search space to the number of unique subpatterns for each pattern.

Algorithm 1 Efficient Alternative Patterns

Input: Set of query patterns P and their S-DAG^P
Output: Low-cost alternative pattern set S

```
1: initializePatternCosts(S-DAG^P)
2: procedure selectPatterns(P, S-DAG^P)
3:     S ← P
4:     while S not converged do
5:         for each p ∈ ∪_{s∈S} parents(S-DAG^P, s) do
6:             for each C ∈ ℙ(children(S-DAG^P, p)) where C ⊆ S do
7:                 cost_C ← Σ_{c∈C} initial_cost(c)
8:                 SPC ← ∪_{c∈C} superpatterns(c)
9:                 cost_SPC ← Σ_{spc∈SPC} cost(spc)
10:                if cost_SPC < cost_C then
11:                    S ← (S \ C) ∪ SPC
12:                    for each c ∈ C ∪ SPC do
13:                        setCost(S-DAG^P, c, 0)
14:                    end for
15:                end if
16:            end for
17:        end for
18:    end while
19:    return S
20: end procedure
```

Estimating Relative Pattern Costs. In order to identify cheaper pattern alternatives, the selection algorithm requires pattern costs that represent the estimated relative times to match different patterns. The cost of subgraph matching depends not only on the input patterns and the data graph, but also on the system-specific properties such as the underlying matching algorithm used by the system and on application-specific details like aggregation functions.

Modern graph mining and subgraph matching systems like [146, 145, 192, 95] often incorporate a cost model to compute efficient exploration plans for the given patterns. While these models are useful, they do not account for any application-specific detail since it is irrelevant to their matching plans. Since we are interested in costs that estimate the performance of different patterns on a given application workload, for these systems we piggyback on their existing cost model by enhancing them to include cost of result aggregation.

The data graph is modeled by an abstract probabilistic graph, where two vertices are connected by an edge with a fixed probability. The subgraph matching process is modeled as a series of $V(P)$ nested loops over this abstract graph. After computing the expected number of iterations in each loop, keeping in mind previous loops, the final cost is simply the number of iterations in the innermost loop.

As aggregations are functions on individual matches, their costs are modeled as the number of estimated matches multiplied by the amount of work for the aggregation. The number of estimated matches is already available from the cost model of the underlying system. The amount of work for the aggregation can be estimated by profiling the application-specific UDF to identify how aggregation time scales with the number of matches, or by analyzing

the aggregation operations. For profiling, a set of n dummy matches can be generated by randomly selecting $|V(P)|$ vertices n times, and then the time required to apply the UDF to these n matches gets measured. Repeating this for varying n and integrating the resulting curve yields an approximation of how the aggregation scales. Alternatively, the cost of aggregation for simple or well-known operations can be directly provided as hints to the system. For instance, the counting aggregation is performed using a constant operation per match, and hence incurs no additional cost. On the other hand, FSM application incurs $O(|V(G)|)$ cost to approximate the overheads of merging MNI tables.

Finally, for systems like Peregrine and BiGJoin [17] that do not use any cost model, we compute pattern costs based on their pattern matching details using a similar approach as AutoMine [146]. In addition, we improve the cost estimation using two novel enhancements.

- From profiling, we observed that highest degree data vertices (those in the 95th percentile) contribute the majority (66–99%) of the matches and the majority of the execution time. Hence, the graph model is restricted to the portion of the data graph comprising the highest degree vertices.
- Since partial orders for symmetry breaking [93] impact the input size (*e.g.*, adjacency lists or indexed tuples) for the set operations or joins performed during matching, the neighborhood size is estimated in terms of the expected number of smaller or higher vertex id neighbors.

6.4 Transforming Results

The efficient alternative pattern set is given as input to the pattern matching engine. The next step is to convert the matches generated for these alternative patterns into results for the original query patterns (result transformation in Figure 6.3). As discussed in Section 6.2.3, we use *permutation functions* (*i.e.*, $\phi(p, q)$ in Eq. 6.1 and Eq. 6.2) that convert the results based on isomorphic mappings from the query pattern to the alternative pattern. For seamless conversion, our key insight is to permute the vertex ids in the pattern instead of modifying the results so that the results get mapped for original patterns. We will describe our result conversion strategies in Section 6.4.1 and Section 6.4.2.

Output Modes. Graph mining systems often employ different output modes for returning matches that are suitable for different applications. For example, applications like FSM and MC return the final aggregation values (counts, support values) in the end after all required matches are found. On the other hand, applications like SE return each individual match to the user function for further processing (*e.g.*, filtering) as the matches are explored. To handle these output requirements, our permutation functions can be employed to convert the results either on-the-fly or after matching finishes.

Algorithm 2 Converting Aggregation Results

```
    Input: Set of query patterns P, their alternative pattern set S,
           and aggregation store A holding results of S
    Output: Aggregation store R for P
 1: procedure convertResults(P, S, A)
 2:     for each p ∈ P do
 3:         for each q ∈ alternative(S, p) do
 4:             for each q_key ∈ AggrKeyMap(q) do
 5:                 for each f ∈ φ(p, q) do
 6:                     p_key ← permute(q_key, f)
 7:                     R[p_key] ← reduce(R[p_key], A[q_key])
 8:                 end for
 9:             end for
10:         end for
11:     end for
12:     return R
13: end procedure
```

6.4.1 Post-Matching Conversion

In this case, results get converted after matching for the alternative patterns completes. Then the converted results are returned to output.

Since the final results are often computed by application-specific aggregation functions, one way to convert the results would be by directly modifying the aggregation functions (*e.g.*, map-reduce UDF) to simply change the mapping between results and patterns. While such a change is easy, it still requires capturing the relationship between the query patterns and their alternative patterns into aggregation functions.

To make the conversion process seamless to the application, we instead modify the vertex ids in the patterns, which does not require modifying application-specific logic. This is achieved by applying the permutation function on vertex ids of the alternative patterns. By doing so, we simply invoke the aggregation operation on the query pattern, but with permuted ids, which ends up correctly routing the aggregation results from alternative patterns to the query patterns.

Algorithm 2 shows the result conversion process. The results for alternative patterns are maintained in the aggregation store A as key-value pairs, where keys are of different types depending on the application (*e.g.*, patterns for SC, pattern vertices for MNI). To convert the results (lines 5-8), the aggregation keys for the alternative pattern are permuted based on the permutation function to obtain the key for input pattern, and then the aggregation values for those keys are combined using the application's aggregation function (reduce on line 7).

Example. We illustrate how conversion happens for the MNI aggregation in FSM. To help understand the context, the FSM application code is shown in Figure 6.7. In the process function, the individual vertices are mapped to the respective pattern vertices, and the reduce operation merges the set of pattern vertices (*i.e.*, MNI columns).

```
void process(Pattern p, Match m) {
  for (PatternVertex u : p.getVertices())
    map(u, m(u));
}
MNIColumn reduce(MNIColumn accumulator,
                 MNIColumn new_value) {
  return accumulator.merge(new_value);
}
```

Figure 6.7: FSM Application.

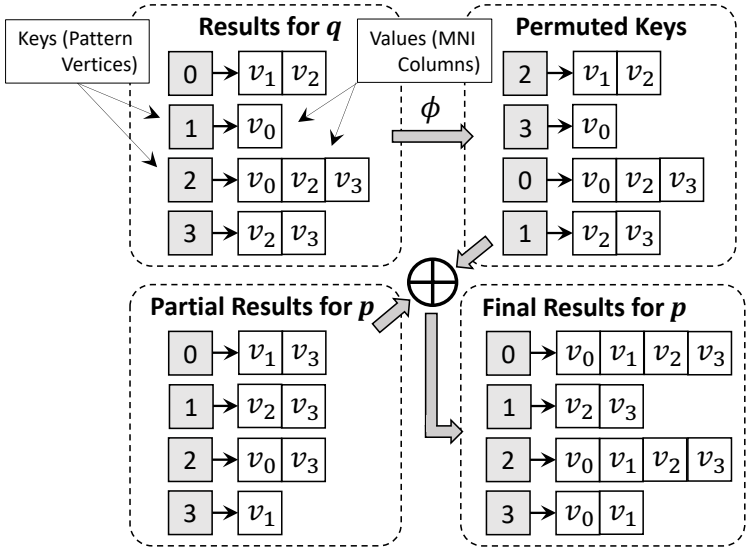


Figure 6.8: Converting MNI aggregation for FSM.

Figure 6.8 shows an example of result conversion. The permutation function permutes the vertex ids for pattern q which results in a change in the mappings between keys (pattern vertices) and values (MNI columns). Hence, for the results on top right, the column for vertex 0 gets remapped to the third column. Then, the aggregation function merges the columns for p with the permuted aggregation.

6.4.2 On-the-Fly Conversion

Here, the results are converted as they get generated by the matching engine, and then the converted results are sent down the application's processing pipeline.

While we can employ the same strategy of converting patterns instead of converting the results, it is possible to directly convert the match generated by the matching engine since it is yet not been modified by the application-specific functions (*i.e.*, converting the match does not require application details). Algorithm 3 shows on-the-fly conversion of matches. Instead of directly calling the application function with the alternative pattern q and its match m, the permutation function is applied on m which generates the match for the query pattern p. These are then supplied to the application function (process on line 6).

Algorithm 3 Converting Matches On-the-Fly

```
1: // Send alternative pattern q to matching engine
2: // Matching engine starts finding matches
3: // Matching engine generates match m for pattern q
4: for each f ∈ φ(p, q) do
5:     m_permuted ← PERMUTE(m, f)
6:     PROCESS(p, m_permuted)      /* Call UDF */
7: end for
```

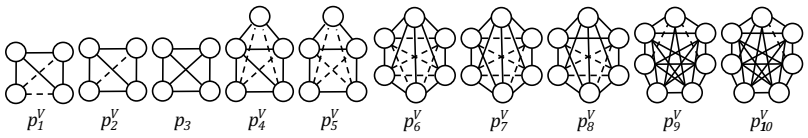


Figure 6.9: Vertex-induced patterns used in evaluation. The edge-induced variants do not contain anti-edges.

6.5 Evaluation

Mining Systems and Implementation Details. We integrated SUBGRAPH MORPHING in four state-of-the-art graph mining and subgraph matching systems: PEREGRINE, AutoZero (an implementation of techniques in both AutoMine [146] and GraphZero [145]), GraphPi [192], and BiGJoin [17]. SUBGRAPH MORPHING is generally applicable to other mining and subgraph matching systems like [95, 144] as well. Systems like Arabesque [205], Fractal [77] and others [45, 219, 52] perform generic BFS or DFS explorations that do not exploit the pattern structure in order to optimize exploration. Hence, they deliver similar performance across different patterns of same size, providing little or no opportunity to exploit performance difference across patterns.

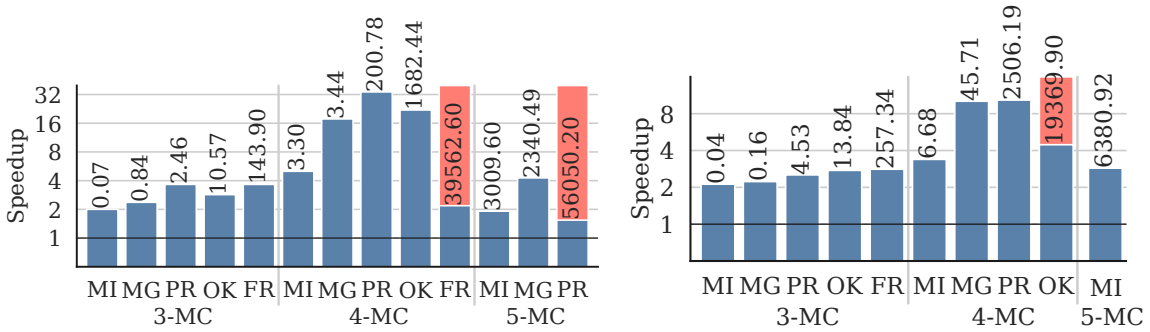
We evaluate using the available mining capabilities of each of the four systems to cover all cases. Hence, we use PEREGRINE and AutoZero for counting motifs and patterns, GraphPi and BiGJoin when counting vertex-induced patterns with a UDF to filter, and PEREGRINE for subgraph enumeration and FSM.

Subgraph Morphing was added in form of two modules external to the subgraph matching engines in these systems (see Figure 6.3), *i.e.*, the subgraph matching strategies and optimizations in these systems were left untouched.

AutoMine [146] uses a compilation-based approach to generate matching schedules and GraphZero [145] enhances the schedules using symmetry breaking (similar to [93]). Since neither AutoMine nor GraphZero have source code available, we developed an in-house version by faithfully implementing the symmetry breaking restrictions and performance model for choosing individual pattern schedules from [145]. Unlike AutoMine however, GraphZero does not merge the schedules of multiple input patterns. Hence we augmented our in-house implementation with schedule-merging, so that overlapping loops in different pattern schedules are merged together, and conflicting restrictions are applied separately to avoid under-counting. We name this augmented implementation AutoZero. AutoZero directly gen-

G	$\|V(G)\|$	$\|E(G)\|$	Num. Labels	Max. Deg.	Avg. Deg.
(MI) MiCo [82]	100K	1M	29	1359	22
(MG) MAG [110]	726K	5.4M	349	4779	14
(PR) Products [110]	2.4M	61M	47	17481	52
(OK) Orkut [225]	3M	117M	—	33133	76
(FR) Friendster [225]	65M	1.8B	—	5214	55

Figure 6.10: Real-world graphs used in evaluation.



(a) Peregrine (b) AutoZero

Figure 6.11: Performance improvements from Subgraph Morphing in Peregrine & AutoZero for Motif Counting relative to baseline system without morphing; absolute times (in seconds) for when Subgraph Morphing is enabled are shown on top of the bars. Red bars indicate the cases where baseline did not finish within 24 hours (*i.e.*, speedups for those cases are underestimated).

erates C++ code for subgraph matching schedules, and invokes `g++` version 10 to compile it. Across all the experiments, we did not measure the C++ code generation and compilation time for AutoZero.

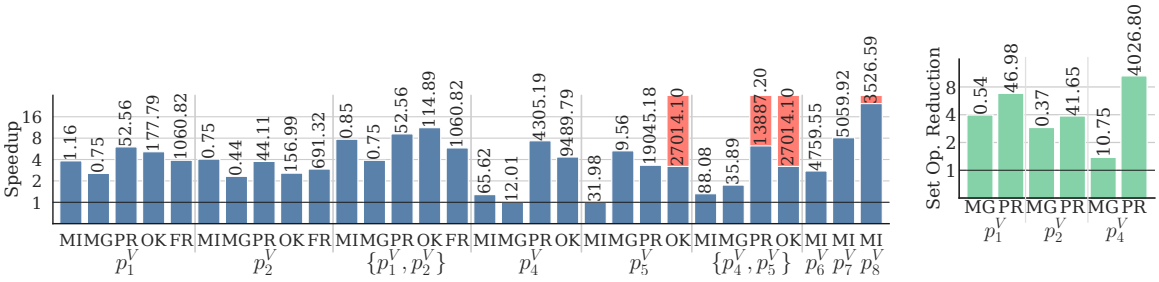
Experimental Setup. We investigate the impact of Subgraph Morphing on the performance bottlenecks identified in Section 6.1 through experiments on a wide array of applications: Motif Counting (MC) of size 3 to 5 vertices, FSM with size-3 and -4, as well as Subgraph Counting (SC) and Enumeration (SE) with patterns in Figure 6.9. Most of these patterns have been used in state-of-the-art evaluations [77, 192], and we have also included some larger and denser patterns in order to stress the systems.

Figure 6.10 lists the data graphs used in our evaluation. MiCo (MI) is a co-authorship graph labeled with each author's research field. MAG (MG) is an academic graph composed of several vertex types. We use the portion representing citations between papers, where papers are labeled by the venue they were published in. Products (PR) is a co-purchasing network, where vertices represent products, labeled by their category, and edges indicate two products are purchased together. Orkut (OK) and Friendster (FR) are unlabeled social network graphs where edges represent friendships between users. MiCo, Orkut and Friendster have been used to evaluate previous systems [205, 77, 146, 95], while MAG and Products are recent graph datasets designed to evaluate data mining tasks [110].

(a) Peregrine

(b) AutoZero

Figure 6.12: Reductions in set operation times from Subgraph Morphing for Motif Counting in Peregrine and AutoZero relative to baseline system without morphing. Bars are marked with mean absolute times (in seconds) from executions where Subgraph Morphing is enabled.



(a) Speedup

(b) Reduction in set operation times.

Figure 6.13: Performance improvements from Subgraph Morphing in Peregrine for Subgraph Counting.

All our experiments were run on a Google Cloud `n2-highcpu-32` instance, equipped with a 2.8GHz Intel Cascade Lake processor with 32 logical cores and 32GB of RAM. Across all experiments, we measured the end-to-end execution time, which includes input pattern transformation, mining computation, as well as result transformation. Since pattern transformation is done on the input patterns, we observed this phase took little time—for instance, transforming patterns of size 4 and 5 took at most 0.7 ms and 7.2 ms, respectively, whereas finding matches for those patterns on large graphs often takes 10s-1000s of seconds.

6.5.1 Morphing for Reducing Set Operation Time

Since counting is heavily bottlenecked by set operations in both Peregrine and AutoZero, we use **Motif Counting (MC)** as a representative benchmark. Figure 6.11 summarizes the results. Figure 6.12 shows that execution time for motif counting is dominated by set operations.

Compared to the baseline systems, applying Subgraph Morphing reduced set operation time in AutoZero and Peregrine by $3-22\times$ and $3.5-30\times$, respectively. This is because morphing the vertex-induced patterns in Motif Counting results in alternative pat-

	G	With Morphing
3-FSM	MI	71.3-195.1s (1.29×)
	MG	89.22s-2.64h (1.94×)
	PR	0.53-5.65h (2.26×)
4-FSM	MI	4.48-4.77h (3.61×)

Figure 6.14: Performance improvements from Subgraph Morphing in Peregrine for Frequent Subgraph Mining. *Minimum* speedups are reported in brackets. Note that 4-FSM on larger graphs did not finish in 24 hours since the complexity grows exponentially, requiring more resources (more machines and time).

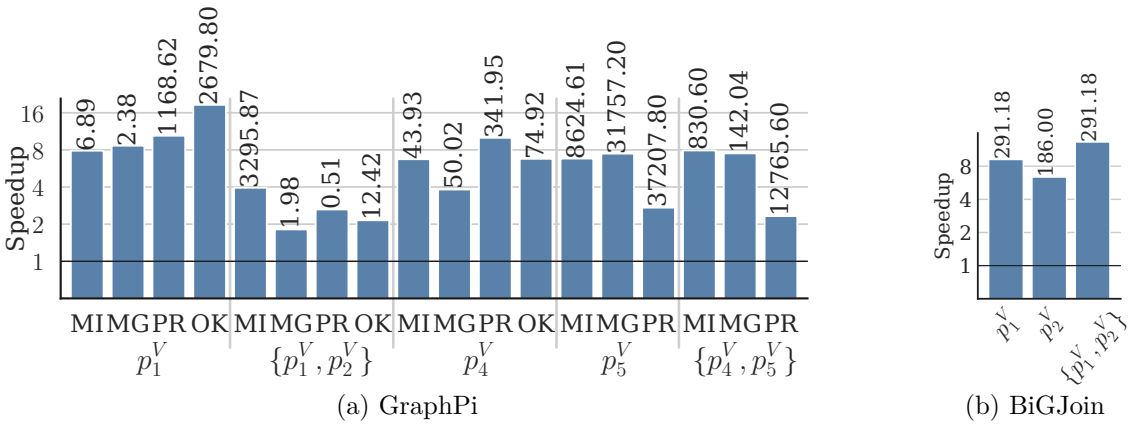
tern sets which contain fewer anti-edges. While anti-edges actively prune the search space and reduce the number of matches generated, each anti-edge necessitates an additional set operation (set difference) in the matching plan. Our selection algorithm identifies that the additional time required for set operations is not justified by the reduction in the number of matches since the counting aggregation is inexpensive.

The reduced set operation times translate to significant speedups in overall execution times, as seen in Figure 6.11a and Figure 6.11b. Subgraph Morphing yields a maximum 34× speedup in Peregrine for 4-MC on PR. The smallest speedup with Peregrine, 1.5× for 5-MC on PR, is an underestimation since the baseline Peregrine (without Subgraph Morphing) did not finish counting even half the patterns in 24 hours. In AutoZero, Subgraph Morphing yields $2-10\times$ speedups for motif counting, including a conservative 5× speedup in the OK 4-MC case, which the baseline system could not complete in 24 hours.

Subgraph Counting (SC). Motif Counting represents a best-case scenario for Subgraph Morphing, since all superpatterns are already contained in the input pattern set. Here we examine the converse situation in Figure 6.13(a-b), matching single patterns and pairs of patterns from Figure 6.9, such that few or no superpatterns are part of the input set. We use Peregrine for these experiments, since it matches patterns one by one, further exacerbating the cost of extra superpatterns. Due to limited space, we skip AutoZero, which gives the best case for Subgraph Morphing since merged matching plans significantly reduce the cost of extra superpatterns. Even with higher costs for superpatterns, Figure 6.13a shows that Subgraph Morphing speeds up Peregrine executions by $1.2-24\times$. The greatest speedup came from the large p_8^V pattern, which Peregrine could not mine without morphing. As expected, set operations are still the main bottleneck when matching individual patterns, and Subgraph Morphing reduces the time spent on them by $1.3-10\times$ (shown in Figure 6.13b), despite having to match extra patterns.

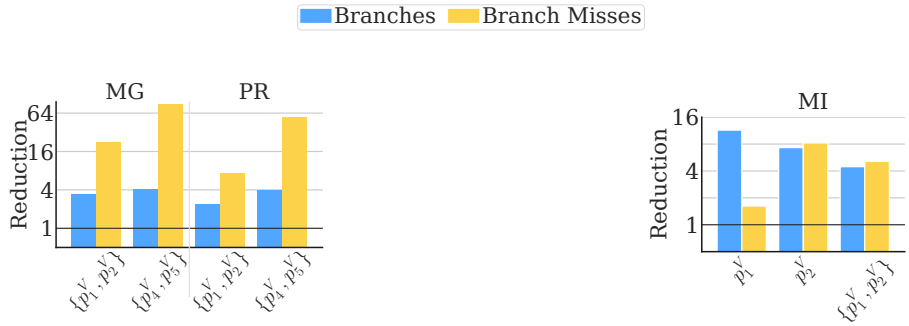
6.5.2 Morphing for Reducing UDF Overheads

We evaluate the effectiveness of Subgraph Morphing to address the key bottleneck in Frequent Subgraph Mining (FSM) and filter-based mining.



(a) GraphPi (b) BiGJoin

Figure 6.15: Performance improvements from Subgraph Morphing in GraphPi and BiGJoin.

