UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Vinícius Vitor dos Santos Dias

Graph Pattern Mining: consolidating models, systems, and abstractions

Belo Horizonte
2023

Vinícius Vitor dos Santos Dias

Graph Pattern Mining: consolidating models, systems, and abstractions

Final Version

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Dorgival Olavo Guedes Neto

Belo Horizonte
2023

© 2023, Vinícius Vitor dos Santos Dias
Todos os direitos reservados

Dias, Vinícius Vitor dos Santos.

D541g Graph pattern mining [recurso eletrônico]: consolidating models, systems, and abstractions / Vinícius Vitor dos Santos Dias – 2023.

1 recurso online (153 f. il, color.) : pdf.

Orientador: Dorgival Olavo Guedes Neto.

Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Referências: f. 139 -153

1. Computação – Teses. 2. Mineração de padrões em grafos – Teses. 3. Sistemas distribuídos – Teses. I. Guedes Neto, Dorgival Olavo. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*73(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Graph Pattern Mining: consolidating models, systems, and abstractions

VINICIUS VITOR DOS SANTOS DIAS

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. DORGIVAL OLAVO GUEDES NETO - Orientador
Departamento de Ciência da Computação - UFMG

PROF. SRINIVASAN PARTHASARATHY
Department of Computer Science and Engineering - Ohio State University

PROF. ARLEI LOPES DA SILVA
Department of Computer Science - Rice University

PROF. ÍTALO FERNANDO SCOTÁ CUNHA
Departamento de Ciência da Computação - UFMG

PROF. VINÍCIUS FERNANDES DOS SANTOS
Departamento de Ciência da Computação - UFMG

PROF. WAGNER MEIRA JÚNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 24 de março de 2023.

I dedicate this work to my mother, my grandmother and all my friends who supported me in this journey.

Acknowledgments

This endeavor would not have been possible without my mother, who always worked hard for me to have a good life with important achievements. I would also like to thank my late grandmother, who always helped to raise me and who would certainly be very happy for me. Special thanks to my girlfriend Denise, who is always around encouraging me not to give up and to stay calm and focused on my goals. That was very important and I hope to be there for her too. Many thanks to my dear friends for fun conversations and advice: Osvaldo, Elverton, Alloma, Samuel Oliveira, Carlos Teixeira, Samuel Ferraz, Paulo, Camila, Júlio, Flávio, Écio, and Thiago. I would like to thank my advisor Professor Dorgival Guedes, who supported me in the decisive final moments of this work. Many thanks to Professor Srinivasan Parthasarathy, who provided me with a great research experience at Ohio State University. Also thanks to Professor Wagner Meira Jr. for making this happen, it was a period of great development for me. Finally, I also would like to acknowledge the funding agencies CNPq/CAPES for financial support and the computational resources of the SDumont supercomputer provided by the National Laboratory for Scientific Computing (LNCC/MCTI).

Resumo

Mineração de padrões em grafos (MPG) se refere a uma classe de problemas envolvendo o processamento de subgrafos extraídos de um único grafo maior. Aplicações para algoritmos de MPG incluem consultas por subgrafos com certas propriedades de interesse, identificação de estruturas em redes biológicas, caracterização de redes sociais, entre outras. Desenvolver algoritmos de MPG é desafiador principalmente pela inerente presença de sub-rotinas não-triviais lidando com conceitos complexos em teoria de grafos, como identificação de isomorfismos. Neste contexto, sistemas de propósito geral para MPG surgem como uma alternativa para melhorar a experiência de usuários com esses algoritmos. Entretanto, sistemas de propósito geral para MPG falham em prover um modelo que seja de fácil entendimento e, ao mesmo tempo, qualificado para exprimir algoritmos alternativos para um mesmo problema usando diferentes paradigmas de enumeração de subgrafos, limitando a integração com fluxos de análise de dados atuais. Além disso, como sistemas de MPG são tão heterogêneos no que se refere aos paradigmas suportados e ambientes de execução, análises experimentais existentes são incapazes de diferenciar se as diferenças encontradas no desempenho dos sistemas são melhor explicadas pelos algoritmos utilizados ou pelos detalhes de implementação. Nesta tese, propomos um modelo para MPG baseado em primitivas, uma implementação distribuída escalável como prova de conceito para o modelo e uma avaliação experimental extensiva dos paradigmas mais usados por sistemas de MPG. Nós demonstramos empiricamente a efetividade de nossas soluções ao observar um desempenho competitivo em relação às propostas existentes sem sacrificar a expressividade dos algoritmos ou a capacidade de composição dos operadores. Nossos resultados mostram ainda que nenhum paradigma é melhor em todo cenário de aplicação e acreditamos que essa e outras de nossas descobertas podem guiar interessados em direção a sistemas de MPG mais otimizados no futuro.

Palavras-chave: Mineração de padrões em grafos. Sistemas distribuídos. Avaliação experimental.

Abstract

Graph Pattern Mining (GPM) refers to a class of problems involving the processing of subgraphs extracted from larger graphs. Applications to GPM algorithms include querying subgraphs with given properties of interest, identifying motif structures in biological networks, characterizing social media, among others. GPM algorithms are challenging to develop due to inherently subroutines that include non-trivial graph theory concepts and methods such as isomorphism. General-purpose GPM systems emerge as a solution to improve the user experience with such algorithms. However, general-purpose GPM systems fail in providing a consistent model that is simple to understand and qualified to express alternative algorithms for the same problem via different paradigms for subgraph enumeration, limiting the integration with modern data analytics pipelines. Furthermore, because GPM systems are so heterogeneous in terms of supported paradigms and computing architecture, existing experimental evaluations are unable to distinguish whether performance differences are best explained by algorithmic strategies or implementation details. In this work we propose a primitive-based model for GPM, a proof of concept distributed implementation of that model, and an extensive experimentation analysis of popular algorithmic paradigms used in GPM systems. We demonstrate empirically the effectiveness of our model by showing competitive performance against state-of-the-art systems without sacrificing the expressiveness of algorithms or the composability of operators. Our experimental results also show that no single paradigm is best for every application scenario, and we believe that our findings may guide practitioner towards more optimized GPM systems in the future.

Keywords: Graph pattern mining. Distributed systems. Experimental evaluation.

List of Figures

2.1 Example of input graph, subgraphs, and pattern. Vertex colors denote labels. A single edge label is illustrated in this Figure: solid black lines. 28
2.2 Example of input graph, subgraph, and equivalent codes. In the provided example it is sufficient to specify vertex ordering, as each vertex in the ordering induces new edges connecting it to previous vertices in the ordering. 29
2.3 Example of an aggregation over subgraphs of size 3 that counts the number of subgraphs per pattern. Only a few subgraphs of size 3 are shown. 32
2.4 Example of subgraph enumeration from the input graph in Figure 4.2. 33
2.5 GPM paradigms: pattern-oblivious (*POSE*) *vs.* pattern-aware (*PASE*). . . . 41

4.1 Extension primitive scheme. 49
4.2 Types of subgraph extension: the subgraph above (composed of vertices and edges in solid lines) 51
4.3 Example of input graph, subgraph, and equivalent codes. 53
4.4 Aggregation primitive scheme. 58
4.5 Filtering primitive scheme. 59
4.6 Algorithm design: pattern-oblivious (POSE) vs. pattern-aware (PASE). . . . 61

5.1 Execution environment. (a) The master communicates with workers for scheduling. (b) Workers execute the tasks assigned to them. 67
5.2 Breadth-first subgraph enumeration for induced subgraphs from the input graph in Figure 4.2. 69
5.3 Depth-first exploration for subgraph enumeration. In this example, the target is subgraphs with three vertices. 70
5.4 Subgraph enumerator abstraction. 70
5.5 Multi-step application design alternatives for FSM kernel. Our approach adopts a from-scratch computation to ensure a space-bounded processing. . . . 73
5.6 Enumeration cost of each step against the cumulative enumeration of previous steps for an exemplar GPM application - it suggests trading off memory for redundant computation can be effective. 74

5.7 Subgraph enumeration without any work balancing. The x-axis represents relative utilization of the worker, considering all available cores. The y-axis represents the timeline of application execution. In this case, CPU is not well utilized due to skewness: utilization drops very quickly within a few seconds of execution. 76
5.8 Work stealing happens (a)(c) internally among threads of the same worker or (b) externally among threads of different workers. To reduce network communication, we use remote work stealing only when no other local thread has work to share. 78
5.9 Generalized graph pattern mining framework. 81
5.10 Fractal initialization. 84
5.11 Fractal step scheduling. 85
5.12 Reliable downstream processing of subgraph aggregations 88
5.13 Fractal: Standard API 90
5.14 Motifs runtime on Mico-SL and Youtube-SL. 94
5.15 Performance on Triangles benchmark. GraphFrames ran out of memory for Orkut-SL. 95
5.16 Cliques runtime on Mico-SL and Youtube-SL. GraphFrames often ran out of memory. 96
5.17 FSM performance. Fractal's stateless characteristic allows competitive scalability with Scalemine. 96
5.18 Queries for pattern querying evaluation. 97
5.19 Fractal outperforms SEED in most configurations, except for those where SEED's execution plan leverages the overlapping substructures (e.g., q_7). 97
5.20 4-*CL* with and without dynamic work balancing. Good CPU utilization throughout the whole execution. Runtime: 73.8s. 100
5.21 Work stealing evaluation. Imbalance becomes evident with load balancing strategies disabled *(1.Disabled)*. Internal work stealing allows a good intra-worker load balancing at low communication cost *(2.Internal)*. External work stealing allows an efficient load balancing at a higher communication overhead *(3.External)*. Applying both strategies gives the best trade-off between load balancing and communication overhead *(4.Internal+External)*. Times of each step are noted on top of each chart. 102
5.22 COST analysis: number of cores that Fractal needs to reach state-of-the-art single-thread methods. 103
5.23 Triangles COST analysis. 104
5.24 Fractal scalability. 105
5.25 Worst case runtime overhead of keeping memory bounded with re-computation if redundant computation and load balancing come for free. 106

6.1 Taxonomy: categories represent problems with similar algorithm design requirements. 108
6.2 Extension primitive operations. The example assumes a vertex-oriented extension type. 110
6.3 Custom: algorithms with specialized extension methods. 110
6.4 Fractal: the subgraph enumerator API. 113
6.5 Mapping primitive: mapping subgraph codes. 115
6.6 Hybrid PASE+POSE via mapping primitive (`M`). 116
6.7 PASE+POSE: hybrid multi-paradigm algorithms. 117
6.8 Fractal: the subgraph mapping API. 118
6.9 Abstraction: graph filtering. 120
6.10 POSE+GF: effective pruning via graph filtering (GF). 120
6.11 Graph filtering example: selecting vertices having specific labels. 121
6.12 Fractal: the graph filtering API. 122
6.13 Pattern and label queries used to evaluate algorithms. 126
6.14 Throughput: Cliques (k-CL). 127
6.15 Throughput: Pattern querying (ρ-PQ). 127
6.16 Runtime: Frequent subgraph mining (k-FSM-α). 129
6.17 Throughput: Quasi cliques (k-QC-α). 130
6.18 Throughput: Query specialization (ρ-QS). 131
6.19 Throughput: Label search (k-LS-$\mathcal{L}$). 133
6.20 Number of Subgraphs per Pattern concerning label search results (Fig. 6.19): pattern-aware (PASE) is penalized by skewness of application steps. On Mico dataset (MI) this skewness is more expressive, resulting in larger drops of performance as subgraph size k increases. A similar behavior can be observed on Youtube dataset (YO) but at a lower scale. 134
6.21 Throughput: Minimal keyword search (k-MKS-$\mathcal{K}$). “not applicable” indicates that none of the algorithms produced output for that particular problem instance. 135

List of Tables

2.1 Basic notations. 25

4.1 Compatibility matrix among general-purpose extension types and methods. . . 57

5.1 Estimated memory demands - the number of induced subgraphs grows exponentially, and so the space required to store them. 68
5.2 Fractal evaluation: real-world datasets used in the experiments. 92
5.3 Motifs runtime on Mico-ML and Youtube-ML. 94
5.4 Memory per worker. 99

6.1 Summary of algorithms evaluated. *Categories* represent problems with similar requirements concerning algorithm design. *Algorithms* represent standard paradigms (POSE and PASE in Figure 4.6) and derived strategies for solution (Custom in Figure 6.3, PASE+POSE in Figure 6.7 , and POSE+GF in Figure 6.10). 124
6.2 Real-world datasets used in the experiments. 124

Contents

1 Introduction **15**
1.1 Challenges . . . 18
1.2 Thesis statement . . . 21
1.3 Objectives . . . 21
1.4 Contributions . . . 21
1.5 Organization . . . 23

2 Preliminaries **25**
2.1 Graph theory . . . 25
2.1.1 Graphs and subgraphs . . . 26
2.1.2 Patterns and isomorphism . . . 27
2.1.3 Subgraph code and automorphism . . . 29
2.2 Problem definition . . . 30
2.3 Graph pattern mining problems . . . 33
2.3.1 Motifs kernel (k-MC) . . . 33
2.3.2 Cliques kernel (k-CL) . . . 34
2.3.3 Pattern querying kernel (ρ-PQ) . . . 35
2.3.4 Frequent subgraph mining kernel (k-FSM-α) . . . 35
2.3.5 Quasi-cliques kernel (k-QC-α) . . . 36
2.3.6 Query specialization kernel (ρ-QS) . . . 37
2.3.7 Label search kernel (k-LS-$\mathcal{L}$) . . . 37
2.3.8 Minimal keyword search kernel (k-MKS-$\mathcal{K}$) . . . 38
2.4 Graph pattern mining paradigms . . . 38