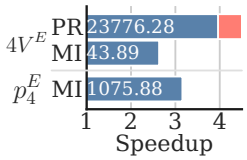


(a) GraphPi (b) BiGJoin

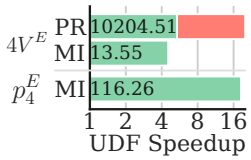
Figure 6.16: Reduction in branches and branch misses in GraphPi and BiGJoin relative to execution without Subgraph Morphing.

Reducing UDF overheads in FSM. Figure 6.14 summarizes the performance results when employing Subgraph Morphing in Peregrine for FSM. Subgraph Morphing alleviates the UDF bottleneck in 4-FSM on MiCo by morphing the patterns predicted to be most frequent into vertex-induced variants which will have fewer matches. For example, the edge-induced 4-Star pattern with all of its vertices sharing the most frequent label in the data graph is one of the most expensive patterns to mine in MiCo FSM due to the few constraints in the pattern combined with the frequent labeling, leading to over 5.7B matches. Morphing it generates 3 additional superpatterns (Tailed Triangle, Chordal 4-Cycle, and 4-Clique, all vertex-induced and with the same labeling), but results in 1.4B fewer matches, and as many fewer UDF calls (a reduction of 24%).

Similarly for other expensive patterns, the morphed patterns saved over 13 hours of time spent on UDFs while spending only 1 additional hour on set operations, yielding 3.6× speedup. 3-FSM on MiCo shows the least improvement, as both the patterns and the data graph are small, making the input patterns easy to match. Note that FSM operates on labeled patterns, which generally require more superpatterns during morphing.



(a) On-the-Fly



(b) On-the-Fly

Figure 6.17: Performance improvements from Subgraph Morphing in Peregrine for Subgraph Enumeration with On-the-Fly conversion (absolute times in seconds).

Eliminating Filter UDFs. We apply Subgraph Morphing to vertex-induced subgraph matching with GraphPi [192] and BiGJoin [17]. These systems lack native support for mining vertex-induced patterns; hence, extracting vertex-induced results requires matching the edge-induced variants and using a Filter UDF to remove matches with extra edges. With Subgraph Morphing, we compute vertex-induced results with edge-induced patterns without invoking any UDFs.

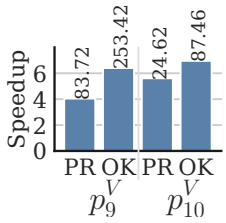
As shown in Figure 6.15, Subgraph Morphing significantly speeds up GraphPi and BiGJoin, by $1.4 - 18\times$ and $6.3 - 13.3\times$ respectively. This is because the native matching capabilities in these systems outweigh the expensive edge lookups in Filter UDFs even when multiple patterns must be matched.

We observed that 98% of execution time in the baseline system (without Subgraph Morphing) was spent in UDF calls. Drilling deeper reveals that the poor performance is primarily due to branches incurred on every match by the Filter UDF. Figure 6.16a and Figure 6.16b show that eliminating the UDFs using Subgraph Morphing reduces the number of branch misses by $30\times$ on average ($1.7 - 88\times$).

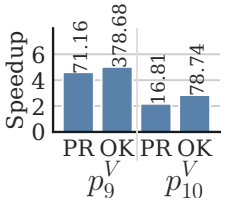
6.5.3 On-the-Fly Conversion

We evaluate the benefits of Subgraph Morphing for **Subgraph Enumeration (SE)** where on-the-fly conversion is employed to handle a stream of matches. We used Peregrine to enumerate matches comprised of vertices whose average weight is within a standard deviation of the distribution mean (vertex weights were assigned from a normal distribution). All edge-induced 4-vertex patterns ($4V^E$) on MiCo and Products, as well as p_4^E on MiCo were used in this experiment. Neither p_4^E nor its alternative pattern sets could be matched on Products within 24 hours.

Figure 6.17 summarizes the results. Since the filter is only dependent on the matched vertices, Subgraph Morphing trades the time spent filtering matches for additional set operations by morphing into vertex-induced patterns which have fewer matches, and then converting the matches that pass the filter on-the-fly. This reduces the time spent in UDFs by $5 - 16\times$, and as a result, speeds up the execution by $2.6 - 4\times$.



(a) Peregrine



(b) GraphPi

Figure 6.18: Performance improvements from Subgraph Morphing for large patterns.

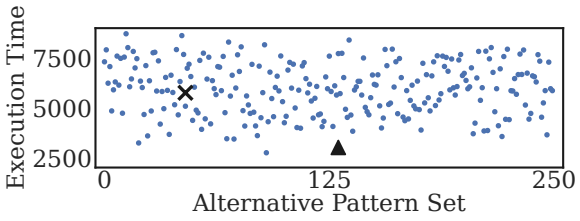


Figure 6.19: The space of alternative pattern sets for 5-motifs and their performance (in seconds) using Peregrine on MiCo graph. The input pattern set is marked by the cross and the set selected by the cost model is marked by the triangle.

6.5.4 Scaling to Large Patterns

We use patterns p_9 and p_{10} that contain 7 vertices. Such large patterns are uncommon in evaluations of graph mining systems. This is because graph mining workloads scale exponentially with pattern size (theoretical bottleneck due to NP-complete nature), making large patterns difficult to mine on single machine systems even for medium-sized data graphs. Since our goal is to show the effectiveness of Subgraph Morphing on large pattern workloads, we control the data graph size to limit the workload size so that executions can finish on a single machine in reasonable time. We do so by partitioning the Products and Orkut graphs using METIS [121], and using Peregrine and GraphPi to mine p_9^V and p_{10}^V within the partitions. This way, the edges between partitions are dropped out, which reduces the workload size.

As shown in Figure 6.18a and Figure 6.18b, morphing speeds up enumeration on Peregrine by $4-7\times$, while it also improves enumeration on GraphPi by $2-5\times$. We observed that the analyses from Section 6.5.1 and Section 6.5.2 apply to large patterns as well. With Subgraph Morphing incorporated, Peregrine spent $3-11\times$ less time on set operations, while GraphPi incurred $2.2-46\times$ fewer branches.

6.5.5 Cost Model Effectiveness

A given input pattern can have exponentially many alternative sets, leading to potentially large gaps in performance. We study the effectiveness of our cost model in identifying the right alternative pattern set that delivers performance. Figure 6.19 shows the performance of 250 alternative pattern sets for 5-motif counting on MiCo, including the query pattern

set and the set chosen by the cost model. The optimal set is over 3× faster than the slowest. The cost model chose an alternative pattern set which performs within 10% of the optimal one.

Several alternative sets perform worse than the query set; while this is visible in Figure 6.19 for 5-motif counting, it is especially clear in FSM which involves many patterns. For 3-FSM on PR with support threshold 140K, blindly morphing all input patterns leads to an execution time of over 22 hours, whereas the query pattern set takes 14 hours and the cost model selects a set taking only 5.65 hours.

6.6 Conclusion

We presented Subgraph Morphing, a general technique to accelerate graph mining workloads across various graph mining systems. We exposed key factors that impact the performance of graph mining workloads, and observed there is no singular bottleneck that is common across the different workloads running on different graph mining systems. Subgraph Morphing exploits performance differences across pattern structures while also incorporating key system-level and application-level characteristics to deliver high performance. We formalized Subgraph Morphing and developed efficient strategies to enable it in practice. Our extensive evaluation showed promising results.

Chapter 7

OsirisBFT: Application Semantics in Distributed Architecture

This chapter explores the potential of application semantics to address crucial and ubiquitous system design challenges beyond those covered by the static single-machine setting used in Peregrine and the system-agnostic approach of Subgraph Morphing. Namely, this chapter centres distributed graph mining on dynamically changing data in order to expose scalability and fault tolerance challenges intrinsic to developing available and reliable high-performance analytics services.

Many graph mining use cases take place in an online setting, where application values must be maintained as the underlying dataset changes. For instance, the graph anomaly detection application seeks to identify anomalous structures in a rapidly changing graph [63]. Because individual changes (*i.e.*, edge additions or deletions) only have local effects on subgraph structure, processing the whole graph in response to each modification is redundant. Instead, *incremental graph mining* applications compute only the change in aggregation values by computing the difference that a graph modification causes in the application-defined interesting subgraph set S [34, 17, 125, 120].

To magnify the scalability and fault tolerance challenges further, we target the *Byzantine failure model*, where faulty machines can behave arbitrarily. Although incremental graph mining applications are common in various settings, Byzantine failures are seldom considered despite occurring frequently in practice [153, 163, 119]. For instance, applications in settings like cybersecurity [112], business intelligence [204], and fraud detection [207] are open to threats where adversaries are incentivized to cause failures in order to gain access to user data, create outages, or commit unchecked fraud. Similarly, accidental Byzantine failures are also pernicious. Physical failures like memory corruption can create subtle flaws in the output of a computation even despite the presence of Error-Correcting Codes [188, 153, 136], with significant humanitarian [155] and legal [57] consequences.

Use Case: Graph Anomaly Detection. This application maintains an up-to-date version of the graph using a continuous stream of edge updates (insertion/deletion of edges),

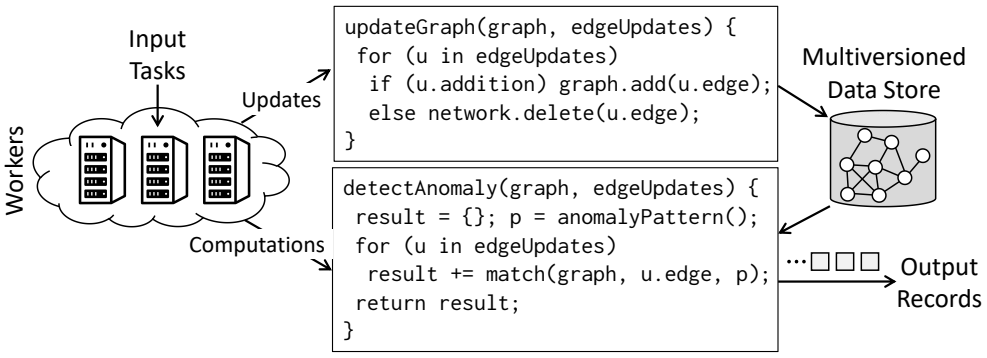


Figure 7.1: Anomaly Detection. Update tasks modify the data store and computation tasks perform pattern matching on the modified graph.

and performs pattern matching on the updated portion of the graph to identify matches of anomalous patterns [63]. As shown in Figure 7.1, the updates are applied to a multiversioned data store and multiple tasks perform pattern matching in parallel using the appropriate versions of the network. The pattern matching computation (`detectAnomaly()` in Figure 7.1) is orders of magnitude more expensive than performing edge updates in the data store (`updateNetwork()` in Figure 7.1). Graph anomaly detection is a strong representative for incremental graph mining applications since it captures the core computation (*i.e.*, computing matches) and generates large output (*i.e.*, all matches of an anomalous pattern) to quickly expose system scalability bottlenecks and will be used as the primary example throughout this chapter.

Here, Byzantine failures can affect the graph in the data store as well as the pattern matching computation. In the first situation, faulty workers can apply incorrect/malicious edge updates resulting in inconsistent views of the graph, hence generating unreliable results computed from inconsistent data. To safeguard against such failures, various Byzantine fault tolerant protocols for managing state have been developed, for example Kauri [160], Basil [200] and others [41, 1, 66, 129, 106, 229, 19, 126, 8]. These protocols target applications composed of read/write transactions where ordering requests via BFT consensus is the major bottleneck.

In the second situation, even if the data store is maintained correctly, faulty workers performing pattern matching on correct data can result in incorrect output. For this, solutions like Medusa [65] and others [198, 64, 169, 157] enable BFT for applications dominated by computation instead of consensus. Like these works, this chapter targets scalable computation and high output record throughput.

Unlike traditional crash failures, Byzantine failures cannot be overcome with straightforward crash recovery mechanisms like checkpointing, because faulty processes can silently corrupt intermediate results. As a result, at the heart of these BFT solutions are replicated state machine protocols (RSM) that replicate both application state and task execution [186]. Specifically, workers are divided into independent subsets of replicas that all

maintain the same application state and execute the same tasks, such that different subsets maintain distinct partitions of application state and execute different tasks in parallel. In such an approach, safety (*i.e.*, results should be correct) and liveness (*i.e.*, downstream processes should receive results) are guaranteed if all subsets contain $O(f)$ workers and at most f workers in each subset are faulty, as a majority of replicas will compute the correct result. However, replicating application tasks in such RSM-based systems imposes significant theoretical and practical limits on scalability and processing throughput.

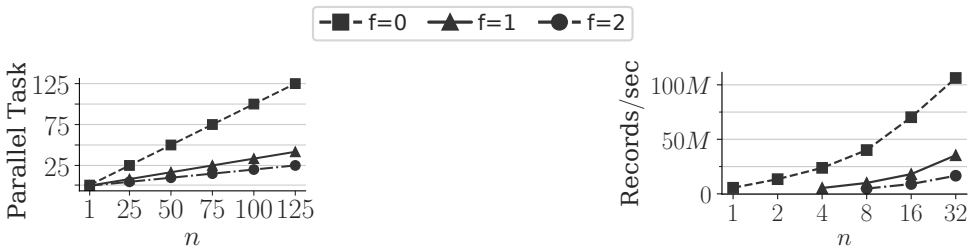
Limited Task Capacity. Replication theoretically limits the number of tasks that can be executed in parallel to a fraction of the cluster's actual hardware capacity. A cluster of n workers can execute at most $\lfloor n/(2f+1) \rfloor^{\dagger}$ replicated tasks in parallel. This means even with minimum fault tolerance $f = 1$, processing with RSM requires $3\times$ the computation resources as a regular execution. Figure 7.2a shows the number of tasks that can be executed in parallel with RSM as a function of cluster size, and we corroborate this analysis by measuring the processing throughput in terms of number of result records generated per second for Anomaly Detection in Figure 7.2b. As seen, RSM-based processing on 32 nodes with $f = 1$ achieves similar throughput to only 8 nodes without fault tolerance.

Local Failure Bounds. Scaling an RSM-based system in practice requires that the failure bounds of each replica group can be realistically achieved. In traditional Byzantine fault tolerant applications like data stores which run on small clusters, techniques like n-version programming and geo-distribution can be used to prevent correlated failures from quickly overwhelming cluster redundancy and compromising safety or liveness. However, such solutions quickly become cost-prohibitive for large analytics clusters.

Key Insight. The computation in incremental graph mining applications involves multiple steps that are performed iteratively (*i.e.*, matching the pattern step-by-step). Hence, computation tasks are often orders of magnitude more expensive than state updates since the latter only involve agreement on the ordering of updates and modifying the underlying graph. Although incremental graph mining tasks are time-consuming to execute, their results can be verified much more quickly. This is because verification simply involves checking whether the results satisfy application semantics (*e.g.*, whether the reported anomalous subgraphs indeed match the pattern) which is much simpler than computing the solution itself. Hence, *with verification being much faster than the original computation, BFT executions can be guaranteed without replicating the computation by judiciously verifying results.*

Our Approach. OsirisBFT separates state management from computation task execution and explores the possibility of BFT without replicating expensive graph mining tasks. The result is a distributed BFT architecture for incremental graph mining applications backed by two components: a BFT data store for managing global state, and verification-based

$^{\dagger}\lfloor n/(2f+1) \rfloor$ tasks using non-equivocation from modern RDMA networks [6] or trusted hardware [135]; otherwise the bound degrades to $\lfloor n/(3f+1) \rfloor$.



(a) Task Capacity (b) Throughput

Figure 7.2: Scaling of RSM-based processing for Anomaly Detection (*i.e.*, with `detectAnomaly()` replicated) assuming at most f failures per replica group.

processing for application computation. Prior solutions [6, 126, 28, 229] already provide efficient BFT state management as discussed above. We incorporate an existing RSM-based BFT design [6] for our data store, and primarily focus on developing an efficient verification-based processing architecture.

OsirisBFT. OsirisBFT decouples task computation from fault tolerance by offloading the responsibility of detecting faults to a special subset of workers called *verifiers*. Regular workers execute computation tasks, and their results are analyzed by verifiers to protect against failures. Hence, workers executing computation tasks need not be replicated to ensure safety, enabling scalable execution. Furthermore, verifiers check the generated results independently, and only perform consensus to linearize input tasks. Hence, OsirisBFT can execute $n - O(f)$ parallel tasks in a cluster with n workers, as opposed to $n/O(f)$ in RSM.

A verification-based BFT processing architecture seems promising, as reducing replication addresses both the task capacity and failure bound assumptions that hinder scalability. However, realizing OsirisBFT in practice poses several challenges.

The first challenge is *how to distinguish Byzantine executions from graceful executions that are not impacted by Byzantine failures?* Byzantine failures can impact the execution and violate the application semantics in various ways (*e.g.*, partially executing tasks, repeatedly executing the same task, simply outputting results that appear valid but do not satisfy the task requirements, *w.r.t.*). Developing custom verification protocols to identify these different behaviors can easily become intractable, especially since several of these issues require understanding the application semantics to distinguish a Byzantine behavior from a correct one.

To address this, we develop an output failure model that captures how Byzantine failures impact the application results. Our model groups all possible application failures into three classes of *output failures*. We then formalize *verifiability properties* required to detect each class of output failures, and develop *verification operators* that allow our processing architecture to capture the required incremental graph mining application semantics so that verifiers can safeguard against all classes of output failures. As further evidence for the power of application-awareness, we show that output failures, verifiability properties, and

verification operators all apply not only to incremental graph mining applications, but to a broader class of *task-parallel* applications.

The second challenge is *how to perform verification robustly and efficiently?* While verification operators capture faults that are observable from application results, verifiers themselves can be faulty, which can in turn cause complex failures even if workers correctly execute their tasks.

For verification that is both efficient and resilient, we develop robust and lightweight protocols. Verifiers certify the application results using the verification operators, with zero coordination among the verifiers during graceful executions. To achieve robustness in our verification pipeline, we rely on redundancy in communication between verifiers and other actors in OsirisBFT. Our careful use of cryptography, timeouts, and limited use of heavy communication primitives like non-equivocating multicast capture complex failure cases while retaining efficiency when processes are well-behaved.

The final challenge is *how to maintain resource utilization and processing throughput as processing workload varies over time?* As processing workload changes over time, tasks demanding high computation can keep workers busy even though the verification workload remains low. On the other hand, failed workers leaving the system can result in throughput drops that can persist in the remaining execution. We design a dynamic role-switching strategy to improve resource utilization and processing throughput across different processing conditions.

Results. To the best of our knowledge, this thesis provides the first treatment of enabling Byzantine fault tolerance for incremental graph mining applications (as well as task-parallel applications more broadly) without replicating application computation. OsirisBFT is backed by safety and liveness proofs to ensure correctness under all circumstances and progress even in presence of Byzantine failures. We evaluated OsirisBFT with graph anomaly detection and two other distributed task-parallel applications, as well as across different processing workloads. Our results show that OsirisBFT delivers high processing throughput and better scalability compared to replicated processing, and it scales comparably to a baseline without any fault tolerance. Importantly, OsirisBFT overcomes the performance overheads from ensuring fault tolerance by simply scaling out.

7.1 Overview of OsirisBFT

The system is modeled as a pipeline with three steps: (i) input processes IP generate or ingest tasks and distribute them downstream; (ii) worker processes WP execute the tasks and output a sequence of records; and, (iii) output processes OP receive the results. IP and OP can overlap. Tasks can involve state updates (*e.g.*, `updateNetwork()` in Figure 7.1), computation (*e.g.*, `detectAnomaly()` in Figure 7.1), or both. This pipeline follows the general ar-

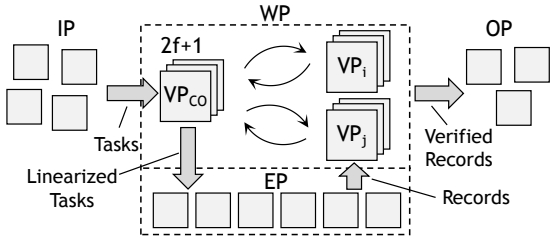


Figure 7.3: Verification-based processing architecture.

chitecture of existing distributed incremental graph mining systems like Delta-BiGJoin [17] and Tesseract [34].

Figure 7.3 shows the verifiable processing architecture. WP is divided into two sub-clusters: the ***execution cluster*** EP and the ***verifier clusters*** VP. The execution processes (or simply, ***executors***) execute computation tasks and output records, whereas the verifier processes (or simply, ***verifiers***) deal with verification of the generated records. A computation task is executed on each input exactly once by an executor in EP (*i.e.*, no task replication). VP is partitioned further into k independent Byzantine fault tolerant sub-clusters $VP_0 ... VP_{k-1}$ with each $|VP_i| \geq 2f + 1$ $(0 \leq i < k)$. One of the verifier sub-clusters is arbitrarily chosen to be responsible for performing consensus to linearize tasks and coordinating the remaining processes throughout the entire execution; we refer to this sub-cluster as the ***coordinator*** VP_{CO}.

State Management. The state management layer resembles the learner architecture [105]. For simplicity and maximal use of hardware resources, the application state is colocated with WP. As we discuss later, we make no assumptions about failures in EP. To safely perform concurrent state updates, the coordinator sub-cluster VP_{CO} linearizes tasks to enforce a global order on state updates, and keeps the rest of WP appraised so correct processes can maintain fresh, globally consistent copies of their state. This design avoids inflating the cost of queries with read requests to a disaggregated storage system or cross-shard transactions in a sharded solution by ensuring all processes maintain a local copy of the state, since analytics queries frequently perform reads.

Verifiable Processing. OsirisBFT enables scalability by placing all responsibility for Byzantine fault tolerance on VP, freeing EP to execute tasks without overheads. Tasks flow from IP to the coordinator VP_{CO}. VP_{CO} linearizes the tasks and broadcasts state updates to WP while distributing computation tasks among EP. State updates mutate local application state, and computation tasks operate on the local state to produce output records. While every state update is sent to all of WP, each computation task is assigned to a single executor at a time and reassigned only if a failure is suspected. Then, the results of computation tasks flow from EP to VP to OP. Each output record is sent to $2f+1$ verifiers in a Byzantine fault tolerant sub-cluster VP_i for verification to ensure output processes only observe correct records.

	Computation Replication	Computation Scalability	Communication Replication	Faults Tolerated
ZFT	1	$\|WP\|$	1	0
RCP	$2f+1$	$\|WP\|/O(f)$	1	$\sum_{WP_i} f$
OsirisBFT	1	$\|WP\| - O(f)$	$2f+1$	$\|EP\| + \sum_{VP_i} f$

Table 7.1: Performance and fault tolerance of OsirisBFT compared to replicated computation strategy (RCP) and a baseline with no fault tolerance (ZFT).

Since only verifiers interact with the downstream and upstream processes, correct processes in IP and OP never observe failures in EP, even though computation tasks are never replicated.

Computation-Communication Tradeoff. Table 7.1 shows the computation redundancy, the communication redundancy, the fault tolerance, and the computation scalability provided by OsirisBFT, compared with the RSM-based replicated computation strategy (RCP) where WP is divided into sub-clusters WP_i of $2f+1$ processes each, and computation is replicated in all processes in a sub-cluster.

OsirisBFT optimizes for application computations. It favors replicating communication rather than computation when possible, leveraging ample bandwidth in high performance networks to maximize utilization of cluster resources. Each output record is replicated over the network to $2f+1$ verifiers in VP_i, and in exchange, computation tasks are not replicated in graceful executions.

Hence, $O(f)$ processes verify records while $|WP| - O(f)$ processes execute tasks. The number of verifier sub-clusters can be kept small relative to $|WP|$, hence achieving higher performance than RCP by not replicating the expensive application computation, and only replicating the lightweight verification. Moreover, OsirisBFT tolerates faults more freely, since no executor is assumed correct. Each VP_i tolerates f failures (similar to WP_i in RCP); in addition, OsirisBFT tolerates complete failure of EP. Hence executors, and the application, can scale independently of f.

Throughput-Latency Tradeoff. OsirisBFT adopts a throughput-focused architecture oriented toward use cases where input tasks or output records pass through the system at high volumes. When there are fewer active computation tasks than there are available workers and the application produces little output, verifying results introduces additional latency with no benefit to throughput while a replicated architecture could communicate results directly to downstream processes. By concentrating on throughput, OsirisBFT scales far better than replicated-compute processing when executing incremental graph mining and other large-scale task-parallel applications, where the arrival rate of input tasks and the output rate of records can quickly saturate system resources.

7.2 System Model

Service Guarantees. OsirisBFT adopts the Byzantine failure model, where processes can behave arbitrarily, including crashes, adversarial failures, and coordination between malicious processes. We define safety and liveness to limit the impact of Byzantine faults in WP. For all i, if at most f processes in VP_i fail, and VP_i contains $2f + 1$ processes that can verify output records from other workers, OsirisBFT is linearizable *(safety):* all correct OP observe records corresponding to a legal sequential execution of correct tasks submitted by IP. Furthermore, all correct OP eventually observe results for every task submitted by IP *(liveness)*. Note that safety is not compromised even if all processes in EP are faulty. However, the system is also bound by assumptions made by its state management layer. As mentioned above, we assume the state is managed by the Byzantine fault tolerant VP processes, and EP learn of state updates from VP. If state must be safely stored on EP using a different approach then additional assumptions about failures in EP may be necessary. We make no assumptions about the number of failures in IP or OP.

We assume that adversaries have finite resources proportionate to correct processes, and cannot overwhelm correct processes with network traffic or break cryptographic primitives like digital signatures. Hence by authenticating all communication, correct processes cannot be impersonated.

Safety can be guaranteed if the system is asynchronous, and we make the standard assumptions from previous work [41, 229, 200, 160] regarding partial synchrony for liveness: there is some known Δ and unknown global synchronization time (GST) such that after GST, all messages between correct processes arrive with maximum latency Δ [78].

Communication Primitives. To achieve fault tolerance with $2f + 1$ processes in a sub-cluster instead of the well-known lower bound of $3f+1$ processes [36], our techniques rely on a multicast primitive that guarantees non-equivocation of certain messages (*e.g.*, Reliable Broadcast using RDMA [6] or trusted hardware [135]). In conjunction with digital signatures, non-equivocating multicast enables atomic delivery of a message to $2f + 1$ processes where f are faulty [59]. Such primitives are relatively heavyweight, and hence they are used sparingly. For situations where non-equivocating multicast is not available, OsirisBFT can operate with $3f + 1$ processes in each sub-cluster. All other messages use reliable links that guarantee messages are not dropped or reordered (*e.g.*, using RDMA RC protocol [118]).

7.3 Identifying Application Faults

OsirisBFT detects violations due to Byzantine failures by verifying the output records produced by executors. In this section, we model the impact of Byzantine failures on application results and develop verification operators to efficiently validate the records returned by executors.

7.3.1 Incremental Graph Mining

Incremental Application Semantics. A graph mining application *App* is *incremental* if it is possible to compute the change in value of $App(g)$ in response to a change in g. This chapter centres incremental graph mining applications as its primary focus, but the techniques developed here apply equally to the broader class of task-parallel applications described in Section 7.3.5.

Consider a graph mining application *App* as defined by Eq. 2.1. Let g_0 be an initial data graph, let $g_1, g_2, \ldots$ be a series of distinct graphs, and let $S_i \subseteq S_{g_i}$ be the subgraph set processed by $App(g_i)$. Since g_i and g_{i-1} are distinct, there is a non-empty set containing the edges present in g_i but not g_{i-1} and the edges present in g_{i-1} but not g_i, denoted by $g_i \triangle g_{i-1}$ (following the notation for symmetric difference of sets). For an aggregation $\langle \mathcal{R}, \oplus \rangle$ forming an *abelian group*[†], we define

$$App(g_i \triangle g_{i-1}) = \bigoplus_{s \in S_i \setminus S_{i-1}} \nu(s) \;\oplus \bigoplus_{s \in S_{i-1} \setminus S_i} -\nu(s)$$

where $-\nu(s)$ is the inverse of $\nu(s)$ in $\mathcal{R}$. Then for $i > 0$, an incremental graph mining application can be written

$$App(g_i) = App(g_{i-1}) \oplus App(g_i \triangle g_{i-1}).$$

Example 7.3.1 (Anomaly Detection)**.** Anomaly detection is a form of pure subgraph matching, with $\mathcal{R} = \mathbb{P}(S_{g_i})$, and $\nu(s) = s$. To ensure that $App(g_i \triangle g_{i-1})$ is well-defined, the aggregation must form an abelian group. Therefore, we define $\oplus$ as symmetric difference (*i.e.*, $\triangle$) instead of set union, since any powerset is an abelian group under symmetric difference [91] with each element acting as its own inverse. This choice of aggregation maintains identical semantics as the Subgraph Matching application presented in Section 2.2.1, since any two subgraph sets generated by a matching algorithm and subsequently aggregated are disjoint, while symmetric difference and set union are equivalent when applied to disjoint sets. The set of anomalous subgraphs computed in $App(g_{i-1})$ is simply S_{i-1}, and $App(g_i \triangle g_{i-1})$ computes the union of $S_{i-1} \setminus S_i$ and $S_i \setminus S_{i-1}$ since the two sets are disjoint by definition, which is conveniently the symmetric difference of S_i and S_{i-1}. Therefore the

[†]An abelian group is a commutative monoid where every element has an inverse. If the subgraph set S_i has fewer subgraphs than S_{i-1}, then the change in aggregation value can only be computed by *removing* the contribution of some subgraphs, necessitating inverses.

symmetric difference of $App(g_{i-1})$ and $App(g_i \triangle g_{i-1})$ is

$$\begin{aligned} App(g_{i-1}) \oplus App(g_i \triangle g_{i-1}) &= S_{i-1} \triangle (S_{i-1} \triangle S_i) \\ &= (S_{i-1} \triangle S_{i-1}) \triangle S_i \\ &= \emptyset \triangle S_i \\ &= S_i = App(g_i). \end{aligned}$$

Hence, anomaly detection has incremental application semantics.

Modeling System Execution. The system consumes a stream of tasks as input and produces a stream of records as output. Formally, applications operate on global states $\mathcal{S} = \{g_0, g_1, \dots\}$, possible output records drawn from the aggregation value set $\mathcal{R}$ (*e.g.*, all possible sets of subgraphs in anomaly detection) and tasks drawn from a set $\mathcal{T} = \mathbb{N} \times \mathbb{N}$ representing potential edges (referred to by pairs of integer vertex ids) to be modified.

Tasks are processed using a pair of functions $\mathcal{U}$ and $\mathcal{A}$, assuming a base state g_0. For a graph $g_i \in \mathcal{S}$ and $e \in \mathcal{T}$, $\mathcal{U}(g_i, e)$ returns a new global state $g_j \in \mathcal{S}$ obtained by adding edge e to g_i if it did not already exist, or removing e from g_i if it already existed. On the other hand, $\mathcal{A}(g_i, e)$ executes $App(g_i \triangle g_{i-1})$ and returns a sequence of records $R = [r_0, r_1, ...]$ such that $\forall_{r_i \in R}\ r_i \in \mathcal{R}$.

By appending an opcode to each task describing whether to execute $\mathcal{U}, \mathcal{A}$, or both, this model captures common use cases such as: (i) event-driven analytics where computation occurs in response to an update (*i.e.*, tasks call for both a state update $\mathcal{U}$ and a computation $\mathcal{A}$); (ii) time-based analytics where computation and updates are decoupled (*i.e.*, some tasks call only $\mathcal{U}$ and appear whenever updates arrive, others call only $\mathcal{A}$ and appear periodically to compute analytics logic); (iii) batch processing where the state is static (*i.e.*, tasks never call $\mathcal{U}$); as well as (iv) classic state management applications (*i.e.*, tasks never call $\mathcal{A}$).

7.3.2 Output Failure Model

A faulty worker can impact the output generated by graph mining applications in various ways. We categorize the impact of arbitrary faults as three types of *output failures*.

[Mismatch] An output record r corresponding to task t is a *mismatch* if it does not satisfy the problem statement of t (*i.e.*, $r \notin \mathcal{R}$ or $r \notin \mathcal{A}(s, t)$). A faulty process in WP can invalidate downstream computations by generating correct records for the wrong task, or simply random records.

[Duplication] A faulty process in WP can perform a replay attack by outputting a record r multiple times. An output record r corresponding to task t is a *duplication* if it has been output previously in the result stream for t. Duplication can skew the output distribution and hence break applications.

[**Omission**] A faulty process in WP can omit portions of the output (*i.e.*, produce a strict subset of $\mathcal{A}(s,t)$). For example, a malicious process can hide suspicious records from downstream analysis in a cybersecurity application.

The output failure model is *complete* with respect to Byzantine failures from the perspective of application output: if a certain sequence of records is expected, incorrect results can only arise due to MISMATCH, DUPLICATION, or OMISSION.

Lemma 7.3.1. *All invalid records produced by an executor correspond to an output failure.*

Proof. We proceed by contradiction. If an executor neglects to produce any records, it will be classified as OMISSION. So suppose that an executor produces an invalid record r which does not qualify as an output failure. To avoid MISMATCH, $r \in \mathcal{R}$ and $r \in \mathcal{A}(s_t, t)$, for some valid task $t \in \mathcal{T}$ and corresponding state $s_t \in \mathcal{S}$. But then either r repeats in $\mathcal{A}(s_t, t)$ (classified as DUPLICATION), or r is valid since it follows all application semantics. □

Lemma 7.3.2. *Correct processes executing $\mathcal{A}$ do not generate output failures. Faithfully executing $\mathcal{A}$ with correct tasks does not generate output failures.*

Proof. Given a valid task $t \in \mathcal{T}$ and corresponding state $s_t \in \mathcal{S}$, $\mathcal{A}(s_t, t)$ does not result in an output failure by definition. Therefore the only way for a correct process to produce an output failure is if $\mathcal{A}$ is executed with an invalid task $t \notin \mathcal{T}$ or a state s such that $s \notin \mathcal{S}$ or s does not correspond to t. But this is impossible by Lemma 7.5.1, which proves that correct processes share the same view of the global application state, corresponding to a consistently ordered sequence of valid tasks. Therefore, no correct process will observe an incorrect state or invalid task while executing $\mathcal{A}$. □

7.3.3 Properties for Verification

An incremental graph mining application is *verifiable* if it satisfies the following four properties:

[**Task-Validity**] For an arbitrary object t, it is possible to determine whether $t \in \mathcal{T}$ (*i.e.*, whether $\mathcal{A}(s,t)$ is defined for arbitrary $s \in \mathcal{S}$).

[**Task-Scope**] For an arbitrary record r, it is possible to determine whether $r \in \mathcal{R}$ (*i.e.*, whether $\mathcal{A}$ can produce r).

[**Task-Ordered**] For every $t \in \mathcal{T}$ and $g \in \mathcal{S}$, $\mathcal{A}(g,t)$ is totally ordered.

[**Task-Bounded**] For every task $t \in \mathcal{T}$ and $g \in \mathcal{S}$, $\mathcal{A}(g,t)$ is finite.

The TASK-VALIDITY property prevents MISMATCH failures where Byzantine input processes submit invalid tasks to be executed. A worker executing a task it was not assigned implies either MISMATCH (*i.e.*, no input process generated the task) or DUPLICATION (*i.e.*,

a different worker was assigned the task). The TASK-SCOPE property distinguishes valid and invalid records, so that MISMATCH failures involving incorrect or nonsensical records can be identified. The TASK-ORDERED property represents the process-local program order of the executing worker. A worker executing a task in a TASK-ORDERED application produces records in a specific order, and hence out-of-order output would imply DUPLICATION or MISMATCH. Finally, the TASK-BOUNDED property requires that applications guarantee termination. Without this property, it is impossible to detect OMISSION because observed output from a worker process cannot necessarily be compared with the expected output of $\mathcal{A}$ (*i.e.*, they can both be infinite), which makes it impossible to identify whether a record is missing.

Example 7.3.2 (Anomaly Detection)**.** The anomaly detection application satisfies all of the properties above.

[**Task-Scope**] It is possible to determine whether a record r is truly an anomalous subgraph by ensuring all the edges in r exist in the graph and there exists a graph isomorphism between r and the anomaly pattern.

[**Task-Ordered**] Any deterministic subgraph matching algorithm implicitly defines an ordering on its output based on program order (*e.g.*, since the algorithm can be viewed as a series of nested loops) and the layout of data graph edges in memory. Hence, there is a total ordering on the set of subgraphs produced by $\mathcal{A}(g, t)$.

[**Task-Bounded**] A finite graph has a finite subgraph set, so $\mathcal{A}(g, t)$ can only produce finitely many anomalous subgraphs.

While many incremental graph mining applications satisfy these properties, even those that do not can be executed in OSIRISBFT by decomposing the application into separate subgraph generation and aggregation steps. As Example 7.3.2 shows, incrementally listing subgraphs is verifiable. Therefore, the results of non-verifiable incremental graph mining applications can be checked by verifying the subgraphs provided as input to the aggregation operator, and then offloading aggregation to the replicated verifiers or to downstream processes (as in Tesseract [34]). In this fashion, aggregation computations are performed by trusted or fault tolerant processes, while subgraph generation is verified.

7.3.4 Output Verification Model

Depending on the nature of the failures, they can be detected by: (a) employing generic protocols in the underlying system; or, (b) verifying output records against application semantics. Generic verification will be discussed in Section 7.4.2. Here we enable application-specific verification.

Algorithm 4 API for verification operators.

```
bool isValid(Record r, Task t);
bool happensBefore(Record a, Record b);
int outputSize(Task t);
```

Verification Operators. We model application-specific *verification operators* that analyze output records. Verifiable applications implement these operators, which are invoked by verifiers (explained later in Section 7.4.2).

Algorithm 4 shows the three verification operators. `isValid()` checks whether a record r is valid (*i.e.*, $r \in \mathcal{R}$) and is generated by the given task t. `happensBefore()` captures the process-local program order of the executing worker by checking whether a record a is ordered before record b. Finally, `outputSize()` returns the number of output records for a task t. The verification operators are also *complete*, *i.e.*, they combine to detect all types of output failures (see proof in Section 7.5.1). MISMATCH is detected by `isValid()` and `outputSize()` that together ensure the output records are the ones expected from the tasks. DUPLICATION is detected using `happensBefore()` and `outputSize()` that identify repeated records arriving from the correct task. OMISSION is detected using `outputSize()`.

Example 7.3.3 (Anomaly Detection)**.** Algorithm 5 illustrates the verification operators for anomaly detection, where the computation primarily involves pattern matching. `isValid()` ensures that each subgraph record is indeed a subgraph of the network graph, matches the pattern, and contains the updated link that resulted in the task to compute that record. `happensBefore()` determines the order between two subgraph records based on their prefix-ordering (prefix-ordering is guaranteed by most pattern matching systems like PEREGRINE or GraphPi [192]). Finally, `outputSize()` simply returns the true number of subgraphs using efficient and exact counting optimizations (*e.g.*, inclusion-exclusion [192] or SUBGRAPH MORPHING) which are orders of magnitude faster than matching each individual subgraph.

Algorithm 5 Verification operators for Anomaly Detection.

```
// Network is network graph. Pattern is pattern to match.
// PatternMatcher contains the matching logic.
bool isValid(Record r, Task t) {
  return isSubgraph(Network, r) && isMatch(Pattern, r)
    && r.links().contains(link(t));
}
bool happensBefore(Record a, Record b) {
  for(int i=0; i<a.length(); ++i) {
    if(a[i] < b[i]) return true;
    if(a[i] > b[i]) return false;
    // if(a[i] == b[i]) continue;
  }
  return false;
}
int outputSize(Task t) {
  PatternMatcher.count(Network, Pattern, t);
}
```

7.3.5 Verifiability Beyond Graph Mining

The output failure model, verifiability properties, and verification operators can be applied to the broader class of *task-parallel* applications that encompasses incremental graph mining.

Task-Parallel Applications. Task-parallel applications also consume a stream of tasks as input and produce a stream of records as output. Instead of graphs, the set $\mathcal{S}$ contains arbitrary application-specific global states, and a task $t \in \mathcal{T}$ consists of application-defined data that will be passed as input to $\mathcal{U}/\mathcal{A}$. Task-parallel applications generate records drawn from a set $\mathcal{R}$ without any particular algebraic structure. Similar to the incremental graph mining case, $\mathcal{U}(s,t)$ updates a global state $s \in \mathcal{S}$ based on task $t \in \mathcal{T}$ and returns a new state; and $\mathcal{A}(s,t)$ executes an application-specific computation on a global state $s \in \mathcal{S}$ and task $t \in \mathcal{T}$ and returns a sequence of records $R = [r_0, r_1, ...]$ such that $\forall_{r_i \in R}\ r_i \in \mathcal{R}$.

Generalizing Verifiability. Lemma 7.3.1 and Lemma 7.3.2 apply naturally to task-parallel applications, therefore the output failure model remains complete with respect to task-parallel applications, and any task-parallel application that implements the verification operators can be executed safely and maintained live by OsirisBFT.

Task-parallel applications include incremental graph mining applications like anomaly detection, but can also model a variety of other use cases like optimization (*e.g.*, motion planning) or cluster analysis (*e.g.*, video analysis).

Example 7.3.4 (Motion Planning)**.** The motion planning application solves NP-complete Mixed Integer Programs (MIP) to determine routes for *e.g.*, airplanes [166] and robots [187], where output failures can lead to human harm. This is a batch-processing workload with no underlying state (*i.e.*, $\mathcal{U}$ is never called); tasks are MIP instances and $\mathcal{A}$ invokes a solver, returning a single record containing a solution with a proof of optimality, or a proof showing the MIP instance is infeasible (*i.e.*, impossible to optimize). Then the verification operators are straightforward: `isValid()` invokes the solver to verify whichever proof is present in the record, while `happensBefore` and `outputSize` are trivial since each task generates a single record.

Example 7.3.5 (Video Analysis)**.** Here, $\mathcal{S}$ consists of frames in a video feed, and tasks are occasionally submitted to the system to compute pixel clusters useful for image segmentation and motion detection [22, 227, 38]. When applied to *e.g.*, security cameras, Byzantine fault tolerance is desirable. $\mathcal{A}$ computes a k-centroids clustering [113] in response to occasional tasks consisting of an integer k, returning locally-optimal centroids in the given video frame. The clustering is valid if running an additional iteration of the k-centroids algorithm shows negligible change in the centroids (*i.e.*, the original execution had converged to an optimum). The `happensBefore` and `outputSize` verification operators are trivial since each task generates k distinct numerical records.

The discussion in the rest of this chapter applies to any verifiable task-parallel application. This generalization demonstrates the power of application-awareness beyond graph mining, as the insights from incremental graph mining semantics enable verification for a wide array of applications.

7.4 Verifiable Processing with OsirisBFT

We present the verifiable processing architecture. We first summarize how tasks, records and state are managed, and then describe the normal execution followed by verification protocols and strategies for workload management.

State Management. Guaranteeing linearizability of computations on concurrently updating state requires efficient mechanisms for isolating state snapshots. Modern data analytics systems [231, 40, 150] employ multiversioning in their data stores to enable concurrent computations over well-defined deterministic snapshots. Specifically, both the state and updates to the state are associated with a logical timestamp, and computations are restricted to specific states based on time intervals or windows. We replicate the timestamped state across all WP to ensure consistency despite failures. Processes which incorrectly update state are caught because they output incorrect results (executor), or ignored as most processes in each sub-cluster operate correctly (verifier).

Replication & Communication. To retain efficiency during normal execution, we develop optimistic protocols that optimize for low replication and fast communication. These decisions lead to graceful executions similar to a system without fault tolerance, but create a larger threat surface.

Task Batches & Record Chunks. To reduce communication overheads, tasks are streamed in *batches*. Likewise, the sequence of records generated by a single computation task is split into disjoint subsequences called *chunks*. Executors output a stream of chunks, allowing verifiers to proceed in parallel instead of waiting for the entire sequence of records.

7.4.1 Normal Execution

Figure 7.4 shows the behavior of the system during graceful executions, divided across four phases (marked [**P1**]-[**P4**]).

Task Flow: $IP \mapsto VP_{CO} \mapsto \{VP, EP\}$

Algorithm 6 shows the protocols for this flow. The input processes send task batches to VP_{CO} [**P1**]. VP_{CO} performs consensus to linearize the tasks, assigning monotonically increasing ids to state updates, which serve as logical timestamps (line 4 in Algorithm 6). Tasks with only computations are given the timestamp of the most recent state update. Since ids are unique throughout the execution, faulty executors computing incorrect tasks can be identified. In the same consensus, VP_{CO} assigns computations to executors. The

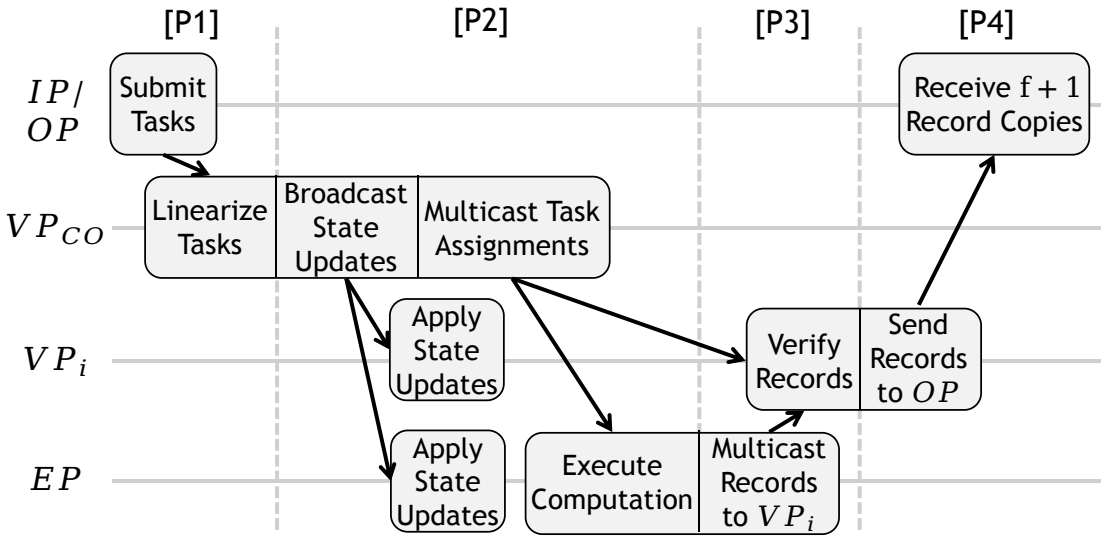


Figure 7.4: Overview of verification-based processing.

tasks are then distributed among the cluster [**P2**]: computations are sent to assigned executors and state updates are broadcast to WP.