3 Related Work **42**
3.1 Graph Analytics . . . 42
3.2 Specialized Graph Pattern Mining . . . 43
3.3 General-purpose Graph Pattern Mining . . . 43
3.4 Understanding subgraph enumeration computation. . . . 45

4 Modeling graph pattern mining applications **47**
4.1 Extension primitive . . . 48
4.1.1 Extension types: supporting multiple graph pattern mining paradigms 49

4.1.2 Extension methods: supporting multiple subgraph extension strategies . . . 52
4.1.2.1 General-purpose extension methods . . . 52
4.1.3 Assembling extension types and methods . . . 55
4.2 Aggregation primitive . . . 57
4.3 Filtering primitive . . . 58
4.4 Design of applications . . . 59
4.4.1 GPM algorithm design . . . 60
4.5 The advantages of primitive-based model . . . 64

5 Implementing graph pattern mining systems **66**
5.1 Efficient subgraph enumeration . . . 67
5.1.1 Depth-first subgraph enumeration . . . 69
5.1.2 From-scratch step execution . . . 72
5.1.3 Space bounded parallel and distributed subgraph processing . . . 75
5.2 Load balancing . . . 75
5.2.1 Hierarchical work stealing . . . 76
5.3 A generalized framework for static graph pattern mining . . . 80
5.3.1 Fractal: A General-Purpose Graph Pattern Mining System . . . 82
5.3.1.1 System initialization . . . 83
5.3.1.2 Scheduling and execution . . . 84
5.3.1.3 Implementing GPM primitives in Fractal . . . 85
5.3.1.4 Implementing work-stealing in Fractal . . . 86
5.3.1.5 Enabling reliable downstream processing in Spark . . . 87
5.3.1.6 Fractal programming interface . . . 88
5.4 Evaluation . . . 92
5.4.1 Fractal: Comparative Performance . . . 93
5.4.2 Fractal Drilldown . . . 98
5.4.2.1 Memory footprint analysis . . . 98
5.4.2.2 Hierarchical Work Stealing . . . 100
5.4.2.3 COST analysis . . . 101
5.4.2.4 Scalability . . . 103
5.5 Understanding limitations associated with design choices . . . 104

6 Consolidating graph pattern mining paradigms and abstractions **107**
6.1 Enabling customized extension methods . . . 109
6.1.1 Incorporating extension abstractions in Fractal . . . 112
6.2 Enabling multi-paradigm algorithms via mapping primitive . . . 113
6.2.1 Incorporating mapping primitive in Fractal . . . 117
6.3 Enabling effective pruning via graph filtering . . . 118

6.3.1 Incorporating filtering abstractions in Fractal 121
6.4 Evaluating GPM paradigms: consolidation and renewed bearing 122
6.4.1 Single-pattern algorithms 125
6.4.2 Multi-pattern algorithms with pattern-driven filter 128
6.4.3 Multi-pattern algorithms with label-driven filter 132

7 Final Remarks **136**
7.1 Limitations and Future work 137

References **139**

Chapter 1

Introduction

Graph pattern mining (GPM) refers to a class of problems marked by the processing of subgraphs extracted from larger graphs. The relevance of GPM computation is multidisciplinary, including applications such as motif extraction from biological networks [92, 57, 3], frequent subgraph mining [34], subgraph searching over semantic data (e.g., RDF) [33], social media network characterization [124, 70], community discovery [28, 9], periodic community discovery [104], temporal hotspot identification [135], identification of surprising dense subgraphs in social networks [54], link spam detection [79], financial fraud detection [52], recommendation systems [142], graph learning [90, 81], among others. We review next some of these exemplary applications for GPM computation.

Social networks. In social networks, small motifs (a.k.a. subgraph patterns) can be used to capture high-order relations between nodes in social networks [141]. Specifically, the authors are capable of improving the performance of user ranking in real-world networks by leveraging the motifs extracted. In the context of temporal networks, several applications leverage from dynamic graph pattern mining strategies. The evaluation of balance in social networks, i.e., the proportion of positive and negative relations in a group of individuals, can be estimated using simple cycles in temporal networks [41]. In terms of temporal data, high-order structures (motifs) also play an important role. Specifically, temporal motifs composed of consecutive events (directed edges) may reveal significant communication patterns that otherwise would remain undetected in a static setting. As an example, [71] shows strong evidences of temporal homophily in a large phone call dataset, i.e., similar individuals appear to have a tendency to participate in specific communication patterns beyond the expected. Other interesting properties studied in social networks through graph pattern mining algorithms include mining periodic communities (subgraphs) from networks of interactions [104], extraction of diversified subgraphs representing temporal hotspots in dynamic networks [135], identifying surprisingly subgraphs (those that one would not expect to be part of some network) [54], searching optimal local communities for a given vertex [24], and extracting antagonistic communities from social networks [40].

Biological networks. In biological field, subgraph enumeration has also been used to extract patterns in multi-layer brain networks, with practical applications to epilepsy identification [57]. Motifs counting can also be used for finding sets of proteins in PPI (protein-protein interaction) networks associated with a given disease [3] – the authors show that standard proximity metrics are insufficient for such tasks and furthermore, that high-order (motifs) structures can improve the quality results of disease identification. Other seminal work uses temporal subgraphs to identify feedback loops in neural networks [31].

Graph learning and recommendation systems. Graph pattern mining algorithms are being used in the context of recommendation systems [142], where the authors leverage motifs to rank users in a network using a motif-aware PageRank algorithm. Graph pattern mining techniques may also be used for predicting the evolution of dynamic networks [105]. In particular, [90] generalizes the link (and node) prediction task [81] to subgraph prediction task. For instance, in a citation network where nodes represent authors, venues, and topics, a subgraph prediction task could unveil high-order relationships that are likely to occur in the future: authors co-authoring papers in new venues or in new topics, for example. An expressive challenge in this context is to efficiently extract training data (subgraphs) and/or to generate subgraph embeddings for downstream learning tasks, both directly related to the scalability of subgraph enumeration systems in modern parallel/distributed architectures. This constitutes an exemplary scenario where a graph pattern mining task requires the enumeration of subgraph instances and not only a few summary statistics of patterns, as in motif counting [92] – count statistics – or frequent subgraph mining [34] – support statistics. Finally, Subgraph enumeration is also required as subroutine for generating graph-pattern association rules (GPARs) in the context of marketing and recommendation systems [37]. In a related scenario, graph pattern mining kernels such as maximal clique finding are used to support learning graph representations in machine learning pipelines [93] or local motif counts may be used to improve the quality of network embeddings [111].

Broader impact. Association rules is a widely known topic in data mining research. Specifically, extracting association rules [4] from temporal graphs can unveil important and timely inference power to dynamic systems. When association rules are composed of events defined by subgraph patterns, one can observe scenarios where an event (subgraph pattern in time) is triggered by a second event (other subgraph pattern in time) with high probability. For example, in communication networks, it may the case that "if a user u joins a group call of his co-workers, he *usually* texts another contact in 5 minutes" [95]. This case could be represented by an observed temporal clique inducing a single edge communication between two users on that clique. Naturally, an inevitable

step for constructing these association rules is the subgraph enumeration of events within the temporal graph, one of the main goals of this work. Other line of work uses subgraph enumeration strategies to detect fraud in financial transactions [52] or in malware detection [99] and thus, could benefit from an efficient subgraph enumeration system capable of providing fast and reliable analytic results.

Graph pattern mining algorithms tend to be complex to develop and to integrate into existing data analysis pipelines. Indeed, workloads for graph pattern mining must handle challenges such as the combinatorial explosion of subgraphs extracted from a larger graph, and non-trivial programming routines for preventing unnecessary enumeration cost (automorphisms) or for grouping subgraphs into equivalence classes (isomorphisms). Moreover, this complexity is only exacerbated if one considers the parallelization of such algorithms as the ideal goal for extracting adequate performance from these solutions. In this context, general-purpose graph pattern mining systems emerge as an alternative for programming and for maintaining GPM applications.

The space of existing general-purpose GPM systems is diverse. In one dimension, concerning the computing architecture and parallelization approaches, some systems are designed for multi-threaded or distributed settings [60, 30, 121, 12, 132], others for emerging GPU architectures [21, 19], and yet others for hardware accelerators [136, 22]. In a second dimension, two main alternative GPM paradigms are adopted by GPM systems and are responsible for ensuring an efficient search-space exploration of subgraphs: (i) the *pattern-aware* paradigm [60] in which subgraph enumeration is accomplished by matching candidate patterns (templates) against the input graph and (ii) the *pattern-oblivious* paradigm [121] in which no pattern information is given to guide subgraph enumeration. Such a range of existing implementations in multiple architectures spanning various algorithm paradigms leads to a lack of understanding on the source of the performance gains, and it also complicates the evaluation of new and existing optimizations.

General-purpose GPM systems provide many common routines from GPM algorithms and hide much complexity from the end-user, improving the overall usability and applicability of the system in solving domain-specific problems. A remarkable example is the subgraph enumeration engine included in most general-purpose GPM systems and responsible for obtaining subgraph candidates for downstream processing. The details of how subgraph extraction is implemented is of little importance while developing a data analysis pipeline that includes searching for specific subgraphs. On the other hand, GPM systems should not be limited to fixed strategies for subgraph enumeration and subgraph selection – generic implementations should be provided as default while optimized solutions should be made possible to advanced users. GPM systems are carefully designed to provide this adequate trade-off between programming productivity and execution efficiency. Naturally, abstractions for GPM computations play an important role in balancing these two sometimes conflicting features – they provide a high-level understanding of the

computation model to the end-user, and they also are easily ported to different systems using different languages on a plethora of existing architectures.

In this work our goals are to provide a standard model for general-purpose GPM that can be composable and portable, to design an efficient parallel and distributed implementation for this model that can be easily integrated into existing processing pipelines, and to provide a comprehensive performance study on the inherent trade-offs that GPM paradigms may exhibit. Specifically, we present solid solutions concerning modeling and implementing efficient general-purpose GPM solutions (Chapters 4 and 5) and a wide experimental evaluation of GPM paradigms and abstractions that may be of interest for future work (Chapter 6). Before getting into those details, in the remaining of this chapter we describe the challenges, the contributions, and the organization of this text.

1.1 Challenges

A proper problem modeling represents an important step in providing automated and consolidated systemic solutions for problems in a given context. The same applies to GPM computations. The first step most existing proposals disregard is towards a solid modeling strategy for representing general-purpose GPM applications. Existing models lack the simplicity, the composability, the integration, and the extensibility necessary to a top-level computation model, the same way SQL is a widely adopted standard for relational data. Moreover, because GPM processing admits modeling algorithms according to multiple paradigms (pattern-oblivious or pattern-aware), a proper model for general-purpose applications should also consider this

The next aspect of challenge concerns the proposal of effective general-purpose solutions in modern architectures. At its core, GPM methods perform subgraph enumeration. In this work we refer to *subgraph enumeration* as a routine for graphs responsible for extracting subgraphs from a *single larger input graph*. Subgraph enumeration may be computationally and storage intensive, where a tremendous amount of intermediate state can be generated even when running on small-scale networks (e.g., 5-10k nodes). This imposes challenges in terms of both runtime and memory performance. Moreover, the irregular topology presented in scale-free graphs makes GPM quite challenging regarding load balancing in parallel and distributed settings.

This complexity, has led to the development of distributed algorithms for specialized (domain-specific) GPM problems, such as frequent subgraph mining [1], motif counting [110], and clique counting [39], that do not generalize to other GPM problems. Systems such as Arabesque [121], and NScale [106] have emerged as first generation,

general-purpose solutions for GPM. While both potentially offer programming interfaces suitable for processing coarse-grained GPM applications, their computation models fail to support fine-grained interactive analysis. In fact, those systems adopt a BFS-style subgraph enumeration to balance the work at the end of each synchronization step, generating and shuffling a huge amount of intermediate state between workers, what leads to increased overhead at larger scales.

Next we list the specific challenges motivating this work:

- *Algorithm model for GPM* ($C1$). Existing general-purpose GPM systems often put all the effort into optimizing performance and providing programming language API, but most lack an underlying abstract model for expressing precise execution plans in different paradigms for subgraph enumeration. Such gap limits GPM solutions in the following aspects: (i) solutions tend to be too strict in terms of the GPM paradigm used for extracting subgraphs, which complicates the processing of reasoning about alternative approaches to solve the same problem; (ii) extensive work must incorporate new abstractions and features in an uncertain GPM model, which slows down extensibility and portability of solutions; and (iii) it is difficult to reason about performance and to understand application semantics in a context where program building blocks (operations and their purpose) are implicit and hard to track [100].

- *Efficiency, programming productivity, and integration of GPM systems* ($C2$). Enumeration of subgraphs is a computational intensive task that quickly consumes the computational resources available, even in a distributed setup with mid-size inputs. The performance of GPM applications are determined by how efficiently we are able to minimize resource idleness, and to overlap communication with computation.