Coordination-Free Task Assignment. Each task has: (a) an assigned executor which computes the task and generates records; and, (b) an assigned verifier sub-cluster to verify those records. While task messages are smaller than the record chunks produced by those tasks, communicating these two assignments separately creates a race condition; the executor may send record chunks to its assigned verifiers before the coordinator can inform them of the assignment, causing them to falsely believe they are faulty.

To avoid this, tasks are assigned using a coordination-free scheme (lines 8-10 in Algorithm 6). VP_{CO} sends signed task assignment messages to both executors and verifiers of the form $\langle t, E, i \rangle$, where t is the task to be executed by $E \in EP$ and verified by VP_i. The executor E receives $f + 1$ signed assignments for task t from different verifiers in the coordinator before executing t. As record chunks are generated for t, the task assignment messages (signed originally by verifiers in VP_{CO}) are prepended to each chunk and sent to VP_i. Likewise, verifiers in VP_i begin computing `outputSize(`t`)` upon receiving $f + 1$ assignment messages, in order to overlap verification and execution. Verifiers in VP_i each ensure the assignment messages were signed by VP_{CO} processes.

Output Flow: $EP \mapsto VP \mapsto OP$

As an executor computes a task, it sends each record chunk C to the assigned verifiers VP_i, alongside a digest $\sigma(C)$ [**P3**] using non-equivocating multicast (lines 25-28 in Algorithm 6). The final chunk for a task is tagged to signal its completion.

Verifiers independently check that they received a valid digest for C and verify the records in C are correct. Chunks are buffered until verification is complete before forwarding to downstream processes. To reduce message sizes, the leader verifier sends $\langle C, \sigma(C) \rangle$ to the process in OP, while every other verifier sends only $\sigma(C)$. An output process accepts C if

Algorithm 6 Task Flow protocol.

```
1  // [P1] Coordinator receives task from input process
2  Void onRecvTask(Task t) {
3    if (!validTask(t)) return; // t ∉ T
4    t.timestamp = consensus(t, getTimestamp())// Linearize
5    // [P2] Broadcast state updates and assign computations
6    if (hasStateUpdate(t)) broadcast(t);
7    if (hasComputation(t)) {
8      <e, vpi> = getNextExecutorAndVP();
9      send(e, <t, e, vpi>);
10     multicast(vpi, <t, e, vpi>);
11     startReassignmentTimeout(t);
12   }
13 }
14 // [P2] All other workers receive f+1 copies from VP_CO
15 Void onRecvStateUpdate(Task t) { applyStateUpdate(t); }
16 //[P2] Verifier in VP_i receives f+1 copies from VP_CO
17 Void onRecvAssignment(TaskAssignment <t, e, vpi>) {
18   if (!validAssignment(<t,e,vpi>) || !hasComputation(t)) return;
19   numRecords[t] = outputSize(t);
20 }
21 // [P2] Executor receives f+1 copies from VP_CO
22 Void onRecvAssignment(TaskAssignment <t, e, vpi>) {
23   if (!validAssignment(<t,e,vpi>) || !hasComputation(t)) return;
24   // [P3] Send output to assigned verifiers
25   for (chunk in compute(t)) {
26     multicast(vpi, chunk);
27     nonEquivocatingMulticast(vpi, σ(chunk));
28   }
29 }
```

it receives $f + 1$ matching digests (including the one that accompanied C) from the same verifier sub-cluster [**P4**].

7.4.2 Detecting Failures

Failures manifest where messages flow between fault tolerant verifiers and processes without fault tolerance.

Application-Specific Failures

Algorithm 6 and Algorithm 7 show the verification protocols run by the verifiers in the Task Flow and Output Flow, respectively.

Task Verification. MISMATCH caused by Byzantine IP in [**P1**] is handled by validating input tasks before distributing them (`isValid()` on line 3 in Algorithm 6). Byzantine executors can also cause MISMATCH failures in [**P3**] by sending chunks that correspond to invalid tasks. Verifiers check that the task corresponding to every chunk has been assigned to that executor and verifier sub-cluster (line 35 in Algorithm 7).

Record Chunk Verification. Records in every chunk are verified against MISMATCH and DUPLICATION (lines 56-58 in Algorithm 7). Each record is checked for validity and whether it originates from the correct task. Finally, the records are verified to be in sorted order by applying `happensBefore()` to every adjacent pair of records.

Algorithm 7 Verifier Output Flow protocol.

```
30 // [P3] Verifier receives from executor
31 Void onRecvRecords(RecordMessage msg, String digest) {
32   TaskAssignment <t, e, vpi> = msg.getAssignment();
33   Executor sender = msg.getSender();
34   RecordList chunk = msg.getChunk();
35   if (!validAssignment(<t, e, vpi>, sender)
36       || digest != computeDigest(chunk)
37       || !verify(chunk, t, e)) {
38     markByzantineExecutor(sender);
39     allChunks[t].clear();
40     reassignAllTasks(sender);
41   } else if (chunk.taskFinished()) {
42     cancelReassignmentTimeout(t);
43     sendDownStream(t, allChunks[t].append(chunk));
44   } else {
45     resetReassignmentTimeout(t);
46     seenRecords[t] += chunk.size();
47     allChunks[t].append(chunk);
48   }
49 }
50 Bool verify(RecordList chunk, Task t, Executor e) {
51   // ensure t is ongoing and chunks are sorted
52   RecordList prevChunk = allChunks[t][-1];
53   if (prevChunk != null && (prevChunk.taskFinished()
54         || !happensBefore(prevChunk[-1], chunk[0])))
55       return false;
56   for (r in chunk) // ensure all chunks are valid
57     if (!isValid(r, t) || !happensBefore(r, next(r)))
58       return false;
59   if (chunk.taskFinished()) // ensure nothing is missing
60     if (seenRecords[t] + chunk.size() != numRecords[t])
61       return false;
62   return true;
63 }
```

Inter-Chunk Ordering. A Byzantine executor can attempt to hide DUPLICATION across chunk boundaries, for example by sending a correct chunk twice. Verifiers protect against this by comparing the last record in the previous chunk with the first record of the newly received chunk, using the `happensBefore()` operator (lines 52-55 in Algorithm 7).

Missing Records. Finally, OMISSION is detected by comparing the number of records sent by an executor with the true number of records corresponding to the task when its final chunk is received. The true count is available from `outputSize()` which runs asynchronously while the executor produces records (line 19 in Algorithm 6).

Generic Protocol Failures

Generic failures range from impersonating processes to sophisticated attacks by different Byzantine processes cooperating across multiple phases in order to prevent output and compromise liveness.

Speculative Task Reassignment. Byzantine executors can cause OMISSION faults and compromise liveness by responding to most messages but neglecting to send a final chunk, making them indistinguishable from a correct executor working on a difficult task. We address this issue using a speculative reassignment scheme. In the case where the final chunk is not marked or no output is received at all, when sufficient time passes after Δ,

the task times out (line 11 in Algorithm 6) and VP_{CO} assigns the task to another executor. Verifiers accept results from whichever executor finishes first. To avoid tying up all of EP on one large task, the timeout duration for a given task is increased using exponential backoff.

Faulty Verifiers & Output Processes. A faulty verifier can compromise liveness by never forwarding chunks to OP when it serves as leader of its sub-cluster. If an output process receives $f + 1$ digests $\sigma(C)$ from VP_i but does not receive a matching chunk C in some time after Δ has passed, it multicasts messages to VP_i to report a *negligent leader*. When verifiers receive a negligent leader report, they initiate an election for a new leader, and the new leader sends C instead.

However, a negligent leader report is not sufficient to conclude a verifier is faulty due to the possibility of faulty output processes. Since there can only be f failures in VP_i, verifiers track which leaders have been reported and assume an output process is Byzantine if it reports $f + 1$ different leaders in the same sub-cluster. Finally, to avoid spurious reports due to innocent network delays in communicating chunks, correct output processes apply exponential backoff to their timeout duration after each negligent leader report.

Limited Equivocation. Equivocation occurs if a faulty process sends different messages to different verifiers in a sub-cluster when it was expected to send identical messages. This is expected in three situations: (1) In **[P1]** when an input process sends tasks to VP_{CO}; (2) In **[P3]** when an executor sends record chunks to assigned verifiers of a task; and, (3) In **[P4]** when an output process sends negligent leader reports to all verifiers in the sub-cluster that sent chunk digests.

In **[P1]**, equivocation by an input process does not affect the system because VP_{CO} performs a Byzantine agreement protocol to linearize tasks, and conflicting task messages will simply not be agreed upon. Similarly in **[P4]**, equivocation by an output process has no effect since $f + 1$ verifiers must initiate a leader election.

Finally, equivocation in **[P3]** is avoided by requiring executors to send chunk digests using non-equivocating multicast, and having correct output processes that receive at least one but fewer than $f+1$ digests $\sigma(C)$ send a report containing $\sigma(C)$ to the verifiers, similar to negligent leader reports. Upon receiving the report, correct verifiers which have chunk C broadcast it to the rest of the sub-cluster. The verifier that had not previously received C but had received $\sigma(C)$ now processes C as if it were sent from the original executor, eventually forwarding a digest to the output process.

7.4.3 Dynamic Role-Switching

The task execution workload and the verification workload can remain incongruous across various scenarios, impacting processing throughput. For example, tasks producing few results can leave verifiers idle despite executors being busy. Moreover, executors failing and leaving the system can drop processing throughput until new executors join the cluster.

To maintain throughput in such situations, verifiers can switch roles. When verifier resource utilization is low and there are many outstanding computation tasks, VP_{CO} assigns tasks to verifiers from an underutilized sub-cluster VP_i as if they were executors, and their output is routed through another sub-cluster VP_j. Verifiers in VP_i finish their verification work and then execute assigned tasks. In the meantime, VP_{CO} avoids assigning VP_i as verifiers of tasks.

7.5 Safety and Liveness

This section proves correctness guarantees of OsirisBFT.

7.5.1 Safety

OsirisBFT satisfies safety; every correct output process observes records corresponding to a legal sequential execution of correct tasks submitted by input processes.

Lemma 7.5.1. *The Task Flow results in a globally consistent ordering of tasks and task assignments to executors.*

Proof. In **[P1]**, input processes act as clients to VP_{CO} in a Byzantine agreement protocol (correct by [6]), hence the tasks are safely linearized and correct verifiers agree on which executor is assigned the task in **[P2]**. A correct executor only accepts task assignments accompanied by $f + 1$ signatures, hence it can never be fooled into performing incorrect tasks. Correct executors and verifiers have a consistent view of task ordering and assignment, because network messages cannot be reordered and reassignment does not occur until after Δ has passed, so initial task assignment messages are received strictly before reassignment messages. Furthermore, correct processes in WP have a consistent view of the state. Monotonic timestamps mean that if a correct process receives $f + 1$ copies of a task with timestamp k before receiving sufficient copies of a task with timestamp $k - 1$, the process simply waits to receive tasks in order before executing. A correct process receiving $f+1$ correctly timestamped task assignments before the corresponding state update simply applies the state update before performing the computation. □

Lemma 7.5.2. *Let $t_1, t_2, \ldots$ be the global (linearized) ordering of tasks submitted by IP, where $\forall i, t_i \in \mathcal{T}$. Let $s_t \in \mathcal{S}$ be the state obtained by applying all state updates from tasks $t_1, \ldots, t$ to the initial application state in order.*

Correct verifiers send OP a sequence of records R corresponding to a task t if and only if $R = \mathcal{A}(s_t, t)$.

Proof. By Lemma 7.5.1, all correct processes have access to s_t during execution of t. Write R as a concatenation of k chunks, $R = R_1|R_2|\ldots|R_k$, with chunk R_i consisting of l records

$r_{i1}|\ldots|r_{il}$. By Algorithm 7, verifiers forward R to OP whenever the following hold:

$$\bigwedge_{i=1}^{k}\bigwedge_{j=1}^{l}\texttt{isValid}(r_{ij}) \tag{7.1}$$

$$\bigwedge_{i=1}^{k}\bigwedge_{j=1}^{l-1}\texttt{happensBefore}(r_{ij}, r_{i(j+1)}) \tag{7.2}$$

$$\bigwedge_{i=1}^{k-1}\texttt{happensBefore}(r_{il}, r_{(i+1)1}) \tag{7.3}$$

$$\sum_{i}^{k}|R_i| = \texttt{outputSize}(t) \tag{7.4}$$

By (1), for all $r \in R$, $r \in \mathcal{A}(s_t, t)$. Hence we can write $\{r : r \in R\} \subseteq \{r : r \in \mathcal{A}(s_t, t)\}$. By (2) and (3), R is totally ordered according to $\prec$, so every element of R is unique and we can write $|\{r : r \in R\}| = |R|$. Finally, by (4), $|R| = |\mathcal{A}(s_t, t)|$, and we get $\{r : r \in R\} = \{r \in \mathcal{A}(s_t, t)\}$. Since both R and $\mathcal{A}(s_t, t)$ are totally ordered according to $\prec$, we have $R = \mathcal{A}(s_t, t)$. □

Corollary 7.5.1. *Correct output processes only observe correct records.*

Proof. We prove this by contradiction. Suppose a correct output process O observes an incorrect sequence of records. As every sequence is made up of chunks, O must observe an incorrect chunk R_i.

To accept R_i, O receives R_i and f digests $\sigma(R_i)$ from $f + 1$ verifiers in the same sub-cluster. This implies either a correct verifier forwarded an incorrect chunk, contradicting Lemma 7.5.2, or there are more than f failures in the same sub-cluster. □

Theorem 7.5.3. *Every correct output process observes records corresponding to a legal sequential execution of tasks submitted by correct input processes.*

Proof. By Corollary 7.5.1, there is a sequence of tasks T with corresponding states S such that correct output processes only receive records corresponding to $\mathcal{A}(s_t, t)$ for $t \in T$. By Lemma 7.5.1, all correct tasks are consistently ordered and successfully distributed to executors. Correct verifiers reject records corresponding to unassigned tasks and hence T contains only correct tasks submitted by an input process.

Furthermore, correct verifiers have a consistent view of the ordering of tasks when verifying $\mathcal{A}(s_t, t)$. Therefore, $\forall t \in \mathcal{T}, \mathcal{A}(s_t, t)$ follows a legal sequential execution of T. □

7.5.2 Liveness

Reliable links alongside partial synchrony guarantee that sent messages are always delivered without reordering. This constrains potential liveness issues to Byzantine behavior from

processes, namely full or partial unresponsiveness leading to OMISSION failures. We begin by proving that such failures cannot occur.

Lemma 7.5.4. *If there is a non-faulty executor in EP, every correct task is executed.*

Proof. Let t be a correct task and suppose for contradiction that t is never executed. By Lemma 7.5.1, t is correctly distributed to an executor E. If E is correct it will execute t, so E must be faulty.

But then $f+1$ correct verifiers in VP_{CO} will eventually reassign t to a different executor, succeeding once again due to Lemma 7.5.1. If any other executor is correct, t will be executed after enough reassignments. Hence, t would remain unexecuted only when VP_{CO} cannot find a correct executor to reassign t. This means all executors must be faulty, which is a contradiction. □

Lemma 7.5.4 relies on a non-faulty executor in EP. Without this assumption, it is impossible to tell whether all executors are faulty once t has been assigned to every executor because the length of a task is not known *a priori*. To guarantee liveness in this worst case, after a final timeout VP_{CO} can always reassign t to a verifier sub-cluster, where at least $f+1$ correct processes execute it and skip to **[P4]** in the Output Flow. In practice, however, executors can be assumed Byzantine after a sufficiently long timeout and failed over.

Using Lemma 7.5.4, Lemma 7.5.2, and our assumptions about the underlying network, we can now prove liveness.

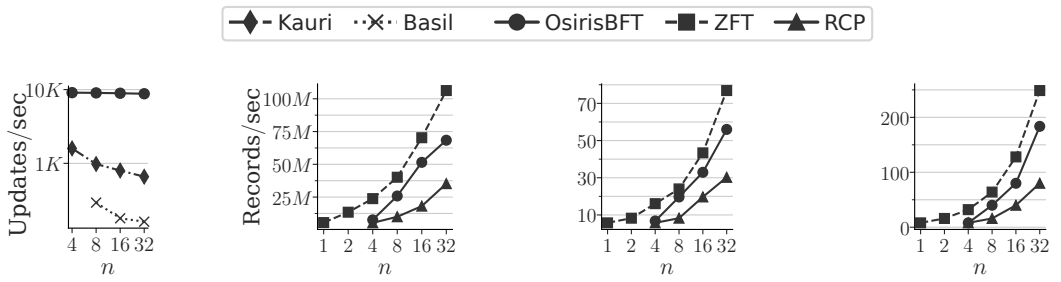
Theorem 7.5.5. *All correct output processes receive output records for every correct task submitted by input processes.*

Proof. The underlying network is partially synchronous and messages are delivered reliably, thus executors can successfully forward output records to $f + 1$ verifiers. Additionally by Lemma 7.5.2, verifiers will successfully forward output records to the output processes. Finally, by Lemma 7.5.4, every correct task is executed. Therefore, all output records will be received by $f + 1$ verifiers whether they are generated by a correct executor or by the verifiers themselves. □

7.6 Evaluation

We seek to understand how OSIRISBFT affects performance and fault tolerance in realistic processing scenarios.

System Details. All experiments were conducted using a 40-node cluster with each node containing 8 logical cores and 6GB RAM, implemented as Docker containers like in [160]. Nodes are distributed among a testbed of machines connected by a Mellanox 100Gbps Infiniband network (0.075ms TCP ping latency), each with a 2-socket Intel Xeon Gold 6242R



(a) State Updates. (b) Anomaly Detection. (c) Motion Planning. (d) Video Analysis.

Figure 7.5: Throughput scalability.

CPU. All experiments have a single node acting as both IP and OP, and the remaining nodes allocated to WP.

OsirisBFT *Implementation.* OsirisBFT is implemented in approximately 3500 lines of C++20 code. Regular communications use RDMA RC [118] via the `ibverbs` library, the non-equivocating multicast implementation follows open-source code for Mu [7], and the Fast & Robust algorithm [6] is used for consensus. Processes use one CPU core for network operations, and the rest for cryptography and executing application tasks (executors) or verifying results (verifiers).

Baselines. We compare OsirisBFT performance against a baseline with zero fault tolerance (**ZFT**), as well as a replicated computing processing architecture (**RCP**) based on the RSM philosophy of replicating computation tasks. In ZFT, IP sends tasks to a coordinator worker in WP, which distributes the tasks to other workers who execute $\mathcal{A}$ and simply forward the results. BFT processing systems like Medusa [65] and others [198, 64, 169] target narrow application models such as map-reduce or lack open-source code, and state-of-the-art RSM systems like Kauri [160] focus on consensus and are inappropriate for heavyweight computations. Therefore, we implement RCP using the same network and consensus algorithms as OsirisBFT to capture the essence of the replicated processing design while ensuring prior works are represented fairly. Every worker is replicated to create sub-clusters of size $2f+1$, with a designated coordinator sub-cluster WP_{CO} that linearizes tasks from IP and distributes them among the other sub-clusters to be executed. The worker sub-clusters and OP only accept messages that are sent from $f+1$ processes in a sub-cluster.

ZFT, RCP, and OsirisBFT all use a fully replicated data store since execution is bottlenecked by computations and not state updates. To confirm this, we ran write-only workloads on state-of-the-art BFT state management solutions **Kauri** [160] and **Basil** [200], as well as OsirisBFT. Figure 7.5a shows the results for different cluster sizes. The data store in OsirisBFT (and therefore the baselines) performs better as it does not incur overheads from transactional safety (Basil) or hashing blocks (Kauri), while also leveraging RDMA.

Applications. We consider three applications to evaluate performance under diverse conditions.

Anomaly Detection: Anomaly Detection computes anomalous subgraphs that emerge as a result of graph updates [63]. We built the application on top of OsirisBFT by integrating components from state-of-the-art pattern matching systems [192, 17] with verification operators implemented in only 100 lines of code.

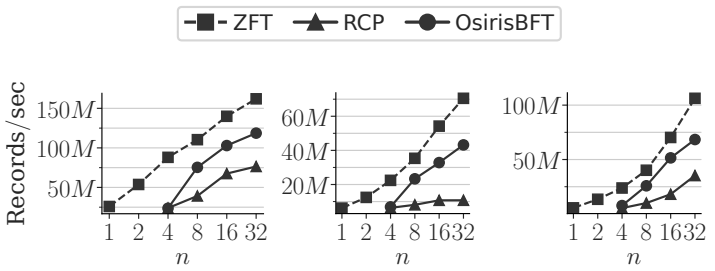
Motion Planning: Motion Planning solves Mixed Integer Programs (MIP) to determine routes for *e.g.*, airplanes [166] and robots [187], where output failures can lead to human harm. This is a batch-processing workload with no underlying state; tasks are drawn from a set of 107 standard MIP instances [62]. Executors use the state-of-the-art SCIP suite [30] to solve MIP instances. In OsirisBFT experiments, SCIP is configured to append a proof of optimality or infeasibility to each record [56]. The verification operators use built-in SCIP methods for validating the proof.

Video Analysis: This application operates on frequently updating video feed and periodically computes pixel clusters useful for image segmentation and motion detection [22, 227, 38] for, *e.g.*, security cameras, where Byzantine fault tolerance is desirable. It uses clustering [113] where executors return the centroids of each cluster, and verifiers check the optimality of centroids.

Methodology. Experiments are run 5 times and their results averaged to account for variance. Throughput experiments measure average throughput (output records per second) over 5 minutes, with an initial 30 second warm-up period. IP submits tasks to WP in batches, and results are streamed continuously to OP. Except where specified otherwise, experiments are run with $f = 1$ and 1MB record chunks. In Anomaly Detection, IP streams 1K tasks per second, and EP finds 6-cliques missing 2 edges in the Orkut graph [225], common inputs in previous work [205, 192]. In Video Analysis, IP streams 1K state updates per second and 5 computation tasks per second. In Motion Planning, IP streams 1K tasks per second. Dynamic role-switching is enabled in most experiments, and executions begin with $|WP|/(2f + 1)$ verifier sub-clusters. OsirisBFT converges to a stable number of sub-clusters during the warm-up period. Timeout values are calibrated empirically between 500 milliseconds and 5 seconds for each workload, necessary due to the complexity of the queries (tasks can take hundreds of seconds).

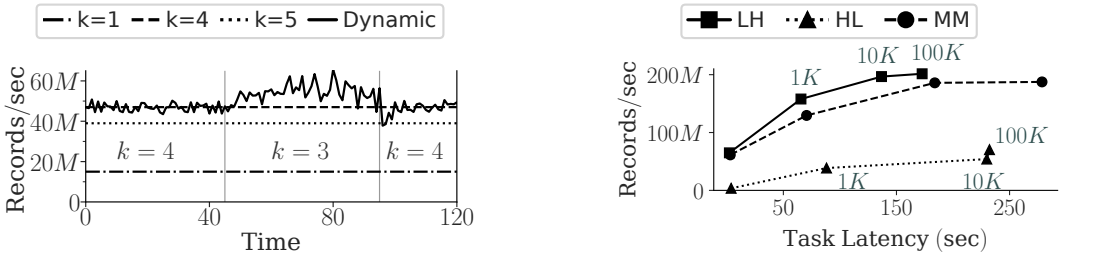
7.6.1 Graceful Execution Performance

We measure how output record throughput scales in OsirisBFT by varying the size of $n = |WP|$ between 1 and 32 nodes for each of the three applications. Figure 7.5 shows the results. OsirisBFT scales nearly as well as ZFT, with 1.2–4× lower throughput. The performance gap between ZFT and OsirisBFT decreases as n grows, with ZFT having 4× higher throughput at $n = 4$ but only 1.4× at $n = 32$ (Video Analysis). The other applications exhibit similar behaviour: in Motion Planning, ZFT initially has 2.3× higher



(a) Low CPU & High Output. (b) High CPU & Low Output. (c) Medium CPU & Output.

Figure 7.6: Throughput scalability across different Anomaly Detection workloads.



(a) Effect of role-switching. k is the number of active verifier sub-clusters.

(b) Throughput-Latency for 100 to 100K tasks submitted per second.

Figure 7.7: (a): effect of role-switching on Anomaly Detection workloads; (b): throughput-latency curve as the number of tasks submitted per second increases.

throughput at $n = 4$ but 1.4× at $n = 32$, whereas the difference is 3.1× to 1.6× for Anomaly Detection. This aligns with our theoretical analysis indicating OsirisBFT scales in $O(n-f)$ instead of $O(n/f)$, as the relative cost of the $O(f)$ overhead reduces as n grows.

Finally, OsirisBFT outperforms RCP in all workloads, achieving 1.9–2.3× higher throughput at $n = 32$. The performance difference can be attributed to lower parallelism in RCP; at $n = 32$ RCP has 10 parallel worker sub-clusters while OsirisBFT varies between 13 and 25 parallel executors based on how many verifiers switch roles.

OsirisBFT scales comparably to ZFT and scales better than RCP. OsirisBFT can reduce the performance penalty of fault tolerance relative to ZFT by scaling out.

7.6.2 Bottleneck Analysis

We performed detailed experiments to study performance across workloads. Results for Anomaly Detection are summarized below. By choosing appropriate queries from the literature, we emphasize stress on the CPU or the network, obtaining three workloads:

Medium CPU & Medium Output (MM): Listing instances of a dense size-6 pattern in the Orkut graph [225], a fairly expensive query with fairly large output.

Low CPU & High Output (LH): Listing 3-hop paths in Amazon Products [110], a computationally cheap query that creates massive result sets, to identify network bottlenecks.

High CPU & Low Output (HL): Listing 6-cliques in the Orkut graph [225], a computationally expensive query with relatively few results, to identify CPU bottlenecks.

Figure 7.6 shows the scalability on these workloads. As before, OsirisBFT scales nearly as well as ZFT, achieving 1.4–3.7× lower throughput, with the gap closing as n grows. Drilling down, we notice that MM and LH lead to worse scaling than the low output workload HL. By profiling network and CPU usage of the workloads in ZFT and OsirisBFT at $n = 32$, we discover that bandwidth usage on the link between OP and WP is similar during the high output workloads. In OsirisBFT WP sends messages to OP at a rate of 2.2GB/s in LH, 2.0GB/s in MM, but 1.8GB/s in HL, and in ZFT the rates are 3.4GB/s in both LH and MM, and 2.7GB/s in HL. Meanwhile average CPU usage of executors in OsirisBFT and ZFT is 93–95% during HL but 79–84% in LH and MM.

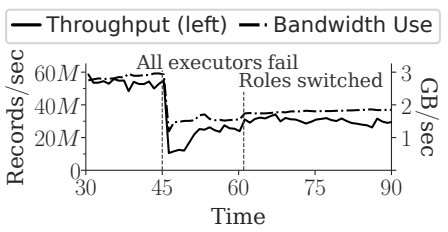
Finally, comparing OsirisBFT to RCP shows that with different workloads, OsirisBFT achieves 1.5–4× higher throughput at $n = 32$, due to better parallelism. We observe that in network-bound LH, RCP has 2.1×/1.5× lower throughput than ZFT/OsirisBFT, since parallelism is least important, but 6.5×/4× lower than ZFT/OsirisBFT in CPU-bound HL, where parallelism is most important. This follows our performance analysis in Section 7.1, as OsirisBFT is CPU-efficient.

Locating the Network Bottleneck. Since output rates during LH and MM are nearly identical and higher than HL, and CPU utilization is low, we confirm these workloads are bottlenecked by record communication in both OsirisBFT and ZFT. Importantly, this bottleneck only occurs at the link to OP, where records converge. The replicated communication between executors and verifier sub-clusters is parallelized over multiple links, and avoids this bottleneck. To further support this claim, we fix $n = 32$ and vary system load by controlling the rate of task submission between 100 per second and 100K per second, measuring task execution latency and output record throughput. Figure 7.7b shows the results.

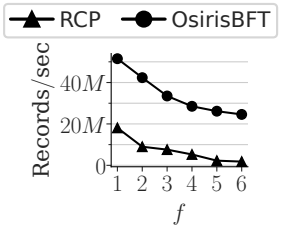
In LH and MM, heavy task loads severely impact latency as network bandwidth to OP saturates. Increasing from 10K to 100K tasks per second leads to slim increases in throughput compared to the increase in latency. However, in the CPU-bound HL workload OsirisBFT continues to achieve higher throughput as load increases. Mean latency was not affected from 10K to 100K tasks/sec since tasks in HL are expensive, and the cluster has sufficient parallelism and bandwidth.

7.6.3 Dynamic Role-Switching

We investigate whether role-switching balances verification and execution by comparing the throughput with executions where verifier sub-clusters are kept static (*i.e.*, verifiers cannot switch roles). Figure 7.7a shows the average throughput of the static executions, and plots the throughput over 2 minutes of execution with dynamic role-switching. The best static configuration is 4 sub-clusters, with 5 sub-clusters leaving verifiers idle and fewer than 4



(a) Simultaneous failure of all executors in EP (*i.e.*, not the ones role-switched from verifiers).



(b) Throughput for varying verifier fault tolerance level f.

Figure 7.8: Performance with Byzantine faults.

sub-clusters causing a verification bottleneck. The role-switching execution began with 5 sub-clusters but settled at 4 during its warm-up phase. Two other role-switches occur at near 45 and 95 seconds to transition from 4 sub-clusters to 3 when the verification workload dips due to a few consecutive batches of cheap tasks, then back to 4 sub-clusters when output records become too many to handle. Overall, dynamic role-switching results in 11% higher average throughput and 31% higher peak throughput than the best static configuration.

7.6.4 Performance Under Failures

Executor Failures. OsirisBFT theoretically tolerates the failure of all executor processes. We investigate the behaviour of OsirisBFT when executors fail by injecting output failures in every process from EP. Figure 7.8a shows the throughput and bandwidth observed at OP during an execution of MM with $f = 1$, $|VP| = 15$ and $|EP| = 16$. At 45 seconds, each executor corrupts the final record in the next chunk it outputs to cause a MISMATCH. The failures are detected quickly, and throughput does not drop to 0, because 3 verifiers had previously switched roles to act as executors. OsirisBFT automatically recovers to half its previous throughput by 61 seconds, as 6 more verifiers switch roles to make up for faulty executors. We repeated this experiment with other failure types and observed that OsirisBFT always recovers to approximately half its previous throughput seconds after fault detection.

Verifier Failures. Faulty verifiers mainly affect performance when sub-cluster leaders do not forward chunks as expected and require OP to report them. We repeat the previous experiment but instead of faulty executors, verifier sub-cluster leaders do not send chunks to OP. We observe that throughput is only affected until a new leader is elected, and OsirisBFT recovers to the same level since the executors are still correct.

Fault Scalability. We evaluate how OsirisBFT copes as more possibly faulty verifiers must be tolerated. Figure 7.8b compares executions of MM by OsirisBFT and RCP with $n = 32$ and varying fault tolerance levels f. OsirisBFT with role-switching ran with up to 2 verifier sub-clusters and 9–20 executors. We observe OsirisBFT executing with $f = 6$ achieves 2.7× higher throughput than RCP with $f = 2$.

7.7 Conclusion

We presented OsirisBFT, a verification-based Byzantine fault tolerant processing architecture for distributed task-parallel applications that does not replicate computation tasks. We formalized the application failures and developed generic verification operators to capture the required application semantics for verification. OsirisBFT incorporates efficient verification protocols that capture Byzantine failures with little coordination. OsirisBFT does not replicate computation tasks, hence delivering high processing throughput and scalability, for the first time allowing the performance gap between BFT and unreliable systems to close through horizontal scaling.

Chapter 8

Related Work

This thesis leverages insights from different domains and spans several literatures. Here, previous work in these domains and literatures is organized thematically and discussed critically.

8.1 General-Purpose Graph Mining Systems

Several general-purpose graph mining systems have been developed [205, 219, 111, 77, 146, 45]. Arabesque [205] is a distributed graph mining system that follows a filter-process model developed on top of map-reduce. It proposed the "Think Like an Embedding" (TLE) processing model. Pangolin [52], Kaleido [235], and Tesseract [34] all adopt this model, and differentiate themselves through support for GPUs, disk spilling, and streaming graphs. Fractal [77] extends TLE to the concept of *fractoids*, which expose parts of the user program to the system; in conjunction with depth-first exploration, fractoids allow the system to more intelligently plan its execution. G-Miner [45] is a task-oriented distributed graph mining system that enables building custom graph mining use cases using a distributed task queue. RStream [219] is a single machine out-of-core graph mining system that leverages SSDs to store intermediate solutions. It uses relational algebra to express mining tasks as table joins. SumPA [95] enhances batching in pattern-aware matching plans by combining the input patterns into abstract patterns in order to eliminate redundancies during exploration. AutoMine [146] compiles input patterns into exploration programs consisting of set operation schedules. While AutoMine batches the schedules of multiple input patterns, the schedules remain oblivious to the pattern substructures and symmetries, and hence end up exploring redundant matches, as shown in Section 5.6.6.

As discussed in Section 5.1, none of these systems are fully pattern-aware the way Peregrine [114] is: these systems perform unnecessary explorations and computations, require large memory (or storage) capacity, and lack the ability to easily express mining tasks at a high level. While Fractal uses symmetry breaking for pattern matching use case, other applications like FSM and motif counting are not guided by symmetry breaking,

and hence they end up performing unnecessary explorations. Similarly, AutoMine also does not employ symmetry breaking for any of the use cases, requiring users to filter duplicate matches by individually examining every single match when enumerating patterns. Lack of full pattern-awareness not only makes these systems slower, but also limits their applicability to more complex mining use cases.

More recent works have adopted the pattern-aware philosophy of Peregrine [114], and propose generic runtime techniques for improving graph mining performance based on input patterns. These techniques can be classified as follows.

Exploration Strategies. Several works propose hybrid breadth-first and depth-first techniques for graph mining in order to achieve high parallelism from breadth-first exploration while limiting memory use [51, 47, 50, 208]. The high-level idea in all these works is to chunk explorations: perform a fixed number of extensions in breadth-first manner to obtain a chunk of partial matches, and traverse the chunks in depth-first manner. To enable this, they analyze input patterns and the relationships between them in the same manner as Peregrine [114] and Subgraph Morphing [116].

Contigra [44] develops strategies for graph mining runtimes to exploit containment constraints in graph mining applications. Making such constraints transparent to underlying systems remains a challenging problem, as anti-vertex is currently the only declarative construct for expressing constraints on subgraph neighbourhoods.

Using Disks. RStream [219] is a general-purpose disk-based graph mining system that combines a relational data model with traditional graph processing techniques to allow mining graphs on a single machine without exhausting memory. On the other hand, RStream can easily exhaust disk space due to its breadth-first model since it stores matches in uncompressed tables. Kaleido [235] seeks to remedy the disk exhaustion problem with a compressed sparse match data structure that more succinctly stores matches.

Qiao et al [172] also identified this "output crisis", but in the context of subgraph matching, and proposed a novel vertex-cover-based compression scheme (VCBC) to store sets of matches on disk in compact format, as well as a distributed algorithm to automatically and efficiently join compressed match sets to form new ones. This approach makes it more practical to store massive match sets, and offers the practical advantage that the match sets can act as an index of the output space to make answering subsequent queries faster.

Hardware Acceleration. Pangolin [52] is the first programmable graph mining system to leverage GPU hardware acceleration, but it is not pattern-aware. It adopts a breadth-first exploration model which enables it to take advantage of SIMT parallelism. To broach pattern-awareness, PBE [96] matches patterns using GPUs, operating on partitioned graphs to stay within the limits of VRAM. G^{2}Miner [50] is the latest GPU-based graph mining sys-

tem, and automatically generates CUDA code to efficiently match graph patterns on GPU. It explicitly adopts pattern-aware optimizations such as data preprocessing and counting-specific pruning.

There has also been work on novel hardware accelerators for graph mining, incorporating intelligent caches and graph mining-specific processing elements to boost efficiency. FlexMiner [53] develops hardware support for existing techniques like connectivity maps [51]. FINGERS [48] augments the large amounts of coarse-grain parallelism exploited across processing elements with additional fine-grain parallelism within each processing element. These insights are carried to the streaming graph mining setting by PSMiner [170], allowing for pattern-aware incremental graph mining. Other works focus on processing-in-memory [39, 29] and near-memory [68, 202] graph mining. Shogun [222] studies how best to schedule graph mining tasks on processing elements.

These works are all orthogonal to the issue of application-aware design. The systems and frameworks developed in this thesis are compatible with hardware accelerators, novel exploration styles, and compression schemes. Instead, this thesis focuses on understanding the semantics of graph mining applications and exploiting them for performance and fault tolerance.

8.2 Approximation

Approximation has been used extensively in specific graph mining tasks to achieve scalability and efficiency, especially for counting matches. There is various work on algorithms for approximate triangle counting [209, 174, 127, 194, 167], approximate motif counting [24, 93, 173, 196, 88, 139, 35], and approximate counts for arbitrary patterns [18, 195]. These works typically either adapt existing sampling techniques into novel parallel algorithms [209, 88, 18, 194], or propose novel sampling methods optimized for different graph settings (*e.g.*, uncertain graphs [139], streaming graphs [167], semistreaming graphs [127]).

There is also work on approximation for FSM [182, 12, 104, 237, 31]. Some works [182, 12, 104] employ Markov Chain Monte Carlo sampling schemes to directly sample subgraphs from frequent patterns by first finding frequent single-vertex patterns and then exploiting the anti-monotonicity of frequency to guide their sampling. Other works [237, 31] begin by sampling a representative graph from the original data graph, then run an exact FSM algorithm on the representative graph and extrapolate the results to the original graph. Reference [237] uses a similar idea to neighbourhood sampling, called *random areas sampling*, where subgraphs are sampled from the neighbourhoods of seed vertices chosen uniformly at random, and the representative graph is the union of these sampled subgraphs. Reference [31] ensures that the representative graph has the same degree distribution as the original graph, by first grouping vertices with similar degrees (*i.e.*, within a similarity range

given by a hyper-parameter δ) into buckets and proportionally sampling vertices from each bucket. ScaleMine [2] is an exact FSM algorithm that uses an approximation phase to compute bounds on the frequency of a pattern and thereby guide its exploration.

ASAP [111] generalizes *neighbourhood sampling*, a technique developed by Pavan et al [167] to quickly compute approximate triangle counts in streaming graphs, to enable unbiased approximation of counts for any pattern in a distributed graph setting. In neighbourhood sampling the first edge is sampled at random, but subsequent edges are sampled from the neighbours of a previously sampled edge. This increases the probability that an individual trial finds a match of the desired pattern, and thus reduces the number of trials for reasonable accuracy. As [167] developed neighbourhood sampling in the context of streaming graphs, ASAP also treats the graph as a stream of edges, and can support changing graphs (*i.e.*, the stream of edges can be infinite).

8.3 Graph Querying

Graph Query Languages. Graph query languages and their data models have been extensively researched [21]. SPARQL [103] is one of the first graph query languages to provide pattern matching alongside SQL constructs, and operates on sets of RDF triples. It can support property graphs through [206], which translates SPARQL queries into Gremlin queries. Cypher [86] is a query language on property graphs first developed as part of Neo4j [161] that introduced "ASCII-art" syntax to specify path patterns. PGQL [212] offers regular path expressions in the pattern matching syntax, and introduces novel operators to construct new graphs as the result of a query. G-CORE [20] proposes a new graph query language using similar syntax to Cypher and PGQL, but operating in the *path property graph* data model, where paths are treated as a first-class entity with labels and properties. GSQL [75] allows for computing aggregate values from the results of graph queries for sophisticated graph analytics. GQL [74] is a recent effort to create a standard graph query language for property graphs. It provides several novel constructs for query expression, such as partial edge direction restrictions and edge predicates. Gremlin [178] is a functional graph traversal language with a simple grammar meant to facilitate embedding within a general-purpose programming language. Unlike the SQL-like syntax, Gremlin users define queries as trees of functions through method-chaining in a host language.

These query languages expose syntax and operators for specifying edges, paths, and constraints on query results, but cannot easily express neighbourhood constraints. The anti-vertex construct provides a declarative method for specifying neighbourhood constraints. Anti-vertex is a generic concept and can be incorporated in any modern query language in a similar fashion to our proposed extensions to Cypher.

Querying Constructs. There has also been work on subgraph query models and programming constructs for subgraph queries.

[85] allows expressing functional dependencies on graphs (GFD). GFDs cannot be used to implement the anti-vertex construct, because they only constrain vertices within a match, without access to the surrounding data graph. [83] proposes the concept of conditional graph pattern (CGP) which enforces conditions on edges, but it cannot express absence of a vertex like the anti-vertex construct.

Absence of entities has been studied in other contexts. Graph grammars [80] provide rule-based mechanisms for generating and manipulating graphs, where the productions are applied to a graph in order to obtain its derived graph when certain application conditions are met. [97] studies negative application conditions that include non-existence of nodes and edges in order to restrict how and where productions get applied. In relational algebra [61], the antijoin operator is similar to semijoin, except its result contains tuples from one relation that do not match on the common attribute from the other relation. Antijoins in SQL are achieved using WHERE clause coupled with logical operators like NOT EXISTS, limited in a similar manner as shown in Figure 4.2 and Figure 4.3.

Graph Query Engines. The backends to graph query languages are graph query engines. Recent works include PGX.D [109, 180] using PGQL [212]; GraphFlow [120] using Cypher [86]; and GAIA [171] using Gremlin [178]. These works consider backend systems details regarding efficiently executing graph queries which have few results and are typically not aggregated. This thesis instead develops new ways of expressing graph queries through anti-edges and anti-vertices, and analyzes graph mining applications involving massive results and requiring aggregation.

8.4 Application-Specific Graph Mining

Purpose-Built Graph Mining Solutions. These works efficiently perform specific graph mining tasks. ApproxG [147] is an efficient system for computing approximate graphlet (motif) counts with accuracy guarantees. [9] uses combinatorial arguments to obtain counts for size 3 and 4 motifs after counting smaller motifs. [71] efficiently lists k-cliques in sparse graphs and [27] is aimed at k-plexes which are clique-like structures. GraMi [82] leverages anti-monotonicity for FSM on a single machine while ScaleMine [2] is a distributed system for FSM that uses efficiently computable approximate stats to inform its graph exploration. [203] is also a distributed system focusing on FSM. [233, 197] are recent works aimed at analyzing small graphs whose edges have large attribute sets.

Several systems aim to perform efficient pattern matching. OPT [124] is a fast single-machine out-of-core triangle-counting system whose techniques are generalized by DualSim [123] to match arbitrary patterns. [194] proposes several provably cache-friendly parallel triangle-counting algorithms which provide order-of-magnitude speedups over previous algorithms. DistTC [107] presents a distributed triangle-counting technique that leverages a novel graph partitioning strategy to count triangles with minimal communication overhead.

[134] is a distributed map-reduce based pattern matching system that first finds small patterns and joins them into large ones. QFrag [190] is another map-reduce based distributed pattern matching system that focuses on searching graphs for large patterns using the TurboISO [102] algorithm. PruneJuice [176] is a distributed pattern matching system that focuses on pruning data graph vertices that cannot contribute to a match. [101] is a scalable subgraph isomorphism algorithm while TurboFlux [125] performs pattern matching on dynamically changing data graphs. [154] presents a pattern matching plan optimizer incorporated in Graphflow [120] that uses both binary and multi-way joins. [180] is a resource-aware distributed graph querying system for property graphs.

Subgraph Matching. There is a broad literature concerned with matching subgraphs in large graphs according to isomorphism semantics, spread especially by the databases community [102, 125, 123, 133, 134, 175, 172, 33, 101, 32, 17, 154, 226], but also in systems [190, 144, 145], and high-performance computing [192, 176]. Algorithms developed in the databases community are evaluated comprehensively in a recent study [199]. Dryadic [144] leverages an intermediate computation tree structure to generate efficient code for distributed pattern matching. GraphPi [192] uses a performance model to select efficient matching orders for subgraph matching. These works can be incorporated in application-aware graph mining systems since generating the subgraph set S is an expensive step in most applications, but they are not application-aware because they do not consider the wide array of possible graph mining aggregations.