  Part of such intensity comes from the memory demand of GPM workloads. Indeed, subgraph enumeration often explores an exponential space of candidate like any other combinatorial problem. In this scenario, the intermediate state of subgraph candidates can quickly become an issue. Existing data-parallel processing systems work over the abstraction of distributed collections, which is infeasible for applications dealing with huge amounts of intermediate data. The amount of intermediate data during subgraph enumeration grows exponentially, making memory a critical bottleneck. Moreover, the intense use of memory in garbage collected languages – which are widespread amongst popular stacks for data analytics – introduces performance overheads and unpredictability [42, 84, 98, 97].

  Other part of this computational intensity comes from the irregular nature of typical input to graph pattern mining algorithms. Graphs used in GPM tasks usually follow a power-law degree distribution (scale-free) [140], which generates highly irregular and imbalanced workloads that are hard to characterize apriori.

On top of all that, the design and implementation of GPM solutions are hard even for experts, especially for distributed architectures. In this context, programming productivity should be faced as a top-level design choice, with a simple and transparent interface that portrays typical and integrated pipelines for GPM analytics.

- *Evaluation of GPM paradigms* ($C3$). GPM algorithms built over general-purpose systems often include user-specific semantics, and multiple algorithms to a problem may exist. This means that our experimental methodology must accommodate both pattern-aware and pattern-oblivious paradigms within the same system implementation and underlying application model – which is non-trivial since existing GPM paradigms differ in how they build subgraph candidates during enumeration. To get around this issue, most existing GPM systems support only a single GPM paradigm (either pattern-aware or pattern-oblivious) [60, 121], and those that support both fail in providing means to combine paradigms and in understanding the impacts of the implementation details (programming language, architecture, optimizations) [20], which makes them not ideal for a systematic experimental evaluation since a direct comparison may not distinguish whether the performance differences are explained by the GPM paradigm or the implementation details.

  In order to identify the trade-offs between alternative GPM paradigms, one must consider a diverse set of applications and scenarios. Moreover, an experimental evaluation with these goals must not only report comparative performance measurements, instead it must be able to provide a comprehensive and insightful diagnostic on the possible variables responsible for the reported results. Existing works on general-purpose GPM provide only a narrow perspective of possible application scenarios [30, 121, 60, 21, 19] and lack a deep understanding on the sources of performance discrepancies making it difficult to understand when one paradigm may be preferred over the other [60].

1.2 Thesis statement

The thesis of this work is that GPM systems can greatly benefit from a strong, well-defined model for algorithms that is independent of implementation details such as system architecture, programming language, and parallelization strategies. Such model can improve the productivity for applications by simplifying the design of complex non-trivial algorithms and facilitate the design and implementation of scalable systems for GPM by allowing a more modularized view of the architecture. The existence of such model can also allow a more comprehensive and fair evaluation of existing GPM paradigms and provide the basis for the future of automated, cost-based GPM systems.

1.3 Objectives

Our goal with this work is to motivate, to model, to design, and to implement a unified platform for Graph Pattern Mining at scale. We list the following specific objectives for this work:

1. Propose a simple and expressive algorithm model for general-purpose GPM.
2. Present design and implementation of a GPM system that adopts the proposed model and that deals with system challenges concerning the efficiency and the programming productivity of GPM applications.
3. Present a wide evaluation study of existing GPM paradigms and application scenarios over a single framework model to both consolidate collective knowledge about GPM processing and to identify promising future work.

1.4 Contributions

In this work we propose effective models for general-purpose GPM computations that are capable of expressing different subgraph enumeration strategies, we validate our model with an efficient and scalable system for GPM that can be easily integrated into modern data-analytics pipelines, and we provide new perspectives on new abstractions

for GPM systems that may push the state-of-the-art into another level in terms of new optimization possibilities and performance gains. To this end, we develop the following contributions in this work:

1. *Primitive-based model for general-purpose GPM applications (Chapter 4).* We propose a formal description of general-purpose GPM applications, unveiling important building blocks for standardized application design. Also, a rigorous definition of GPM applications enables more consistent reasoning about system performance because bottlenecks become easily traceable from the set of operators (building blocks), allowing new emerging performance-driven abstractions and existing subgraph enumeration paradigms to be more effective.

2. *Proof of concept implementation and experiments (Chapter 5).* We provide a proof of concept implementation of the proposed model as an available[1] open-source general-purpose distributed and parallel GPM system called Fractal [30]. In addition to the implementation we present an extensive set of experiments on real-world datasets showing the effectiveness of our proposal. Fractal's design includes the following system contributions:

   - *Flexible, expressive and compositional API.* Fractal's API and programming model are designed from the ground up to be simple and to reflect fundamental GPM operations (denoted as primitives). In this work, we show how the API is flexible to compose a wide range of GPM applications interactively and efficiently using just a few lines of code. To the best of our knowledge no other system can tackle such a range of GPM applications while being user-friendly.
   - *Mitigating irregular memory demand.* The amount of intermediate state in GPM applications can grow exponentially and it is irregular, making it is hard to predict, being a significant source of overhead. The use of modern garbage collected languages – widely used in popular stacks for data analytics – compounds this issue leading to high unpredictability in performance [42, 84, 98, 97]. Fractal combines a depth-first subgraph exploration strategy with a "from scratch processing" paradigm to keep the memory requirements bounded by the size of the input. Our results show that it can improve performance and reliability of GPM applications: executions up to three orders of magnitude faster than comparable systems or specialized baselines, while leaving more memory for the user application.
   - *Adaptive load balancing.* GPM algorithms are irregular by nature and dependent on both input parameters and data characteristics. Balancing the load

[1] https://github.com/dccspeed/fractal

in such problems and minimizing communication overhead is central to performance efficiency. Fractal incorporates a novel hierarchical work stealing and communication mitigating strategy that is aware of task locality and reduces the communication overhead. The results reveal that our technique achieves a nearly-ideal load balancing in many scenarios.

3. *Experimental evaluation of GPM paradigms (Chapter 6).* We provide an extensive experimental evaluation of GPM workloads, including a wide range of application scenarios considering multiple algorithms and over real-world datasets. Our evaluation exposes the trade-offs between standard GPM paradigms (pattern-oblivious and pattern-aware) and show how these trade-offs may be exploited for choosing the most adequate paradigm for a given workload scenario. Our experimental setting and general-purpose modeling also expose key takeaways and opportunities for deploying existing and new optimization strategies for both existing and future GPM systems. Along the way we also provide a discussion on how to identify opportunities of optimization given the characteristics of specific workloads. To our knowledge this is the first work to digest such variety of use case scenarios for GPM workloads and to consider the inherent trade-offs among these algorithms and respective paradigms.

1.5 Organization

This work is organized as follows:

Chapter 2 – Preliminaries: In this chapter we provide concepts and definitions used throughout this work.

Chapter 3 – Related Work: In this chapter we review the literature of general-purpose GPM, specialized solutions, real-world applications, and characterization of related GPM workloads.

Chapter 4 – Modeling graph pattern mining applications: In this chapter we propose a modeling approach to general-purpose GPM and argue why this alternative may be effective towards flexibility and extensibility.

Chapter 5 – Implementing graph pattern mining systems: In this chapter we provide design and implementation of a distributed and parallel general-purpose GPM

system supported by the modeling proposed and we validate our proof of concept with an experimental evaluation.

Chapter 6 – Consolidating graph pattern mining paradigms and abstractions: In this chapter we provide the experimental study on GPM paradigms to expose their trade-offs and to indicate promising research directions.

Chapter 7 – Final Remarks: In this chapter we summarize our findings and discuss opportunities for future work.

Chapter 2

Preliminaries

In this chapter we describe preliminary definitions and concepts that are used in this text. In Section 2.1 we present the graph theory background related to graph pattern mining computations. Next, in Section 2.2, we provide a precise definition on the general problem studied in this work. Finally, in Section 2.3, we instantiate the general problem with real-world graph pattern mining problems. Such problems are referenced in examples and for evaluation throughout this work. Table 2.1 summarizes the notations used in this text.

Table 2.1: Basic notations.

Notation	**Description**
$V(G)$	Vertices of (sub)graph G.
$E(G)$	Edges of (sub)graph G.
$L(G)$	Labels present in (sub)graph G.
$L(v)$	Labels of vertex v.
$L(e)$	Labels of edge e.
$\rho(G)$	Pattern of (sub)graph G.
$\rho_c(G)$	Canonical pattern of (sub)graph G.
$\gamma(G)$	Code of (sub)graph G, a totally ordered representation of its vertices and edges.
$\gamma_c(G)$	Canonical code of (sub)graph G
$N(S)$	the set of neighbors of a subgraph S in G

2.1 Graph theory

In Section 2.1.1 we define the input graphs and subgraphs considered for static GPM processing. In Section 2.1.2 we define the subgraph isomorphism problem and pattern, recurring concepts in any GPM computation. Finally, in Section 2.1.3, we define the concept of subgraph codes, related to graph automorphism and important for subgraph

extraction from the input graph.

2.1.1 Graphs and subgraphs

Without loss of generality, we adopt in this work an input graph G with vertices and non-directed edges which may have multiple labels (Definition 1). We adopt this model for graphs for convenience, but it can be easily extended for directed graphs (an also edge-labeled graphs) by making labels carry out these additional properties.

Definition 1. (Graph) A graph G is represented by three sets, $V(G)$, $E(G)$ and $L(G)$ which are the sets of vertices, edges, labels of G and one map function L. Each edge $e = (v, u) \in E(G)$ connects a pair of vertices v and $u \in V(G)$. The edges are not directed and there are no self-loops in G. Formally, $(v_i, v_j) = (v_j, v_i)$ and $e = (v_i, v_i) \notin E(G)$. The labels of a vertex or an edge are defined according the function $L : V(G) \cup E(G) \rightarrow \mathcal{P}(L(G))$, where $\mathcal{P}$ is the power set of a set.

Definition 2. (Subgraph) Let G and S be graphs. We say that S is an subgraph of G iff $V(S) \subseteq V(G)$ and $E(S) \subseteq E(G)$. A subgraph is connected iff there is a path between each pair of vertices $u, v \in V(S)$.

The most fundamental routine in GPM algorithms concerns the processing of subgraphs extracted from a larger input graph G. According to Definition 2, a subgraph S is represented by a set of vertices and edges embedded in the input graph G. In this work, we are interested in *connected subgraphs*: there must be a path between any pair of nodes in $V(S)$ comprising edges in $E(S)$. Thus, if not otherwise specified, when we mention the word "subgraph" in this work we actually mean *connected subgraph*. In this work, we use *subgraph* (Def. 2) to refer to a subgraph *instance* in an input graph G.

Some GPM applications may also be interested in a more restrict definition for subgraphs that can be induced (i.e., completely represented) by its set of vertices. According to Definition 3, an induced subgraph is a subgraph such that its set of edges can be implicitly obtained from the input graph G: given a vertex set for a subgraph, its edge set comprises all existing edges from G among those vertices.

Definition 3. (Induced Subgraph) An induced subgraph S of G comprises a vertex set $V(S) = I_V$, where $I_V \subseteq V(G)$. The edge set comprises all the *induced* edges: $E(S) = \{(u, v) \mid (u, v) \in E(G), u \in I_V, v \in I_V\}$.

2.1.2 Patterns and isomorphism

Subgraphs extracted from a graph can be mapped to a naive representation of its structural and labeling information, referred simply as *pattern* (Definition 5). Note that subgraph patterns discard the identification of individual vertices and edges from the input graph and may be represented by a set of pattern edges (Definition 4). Each pattern edge describes the template for an edge, identified by a source index s of the edge together with a destination index d of the edge. Moreover, because we consider labeled graphs as input, each index is followed by its label sets l_s or l_d. Indexes in this context have nothing to do with the input graph identifiers for vertices and edges, it is only a reference for an arbitrary vertex in the template. In practice, a pattern $\rho(S)$ is a template for a subgraph S and, thus, a subgraph is an instance of its pattern.

Definition 4. (Pattern Edge) Given a graph G, a pattern edge is 5-tuple (s, d, l_s, l_d, l), where $s, d \in \mathbb{N}_0$ represent respectively the source and destination vertices of the edge, $l_s, l_d, l \subseteq L(G)$ represent respectively the set of labels of the source vertex, destination vertex, and edge.

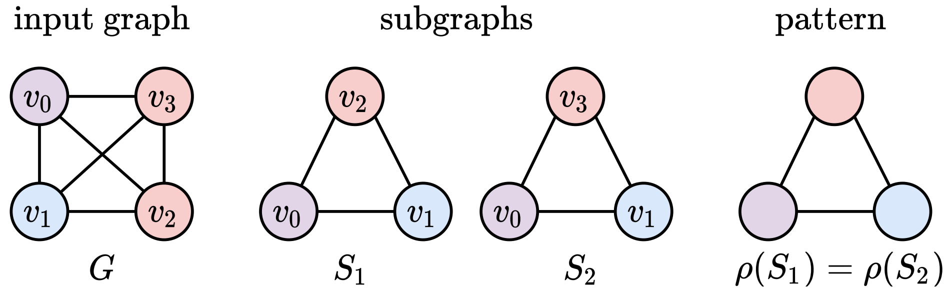
Definition 5. (Pattern) Given a subgraph S of graph G, the pattern of S is a set of pattern edges $\rho(S)$ such that $(u, v) \in E(S)$ iff $(\pi(u), \pi(v), L(u), L(v), L((u, v))) \in \rho(S)$, where π is an isomorphism between S and $\rho(S)$ (Definition 6). We say that a pattern is *induced* iff it is derived from an induced subgraph.