Counting Subgraphs. A myriad of research has been conducted on algorithms for counting motifs [93, 9, 141, 108, 151, 152, 234, 168, 173, 147]. [9] uses combinatorial identities for counting size 3 and 4 motifs. RAGE [141] provides a method for computing edge-induced size-4 motifs, and for converting the results to those for vertex-induced motifs. [108] uses automorphism groups of pattern vertices to compute counts for motifs with 2-5 vertices, while [151, 152, 234] optimize orbit-local counting using equations for arbitrary pattern sizes. [168] computes counts for all size 5 motifs using global and local counts for smaller patterns.

As discussed in Section 6.2.4, none of these works are applicable for general-purpose graph mining systems since they focus (a) only on converting counts, (b) only for certain specific patterns, and (c) only on certain specific way to convert counts. Hence for instance, their combinatorial strategies (*e.g.*, scalar möbius function in [234]) cannot be generalized to arbitrary aggregations, and they cannot generate multiple alternatives, which is crucial. In comparison, Subgraph Morphing [116] is general and captures system-level nuances and application-level characteristics, making it practical for graph mining systems.

Frequent Subgraphs. Works like [82, 2, 3] develop solutions for mining frequent patterns, however none of these are pattern-based and they instead view the FSM computation in terms of arbitrary subgraphs of the data graph.

8.5 Byzantine Fault Tolerance

Byzantine fault tolerance is a well-studied research area. We summarize the proposed solutions below.

Byzantine Fault Tolerant Data Processing. [198, 64, 65, 157, 169] enable Byzantine fault tolerance in data processing systems. ClusterBFT [198] replicates data-flow nodes in data-flow systems $2f + 1$ times. [64] replicates mapper and reducer tasks in MapReduce so that an answer is correct when a quorum of $2f+1$ tasks achieve the same result. Medusa [65] extends this fault tolerance to MapReduce clusters in multi-cloud environments, where failures can affect entire data centres. [157] also replicates MapReduce tasks and chooses results that occur most frequently. Finally, Greft [169] is a BFT graph processing system that replicates vertex functions, relying on a trusted master process to detect if values differ.

These works rely on replication of core computation, limiting their scalability, while OsirisBFT [115] enables BFT processing without replicating application tasks.

Byzantine Fault Tolerance Protocols. Research regarding BFT consensus spans decades, with seminal works like PBFT [41] inspiring many works that improve usability, resiliency, and performance [42, 177, 130, 129, 214, 60, 10, 25, 1, 66, 28, 228, 8]. [5] proposed the message-and-memory model used by [6] to achieve BFT consensus with $2f + 1$ replicas. [228] divides workers into an agreement sub-cluster and execution sub-clusters; however, both the sub-clusters replicate tasks.

More recently, the popularity of permissioned blockchains has caused a resurgence in BFT research. HotStuff [229], Kauri [160], Fabric [19], Narwhal and Tusk [69], Damysus [73], and others [70, 11, 126, 122] develop efficient consensus strategies using optimized communication and transaction scheduling techniques as well as trusted components.

There has also been ample work on Byzantine fault tolerance for databases, like [213, 106, 15, 16, 165, 200, 224] that focus on serializable concurrent execution of transactions, and [159, 81, 87] that marry blockchain and database features.

All these works focus on consensus in the client-server model, where agreeing on an ordering of client requests is the only consistency requirement. As such, they relate only to the Task Flow of OsirisBFT [115], where tasks from IP are linearized. However, we target task-parallel processing and focus on computation and not state management, and our solution ensures the computation is not replicated.

Byzantine Fault Detection. PeerReview [99] and others [98, 92] propose failure detectors for Byzantine faults. These are modules in each node which only eventually detect simple deviations from a protocol, whereas a faulty executor can communicate correctly with other nodes while outputting incorrect records. OsirisBFT [115] does not have such limitations since all communication with downstream nodes occurs through verifiers which can detect all output failures.

8.6 Graph Processing Systems

Several works enable processing static and dynamic graphs [140, 89, 162, 193, 143, 181, 90, 236, 216, 76, 131, 217, 218, 184, 109]. These systems typically compute values on vertices and edges rather than analyzing substructures in graphs. They decompose computation at vertex and edge level, which is not suitable for graph mining use cases.

[140, 89, 162, 193, 181, 90, 236, 216, 143, 223, 142, 215, 184, 109, 217] and others primarily focus on graph processing problems that compute values on vertices and edges, as opposed to graph mining problems that are concerned with subgraph structures. aDFS [208] enhances the graph processing system PGX.D [109] with a hybrid depth-first/breadth-first graph exploration strategy for pattern matching queries. On the other hand, techniques like [132] develop custom transformations for specific subgraphs in the data graph in order to speed up value propagation.

As explained in Chapter 2, graph processing workloads are fundamentally different from graph mining. Graph processing applications iteratively perform matrix computations, whereas graph mining aggregates subgraphs.

Chapter 9

Conclusion

This thesis investigated the use of application semantics in improving general-purpose graph mining systems, as well as how such semantics can be expressed transparently. These questions were approached from the perspectives of a general-purpose programmable graph mining system Peregrine, a system-agnostic framework Subgraph Morphing, as well as a Byzantine fault tolerant distributed protocol OsirisBFT. Each perspective made the case for application-aware design by advancing the current understanding of a different facet of graph mining systems. Peregrine demonstrated the performance impact of employing application-specific techniques more broadly, and provides a foundation for using input patterns as a declarative query language to communicate application semantics to the system without resorting to imperative and opaque callbacks. Subgraph Morphing studied how application-awareness can be exploited without tightly coupling to low-level details of a processing model, and provides techniques for generalizing theoretical application-specific insights. Finally, OsirisBFT exploits a fundamental algorithmic fact about graph mining applications to develop a novel distributed architecture for Byzantine fault tolerance that enables scalability without sacrificing safety. These works represent the first push in the graph mining systems literature towards explicitly leveraging the user application semantics to develop graph mining systems, and inspiring extensive use of application-awareness by the community, particularly to develop novel pattern-aware graph mining and pattern matching solutions [47, 50, 46, 94, 44, 53, 202, 222, 48, 170, 39].

As the field matures, there are several paths forward for graph mining systems research. Up to now, significant research has been dedicated to application-aware subgraph matching in order to generate the subgraph set S for more complex applications, and this line of research will no doubt continue as novel algorithms, techniques, and hardware capabilities evolve. However as shown in Chapter 6, Peregrine and similar pattern-based systems are seldom bottlenecked by subgraph matching when executing complex applications, instead hitting fundamental scalability limits due to the number of subgraphs that filters and aggregations must process. Yet outside of Subgraph Morphing and OsirisBFT, little research has been done on leveraging application semantics beyond the backend, leaving

gaps in application-aware frontends, middle-ends, and high-level architecture designs. This thesis has demonstrated that application semantics are a largely untapped resource, and the impact of application-awareness will extend beyond pattern-aware subgraph matching.

By developing novel insights at different layers of the system, more ambitious work can be done on comprehensive systems that understand and optimize graph mining applications end-to-end, from high-level specifications through to execution. The path toward such a system requires several key developments. First, we need richer domain-specific languages that can fully capture the intent of graph mining applications. The pattern-based abstractions developed in this thesis are just the beginning. Future systems will extend pattern semantics to express an even broader range of structural constraints and neighborhood properties, allowing users to directly specify what subgraphs they want rather than writing complex filters and callbacks. The distinction between edge-induced and vertex-induced exploration, which this thesis unified through anti-edges, hints at how many other seemingly fundamental implementation choices could be abstracted away through the right semantic models.

Second, these declarative specifications must be coupled with configurable execution strategies. Users should be able to easily express their requirements around fault tolerance, memory usage, and parallelism through high-level knobs, and the system would then automatically select appropriate techniques that make intelligent tradeoffs based on the application semantics. While this thesis presented options and techniques for making such tradeoffs, *e.g.*, when to use verification versus replication for fault tolerance, or when to morph patterns for better performance, making a combination of runtime choices across the system can lead to compelling novel insights.

Finally, and perhaps most importantly, the portion of applications that requires custom user code should continue to shrink. As pattern semantics grow richer and systems better understand common aggregations, what users express imperatively today through opaque callbacks will increasingly move into the declarative specification of the subgraph set S and composable aggregation operators. By replacing the user-defined functions and filters that dominate current graph mining programs with higher-level specifications, future graph mining systems can better optimize execution without additional burden on users.

This vision builds on the fundamental insight of this thesis: that understanding application semantics enables better systems. The challenge ahead is to develop even richer semantic models that can capture more of what users want to express, while maintaining the performance benefits demonstrated by pattern-aware designs. Application-aware design leads not only to faster graph mining, but also to greater scalability and fault tolerance for task-parallel, and offer more declarative interfaces that abstract implementation details out of user programs.

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '05, page 59–74, 2005.

[2] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 1–12, 2016.

[3] Ehab Abdelhamid, Mustafa Canim, Mohammad Sadoghi, Bishwaranjan Bhattacharjee, Yuan-Chi Chang, and Panos Kalnis. Incremental Frequent Subgraph Mining on Large Evolving Graphs. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1767–1768, 2018.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995.

[5] Marcos K. Aguilera, Naama Ben-David, Irina Calciu, Rachid Guerraoui, Erez Petrank, and Sam Toueg. Passing Messages While Sharing Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC '18, page 51–60, 2018.

[6] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The Impact of RDMA on Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 409–418, 2019.

[7] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond Consensus for Microsecond Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 599–616, November 2020.

[8] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. UBFT: Microsecond-Scale BFT Using Disaggregated Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '23, page 862–877, 2023.

[9] Nesreen K. Ahmed, Jennifer Neville, Ryan A. Rossi, and Nick G. Duffield. Efficient Graphlet Counting for Large Networks. In *2015 IEEE International Conference on Data Mining*, ICDM 2015, pages 1–10, 2015.

[10] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR Fault Tolerance for Cooperative Services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '05, page 45–58, 2005.

[11] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A Sharded Smart Contracts Platform. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[12] Mohammad Al Hasan and Mohammed J. Zaki. Output Space Sampling for Graph Patterns. *Proceedings of the VLDB Endowment*, 2(1):730–741, August 2009.

[13] Roberto Alonso and Stephan Günnemann. Mining contrasting quasi-clique patterns. *CoRR*, abs/1810.01836, 2018.

[14] Amazon, Inc. Amazon Neptune, 2022. Version 1.1.0.0.

[15] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. CAPER: A Cross-Application Permissioned Blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, July 2019.

[16] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 76–88, 2021.

[17] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment*, 11(6):691–704, February 2018.

[18] Pranay Anchuri, Mohammed J. Zaki, Omer Barkol, Shahar Golan, and Moshe Shamy. Approximate Graph Mining with Label Costs. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, page 518–526, 2013.

[19] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[20] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1421–1432, 2018.

[21] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys*, 50(5), September 2017.

[22] Borislav Antić, Dragan Letić, Dubravko Ćulibrk, and Vladimir Crnojević. K-means based segmentation for real-time zenithal people counting. In *2009 16th IEEE International Conference on Image Processing*, ICIP '09, pages 2565–2568, 2009.

[23] Apache Software Foundation. Giraph, 2011. Accessed: 2024-05-15.

[24] V. Arvind and Venkatesh Raman. "approximation algorithms for some parameterized counting problems". In Prosenjit Bose and Pat Morin, editors, *Algorithms and Computation*, pages 453–464, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[25] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant Byzantine Fault Tolerance. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 297–306, 2013.

[26] Lowell W Beineke, Robin J Wilson, and Peter J Cameron, editors. *Topics in Algebraic Graph Theory*. Encyclopedia of mathematics and its applications. Cambridge University Press, Cambridge, England, October 2004.

[27] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. Efficient enumeration of maximal k-plexes. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '15)*, pages 431–444, 2015.

[28] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, 2014.

[29] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 282–297, 2021.

[30] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. The SCIP Optimization Suite 8.0. Technical report, Optimization Online, December 2021.

[31] Vandana Bhatia and Rinkle Rani. Ap-FSM: A parallel algorithm for approximate frequent subgraph mining using Pregel. *Expert Systems with Applications*, 106:217–232, 2018.

[32] Bibek Bhattarai, Hang Liu, and H. Howie Huang. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1447–1462, 2019.

[33] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1199–1214, 2016.

[34] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 458–473, 2021.

[35] Ilaria Bordino, Debora Donato, Aristides Gionis, and Stefano Leonardi. Mining large networks with subgraph counting. In *2008 Eighth IEEE International Conference on Data Mining*, ICDM '08, pages 737–742, 2008.

[36] Gabriel Bracha and Sam Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[37] Björn Bringmann and Siegfried Nijssen. What Is Frequent in a Single Graph? In *Advances in Knowledge Discovery and Data Mining, 12th Pacific-Asia Conference*, volume 5012 of *Lecture Notes in Computer Science*, pages 858–863, 2008.

[38] Butler, Darren and Bove Jr, V. Michael and Sridha, Sridharan. Real-Time Adaptive Foreground/Background Segmentation. *EURASIP Journal on Advances in Signal Processing*, 2005, August 2005.

[39] Shuangyu Cai, Boyu Tian, Huanchen Zhang, and Mingyu Gao. PimPam: Efficient Graph Pattern Matching on Real Processing-in-Memory Hardware. *Proceedings of the ACM on Management of Data*, 2(3), May 2024.

[40] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4), 2015.

[41] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, New Orleans, LA, February 1999.

[42] Miguel Castro and Barbara Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 4th Conference on Symposium on Operating Systems Design & Implementation - Volume 4*, OSDI '00, USA, 2000.

[43] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719, 2016.

[44] Joanna Che, Kasra Jamshidi, and Keval Vora. Contigra: Graph mining with containment constraints. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 50–65, 2024.

[45] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. G-Miner: An Efficient Task-Oriented Graph Mining System. In *Proceedings of the 13th EuroSys Conference*, EuroSys '18, pages 1–12, 2018.

[46] Jingji Chen and Xuehai Qian. Decomine: A compilation-based graph pattern mining system with pattern decomposition. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 47–61, 2022.

[47] Jingji Chen and Xuehai Qian. Khuzdul: Efficient and scalable distributed graph pattern mining engine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 413–426, 2023.

[48] Qihang Chen, Boyu Tian, and Mingyu Gao. FINGERS: exploiting fine-grained parallelism in graph mining accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 43–55, 2022.

[49] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[50] Xuhao Chen and Arvind. Efficient and scalable graph pattern mining on GPUs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 857–877, Carlsbad, CA, July 2022. USENIX Association.

[51] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. Sandslash: a two-level framework for efficient graph pattern mining. In *Proceedings of the 35th ACM International Conference on Supercomputing*, ICS '21, page 378–391, 2021.

[52] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *Proceedings of the VLDB Endowment*, 13(10):1190–1205, April 2020.

[53] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind Arvind. Flexminer: A pattern-aware accelerator for graph pattern mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 581–594, 2021.

[54] James Cheng, Yiping Ke, Ada Wai-Chee Fu, Jeffrey Xu Yu, and Linhong Zhu. Finding Maximal Cliques in Massive Networks. *ACM Transactions on Database Systems*, 36(4), December 2011.

[55] Xu Cheng, C. Dale, and Jiangchuan Liu. Statistics and social network of youtube videos. In Hans van den Berg and Gunnar Karlsson, editors, *Quality of Service, 2008. IWQoS 2008. 16th International Workshop on*, pages 229–238. IEEE, June 2008.

[56] Kevin K. H. Cheung, Ambros Gleixner, and Daniel E. Steffy. Verifying Integer Programming Results. In *Integer Programming and Combinatorial Optimization*, pages 148–160. Springer International Publishing, 2017.

[57] Alexandra Chouldechova. Fair Prediction with Disparate Impact: A Study of Bias in Recidivism Prediction Instruments. *Big Data*, 5(2):153–163, June 2017.

[58] Wei-Ta Chu and Ming-Hung Tsai. Visual Pattern Discovery for Architecture Image Classification and Product Image Search. In *Proceedings of the 2nd ACM International Conference on Multimedia Retrieval*, ICMR '12, pages 1–8, 2012.

[59] Allen Clement, Flavio Junqueira, Aniket Kate, and Rodrigo Rodrigues. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, PODC '12, page 301–308, 2012.

[60] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, page 153–168, USA, 2009.

[61] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[62] William J. Cook, Thorsten Koch, Daniel E. Steffy, and Kati Wolter. A Hybrid Branch-and-Bound Approach for Exact Rational Mixed-Integer Programming. *Mathematical Programming Computation*, 5(3):305–344, 2013.

[63] Corinna Cortes, Daryl Pregibon, and Chris Volinsky. Communities of Interest. In *Proceedings of the 4th International Conference on Advances in Intelligent Data Analysis*, IDA '01, page 105–114, Berlin, Heidelberg, 2001. Springer-Verlag.

[64] Pedro Costa, Marcelo Pasin, Alysson N. Bessani, and Miguel Correia. Byzantine Fault-Tolerant MapReduce: Faults are Not Just Crashes. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 32–39, 2011.

[65] Pedro A. R. S. Costa, Xiao Bai, Fernando M. V. Ramos, and Miguel Correia. Medusa: An Efficient Cloud Fault-Tolerant MapReduce. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '16, pages 443–452, 2016.

[66] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 177–190, USA, 2006.

[67] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. Technical report, W3 Consortium, 2014.

[68] Guohao Dai, Zhenhua Zhu, Tianyu Fu, Chiyue Wei, Bangyan Wang, Xiangyu Li, Yuan Xie, Huazhong Yang, and Yu Wang. Dimmining: pruning-efficient and parallel graph mining on near-memory-computing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 130–145, 2022.

[69] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, 2022.

[70] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards Scaling Blockchain Systems via Sharding. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 123–140, 2019.

[71] Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing K-Cliques in Sparse Real-World Graphs. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 589–598, 2018.

[72] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[73] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. DAMYSUS: Streamlined BFT Consensus Leveraging Trusted Components. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 1–16, 2022.

[74] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Hannes Voigt, Oskar van Rest, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. Graph Pattern Matching in GQL and SQL/PGQ. *CoRR*, abs/2112.06217, 2021.

[75] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '20, page 377–392, 2020.

[76] Laxman Dhulipala, Guy E Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, pages 918–934, 2019.

[77] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1357–1374, 2019.

[78] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[79] William Eberle and Lawrence Holder. Discovering structural anomalies in graph-based data. In *Seventh IEEE international conference on data mining workshops (ICDMW 2007)*, pages 393–398. IEEE, 2007.

[80] Hartmut Ehrig, Annegret Habel, and Hans-Jörg Kreowski. Introduction to Graph Grammars with Applications to Semantic Networks. *Computers & Mathematics with Applications*, 23(6-9):557–572, 1992.

[81] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A Shared Database on Blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, July 2019.

[82] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment*, 7(7):517–528, March 2014.

[83] Grace Fan, Wenfei Fan, Yuanhao Li, Ping Lu, Chao Tian, and Jingren Zhou. Extending graph patterns with conditions. In *Proceedings of the ACM International Conference on Management of Data*, pages 715–729, 2020.

[84] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Tianyi Chen, Zhenzhou Tian, Xiaodong Zhang, Qinghua Zheng, and Ting Liu. Frequent Subgraph Based Familial Classification of Android Malware. In *27th IEEE International Symposium on Software Reliability Engineering*, ISSRE 2016, pages 24–35, 2016.

[85] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional Dependencies for Graphs. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1843–1857, 2016.

[86] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD '18, page 1433–1445, 2018.

[87] Zerui Ge, Dumitrel Loghin, Beng Chin Ooi, Pingcheng Ruan, and Tianwen Wang. Hybrid Blockchain Database Systems: Design and Performance. *Proceedings of the VLDB Endowment*, 15(5):1092–1104, May 2022.

[88] Mira Gonen and Yuval Shavitt. Approximating the Number of Network Motifs. *Internet Mathematics*, 6(3):349–372, 2009.

[89] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 17–30, 2012.

[90] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 599–613, 2014.

[91] J.A. Green. *Sets and Groups: A First Course in Algebra*. Library of Mathematics. Routledge & Kegan Paul, 1988.

[92] Fabiola Greve, Murilo Santos de Lima, Luciana Arantes, and Pierre Sens. A Time-Free Byzantine Failure Detector for Dynamic Networks. In *2012 Ninth European Dependable Computing Conference*, pages 191–202, 2012.

[93] Joshua A. Grochow and Manolis Kellis. Network Motif Discovery Using Subgraph Enumeration and Symmetry-Breaking. In *Research in Computational Molecular Biology*, pages 92–106, 2007.

[94] Chuangyi Gui, Xiaofei Liao, Long Zheng, and Hai Jin. Cyclosa: Redundancy-Free Graph Pattern Mining via Set Dataflow. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 71–85, Boston, MA, July 2023. USENIX Association.

[95] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. SumPA: Efficient Pattern-Centric Graph Mining with Pattern Abstraction. In *30th International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, pages 318–330, 2021.

[96] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. Gpu-accelerated subgraph enumeration on partitioned graphs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1067–1082, 2020.

[97] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.

[98] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. The Case for Byzantine Fault Detection. In *Proceedings of the 2nd Conference on Hot Topics in System Dependability - Volume 2*, HOTDEP '06, page 5, USA, 2006.

[99] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '07, page 175–188, 2007.

[100] Bronwyn Hall, Adam Jaffe, and Manuel Trajtenberg. The NBER Patent Citation Data File: Lessons, Insights and Methodological Tools. *NBER Working Paper 8498*, 2001.

[101] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1429–1446, 2019.

[102] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. TurboISO: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of the 2013 International Conference on Management of Data*, SIGMOD '13, pages 337–348, 2013.

[103] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. Technical report, W3 Consortium, 2013.

[104] Mohammad Al Hasan and Mohammed Zaki. Musk: Uniform Sampling of k-Maximal Patterns. In *Proceedings of the 2009 SIAM International Conference on Data Mining (SDM)*, pages 650–661, 2009.

[105] Jelle Hellings and Mohammad Sadoghi. Coordination-Free Byzantine Replication with Minimal Communication Costs. In Carsten Lutz and Jean Christoph Jung, editors, *23rd International Conference on Database Theory*, volume 155 of *ICDT '20*, pages 17:1–17:20, Dagstuhl, Germany, 2020.

[106] Jelle Hellings and Mohammad Sadoghi. ByShard: Sharding in a Byzantine Environment. *Proceedings of the VLDB Endowment*, 14(11):2230–2243, October 2021.

[107] Loc Hoang, Vishwesh Jatala, Xuhao Chen, Udit Agarwal, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Disttc: High performance distributed triangle counting. In *IEEE High Performance Extreme Computing Conference (HPEC '19)*, pages 1–7, 2019.

[108] Tomaz Hocevar and Janez Demsar. A Combinatorial Approach to Graphlet Counting. *Bioinformatics*, 30(4):559–565, 2014.

[109] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. PGX.D: A Fast Distributed Graph Processing Engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 1–12, 2015.

[110] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *CoRR*, abs/2005.00687, 2020.

[111] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. ASAP: Fast, Approximate Graph Pattern Mining at Scale. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 745–761, 2018.

[112] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. JACKSTRAWS: picking command and control connections from bot traffic. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, page 29, USA, 2011.

[113] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, Inc., USA, 1988.

[114] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[115] Kasra Jamshidi and Keval Vora. Osirisbft: Say no to task replication for scalable byzantine fault tolerant analytics. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '24, page 94–108, 2024.

[116] Kasra Jamshidi, Harry Xu, and Keval Vora. Accelerating Graph Mining Systems with Subgraph Morphing. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 162–181, 2023.

[117] Tommi Junttila and Petteri Kaski. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 135–149, 2007.

[118] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference*, ATC '16, page 437–450, 2016.

[119] Miltiadis Kandias, Nikos Virvilis, and Dimitris Gritzalis. The Insider Threat in Cloud Computing. In Sandro Bologna, Bernhard Hämmerli, Dimitris Gritzalis, and Stephen Wolthusen, editors, *Critical Information Infrastructure Security*, pages 93–103, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[120] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1695–1698, 2017.

[121] George Karypis and Vipin Kumar. Parallel Multilevel K-Way Partitioning Scheme for Irregular Graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, pages 35–es, 1996.

[122] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 357–388. Springer, 2017.

[123] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1231–1245, 2016.

[124] Jinha Kim, Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, and Hwanjo Yu. OPT: A New Framework for Overlapped and Parallel Triangulation in Large-Scale Graphs. In *Proceedings of the 2014 International Conference on Management of Data*, SIGMOD '14, pages 637–648, 2014.

[125] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 411–426, 2018.

[126] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy*, SP '18, pages 583–598, 2018.

[127] Mihail N. Kolountzakis, Gary L. Miller, Richard Peng, and Charalampos E. Tsourakakis. Efficient Triangle Counting in Large Graphs via Degree-Based Vertex Partitioning. *Internet Mathematics*, 8(1-2):161–185, 2012.

[128] Anton Korshunov, Ivan Beloborodov, Nazar Buzun, Valeriy Avanesov, Roman Pastukhov, Kyrylo Chykhradze, Ilya Kozlov, Andrey Gomzin, Ivan Andrianov, Andrey Sysoev, Stepan Ipatov, Ilya Filonenko, Christina Chuprina, Denis Turdakov, and Sergey Kuznetsov. Social network analysis: Methods and applications. In *Proceedings of the Institute for System Programming of RAS*, pages 439–456, 2014.

[129] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *ACM Transactions on Computer Systems*, 27(4), January 2010.

[130] Ramakrishna Kotla and Mike Dahlin. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, DSN '04, pages 575–584, 2004.

[131] Pradeep Kumar and H Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST '19)*, pages 249–263, 2019.

[132] Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. Efficient Processing of Large Graphs via Input Reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 245–257, 2016.

[133] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable Subgraph Enumeration in MapReduce. *Proceedings of the VLDB Endowment*, 8(10):974–985, June 2015.

[134] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable Distributed Subgraph Enumeration. *Proceedings of the VLDB Endowment*, 10(3):217–228, November 2016.

[135] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '09, page 1–14, USA, 2009.

[136] Xin Li, Michael C. Huang, Kai Shen, and Lingkun Chu. An Empirical Study of Memory Hardware Errors in a Server Farm. In *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability*, HotDep'07, page 13–es, 2007.

[137] Zhepeng (Lionel) Li, Xiao Fang, and Olivia R. Liu Sheng. A Survey of Link Recommendation for Social Networks: Methods, Theoretical Foundations, and Future Research Directions. *ACM Transactions on Management Information Systems*, 9(1), October 2017.

[138] David Liben-Nowell and Jon Kleinberg. The Link Prediction Problem for Social Networks. In *Proceedings of the Twelfth International Conference on Information and Knowledge Management*, CIKM '03, page 556–559, 2003.

[139] Chenhao Ma, Reynold Cheng, Laks V. S. Lakshmanan, Tobias Grubenmann, Yixiang Fang, and Xiaodong Li. LINC: A Motif Counting Algorithm for Uncertain Graphs. *Proceedings of the VLDB Endowment*, 13(2):155–168, October 2019.

[140] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[141] Dror Marcus and Yuval Shavitt. RAGE: A Rapid Graphlet Enumerator for Large Networks. *Computer Networks*, 56(2):810–819, February 2012.

[142] Mugilan Mariappan, Joanna Che, and Keval Vora. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 83–98, 2021.

[143] Mugilan Mariappan and Keval Vora. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–16, 2019.

[144] Daniel Mawhirter, Sam Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu. Dryadic: Flexible and Fast Graph Pattern Matching at Scale. In *30th International Conference on Parallel Architectures and Compilation Techniques*, PACT '21, pages 289–303, 2021.

[145] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. GraphZero: A High-Performance Subgraph Matching System. *SIGOPS Operating Systems Review*, 55(1):21–37, June 2021.

[146] Daniel Mawhirter and Bo Wu. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 509–523, 2019.

[147] Daniel Mawhirter, Bo Wu, Dinesh Mehta, and Chao Ai. ApproxG: Fast Approximate Parallel Graphlet Counting through Accuracy Control. In *Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '18, pages 533–542, 2018.

[148] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[149] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what COST? In *Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[150] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential Dataflow. In *CIDR*, 2013.

[151] Ine Melckenbeeck, Pieter Audenaert, Didier Colle, and Mario Pickavet. Efficiently Counting All Orbits of Graphlets of Any Order in a Graph Using Autogenerated Equations. *Bioinformatics*, 34(8):1372–1380, November 2017.

[152] Ine Melckenbeeck, Pieter Audenaert, Thomas Van Parys, Yves Van De Peer, Didier Colle, and Mario Pickavet. Optimising Orbit Counting of Arbitrary Order by Equation Selection. *BMC Bioinformatics*, 20(1), January 2019.

[153] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 415–426, 2015.

[154] Amine Mhedhbi and Semih Salihoglu. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proceedings of the VLDB Endowment*, 12(11):1692–1704, July 2019.

[155] Nema Milaninia. Biases in Machine Learning Models and Big Data Analytics: The International Criminal and Humanitarian Law Implications. *International Review of the Red Cross*, 102(913):199–234, 2020.

[156] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298(5594):824–827, 2002.

[157] Mircea Moca, Gheorghe Cosmin Silaghi, and Gilles Fedak. Distributed Results Checking for MapReduce in Volunteer Computing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 1847–1854, 2011.

[158] Aida Mrzic, Pieter Meysman, Wout Bittremieux, Pieter Moris, Boris Cule, Bart Goethals, and Kris Laukens. Grasping Frequent Subgraph Mining for Bioinformatics Applications. *BioData Mining*, 11(1):20, 2018.

[159] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *Proceedings of the VLDB Endowment*, 12(11):1539–1552, July 2019.

[160] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '21, page 35–48, 2021.

[161] Neo4j, Inc. Neo4j Graph Database, 2022. Version 4.4.

[162] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, 2013.

[163] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, page 343–356, 2011.

[164] Caleb C. Noble and Diane J. Cook. Graph-Based Anomaly Detection. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, page 631–636, 2003.

[165] Ricardo Padilha and Fernando Pedone. Augustus: Scalable and Robust Storage for Cloud Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 99–112, 2013.

[166] Lucia Pallottino, Eric M Feron, and Antonio Bicchi. Conflict resolution problems for air traffic management systems solved with mixed integer programming. *IEEE Transactions on Intelligent Transportation Systems*, 3(1):3–11, 2002.

[167] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and Sampling Triangles from a Graph Stream. *Proceedings of the VLDB Endowment*, 6(14):1870–1881, September 2013.

[168] Ali Pinar, Comandur Seshadhri, and Vaidyanathan Vishal. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 1431–1440, 2017.

[169] Daniel Presser, Lau Cheuk Lung, and Miguel Correia. Greft: Arbitrary Fault-Tolerant Distributed Graph Processing. In *2015 IEEE International Congress on Big Data*, pages 452–459, 2015.

[170] Hao Qi, Yu Zhang, Ligang He, Kang Luo, Jun Huang, Haoyu Lu, Jin Zhao, and Hai Jin. PSMiner: A Pattern-Aware Accelerator for High-Performance Streaming Graph Pattern Mining. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2023.

[171] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '21, pages 321–335. USENIX Association, April 2021.

[172] Miao Qiao, Hao Zhang, and Hong Cheng. Subgraph Matching: On Compression and Computation. *Proceedings of the VLDB Endowment*, 11(2):176–188, October 2017.

[173] Mahmudur Rahman, Mansurul Bhuiyan, and Mohammad Al Hasan. GRAFT: An Approximate Graphlet Counting Algorithm for Large Graph Analysis. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, page 1467–1471, 2012.

[174] Mahmudur Rahman and Mohammad Al Hasan. Approximate triangle counting algorithms on multi-cores. In *2013 IEEE International Conference on Big Data*, Big Data '13, pages 127–133, 2013.

[175] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. Fast and Robust Distributed Subgraph Enumeration. *Proceedings of the VLDB Endowment*, 12(11):1344–1356, 2019.

[176] Tahsin Reza, Matei Ripeanu, Nicolas Tripoul, Geoffrey Sanders, and Roger Pearce. PruneJuice: Pruning Trillion-Edge Graphs to a Precise Pattern-Matching Solution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 265–281, 2018.

[177] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using Abstraction to Improve Fault Tolerance. *SIGOPS Operating Systems Review*, 35(5):15–28, October 2001.

[178] Marko A. Rodriguez. The Gremlin Graph Traversal Machine and Language. In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, page 1–10, 2015.

[179] Rahmtin Rotabi, Krishna Kamath, Jon Kleinberg, and Aneesh Sharma. Detecting Strong Ties Using Network Motifs. In *Proceedings of the 26th International Conference on World Wide Web Companion*, WWW '17 Companion, pages 983–992, 2017.

[180] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *Proceedings of the Fifth International Workshop on Graph Data-Management Experiences & Systems*, GRADES '17, pages 1–6, 2017.

[181] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, 2015.

[182] Tanay Kumar Saha and Mohammad Al Hasan. FS3: A sampling based method for top-k frequent subgraph mining. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 72–79, 2014.

[183] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The Future is Big Graphs: A Community View on Graph Processing Systems. *Communications of the ACM*, 64(9):62–71, August 2021.

[184] Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 1–12, 2013.

[185] Soumajyoti Sarkar, Ruocheng Guo, and Paulo Shakarian. Using Network Motifs to Characterize Temporal Network Evolution Leading to Diffusion Inhibition. *CoRR*, abs/1903.00862, 2019.

[186] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Survey*, 22(4):299–319, December 1990.

[187] Tom Schouwenaars, Bart De Moor, Eric Feron, and Jonathan How. Mixed integer programming for multi-vehicle path planning. In *2001 European Control Conference*, ECC '01, pages 2603–2608. IEEE, 2001.

[188] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. *Communications of the ACM*, 54(2):100–107, February 2011.

[189] Uwe Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37(3):312–323, 1988.

[190] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. QFrag: Distributed Graph Search via Subgraph Isomorphism. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 214–228, 2017.

[191] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. Parallel Subgraph Listing in a Large-Scale Graph. In *Proceedings of the 2014 International Conference on Management of Data*, SIGMOD '14, pages 625–636, 2014.

[192] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. GraphPi: High Performance Graph Pattern Matching through Effective Redundancy Elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, pages 1–14, 2020.

[193] Julian Shun and Guy E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, 2013.

[194] Julian Shun and Kanat Tangwongsan. Multicore triangle computations without tuning. In *2015 IEEE 31st International Conference on Data Engineering*, pages 149–160, 2015.

[195] George M. Slota and Kamesh Madduri. Fast approximate subgraph counting and enumeration. In *2013 42nd International Conference on Parallel Processing*, ICPP '13, pages 210–219, 2013.

[196] George M. Slota and Kamesh Madduri. Complex network analysis using parallel approximate motif counting. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 405–414, 2014.

[197] Qi Song, Mohammad Hossein Namaki, and Yinghui Wu. Answering why-questions for subgraph queries in multi-attributed graphs. In *IEEE International Conference on Data Engineering (ICDE '19)*, pages 40–51, 2019.

[198] Julian James Stephen and Patrick Eugster. Assured Cloud-Based Data Analysis with ClusterBFT. In *14th International Middleware Conference*, volume LNCS-8275 of *Middleware '13*, pages 82–102, Beijing, China, December 2013. Springer. Part 1: Distributed Protocols.

[199] Shixuan Sun and Qiong Luo. In-Memory Subgraph Matching: An In-Depth Study. In *Proceedings of the 2020 International Conference on Management of Data*, SIGMOD '20, pages 1083–1098, 2020.

[200] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID (Transactions). In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '21, page 1–17, 2021.

[201] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 320–331, 2016.

[202] Nishil Talati, Haojie Ye, Yichen Yang, Leul Belayneh, Kuan-Yu Chen, David Blaauw, Trevor Mudge, and Ronald Dreslinski. NDMiner: accelerating graph pattern mining using near data processing. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 146–159, 2022.

[203] N. Talukder and M. J. Zaki. "a distributed approach for graph mining in massive networks". *Data Mining and Knowledge Discovery*, 30(5):1024–1052, September 2016.

[204] Michiaki Tatsubori and Shohei Hido. Opportunistic Adversaries: On Imminent Threats to Learning-Based Business Automation. In *Proceedings of the 2012 Annual SRII Global Conference*, SRII '12, page 120–129, USA, 2012.

[205] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 425–440, 2015.

[206] Harsh Thakkar, Dharmen Punjani, Jens Lehmann, and Sören Auer. Two for One: Querying Property Graph Databases Using SPARQL via Gremlinator. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, 2018.

[207] Tian Tian, Jun Zhu, Fen Xia, Xin Zhuang, and Tong Zhang. Crowd Fraud Detection in Internet Advertising. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, page 1100–1110, 2015.

[208] Vasileios Trigonakis, Jean-Pierre Lozi, Tomás Faltín, Nicholas P. Roth, Iraklis Psaroudakis, Arnaud Delamare, Vlad Haprian, Calin Iorgulescu, Petr Koupy, Jinsoo Lee, Sungpack Hong, and Hassan Chafi. aDFS: An Almost Depth-First-Search Distributed Graph-Querying System. In *2021 USENIX Annual Technical Conference*, ATC '21, pages 209–224, 2021.

[209] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. DOULION: Counting Triangles in Massive Graphs with a Coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, page 837–846, 2009.

[210] Julian Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics*, 15:1–1, 2011.

[211] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[212] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, 2016.

[213] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '07, page 59–72, 2007.

[214] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, page 135–144, USA, 2009.

[215] Keval Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference*, ATC '19, pages 429–442, July 2019.

[216] Keval Vora, Rajiv Gupta, and Guoqing Xu. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 237–251, 2017.

[217] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM. In *Proceedings of the 2014 International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 861–878, 2014.

[218] Keval Vora, Chen Tian, Rajiv Gupta, and Ziang Hu. CoRAL: Confined Recovery in Distributed Asynchronous Graph Processing. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 223–236, 2017.

[219] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. RStream: Marrying Relational Algebra with Streaming for Efficient Graph Mining on a Single Machine. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 763–782, 2018.

[220] Li Wang, Hongying Zhao, Jing Li, Yingqi Xu, Yujia Lan, Wenkang Yin, Xiaoqin Liu, Lei Yu, Shihua Lin, Michael Yifei Du, Xia Li, Yun Xiao, and Yunpeng Zhang. Identifying Functions and Prognostic Biomarkers of Network Motifs Marked By Diverse Chromatin States in Human Cell Lines. *Oncogene*, 39(3):677–689, September 2019.

[221] Xuyun Wen, Wei-Neng Chen, Ying Lin, Tianlong Gu, Huaxiang Zhang, Yun Li, Yilong Yin, and Jun Zhang. A Maximal Clique Based Multiobjective Evolutionary Algorithm for Overlapping Community Detection. *IEEE Transactions on Evolutionary Computation*, 21(3):363–377, June 2017.

[222] Yibo Wu, Jianfeng Zhu, Wenrui Wei, Longlong Chen, Liang Wang, Shaojun Wei, and Leibo Liu. Shogun: A Task Scheduling Framework for Graph Mining Accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, 2023.

[223] Chengshuo Xu, Keval Vora, and Rajiv Gupta. PnP: Pruning and Prediction for Point-To-Point Iterative Graph Analytics. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 587–600, 2019.

[224] Hiroyuki Yamada and Jun Nemoto. Scalar DL: Scalable and Practical Byzantine Fault Detection for Transactional Database Systems. *Proceedings of the VLDB Endowment*, 15(7):1324–1336, June 2022.

[225] Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities based on Ground-Truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[226] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2049–2062, 2021.

[227] Hong Yao, Qingling Duan, Daoliang Li, and Jianping Wang. An improved K-means clustering algorithm for fish image segmentation. *Mathematical and Computer Modelling*, 58(3):790–798, 2013.

[228] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '03, page 253–267, 2003.

[229] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, 2019.

[230] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, April 2012.

[231] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, 2013.

[232] Stéphane Zampelli, Yves Deville, and Pierre Dupont. Approximate Constrained Subgraph Matching. In Peter van Beek, editor, *Principles and Practice of Constraint Programming*, CP 2005, pages 832–836, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[233] Gensheng Zhang, Damian Jimenez, and Chengkai Li. Maverick: Discovering exceptional facts from knowledge graphs. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '18)*, pages 1317–1332, 2018.

[234] Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, and Hong Cheng. Distributed Subgraph Counting: A General Approach. *Proceedings of the VLDB Endowment*, 13(12):2493–2507, July 2020.

[235] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. Kaleido: An Efficient Out-of-core Graph Mining System on A Single Machine. In *36th IEEE International Conference on Data Engineering*, ICDE '20, pages 673–684, 2020.

[236] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 301–316, 2016.

[237] Ruoyu Zou and Lawrence B. Holder. Frequent Subgraph Mining on a Single Large Graph Using Sampling Techniques. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, page 171–178, 2010.

Appendix A

Anti-Vertex: Generalizations and Further Examples

A.1 Generalizations

A.1.1 Other Matching Semantics

The anti-vertex definition provided in Section 4.2 can be used in other matching semantics as well, including those provided by some graph database management systems like Neo4j [161].

Homomorphism. In homomorphism semantics, m only needs to satisfy the vertex labels and edge types, and preserve edge relationships. Hence, any data vertex with the correct edges and labels can fulfill the anti-vertex requirement and invalidate the match, including previously mapped vertices. C is defined as follows.

$$C(\overline{u}) = \bigcap_{v:(\overline{u},v)\in E(p)} N(m(v),(\overline{u},v),\overline{u})$$

No-Repeated-Edge. No-repeated-edge semantics requires that m provide an injective mapping from edges in P to edges in G. Hence, for anti-vertices, the data edges that are already mapped by m cannot invalidate the match, but vertices from m can satisfy the anti-vertex requirement (i.e., allow repeated vertices). C is defined as follows.

$$C(\overline{u}) = \bigcap_{v:(\overline{u},v)\in E(p)} N(m(v),(\overline{u},v),\overline{u}) \setminus m(N(v,(\overline{u},v),\overline{u}))$$

A.1.2 Property Graphs

When storing and processing graph-structured data it is often convenient to consider not only the structural information encoded by connections between vertices, but also the myriad information associated with each vertex and edge. Two popular models for rich graph

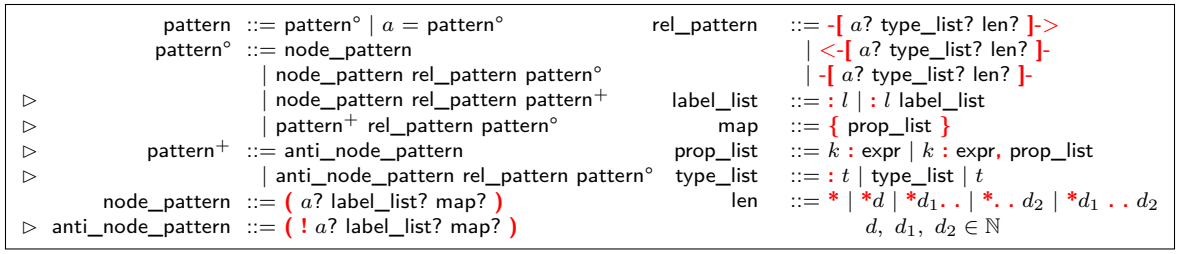


Figure A.1: Syntax of Cypher patterns with anti-vertex. Enhancements to the original grammar are marked with ▷.

data are the Resource Description Framework (RDF) [67] and property graphs [21]. Property graphs can model more complex structures than RDF by allowing edges and vertices to be associated with arbitrary key-value pairs called properties. As a result, property graphs have gained widespread adoption by both commercial and academic graph databases [74, 86, 212, 20].

We define property graph based on the definition in Cypher [86], while including anti-vertices. A *property graph* is a graph g with additional functions $\rho : E(p) \to V(p) \times V(p)$ and $\pi : V(p) \cup E(p) \to \mathbb{P}(\mathcal{K} \times \mathcal{V})$, where $\mathcal{K}$ and $\mathcal{V}$ are sets of property keys and values, respectively. ρ maps each edge to an ordered pair of endpoints, so that a pair of vertices can have multiple edges between them. The definition of the (e, u)-neighbourhood of a vertex v generalizes easily to property graphs by using ρ to obtain all edges involving v and by considering the properties of e and u in addition to their type and labels.

Anti-vertex also naturally generalizes to property graphs. In a property graph, each anti-vertex $\overline{u}$ can be incident on multiple edges with directions. Thus, to transform the previous definitions of $C(\overline{u})$ to fit property graphs, it suffices to intersect the $(e, \overline{u})$-neighbourhoods of v for every $e \in E(p)$ where $\rho(e) = (\overline{u}, v)$ or $\rho(e) = (v, \overline{u})$, and perform the same set differences.

Let g and p be property graphs. The semantics with isomorphism in property graphs can be expressed by defining C as follows.

$$C(\overline{u}) = \bigcap_{\substack{e \in E(p): \\ \rho(e) = (\overline{u}, v)}} N(m(v), e, \overline{u}) \cap \bigcap_{\substack{e \in E(p): \\ \rho(e) = (v, \overline{u})}} N(m(v), e, \overline{u}) \setminus m(V(p))$$

Similarly, semantics of anti-vertices with homomorphism and no-repeated-edge semantics in property graphs can be defined by translating the definitions from Section 4.2.2.