Figure 2.1 provides an example of these concepts. Colors represent vertex labels and numbers represent vertex unique identifiers. In this example, the input graph is a clique with 4 vertices, which contains two subgraphs $S_1 = \{0, 1, 2\}$ and $S_2 = \{0, 1, 3\}$, both sharing the same pattern: a labeled triangle. This labeled triangle in turn can be defined as $\rho_\triangle = \{(0, 1, 0, 1, 0), (0, 2, 0, 2, 0), (1, 2, 1, 2, 0)\}$[1], if we assume the "red" vertex represents label 0, the "blue" vertex represents the label 1, and the "purple" vertex represents the label 2. In this example we assume all edges have the same label (depicted as black lines). For instance, pattern edge $(0, 1, 0, 1, 0)$ is described as "vertex in position 0, which has label 0, is connected to vertex in position 1, which has label 1, through an edge that has label 0 (i.e. a single label represented by solid black lines)". Although we do not consider in this work application scenarios where edge-labeled subgraphs are mined, we provide this more generic definition to highlight that this is supported.

Different patterns extracted from G may exhibit the same structural template and labeling information. We say that such subgraphs belong to the same equivalence class and that they are *isomorphic* to each other. Graph isomorphism (Definition 6) is

[1] abuse of notation: we omit the *set notation* for labels whenever the set is a singleton (unit set)

Figure 2.1: Example of input graph, subgraphs, and pattern. Vertex colors denote labels. A single edge label is illustrated in this Figure: solid black lines.



Source: Made by the author.

the problem of verifying whether two (sub)graphs have an identical structure (topology), being fundamental to a variety of GPM applications such as motif counting, frequent pattern mining and graph matching. Given a set of (sub)graphs $\mathbb{S} = \{S_i, S_2, \ldots, S_n\}$, the isomorphism relation divides $\mathbb{S}$ into equivalence classes, where each class contains graphs that are isomorphic among themselves.

Definition 6. (Isomorphism) Two (sub)graphs G and H are isomorphic iff there is a bijective function π: $V(G) \rightarrow V(H)$ such that there is an edge $(v_i, v_j) \in E(G)$ iff $(\pi(v_i), \pi(v_j)) \in E(H)$.

Therefore, for our purposes, the definition of pattern is useful but imprecise. Consider again the triangle pattern in Figure 2.1. An alternative representation for this pattern is $\rho'_\triangle = \{(0, 1, 1, 0, 0), (0, 2, 1, 2, 0), (1, 2, 0, 2, 0)\}$, if we assume the labels have the same coloring as before but the "blue" vertex is at position 0, the "red" vertex is at position 1, and the "purple" vertex is at position 2. Such ambiguity can be a problem when comparing patterns: two representation are indeed different patterns or they are two alternatives for defining the same pattern? To handle the issues that arise with this question, we need a stronger definition for patterns: the **canonical patterns**.

In fact, two (sub)graphs G and H in the same equivalence class have the same *canonical pattern*, a common and unique representation for each pattern (Definition 7). Many canonical labeling algorithms exist for mapping a subgraph to its canonical pattern [73, 56, 133, 14]. In this work, we adopt Bliss algorithm [64] to determine the canonical labeling of a labeled (sub)graph S, which is given by the function $\rho_c(S)$. Basically, the canonical labelling is a string that represents the pattern of a given (sub)graph through the ordering of its edges. This is a popular and an efficient algorithm to perform isomorphic checks (i.e., comparison of strings) between (sub)graphs.

Definition 7. (Canonical pattern) A pattern $\rho_c(S)$ of subgraph S is canonical iff its pattern edges are ordered according to some *canonical labeling* algorithm c.

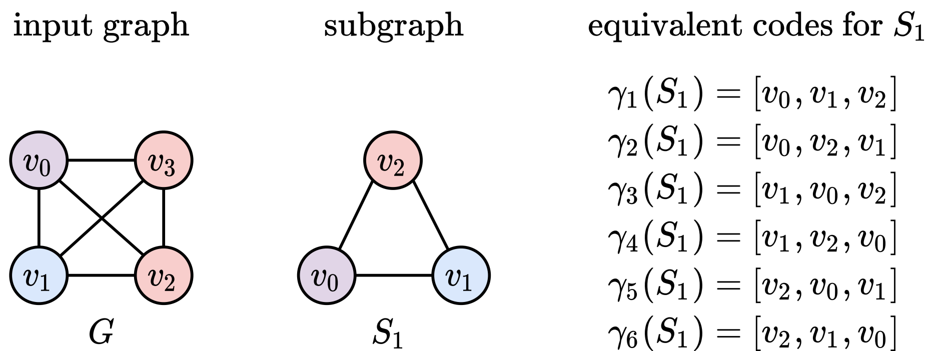
2.1.3 Subgraph code and automorphism

Definition 8. (Subgraph code) A subgraph code of a subgraph G, denoted by $\gamma(G)$, is a totally ordered set of vertices and edges. A subgraph code represents an order in which each subgraph may be extracted from the input graph.

The enumeration of subgraph instances (not patterns) from a graph is also related to the concept of isomorphism. Specifically, subgraphs are extracted by the enumeration of connected vertices and edges from G in a specific order, incrementally. Therefore, any permutation of vertices and edges represents an enumeration ordered code for the *same* subgraph instance in G. We refer to these codes as subgraph codes (Definition 8). We say that these equivalent orders representing the same subgraph are automorphic to each other – i.e. they represent isomorphisms from the subgraph to itself.

Figure 2.2 illustrates the issue of multiple subgraph codes that may arise from subgraph extraction. For this reason, GPM systems strict themselves to enumerating only a single *canonical representative code*[2] for each subgraph to prevent redundant and unnecessary work. Many *canonical ordering algorithms* used to ensure that the enumeration process outputs only a single subgraph code per subgraph exist. The algorithms adopted in this work to accomplish this task are presented and discussed in Section 4.1.2 because, as we may see, they are dependent and closely related to the GPM paradigm used.

Figure 2.2: Example of input graph, subgraph, and equivalent codes. In the provided example it is sufficient to specify vertex ordering, as each vertex in the ordering induces new edges connecting it to previous vertices in the ordering.



Source: Made by the author.

[2] not to be confused with *canonical pattern*

2.2 Problem definition

In this section we present the concepts needed to properly define a general-purpose graph pattern mining problem, the problem targeted in this work. Let G be input graph and $\mathcal{S} = \{S_1, \ldots, S_n\}$ be the set all of distinct[3] subgraphs in G. We define next the class of problems solved by general-purpose GPM systems. The first definition concerns the enumeration of subgraphs of interest according to a given size and predicate. We refer to this problem as the conditional subgraph enumeration problem (Definition 9). This predicate can be defined over any property that may be obtained from subgraphs in G. For example, to cite a few, predicates may refer to the number of edges of a subgraph, the labels of the subgraph, the pattern of the subgraph, or some density measure target for the subgraph.

Definition 9. (Conditional subgraph enumeration problem) Given a graph G, a subgraph size k, and a subgraph predicate C. The conditional subgraph enumeration problem seeks to list connected subgraphs $\mathcal{S}' = \{S \in \mathcal{S} \mid \textsc{size}(S) = k \text{ and } C(S)\}$, such that: $\textsc{size}(S) = |V(S)|$ iff the problem seeks *induced subgraphs* (Definition 3); $\textsc{size}(S) = |E(S)|$ iff the problem seeks general subgraphs (Definition 2); and $C(S)$ is satisfied.

Another definition we need to understand the space of problems in general-purpose GPM is the concept of *subgraph aggregation* (Definition 11). This is important since the space of subgraph candidates extracted from the input graph grows exponentially and thus, it becomes infeasible generate an output that large. Fortunately, graph pattern mining problems usually seek for summarizations or aggregations over this exponential output. Given a set of subgraphs $\mathcal{S}$, a subgraph aggregation performs a summarization of these subgraphs according to input functions of mapping and reduction. We are given three functions: $(g : \mathcal{S} \rightarrow K)$ a mapping function to obtain keys from subgraphs, $(h : \mathcal{S} \rightarrow V)$ a mapping function to obtain values from subgraphs, and $(r : V \times V \rightarrow V)$ a reduction function to reduce two values into one. A subgraph aggregation in this context over a set of subgraphs $\mathcal{S}$ first obtain key/value pairs using functions g and h and then, reduce the values sharing the same key using the reduction function r. This execution paradigm is known as Map/Reduce [27]. In fact, the reduction phase is a set reduction (Definition 10) over the set of subgraphs. Examples of reductions include sums over sets, averaging, counting by key, among others.

Definition 10. (Set reduction) Given a set $A \neq \emptyset$ and an associative and commutative reduction function $r : (A \times A) \rightarrow A$, a set reduction over A given r, denoted as $\mathcal{R}(A, r)$, is defined recursively:

[3] $\forall S_i, S_j \in \mathcal{S}, S_i$ is not automorphic to S_j

$$\mathcal{R}(A,r) = \begin{cases} a & \text{if } A = \{a\} \\ \mathcal{R}(A',r) & \text{if } a,b \in A,\ a \neq b,\ c = r(a,b), \text{ and } A' = (A \setminus \{a,b\}) \cup \{c\} \end{cases}$$

Definition 11. (Subgraph aggregation) Given a set of subgraphs $\mathcal{S} = \{S_1, S_2, \cdots, S_n\}$, a user-defined function $g : \mathcal{S} \rightarrow K$ mapping subgraphs into arbitrary keys in K, a user-defined function $h : \mathcal{S} \rightarrow V$ mapping subgraphs into arbitrary values in V, and a user-defined reduction function $r : (V \times V) \rightarrow V$, the subgraph aggregation over $\mathcal{S}$ given g, h and r, denoted as $\mathcal{A}(\mathcal{S}, g, h, r)$, is defined as $\mathcal{A}(\mathcal{S}, g, h, r) = \{(k_i, v_i) \mid \exists S_j \in \mathcal{S}, g(S_j) = k_i, v_i = \mathcal{R}(\mathcal{S}', r), \mathcal{S}' = \{h(S_j) \mid g(S_j) = k_i\}\}$.

Figure 2.3 shows an example of aggregation over a set of subgraphs with 3 vertices. Informally, this aggregation obtains the frequencies of patterns, i.e., the number of subgraphs of each pattern. We describe this operation next, considering the definitions of subgraphs (Definition 2), patterns (Definition 4), and subgraph aggregation (Definition 11). The result of such aggregation is a mapping composed of tuples of patterns (ρ_1 and ρ_2) and their respective frequencies (2 and 3):

$\mathcal{A}(\mathcal{S}, g, h, r) = \{(\rho_1, 3), (\rho_2, 2)\}$, where:

$\rho_1 = \{(0,1,0,0,0), (0,2,0,0,0)\}$

$\rho_2 = \{(0,1,0,0,0), (0,2,0,0,0), (1,2,0,0,0)\}$

$\mathcal{S} = \{S_1(\{v_0, v_1, v_3\}, \{e_0, e_2\}), S_2(\{v_0, v_1, v_2\}, \{e_0, e_1, e_6\}), S_3(\{v_0, v_1, v_4\}, \{e_0, e_3, e_7\}), S_4(\{v_0, v_1, v_5\}, \{e_0, e_4\}), S_5(\{v_0, v_1, v_6\}, \{e_0, e_5\})\}$

$g:\ S_i \rightarrow \mathit{pattern}(S_i)$ // key is the canonical pattern of S_i

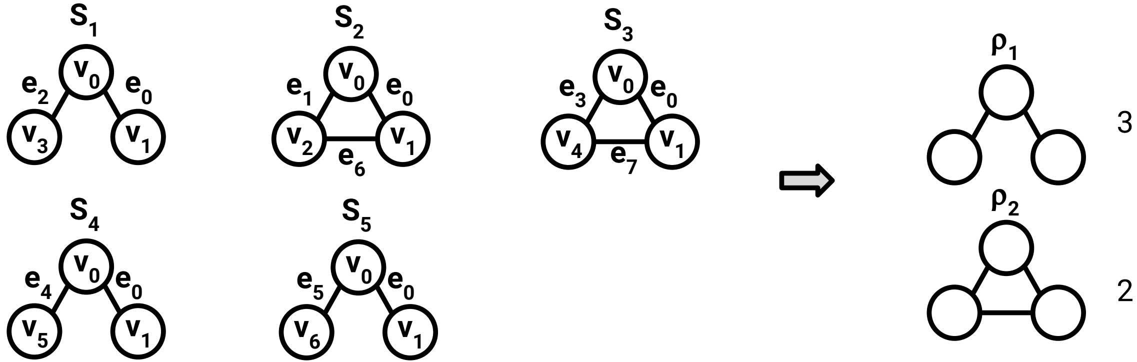
$h:\ S_i \rightarrow 1$ // value is 1

$r:\ (a,b) \rightarrow a + b$ // sum reduction

Definition 12 formalizes the general problem of this work: *subgraph aggregation problem*. Specifically, the subgraph aggregation problem combines conditional subgraph enumeration with subgraph aggregation to produce the results. The following steps characterize this problem:

1. *Subgraph enumeration* to produce subgraph candidates $\mathcal{S}$
2. *Subgraph pruning* given a predicate C that each subgraph in $\mathcal{S}$ should satisfy. The result of this step is a subset of the original subgraph candidates: $\mathcal{S}' \subseteq \mathcal{S}$.
3. *Subgraph aggregation* over $\mathcal{S}'$, generating the final results.

Figure 2.3: Example of an aggregation over subgraphs of size 3 that counts the number of subgraphs per pattern. Only a few subgraphs of size 3 are shown.



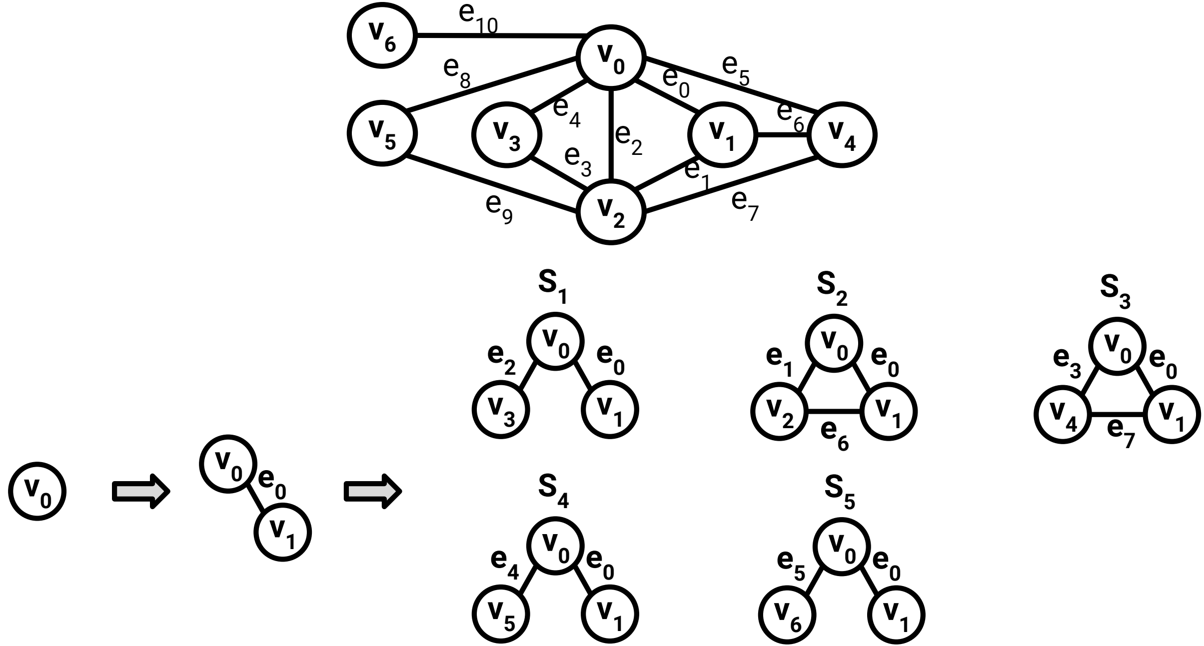
Source: Made by the author.

Definition 12. (Subgraph aggregation problem) Given a graph G, a subgraph size k, a subgraph predicate C, a user-defined function $g : \mathcal{S} \to K$ mapping subgraphs into arbitrary keys in K, a user-defined function $h : \mathcal{S} \to V$ mapping subgraphs into arbitrary values in V, and a user-defined reduction function $r : (V \times V) \to V$, the subgraph aggregation problem seeks to obtain a subgraph aggregation $\mathcal{A}(\mathcal{S}', g, h, r)$ (Definition 11) over a set $\mathcal{S}'$ of subgraphs enumerated from G that satisfy predicate C: $\mathcal{S}' = \{S_j \in \mathcal{S} \mid \textsc{size}(S_j) = k \text{ and } C(S_j)\}$ (Definition 9).

To give an idea of what an instance of this problem may look like, see Figure 2.4. This figure represents the enumeration and pruning steps of the subgraphs aggregated in Figure 2.3. That aggregation is already defined and thus, step 3 is complete. Steps 1 and 2 are implicitly defined in this figure for subgraphs from the graph of Figure 4.2 starting with vertex v_0. In particular, this instance represents the subgraph enumeration over the input graph with 3 vertices. Thus, we apply the aggregation illustrated in Figure 2.3 over this reduced set of subgraphs. We define this problem instance as follows:

- Graph $G(V, E)$ from Figure 4.2, where $V = \{v_0, v_1, \cdots, v_6\}$ and $E = \{e_0 = (v_0, v_1), e_1 = (v_1, v_2), e_2 = (v_0, v_2), e_3 = (v_2, v_3), e_4 = (v_0, v_3), e_5 = (v_0, v_4), e_6 = (v_1, v_4), e_7 = (v_2, v_4), e_8 = (v_0, v_5), e_9 = (v_2, v_5), e_{10} = (v_0, v_6)\}$;
- Subgraph size $k = 3$, where k represents the number of vertices (induced subgraphs);
- Predicate $C(S)$ is satisfied for all $S \in \mathcal{S}$
- Mapping function $(g : \mathcal{S} \to \mathcal{P}) = S \to \rho(S)$, where $S \in \mathcal{S}$;
- Mapping function $(h : \mathcal{S} \to \mathbb{N}) = S \to 1$, where $S \in \mathcal{S}$;
- Reduction function $(r : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}) = (a, b) \to a + b$, where $S \in \mathcal{S}$.

Figure 2.4: Example of subgraph enumeration from the input graph in Figure 4.2.



Source: Made by the author.

2.3 Graph pattern mining problems

Some graph pattern mining tasks are so fundamental that we may define them as GPM *kernels*, because they implement core operation analysis over graphs and they are often seen in several applications [92, 34, 5]. Next, we review popular GPM kernels studied in the literature, instantiated as subgraph aggregation problems (Definition 12), and used as workload in this work.

2.3.1 Motifs kernel (k-MC)

A *motif* P is defined as a connected and induced subgraph pattern in an input graph G. The goal is to count frequencies of all motifs (patterns) having k vertices. This kernel usually ignores the labels in G and it is widely used in bioinformatics [102, 92]. Specifically, let $G(V, E)$ be an input graph, $\mathcal{S} = \{S_1, S_2, \cdots, S_n\}$ be the set of induced subgraphs of G, S be an arbitrary induced subgraph in $\mathcal{S}$ ($S \in \mathcal{S}$), and k the number of vertices targeted for the subgraphs, the following items define an instance of the motifs kernel:

- Induced subgraphs with size k;
- Predicate $C(S)$ is satisfied for all $S \in \mathcal{S}$;
- Mapping function $(g : \mathcal{S} \rightarrow \mathcal{P}) = S \rightarrow \rho(S)$;
- Mapping function $(h : \mathcal{S} \rightarrow \mathbb{N}) = S \rightarrow 1$;
- Reduction function $(r : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}) = (a, b) \rightarrow a + b$.

The motifs kernel has only one constraint that limits the size of induced subgraphs to k. The mapping function g extracts the key for aggregation from the subgraphs, representing the canonical patterns obtained. The mapping function h extracts the value function from a subgraph and in this case it is a constant 1 for counting. Finally, the reduction function r defines the associative operation of summing the partial counts of the same pattern. Figure 2.3 illustrates the aggregation part in the motifs kernel.

2.3.2 Cliques kernel (k-CL)

A k-clique is a complete subgraph having k nodes in an input graph. In this case, only the topology of the subgraphs is considered. The cliques kernel is used in [28, 25]. Formally, let $G(V, E)$ be the input graph, k be the number of vertices in a clique, $\mathcal{S}$ be the set of induced subgraphs of G, S be an arbitrary induced subgraph ($S \in \mathcal{S}$), the following items define the k-cliques kernel:

- Subgraphs (also induced subgraphs) with size k;
- Predicate $C(S) \iff \forall u, v \in V(S), (u, v) \in E(G)$;
- Mapping function $(g : \mathcal{S} \rightarrow \{cliques\}) = S \rightarrow cliques$;
- Mapping function $(h : \mathcal{S} \rightarrow \mathbb{N}) = S \rightarrow 1$;
- Reduction function $(r : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N}) = (a, b) \rightarrow a + b$.

In this kernel, we have two constraints: valid subgraphs have k vertices and each one of them is connected to $k - 1$ other vertices, yielding a total of $\frac{k(k-1)}{2}$ edges. Because we are interested in counting a single pattern (i.e. the k-clique pattern), mapping key function g returns the *cliques* always. The other mapping function h and the reduction function r represent the sum over all valid subgraphs. The result of this kernel is a mapping with a single entry representing the number of k-cliques, denoted by m: $\{(cliques, m)\}$. A particular instance of the cliques kernel is the *triangle kernel*, i.e., cliques with 3 vertices.

2.3.3 Pattern querying kernel (ρ-PQ)

Querying a subgraph pattern is maybe the naivest GPM application known. The task is to list and count all the subgraphs in an input graph G that are isomorphic to a user-defined pattern p. The following items describe the subgraph querying kernel in terms of the subgraph aggregation problem:

- Subgraphs with size $k = |E(p)|$
- Predicate $C(S) \iff \rho(S) = p$
- Mapping function $(g : \mathcal{S} \to \mathcal{P}) = S \to P$.
- Mapping function $(h : \mathcal{S} \to \mathbb{N}) = S \to 1$.
- Reduction function $(r : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}) = (a, b) \to a + b$.

Valid subgraphs have exactly the same number of vertices as p, the same number of edges as p, and have the same template as p. The output of this kernel is a mapping with only one entry, representing the number of subgraphs m isomorphic to pattern p in G: $\{(p, m)\}$.

2.3.4 Frequent subgraph mining kernel (k-FSM-α)

A Frequent Subgraph Mining (FSM) kernel seeks to obtain all frequent subgraph patterns from a labeled input graph G. A pattern P is frequent if it has a support $s(P) \in \mathbb{S}$ above a threshold α, *i.e.*, if $s(P) \geq \alpha$. In particular, $s(P)$ is calculated based on the set of isomorphisms of the extracted subgraphs. We adopt the *minimum image-based support* [16] as the support function $s(\cdot)$ to leverage the anti-monotonic property: larger frequent patterns can only be obtained from smaller also frequent patterns. Because of this hierarchical behavior, the FSM kernel is defined as a chain of conditional subgraph enumeration problems, where the output of a computation level (the frequent patterns) serves as input to the next computation level.

Specifically, to compute the frequent patterns with k edges, we split this execution in three steps: (1) computation of frequent patterns with 1 edge; (2) computation of frequent patterns with 2 edges from frequent patterns of size 1; and (3) computation of frequent patterns with 3 edges from frequent patterns of size 2. Nevertheless, the following items describe each step $i = 1, 2, \cdots, k$ of the FSM kernel. Let $G(V, E)$ be the

input graph, $\mathcal{S}$ be the set of subgraphs with i edges of G, i be the current FSM step, and $\mathcal{F}_{i-1}$ be the set of frequent patterns with $i-1$ edges[4]:

- Subgraphs with size k (i.e., number of edges);
- Predicate $C(S) \iff \rho_c(S) \in \mathcal{F}_{i-1}$;
- Mapping function $(g : \mathcal{S} \to \mathcal{P}) = S \to \rho_c(S)$;
- Mapping function $(h : \mathcal{S} \to \mathbb{S}) = S \to s(S)$;
- Reduction function $(r : (\mathbb{S} \times \mathbb{S}) \to \mathbb{S}) = (a, b) \to a + b$[5].

In this case, step i generates valid subgraphs with i edges that can be obtained from frequent patterns ($S' \in \mathcal{F}_{i-1}$). The mapping key function extracts the pattern of the subgraph, the mapping value function extracts the support of the subgraph. Finally, the reduction function combines the supports of the same pattern using an associative "+" operator. The output of step i is $\mathcal{A}(\mathcal{S}, g, h, r)$, in which we derive the input for step $i+1$ representing only the mappings spanning frequent patterns: $\mathcal{F}_{i-1} = \{(\rho, s) \in \mathcal{A}(\mathcal{S}, g, h, r)$ such as $s \geq \alpha\}$.

2.3.5 Quasi-cliques kernel (k-QC-α)

Dense subgraph extraction can assist in fraud detection for social networks [55], in unveiling structural correlations for attributed graphs [115], among others. A α-quasi-clique of size k is a subgraph that has k nodes and every vertex of the subgraph has a degree of at least $\lceil \alpha * (|V(S)| - 1) \rceil$, i.e., each vertex is connected to a fraction of the vertices in the subgraph. The problem seeks to list and count α-quasi-cliques in a graph:

- Subgraphs with k vertices;
- Predicate $C(S) \iff \forall u \in V(S), d_S(u) \geq \lceil \alpha * (|V(S)| - 1) \rceil$
- Mapping function $(g : \mathcal{S} \to \{qc\}) = S \to qc$;
- Mapping function $(h : \mathcal{S} \to \mathbb{N}) = S \to 1$;
- Reduction function $(r : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}) = (a, b) \to a + b$.