To give intuition for how anti-vertex queries work on property graphs, Table A.1 shows various subgraph queries where anti-vertices are used in different ways. The data graphs capture social network information where vertices represent people and edges represent LIKES and FOLLOWS relationships. The patterns contain anti-vertices, and their textual description is provided to help familiarize with the concept by demonstrating how anti-vertices are perceived for social network analysis. For isomorphism and no-repeated-edge matching semantics, the resulting mappings between query vertices and data vertices are shown in relational format.

```
MATCH (a:SCHOOL)--(b:BUSINESS),
      (a)--(c:FIRE_HYDRANT)--(b),
      (a)--(d:FIRE_HYDRANT)--(b)
WHERE NOT EXISTS {
  MATCH (a:SCHOOL)--(b:BUSINESS),
        (a)--(c:FIRE_HYDRANT)--(b),
        (a)--(d:FIRE_HYDRANT)--(b),
        (a)--(e:FIRE_HYDRANT)--(b)
}
RETURN a, b, c, d
```

```
MATCH (a:SCHOOL)--(b:BUSINESS),
      (a)--(c:FIRE_HYDRANT)--(b),
      (a)--(d:FIRE_HYDRANT)--(b)
WHERE NOT EXISTS {
   MATCH (a)--(e:FIRE_HYDRANT)--(b)
   WHERE e<>c
     AND e<>d
}
RETURN a, b, c, d
```

Figure A.2: Cypher queries for the anomaly detection use case.

```
MATCH (a)--(b), (a)--(c), (a)--(d),
      (b)--(c), (b)--(d), (c)--(d)
WHERE NOT EXISTS{
  MATCH (a)--(b), (a)--(c), (a)--(d),
        (b)--(c), (b)--(d), (c)--(d),
        (a)--(e), (b)--(e),
        (c)--(e), (d)--(e)
}
RETURN a, b, c, d
```

```
MATCH (a)--(b), (a)--(c), (a)--(d),
      (b)--(c), (b)--(d), (c)--(d)
WHERE NOT EXISTS{
  MATCH (a)--(e), (b)--(e),
        (c)--(e), (d)--(e)
  WHERE e<>a AND e<>b
    AND e<>c AND e<>d
}
RETURN a, b, c, d
```

Figure A.3: Cypher queries for the maximal cliques use case.

A.2 Anti-Vertex in Graph Query Languages

Recent graph query languages [86, 212, 74, 20] integrate declarative "ASCII-art" pattern expressions with familiar SQL constructs. In particular, Cypher [86] is a popular graph query language, used in both academic and commercial graph databases including Neo4j [161], Amazon Neptune [14], and GraphFlow [120]. The anti-vertex and anti-edge constructs can be incorporated in existing graph query languages to express complex structural and neighborhood constraints as easily as standard subgraph queries. To demonstrate, in this section we examine the ways neighbourhood constraints must currently be implemented in Cypher, and then develop prototype extensions to Cypher's pattern matching syntax to support anti-vertices natively.

A.2.1 Constraining Neighbourhoods in Cypher

In this section we show how neighbourhood constraints can currently be expressed in Cypher by continuing the earlier examples.

Example A.2.1. Consider the anomaly detection and maximal cliques use cases from Example 4.2.1.

1. *Anomaly Detection.* There are two obvious approaches to writing a Cypher query for this problem, both shown in Figure A.2. In the first query, the problem is reformulated as matching subgraphs with two fire hydrants that are not part of subgraphs containing three fire hydrants. Expressing the absence of the third fire hydrant in such an indirect fashion causes tedious repetition and increase in query sizes. This not only makes it challenging to read and manage those queries (e.g., incrementally adjust to add new constraints), but also makes the process of writing complex queries (e.g., with multiple constraints) error-prone. The second query incurs less repetition in the subquery than

the first approach, but requires users to specify additional constraints against the outer query to achieve the desired semantics. Determining the correct constraints to ensure the subquery does not filter too many or too few subgraphs is difficult in larger queries, as users must visualize how their query will be matched against complex graph structures. Both approaches lead to larger, less declarative queries involving error-prone subqueries. Instead, *the absence of another fire hydrant neighbor* can be directly expressed using an anti-vertex.

2. *Maximal Cliques.* While cliques of a certain size can be easily expressed as a Cypher query, the maximality requirement can again be expressed in two ways, shown in Figure A.3. The first query reformulates the problem as finding cliques of size k that are not contained inside cliques of size $k+1$, while the second query finds vertices adjacent but not equal to all k previously matched vertices.

A.2.2 Augmenting Cypher with Anti-Vertex Semantics

Cypher pattern matching syntax allows node patterns, relationship patterns, and path patterns. Since anti-vertices express absence of neighbors, they will be best expressed using relationship patterns and path patterns. However, the anti-vertex semantics developed in Section 4.2.2 do not consider the case where two anti-vertices are connected via an edge (recall the assumption in Chapter 4 when anti-vertices are first introduced). This leaves the semantics of fixed/variable-length path fragments containing anti-vertices ambiguous. While such semantics are left for future work, we envision the grammar to be able to support arbitrary path patterns with anti-vertices.

Hence, we develop two prototype extensions to Cypher's pattern matching syntax: one that allows arbitrary path patterns with anti-vertices (presented in Section A.2.2), and one that limits the grammar to the anti-vertex semantics defined in this thesis (presented in Section A.2.2). Allowing arbitrary path patterns requires fewer changes to the original grammar, and thus easier implementation and validation in existing query engines, at the cost of some ambiguity regarding the semantics of anti-vertices in fixed/variable-length path patterns. Meanwhile, keeping the grammar limited requires more complex changes to the original grammar, but has no ambiguity, and provides flexibility for future work to define the semantics of path patterns involving an anti-vertex consistently (i.e., handle paths with one or both endpoints being an anti-vertex consistently).

Grammar with Arbitrary Path Patterns

Following the same notation as Cypher, Figure A.1 shows an extended pattern matching grammar that supports anti-vertices (extensions added for anti-vertex support are marked with ▷). This syntax only applies within `MATCH` clauses of Cypher; the remainder of Cypher's syntax is unaffected.
An anti-vertex is defined by the anti_node_pattern construct, which is identical to node_pattern

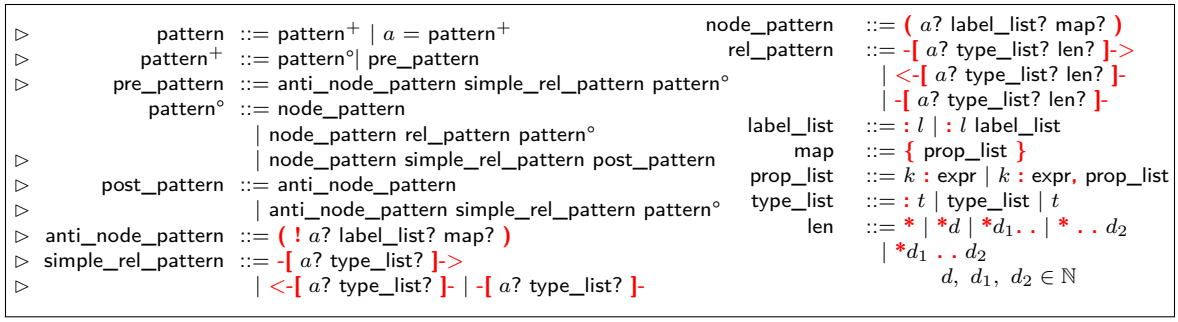


Figure A.4: Syntax of Cypher patterns with anti-vertex, without arbitrary path patterns. Enhancements marked with ▷.

from the original Cypher grammar, but marked with a ! symbol[†]. This construct only appears in pattern$^+$ either by itself or accompanied by rel_pattern. We compose these fragments with Cypher's original pattern definition to allow as much programmer flexibility as possible.

Intuitively, the syntax allows an anti-vertex to be present:

1. at the beginning of the pattern:

```
(!a)--
```

2. in the middle of the pattern:

```
--(!a)--
```

3. at the end of the pattern:

```
--(!a)
```

Since we utilize rel_pattern as defined in the original grammar, fragments of path patterns with anti-vertices can be expressed in this grammar. For example, the following are allowed:

```
(a)-[*3]-(!b)
```

```
(!a)--(b)
```

```
()-[*2]-(!a)-[*2]-()
```

The pattern$^+$ separates the anti_node_pattern from pattern°, which disallows anti-vertices at both endpoints of a relationship.

Grammar without Arbitrary Path Patterns

Here we limit the syntax to only express anti-vertices where semantics are well-defined in this thesis. Figure A.4 shows the extended pattern matching grammar for this case.

The simple_rel_pattern is added to ensure path fragments containing anti-vertices are not length-based (fixed or variable). simple_rel_pattern is simply rel_pattern without a len pa-

[†]The ! symbol typically denotes the *not* operator in programming languages, which fits the meaning for anti-vertex (data vertex *not* present).

```
MATCH (a:SCHOOL)--(b:BUSINESS),
      (a)--(fh1:FIRE_HYDRANT)--(b),
      (a)--(fh2:FIRE_HYDRANT)--(b),
      (a)--(!fh3:FIRE_HYDRANT)--(b)
RETURN a, b
```

(a) Anomaly Detection

```
MATCH (a)--(b), (a)--(c), (a)--(d),
      (b)--(c), (b)--(d), (c)--(d),
      (a)--(!e), (b)--(!e),
          (c)--(!e), (d)--(!e)
RETURN a, b, c, d
```

(b) Maximal 4-Clique

Figure A.5: Cypher queries using anti-vertex for use cases in Example 4.2.1.

rameter. The anti_node_pattern construct only appears in pre_pattern and post_pattern, accompanied by simple_rel_pattern. Intuitively, pre_pattern represents the syntax fragment

```
(!a)--
```

and post_pattern represents the syntax fragments

```
--(!a)    and    --(!a)--
```

These fragments are composed with Cypher's original pattern definition to allow as much programmer flexibility as possible.
A pattern either begins with an anti-vertex (through pre_pattern) or a standard vertex (through pattern°). Anti-vertices in the middle or at the end of a pattern are supported through mutual recursion between pattern° and post_pattern. Two anti-vertices will never form both endpoints of a relationship because pre_pattern never occurs directly before post_pattern.

While fixed/variable length path fragments with anti-vertex are disallowed, regular fixed/-variable length path fragments containing node_pattern can still be expressed (same as defined in original Cypher grammar). For example, the following are allowed:

```
(a)-[*2]-()--(!b)

(!a)--(b)

(a)--(!b)--(c)
```

A.2.3 Examples with the Enhanced Cypher Grammars

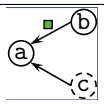
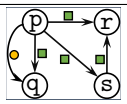
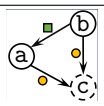
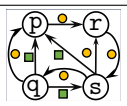
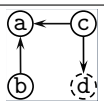
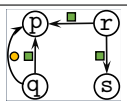
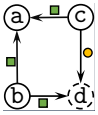
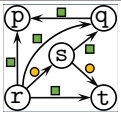
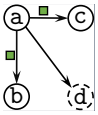
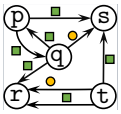
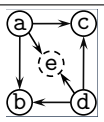
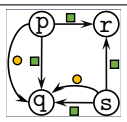
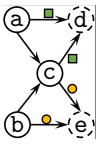
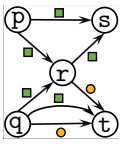
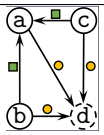
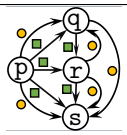
We revisit the use cases from Example 4.2.1 to demonstrate how they can be easily expressed using the modifications to the Cypher grammar. Figure A.5a and Figure A.5b show the example Cypher queries in Figure 4.2 and Figure 4.3 rewritten declaratively with this syntax.

Example A.2.2. Consider the anomaly detection and maximal cliques Cypher queries from Example A.2.1.

1. *Anomaly Detection.* The Cypher query with anti-vertex for anomaly detection is shown in Figure A.5a. Instead of a long subquery which repeats most of the initial `MATCH` clause, or one that must explicitly specify the matching semantics, the query directly expresses the anomalous subgraph using an anti-vertex to denote absence of a third fire hydrant. Anti-vertices are expressed similarly to standard vertices, including specifying labels and properties.

2. *Maximal Cliques.* Figure A.5b shows the Cypher query with anti-vertex for finding maximal cliques of size 4. Previously without anti-vertex support (shown in Figure 4.3), the constraints induced by the matching semantics on the subgraph were explicitly enforced in a `WHERE` clause. Now, the query directly expresses the maximality constraint by connecting all vertices to an anti-vertex, guaranteeing consistency with the underlying matching semantics. The anti-vertex `e` is unconstrained, thus if any data vertex can be mapped to `e` (i.e., there is a vertex adjacent to the matches for all of `a,b,c,d`), then the subgraph will be discarded, as matching `e` would create a clique of size 5.

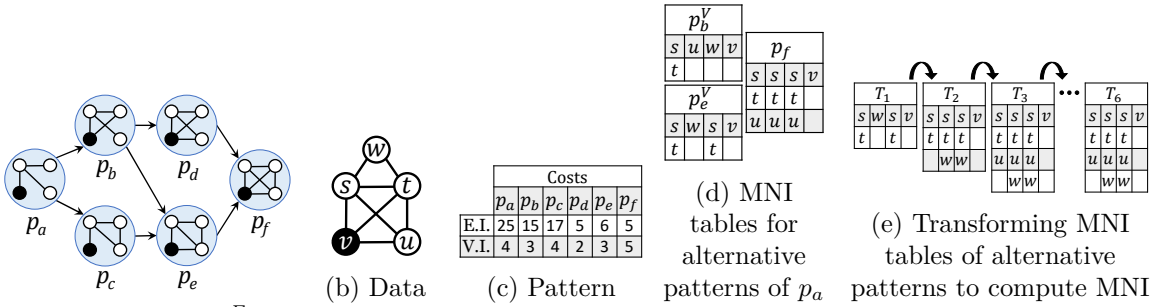
Table A.1: Examples with anti-vertices.

No.	Pattern	Data Graph	Description (w.r.t. Isomorphism)	Subgraph Results		Notes
				Isomorphism	No-Repeated-Edge	
Q1			Find a,b such that i) b FOLLOWs a, and ii) a is not FOLLOWed/LIKEd by another PERSON	**a** **b** s p q p	**a** **b** s p	(q,p) is not a match in no-repeated-edge semantics
Q2			Find a,b such that i) b FOLLOWs a, and ii) a and b do not LIKE another common PERSON	**a** **b** p q s q	**a** **b** p q s q	(p,s) is not a match since p,s both LIKE r (s,q) is a match since only q LIKEs p
Q3			Find a,b,c such that i) b and c FOLLOW/LIKE a, and ii) c does not FOLLOW/LIKE another PERSON	**a** **b** **c** p r q	**a** **b** **c** p q q	(p,q,r) is not a match since r FOLLOWs s
Q4			Find a,b,c such that i) b and c FOLLOW a, and ii) b does not FOLLOW d, and iii) c does not LIKE d	**a** **b** **c** p q r p r q q s r	**a** **b** **c** p q r p r q q s r	(q,r,s) is not a match since r FOLLOWs t and s LIKEs t
Q5			Find a,b,c such that i) a FOLLOWs b and c, and ii) a does not FOLLOW/LIKE another PERSON	**a** **b** **c** p q s p s q t s r t r s	**a** **b** **c** p q s p s q	(t,s,r) and (t,r,s) are not matches with no-repeated-edge semantics since t LIKEs and FOLLOWs r
Q6			Find a,b,c,d such that i) a and d FOLLOW/LIKE b and c, and ii) a and d do not FOLLOW/LIKE another common PERSON	**a** **b** **c** **d** p q r s p r q s s q r p s r q p	**a** **b** **c** **d** - - - -	No matches with no-repeated-edge semantics since p and s LIKE and FOLLOW q
Q7			Find a,b,c such that i) a and b FOLLOW/LIKE c, and ii) a and c do not FOLLOW another common PERSON , and iii) b and c do not LIKE another common PERSON	**a** **b** **c** p r s r p s q r t r q t q p r	**a** **b** **c** p r s r p s q r t r q t q p r q q t	(p,q,r) is not a match since p and r both FOLLOW s, and q and r both LIKE t
Q8			Find a,b,c such that i) b and c FOLLOW a, and ii) the three do not LIKE a common PERSON	**a** **b** **c** s p r s r p	**a** **b** **c** s p r s r p	(r,p,q) is not a match since r,p,q LIKE s. (s,p,r) is a match since r,p LIKE q but s does not LIKE q

Appendix B

Applications with Subgraph Morphing

We will walk through the main steps in applying Subgraph Morphing(*i.e.*, S-DAG generation, pattern selection, and result conversion) on two graph mining use cases: Frequent Subgraph Mining and Subgraph Counting.



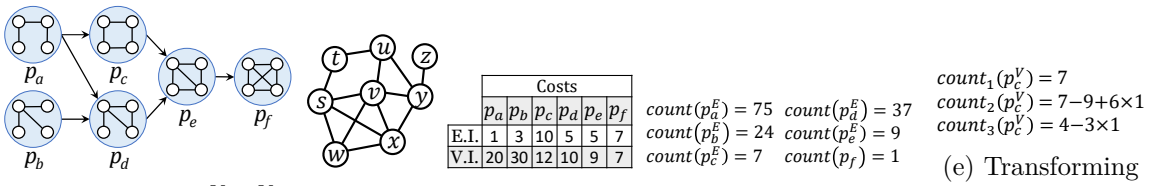
(a) S-DAG for p_a^E (b) Data graph (c) Pattern costs (d) MNI tables for alternative patterns of p_a (e) Transforming MNI tables of alternative patterns to compute MNI for p_a

Figure B.1: Frequent Subgraph Mining (FSM) with Subgraph Reshaping. Key steps in reshaping are shown for pattern p_a.

B.1 Frequent Subgraph Mining

Since FSM explores labeled edge-induced patterns, it can end up matching and computing MNI for a large number of patterns. To simplify exposition, we consider a single pattern p_a^E (edge-induced 4-star). Figure B.1 summarizes the example.

The S-DAG is constructed by recursively adding the superpatterns of p_a^E. The resulting S-DAG is shown in Figure B.1a. Since we are dealing with labeled patterns, some of the superpatterns can have identical structures but different labelings. Patterns p_b and p_c in the S-DAG show this case.



(a) S-DAG for p_a^V, p_b^V and p_c^V (b) Data graph (c) Pattern costs (d) Counts for alternative patterns (e) Transforming counts to compute for p_c^E

Figure B.2: Subgraph Counting (SC) with Subgraph Reshaping. Key steps in reshaping shown for input patterns p_a, p_b and p_c.

Costs are estimated for both variants of each pattern in the S-DAG. The pattern costs for our example are shown in Figure B.1c. Since MNI computations are sensitive to output size, patterns that are estimated to produce more matches have higher costs. For example, p_a^E is the least constrained pattern in the S-DAG, and hence has the highest cost. Similarly, the other superpatterns have lower costs for the vertex-induced variants which cause fewer matches.

Next, the alternative pattern set S is constructed using Algorithm 1. Initially, $S = \{p_a^E\}$. Then we iterate over the direct parents of p_a^E in the S-DAG, beginning with p_b^V. The only child of p_b^V is p_a^E with cost 25 while the superpatterns of p_a^E (including p_a^V) have combined cost 17. As a result, S is updated to contain $\{p_a^V, p_b^V, p_c^V, p_d^V, p_e^V, p_f\}$ and all these patterns have their costs set to 0. The algorithm converges in the next iteration as the alternative pattern set S does not change.

The matching engine explores the subgraphs that match the patterns in S. The final step is to compute the MNI table for p_a^E from the MNI results for patterns in S. To illustrate this, consider the sample data graph shown in Figure B.1b. Figure B.1d shows the MNI tables for the alternative pattern set S. Note that p_a^V, p_c^V and p_d^V do not have any matches in this example, and hence their MNI tables are empty (not shown). Figure B.1e shows how the final MNI table is computed from the tables for alternative patterns. Starting with an empty table, the MNI tables are merged after permuting them using permutation functions. Consider the MNI table for p_e^V. There are two subgraph isomorphisms from p_a^E to p_e^V, which lead to two permutations. The first one is the identity permutation (*i.e.*, unchanged) which results in T_1. The second one sends the first column of the MNI table to the second, the second column to the third, and the third column to the first. Applying this permutation and merging the resulting table with T_1 gives T_2. This process continues with the next alternative pattern p_f and results in T_3. There are two further isomorphisms into p_f, and one into p_b^V, none of which affect the final result, and the process completes with T_6.

B.2 Subgraph Counting

In this application, we are interested in counting the subgraphs that match three unlabeled vertex-induced patterns: a 4-star, a 4-cycle, and a 4-chain. Figure B.2 summarizes the example where the three patterns are named p_a, p_b and p_c.

Similar to the previous example, S-DAG is constructed by recursively adding superpatterns of those three input patterns. The resulting S-DAG is shown in Figure B.2a and the esti-

mated pattern costs are shown in Figure B.2c. In this case, since the patterns are unlabeled and the counting aggregation is a constant time operation, the set operation time is the primary concern. Hence, edge-induced variants of sparse patterns tend to be far cheaper to compute than their vertex-induced variants which require additional set differences.

Using the S-DAG and the pattern costs, Algorithm 1 computes the alternative pattern set S. Initially, S starts with $\{p_a^V, p_b^V, p_c^V\}$. Then, p_a^V is evaluated against its superpatterns (p_a^E, p_c^V, p_d^E, p_e^E, p_f). Since p_a^V costs 20 while its superpatterns cost 30 combined, p_a^V is not morphed in this step. Similarly, p_b^V is not morphed in the next step. However, when $C = \{p_a^V, p_b^V\}$, the cost of C is 50 while the cost of the combined superpatterns (including the variants of the patterns in C) is only 33. Hence, S is updated to replace p_a^V and p_b^V with p_a^E, p_b^E, and the other superpatterns, and the cost of these superpatterns is set to 0. Notice that p_c^V is one of the superpatterns of p_a and p_b. Since the original cost of the superpatterns of p_c^V was greater than the cost of p_c^V, it would not have been morphed. However, since the cost of superpatterns got set to 0, the new cost of superpatterns of p_c^V reduces to 10. Hence, S is updated once again with p_e^E instead of p_e^V. The final alternative pattern set S is $\{p_a^E, p_b^E, p_c^E, p_d^E, p_e^E, p_f\}$.

After matching the alternative patterns, their results are transformed back to counts for p_a, p_b and p_c. We discuss this result conversion process next. Figure B.2b shows an example data graph, and Figure B.2d shows the number of matches in the data graph for the alternative patterns. The permutation function accounts for the subgraph isomorphisms from the original patterns to the alternative patterns. For example, consider pattern p_c^V whose counts can be computed using [SM-V1] in Figure 6.5, i.e., $|M(p_c^V)| = |M(p_c^E)| - |M(p_e^V)| - 3 \times |M(p_f)|$. However, our alternative set contains the morphed patterns for p_e^V, and hence, $|M(p_e^V)|$ is computed as $|M(p_e^E)| - 6 \times |M(p_f)|$. Therefore, $|M(p_c^V)| = 7 - 3 - 3 \times 1 = 1$. Counts for p_a^V and p_b^V are computed in a similar manner.