[4] if $i = 1$, then $\mathcal{F}_{i-1} = \emptyset$

[5] operator "+" must be defined over set of supports $\mathbb{S}$

2.3.6 Query specialization kernel (ρ-QS)

Given a pattern query ρ the goal of query specialization [94, 137] is to unveil new queries that are specializations of ρ, i.e., larger queries containing ρ. The major procedure in unveiling query specializations is to list and to count subgraphs that are isomorphic to specializations of query ρ. This kernel is used in the context of query recommendation and for single graph mining instead of graph databases. Indeed, the latter setting can be viewed as a particular case of the former and consequently, more challenging [62]. For simplicity, we consider specializations containing pattern ρ and an additional edge:

- Subgraph with $k = |E(\rho)| + 1$ edges;
- Predicate $C(S) \iff p \subset \rho(S)$;
- Mapping function $(g : \mathcal{S} \to \{qs\}) = S \to qs$;
- Mapping function $(h : \mathcal{S} \to \mathbb{N}) = S \to 1$;
- Reduction function $(r : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}) = (a, b) \to a + b$.

2.3.7 Label search kernel (k-LS-$\mathcal{L}$)

Graph databases are often represented by entities (vertices) that are related among themselves (edges). In this context, vertices may carry labeled semantics representing roles or types in the database schema. The goal of label-based subgraph search is to extract relevant induced subgraphs from a larger input graph according to labels of interest $\mathcal{L}$. The following items define this kernel in terms of the subgraph aggregation problem:

- Induced subgraph with size k;
- Predicate $C(S) \iff \forall u \in V(S)[L(u) \subseteq \mathcal{L}]$;
- Mapping function $(g : \mathcal{S} \to \{ls\}) = S \to ls$;
- Mapping function $(h : \mathcal{S} \to \mathbb{N}) = S \to 1$;
- Reduction function $(r : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}) = (a, b) \to a + b$.

Valid subgraphs for this kernel are induced, must have a given number k of vertices, and each vertex must have only labels within a label set $\mathcal{L}$. While this kernel may seem simple it is widely used in NoSQL graph databases (e.g. Neo4j) for exploratory analysis in machine learning pipelines.

2.3.8 Minimal keyword search kernel (k-*MKS*-$\mathcal{K}$)

Knowledge bases are often represented as triples of subject-predicate-objects (SPO). A common strategy to model those kinds of relationships is through graphs, where subjects and objects are the vertices and the predicates represent the edges. Moreover, for each vertex or edge we associate a set of keywords (or property) that characterize the semantic of that entity. In this setting, we consider the problem of *keyword search*. Given a knowledge graph and a keyword query represented as a set of keywords, our goal is to find the subsets of triples (or subgraphs) that best answer this query. We consider the subtask of retrieving subgraph candidates for the query.

Given an attributed graph G with keywords as labels on vertices, a subgraph size k, and a query represented by a set of keywords (labels) $\mathcal{K} = \{w_1, \ldots, w_C\}$, the task is to retrieve subgraphs in G as follow. A subgraph is retrieved if each keyword is covered by some vertex in the subgraph [12, 127]. In this case vertices not covering any keyword are allowed in the subgraphs of interest as long as they are strictly necessary to maintain the subgraph connected (minimal). Let $\mathcal{S}$ be the set of subgraphs of G, $S \in \mathcal{S}$ be an arbitrary subgraph of G. The following items define this kernel in terms of the subgraph aggregation problem:

- Induced subgraph with size k;
- Predicate $C(S) \iff \mathcal{K} \subseteq L(S),\ \forall u \in V(S)[L(u) \notin \mathcal{K} \implies V(S) \setminus \{u\}$ is disconnected]
- Mapping function $(g : \mathcal{S} \to \{kws\}) = S \to kws$;
- Mapping function $(h : \mathcal{S} \to \mathbb{N}) = S \to 1$;
- Reduction function $(r : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}) = (a, b) \to a + b$.

2.4 Graph pattern mining paradigms

A fundamental routine for graph pattern mining applications is subgraph listing or extraction. Let G be a graph and k be a subgraph size [6], our goal is to enumerate all subgraphs of size k in G satisfying a given predicate C, which may be applied to the subgraph itself or its pattern. Next we define two well-known alternatives for this task.

[6] number of vertices for induced subgraphs, number of edges otherwise

Definition 13. (Pattern-aware subgraph enumeration) Let C be a predicate representing a set of conditions (or properties) a subgraph in G must exhibit in order to be of interest and let G be a graph. A pattern-aware subgraph enumeration (*PASE*) finds subgraphs in G by: (i) obtaining a set of patterns of interest P that does not violate predicate C; and (ii) enumerating subgraphs in G isomorphic to each $p \in P$.

Let $\mathcal{P} = \{P_1, P_2, \cdots, P_m\}$ be the set of all canonical patterns in G, a pattern-aware subgraph enumeration (Definition 13) explores subgraphs isomorphic to a given set of patterns, and uses such given information to guide the extension of connected subgraphs. Algorithm 1 describes the process in high-level details. The method starts by selecting a set of patterns that may generate subgraphs of interest in G, i.e., patterns that contain k vertices and that satisfy the predicate C (line 1). Next, for each selected pattern, the input graph is matched against it producing isomorphic subgraphs for processing (lines 2-5). The matching of each selected pattern works vertex-by-vertex by recursive calls of: (i) selecting the next vertex to match (to add); (ii) determining pattern edges that connect the current subgraph to the next vertex to match; (iii) retrieving from G the set of vertices that satisfy the connectivity pattern represented by those edges; (iv) removing from those vertices any candidate not satisfying the predicate C or that produces a non-canonical subgraph code (details on the canonical ordering algorithm for a pattern-aware subgraph exploration are presented in Section 4.1.2); (vi) and finally including valid candidates to the current subgraph for matching subsequent vertices.

Algorithm 1 PATTERN-AWARE-SENUM(G, k, C)

```
1:  P' ← {P ∈ P | |V(P)| = k and C(P)}
2:  for P in P' do
3:      for u ∈ V(G) do
4:          for S in MATCH({u}, P) do
5:              AGGREGATE(S)
6:  procedure MATCH(S, P)
7:      if |V(S)| = k then
8:          emit S
9:      else
10:         v_P ← NEXT-VERTEX-TO-MATCH(S, P)
11:         E_P ← {(s, l_s, d, l_d, l) ∈ P | d = v_P}
12:         V' ← {v ∈ ⋃_{u∈V(S)} N(v) | γ(S ∪ {v}) is canonical,  C(S ∪ {v}), and
                                         ρ(S ∪ {v}) = ρ(S) ∪ E_P}
13:         for v ∈ V' do
14:             MATCH(S ∪ {v}, P)
```

Definition 14. (Pattern-oblivious subgraph enumeration) Let C be a predicate representing a set of conditions (or properties) a subgraph in G must exhibit in order to be of interest and let G be a graph. A pattern-oblivious subgraph enumeration (*POSE*) finds subgraphs in G directly using Algorithm 2.

On the other hand, a pattern-oblivious subgraph enumeration (Definition 14) represents the class of enumeration strategies that are not pattern-aware. In such exploration category, canonical subgraph codes are visited but without the help of any reference pattern to guide the subgraph growth. Algorithm 2 describes the overall method in high-level details. This paradigm works by adding valid words ($W(G)$: vertices or edges) connected to the current subgraph until it reaches size k. Subgraph extensions are valid if they generate canonical subgraph codes (details on the canonical ordering algorithm for *POSE* are presented in Section 4.1.2) and satisfy the given subgraph predicate (line 8). Note that in this case, no information about which patterns are being extracted from the input graph is given whatsoever.

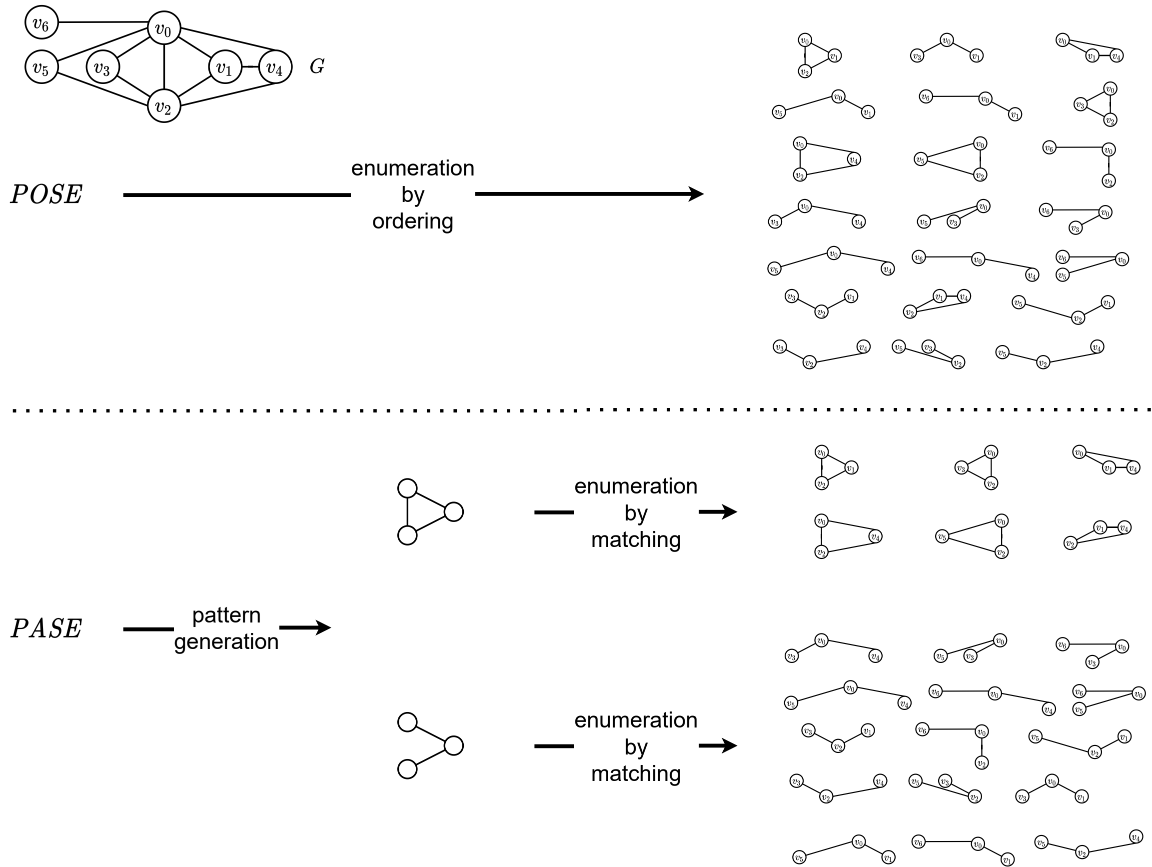
Algorithm 2 PATTERN-OBLIVIOUS-SENUM(G, k, C)

```
1:  for w in W(G) do
2:      for S in EXPLORE({w}) do
3:          AGGREGATE(S)
4:  procedure EXPLORE(S)
5:      if SIZE(S) = k then
6:          emit S
7:      else
8:          W ← {w ∈ N(S) | γ(S ∪ {w}) is canonical and C(S ∪ {w})}
9:          for w in W do
10:             EXPLORE(S ∪ {w})
```

Figure 2.5 illustrates the difference between both subgraph enumeration frameworks. While *POSE* enumerates subgraph candidates directly by having some partial order among subgraphs of different patterns, *PASE* includes a pattern generation step where patterns of interest are first obtained and each are matched against the input graph. We highlight that the output is equivalent since the same subgraph candidates are generated regardless of the paradigm adopted.

We make an informal argument that both methods may be used to solve graph pattern mining applications. In *POSE*, all subgraphs are visited once, and only once, because the algorithm only generate larger subgraph codes that are canonical. Therefore, *all subgraphs are visited in no particular order, meaning that there is no way of determining the pattern of the next subgraph being produced.* Naturally, this is sufficient for visiting all the search space required for the conditional subgraph enumeration problem. On the other hand, in *PASE*, patterns are selected before subgraph exploration. Indeed, the algorithm can always be exhaustive in generating the set of patterns that are going to be matched in the input graph, in case the predicate is unable to select specific patterns. Also each subgraph is visited only once for the same reason as in *POSE*. Therefore, *all subgraphs are visited in a particular order: all subgraphs isomorphic to pattern P_1, then all subgraphs isomorphic to pattern P_2, and so on.* Naturally, this is also sufficient for

Figure 2.5: GPM paradigms: pattern-oblivious (*POSE*) *vs.* pattern-aware (*PASE*).



Source: Made by the author.

visiting all the search space required for the conditional subgraph enumeration problem.

An interesting analogy exists for understanding these two alternative subgraph exploration paradigms for graph pattern mining: querying tuples in a relational database. In this scenario, we can always determine queries (patterns) sufficient to cover all the tuples – this is analog to the *PASE* approach. Alternatively, we may instead retrieve the same set of tuples by fully scanning the database – this is analog to the *POSE* approach. In this work, as we may see, we leverage these alternatives to propose a general modeling for graph pattern mining applications (Chapter 4) and abstractions (Chapter 6).

Chapter 3

Related Work

This chapter is devoted review the GPM literature and related topics. In Section 3.1 we introduce the class of algorithms known as *graph analytics*, and we argue why these algorithms may not exhibit the same challenges as GPM processing. In Section 3.2 we present some specialized ad-hoc solutions for specific GPM problems to motivate why this strategy may be too strict for real-world data analytics pipelines. In Section 3.3 we introduce existing general-purpose GPM systems, which aim at filling the gaps of too specialized GPM solutions. Finally, in Section 3.4 we review a few existing works in which goal is to characterize and understand GPM workloads.

3.1 Graph Analytics

A popular distributed fault-tolerant system, Pregel [85], offers a "think like a vertex" (TLV) programming paradigm, which simplifies the design of graph analytics algorithms (e.g. Pagerank, HITS, belief propagation and shortest path) [15, 69, 43]. In TLV, each vertex of the input graph works as a computation unit, running operations to update its own state value and sharing these processed results with its neighborhood in the input graph. Over the past years many optimizations and variants of Pregel's TLV model have been proposed [44, 122, 131, 116]. Some of these systems [122, 116] present a subgraph-centric model, but it is not transparent to the users and subgraphs are used to reduce data communication among machines. Other matrix-inspired cloud-based graph processing systems such as System-ML [13], PEGASUS [66] and GBASE [65] have also been examined, since for some problems (*e.g.* PageRank), the matrix representation admits the use of fast linear algebraic kernels. However, in such systems every iteration typically requires a full matrix operation, which may be overkill for many applications (e.g. keyword search) where only a small part of the graph needs to be active. Adapting such systems for graph pattern mining problems such as FSM and clique listing is non-trivial.

Single-machine systems such as GreenMarl [53], Galois [96], GraphChi [74], Ligra [113] X-Stream [109] are tightly integrated to the underlying architecture. However, these efforts lack the ability to process iterative problems that accrue large intermediate state and none support the central primitives for GPM algorithms (e.g. frequent graph mining) or work in a heterogeneous, live environment, with machine downtimes (fault tolerance).

3.2 Specialized Graph Pattern Mining

There have indeed been algorithmic advances or specialized frameworks for each of the applications considered, like graph pattern matching [2, 75, 143, 112, 76], graph motif extraction [117], frequent subgraph mining [1, 11, 83, 50, 82], and also RDF and keyword search related problems [33, 58]. BENU [129] is a parallel backtracking algorithm for subgraph querying, designed to leverage redundant operations in subgraph enumeration to generate an optimized query plan. It employs a coarse-grained parallelism by splitting enumeration branches according to a given threshold.

While many of those frameworks are highly efficient for their individual application domain, none of them, to our knowledge, generalize and support different types of applications, while delivering competitive performance.

3.3 General-purpose Graph Pattern Mining

Arabesque [121] and NScale [106] represent the first generation of general purpose distributed systems that operate on a subgraph-centric programming model for graph processing. NScale (built on Hadoop) was not designed to handle FSM and related GPM problems, and it is unclear if it can scale to problems that generate large intermediate state (potentially overwhelming the Hadoop File System). On the other end of the spectrum, G-Miner [106] is a brand new MPI-based C++ framework for graph pattern mining and subgraph exploration. Unlike the above systems and Fractal, G-Miner does not focus on programmer productivity and cannot tolerate machine downtime.

Rstream [128] is a general-purpose single-machine graph pattern mining system that leverages out-of-core environments to store intermediate data. Its core idea is to enumerate and represent graph pattern mining computations through join operations

over a stream of edges. RStream outperforms existing systems – including Arabesque – in several configurations despite being a single-machine system. However, RStream still suffers from a huge cost on intermediate state management, as the results of join operations that represent intermediate subgraphs still must be generated and accessed on disk, which quickly becomes infeasible to handle. The latter observation is empirically studied by a recent work that proposes AutoMine, another GPM system built upon this weakness.

Automine [87] is single-machine system for general purpose graph pattern mining programming. The system, denoted AutoMine, generates C++ optimized code for subgraph enumeration, given template patterns to be enumerated. The main idea is to minimize redundant computation while reducing the memory footprint of subgraph enumeration of induced subgraphs. Although the code generated is parallelizable, it remains unclear what kind of load balancing guarantees AutoMine provides for large-scale skewed input graphs. Nevertheless, it is still a challenge to extend its model for distributed loosely coupled machines, where efficient communication is central for an adequate resource utilization. The same authors proposed the Dryadic [86] system that leverages an intermediate representation for general-purpose pattern matching computations. By using standard compiling optimizations, Dryadic is capable of producing more efficient C++ code than its predecessor (Automine). Despite the advantages of generating specialized code for a set of querying patterns, such approach may become infeasible for patterns with more vertices and edges or for scenarios where the number of patterns of interest is substantial – in these cases the cost for generating the intermediate representation and the cost of maintaining large binaries are not negligible.

Tesseract [12] is a distributed system for incremental graph pattern mining where the input graph may change during computation. Applications in Tesseract are built through two user-defined functions that determines which subgraphs must be pruned from enumeration (*filter*) and which candidates are subgraph of interest (*match*). While this design is simple enough to accommodate many applications, it lacks the flexibility that we aim for: these functions are monolithic in the sense that they are reused in all subgraph enumeration steps for correctness. In this case, it is unclear how to compose subgraph extension primitives with filtering in specific steps of the computation.

Peregrine [60] is a single-machine multi-threaded system for graph pattern mining that uses an execution model centered on the subgraph patterns. In particular, Peregrine expresses any GPM computation as multiple pattern querying routine – in many senses, similar to what Automine [87] and Gtries [108] propose. While this architecture enable applying existing optimizations (symmetry breaking [46], subgraph querying by connected vertex cover [67]) with low cost, some issues still remain unsolved. Its purely pattern-aware approach relies on the pre-computation of the patterns for downstream processing. As we increase the size of subgraphs, an exponential number of pattern queries arise, which

pressures the underlying system – especially, considering the skewed nature of real-world graphs where many patterns may not even exist. Also notice that managing or storing pre-computed patterns on external storage may introduce expressive access overhead. Another concern regarding this model regards the load balancing on distributed settings, where traditional shared memory architectures can not be leveraged to keep adequate resource utilization.

Overall, existing general-purpose systems for GPM either fail to provide a well-defined model for expressing GPM applications, or lack the flexibility to incorporate different graph pattern mining paradigms as solution alternatives. Such features, as we may see, are essential for composability and integration in modern analytics pipelines.

3.4 Understanding subgraph enumeration computation.

GraphMineSuite [10] is a benchmark for evaluating graph mining algorithms. The system provides a set of tools and methodologies for evaluating and tuning properties and behaviors of specific *graph mining algorithms* (i.e., not necessarily related to pattern mining) implemented over a standard set algebra. The authors provide multiple experimental use cases but not from a multi-paradigm perspective. Indeed, the scope is not on abstractions and programming expressiveness for graph pattern mining systems. Besides that, the specificity of each algorithm and the lack of standard strategies for searching the space of subgraphs makes the task of reasoning about different paradigms not trivial, which justifies our approach of evaluating GPM algorithms.

Subgraph enumeration for general-purpose GPM systems has been explored in detail in the context of pattern-aware frameworks. Pattern analysis [61, 68] can be used as an effective tool for optimizing exploration plans of querying patterns. The pattern information about subgraphs of interest is used to reuse computation and to reduce the depth of enumeration. While these optimizations are effective in the pattern-aware scenario, it remains uncertain what are the limitations for GPM problems targeting larger patterns, where the overhead of generating and analyzing multiple patterns may not be trivial. Dryadic [86] is a graph pattern mining system that proposes an intermediate state representation to pattern matching, which allows a more systematic reasoning about optimized exploration plans. While these systems focus on how to optimize set operations and exploration plans for pattern-aware computations, our work takes a step back and handles a more fundamental challenge concerning the trade-offs between pattern-aware

and pattern-oblivious paradigms on real problems. Also, they do not include in their experimental evaluation application scenarios for label-based subgraph querying, so fundamental in the graph database community (e.g. Neo4j).

We also highlight a few community efforts of understanding subgraph mining workloads in the context of specific problems not subject to general-purpose programmable systems. Subgraph matching systems (a.k.a. pattern querying) have been extensively studied: in distributed settings, an experimental study reveal important trade-offs between existing optimizations for subgraph enumeration by joining substructures or by replicating the input data [77]; in the context of labeled graphs [119], the authors provide a detailed comparison among existing algorithms for the problem. Specifically, these solutions rely on very selective query patterns where materializing intermediary query structure before subgraph enumeration is feasible and in this work we are more interested in general-purpose contexts where this assumption may not hold. A detailed study on ordering heuristics for cliques listing [80] evaluates several clique listing algorithms and proposes a new color-based vertex ordering for optimizing the enumeration process. They provide a comprehensive comparison between state-of-the-art clique listing algorithms, along with a discussion about minimizing the enumeration tree by reordering the vertices. While this study gives important contributions on how to improve existing clique listing algorithms, it remains unclear how graph ordering affects other GPM kernels. These focus on fine-tuning specific algorithms for very particular problems (same purpose as GraphMineSuite [10]) and thus, they represent complementary contributions to our goal. In fact, it remains as future work to study which problem-specific optimizations are generalizable.

It is also possible to leverage common memory access patterns observed in GPM workloads to improve memory efficiency on hardware accelerators [136, 18, 120]. These works propose hardware accelerators specifically designed for graph pattern mining applications. For instance, one may explore the power-law distribution in free scale graphs to maintain certain graph data that is more frequently accessed fixed in the hardware cache (or the equivalent to it) [136]. While the insights are very expressive on how to improve data locality in GPM computations, it remains unclear how to leverage this in CPU/GPU architectures where the control over higher-level (low response time) memory chips.

Chapter 4

Modeling graph pattern mining applications

From the subgraph aggregation problem (Definition 12) we can extract three main subtasks that must be performed to produce the expected output: (E) subgraph extraction; (A) subgraph aggregation; and (F) search space pruning. Solving these three subtasks separately and composing the results allow us to solve any instance of the conditional subgraph enumeration problem. Indeed, with these tools we are able to generate the whole search space of subgraphs, to prune out the unimportant ones, and to aggregate the important ones according to some aggregation function.

A naïve approach to model instances of the subgraph aggregation problem is to mix and combine arbitrarily steps of subgraph enumeration, aggregation and pruning in execution time. For example, a GPM algorithm could be described atomically, without clear distinction of any of these parts. Actually, many existing domain-specific systems for GPM adopt such strategy, due to its simplified design [34, 1] or to allow implementation over low-level highly optimized frameworks [17]. Another approach is to define a fixed order for subtask processing and to adopt a BSP (Bulk Synchronous Parallel) [125] runtime to model GPM applications [121]. The order in which GPM subtasks are combined is fixed (E, then F, then A, then E, and so on) to allow splitting GPM tasks into BSP supersteps. Even though the last allows more expressiveness towards general-purpose GPM applications, it lacks the flexibility of combining subtasks in any particular order and the possibility of generating complex GPM programs. In this case, to omit or to jump a subtask, the designer must include empty operations in the application, which can be confusing and inefficient in terms of modeling.

Given these limitations, our proposal is to consider subtasks as computation **primitives**: *extension* (E), *aggregation* (A), and *filtering* (F). Thus, every GPM algorithm can be expressed by a composition of computation primitives, in any particular order that reflects the targeted semantics. Furthermore, because we keep the number of primitives to a minimum, the design of complex GPM applications comes down to building blocks of primitives along with their parameters.

Next we describe in detail the scope of each primitive and which requirements

they should meet. In Section 4.4 we breakdown the design of GPM applications using this primitive-based approach, showing how the designing process of complex applications becomes more concise and effortless.

4.1 Extension primitive

This is a core primitive in our computation model, responsible for generating the solution space of any GPM problem - specifically, it represents the *subgraph enumeration* step of GPM problems. This primitive receives a set of connected subgraphs as input and extends them by using their own neighborhood in G, producing a set of larger subgraphs, also connected. Naturally, it is possible to extend subgraphs using different methodologies, depending on the type of subgraphs targeted and the algorithm used for extending them. Thereby, we make the following distinction between the *type* of subgraph extension and the *method* used for subgraph extension:

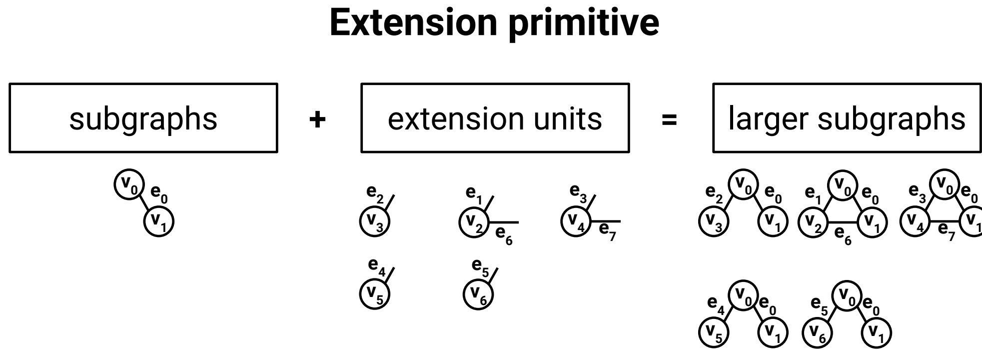
- **Subgraph extension type:** describes the extension units used to obtain a larger subgraph from an existing one. During the extension process, extension units are extracted from the neighborhood of the current subgraph in the input graph. For example, possible extension units include elements like *vertices*, *edges*, or *subgraphs*. We define an extension unit as a tuple (V', E') of vertices and edges that can be added to the current subgraph in order to produce another subgraph (Definition 15). This aspect is important because it defines *what to* search for in the neighborhood of a subgraph.

- **Subgraph extension method:** describes the algorithm used to obtain the subgraph extensions of a given type. For example, one may be interested in all subgraphs that can be generated given an extension type, or yet a sampling of subgraphs that can be generated given an extension type. Moreover, we may be interested in extending subgraphs based on certain priority measure extracted from the application semantics. In the extension method it is usually included some ordering enumeration strategies for only generating a single subgraph code per subgraph to prevent redundant computation. This aspect is important because it defines *how to* explore the neighborhood of a subgraph.

Definition 15. (Extension unit) An extension unit is a tuple (V', E') of vertices and edges that can be added to a subgraph to produce a larger subgraph.

Figure 4.1 shows the scheme of the extension primitive. In summary, the combination of subgraphs and extension units results in larger subgraphs. The extension primitive defined in terms of these two aspects means that type and method works together towards the targeted extension semantics. Thus, on considering subgraph extension, the first point to choose is what is the type of subgraph extension, followed by the specification of the extension algorithm given the type. Therefore, the choice of which extension types to consider is crucial to our modeling, as it is closely related to the expressiveness of the model in designing an wide range of GPM applications. Next, we detail each aspect of the subgraph extension primitive.

Figure 4.1: Extension primitive scheme.



Source: Made by the author.

4.1.1 Extension types: supporting multiple graph pattern mining paradigms

A close look into GPM literature allows us to identify two main classes of applications: *exploratory kernels*, in which the search space of subgraph is explored in disregard with the patterns being discovered [92, 110]; and *searching kernels*, in which the search space is conditioned by characteristics of specific patterns given as input [75, 76]. Surprisingly, a variety of applications can be expressed thinking about these two classes. Based on this, we consider three types of extensions as the standard for most GPM workloads:

- **Edge-oriented extension type (T_E):** expands a subgraph S edge-by-edge, considering its neighborhood, i.e., edges adjacent to the current subgraph not yet included in it. Among extension options, this is the finest-grain of subgraph enumeration that one may perform, being used in algorithms for frequent subgraph

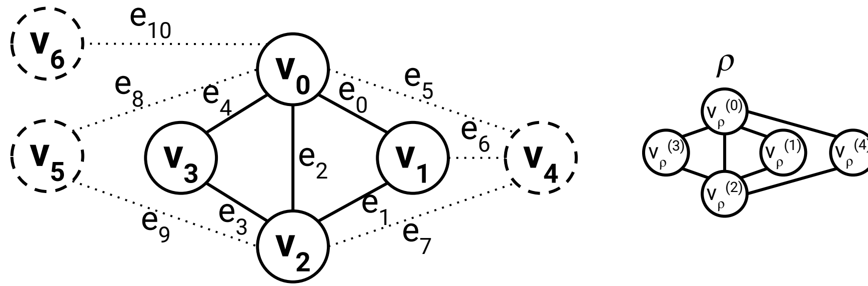
mining [121, 34]. The extension unit of this extension type is a tuple (V', E') representing the set of vertices V' and edges E' being added to the current subgraph: $V' \neq \emptyset$ whenever the edge $e = (v, u) \in E(G)$ being added spans a new vertex outside the current subgraph $((v \notin V(S)) \oplus$ [1] $(u \notin V(S)))$, and $V' = \emptyset$ whenever an edge between two vertices already in the current subgraph is being added $((v \in V(S)) \wedge (u \in V(S)))$, but note that $|E'| = 1$ always because by definition this extension type is edge-by-edge $(e \notin E(S))$. This extension type *outputs general subgraphs* (Definition 2) of the input graph.

- **Vertex-oriented extension type (T_V):** expands a subgraph S vertex-by-vertex, that is, whenever a vertex $v \in V(G)$ is added, all edges that connect v to S are included: $|V'| = 1$ always because by definition this is a vertex-by-vertex extension – $v \notin V(S)$, $|E'| \geq 1$ because at least one edge $e = (v, u) \in E(G)$ connects v with the current subgraph via another vertex u $((v \notin V(S)) \wedge \exists u(u \in V(S)))$. This is often used in exploratory kernels such as motif extraction [108, 32, 110, 121] and clique listing [25, 39, 121]. This extension type *outputs induced subgraphs* (Definition 3) of the input graph.

- **Pattern-oriented extension type ($T_P(\rho)$):** expands subgraphs vertex-by-vertex, but guided by a user-defined reference pattern ρ. The extensions $v \in V(G)$ of a subgraph S will be constrained to those that operate off of ρ: $|V'| = 1$ always because by definition this is an extension vertex-by-vertex, i.e., $v \notin V(S)$ and $|E'| = m$, where m is the number of edges connecting v to the current subgraph according to pattern ρ. Given a subgraph S' isomorphic to a sub-pattern $\rho' \subset \rho$, we say that a vertex $v \in V(G)$ matches a pattern vertex $v_\rho^{(j)}$ iff $S = S' \cup \{v\}$ continues to be isomorphic to some sub-pattern $\rho'' \subseteq \rho$. In other words, the matching vertex can be safely added to the current subgraph in a sense that it is still possible to the current subgraph to grow and to exhibit the given pattern ρ. This notion of matching extends to induced patterns as well: the matching vertex may be required to be incident to only those vertex in the target pattern. Specifically, we assume the vertices of ρ are ordered $v_\rho^{(0)}, v_\rho^{(1)}, \cdots, v_\rho^{(k-1)}$ and thus, a valid extension v matching the j-th pattern vertex $(matches(v_\rho^{(j)}, v))$ must connect each vertex in the current subgraph adjacent to it in the pattern: $\forall(u \in V(S))\forall(i < j)(((v_\rho^{(i)}, v_\rho^{(j)}) \in E(\rho) \ \wedge \ matches(v_\rho^{(i)}, u)) \rightarrow (u, v) \in V(G))$. This is a commonly used option in where analysts interactively provide a reference pattern and ask for instances (subgraphs) of that pattern, for example, pattern querying [76]. This extension type *outputs induced subgraphs iff the reference pattern is induced* (Definition 3) and *outputs general subgraphs iff the reference pattern is not induced* (Definition 2).

[1] "exclusive or" operator

Figure 4.2: Types of subgraph extension: the subgraph above (composed of vertices and edges in solid lines)



Extension Type	Extension Unit V'	E'
Edge-oriented	$\{v_4\}$	$\{e_5\}$
	$\{v_4\}$	$\{e_6\}$
	$\{v_4\}$	$\{e_7\}$
	$\{v_5\}$	$\{e_8\}$
	$\{v_5\}$	$\{e_9\}$
	$\{v_6\}$	$\{e_{10}\}$
Vertex-oriented	$\{v_4\}$	$\{e_5, e_6, e_7\}$
	$\{v_5\}$	$\{e_8, e_9\}$
	$\{v_6\}$	$\{e_{10}\}$
Pattern-oriented given ρ	$\{v_4\}$	$\{e_5, e_7\}$
	$\{v_5\}$	$\{e_8, e_9\}$
Pattern-oriented given induced ρ	$\{v_5\}$	$\{e_8, e_9\}$

Source: Made by the author.

Figure 4.2 shows a running example of these three extension types, where the current subgraph $S(V = \{v_0, v_1, v_2, v_3\}, E = \{e_0, e_1, e_2, e_3, e_4\})$ is highlighted with solid lines and the neighborhood of this subgraph with dashed lines. For simplicity, we assume all vertex and edge labels in this example are 0. The total number of extensions following the edge-oriented approach is 6, the total number of extensions for the vertex-oriented approach is 3, the total number of extensions for the pattern-oriented approach given $\rho = \{(0,0,1,0,0),(0,0,2,0,0),(0,0,3,0,0),(0,0,4,0,0),(1,0,2,0,0),(2,0,3,0,0),(2,0,4,0,0)\}$[2] is 2, and the total number of extensions for the pattern-oriented approach given an induced version of ρ is 1.

We notice the core characteristic of each strategy: edge-oriented extensions always contain single edges, vertex-oriented extensions always contain single vertices but they carry all the incident edges into the subgraph with it, and pattern-induced extensions also always contain single vertices and carry all the corresponding edges respecting the topology of the user-defined pattern ρ. Also, extension types directly correspond and model the existing paradigms for graph pattern mining (Section 2.4): vertex-oriented

[2]By Definition 4, a 5-tuple (s, l_s, d, l_d, l) represents a pattern edge.

(T_V) and edge-oriented (T_E) correspond to the $POSE$ paradigm (Definition 14), while pattern-oriented ($T_P(\rho)$) corresponds to the $PASE$ paradigm (Definition 13).

4.1.2 Extension methods: supporting multiple subgraph extension strategies

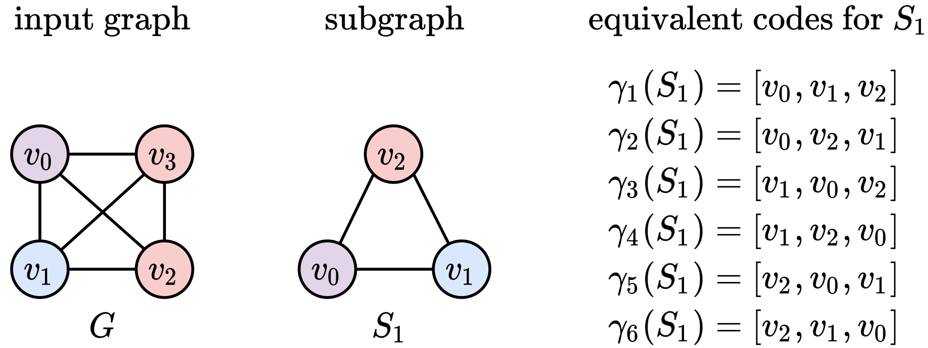
Determining extension types is necessary but not sufficient for a complete extension primitive specification. For example, the application designer may decide to explore the subgraph space of a graph vertex-by-vertex, thereby generating induced subgraphs, but obtaining extensions using a particular strategy: all extensions, a sample of extensions, extensions using an optimized data structure, and so on. Therefore, we create this second level of modeling, denoted as extension methods, to accommodate such requirements and consequently, increase the expressiveness of our model.

4.1.2.1 General-purpose extension methods

An important feature that most extension methods should have is the guarantee of uniqueness for subgraphs enumerated. As in any combinatorial problem, subgraph enumeration must handle symmetric subgraphs (automorphisms of some subgraph). Symmetric subgraphs are valid enumerations that represent the same subgraph S in the underlying input graph, the only difference being the order in which the extension units are composed, i.e., the subgraph code $\gamma(S)$. For example, consider Figure 4.3. A strategy for enumerating this triangle can leverage the vertex-oriented type (T_V) and an extension method that generates this triangle vertices in the following order: 0, then 1, then 2, generating the vertices $V(S) = \{0, 1, 2\}$. However, an alternative enumeration method may generate the same triangle through a different order: 0, then 2, then 1, also generating the vertices $V(S) = \{0, 1, 2\}$. The main issue with symmetries is that they are redundant and can alter the correctness and performance of the algorithm. Thus, extension methods often limit themselves in generating only canonical extensions given a subgraph, i.e., only those subgraph extension units that certainly generate one representative candidate per subgraph. Several techniques for preventing the enumeration of all symmetries of a subgraph exist, like canonical subgraph checking [121] and symmetry breaking for subgraph querying [46]. Separating types from methods for the extension primitive enables these

features to be seemly integrated into the extension GPM task.

Figure 4.3: Example of input graph, subgraph, and equivalent codes.



Source: Made by the author.

Next, we show two extension methods for generating all the extensions of a subgraphs in two contexts: *(1)* exploratory analysis and *(2)* searching of patterns. *All extensions method* is considered the default approach for extension methods since it covers all the search space for subgraph enumeration and thus, it is independent of application semantics. For the algorithms presented, REVERSE(*list*) returns an iterator for *list* in reverse order; FIND-EDGES(ρ, idx) returns all the pattern edges (Definition 4) of pattern ρ in which the source or destination of that edge is idx.

All extensions method for edge-, vertex-oriented types (M_C): Algorithm 3 shows an extension method that generates all the extensions of a subgraph from a graph G. This oulined algorithm is based on a standard enumeration technique known as Reverse Search [8]: the general idea is to (1) define a rule (a function) that obtains smaller connected subgraphs from larger subgraphs by removing the largest vertex from the larger subgraph; and (2) define the inverse of that rule to build-up larger subgraphs from smaller ones. The current subgraph is denoted *code* because this routine works for induced subgraphs (Definition 3) or general subgraphs (Definition 2). For conciseness, we omit elements from the sequence of extension units that can be inferred given the extension type. Thus, in case this method is used for induced subgraphs, $code = \{c_0, c_1, \cdots, c_k\}$ and *extensions* represent lists of vertices and the subgraphs are induced by them. In case this method is used for general subgraphs, *code* and *extensions* represent lists of edges and the subgraphs are the union of those edges.

This procedure implements the canonical subgraph checking [121], which establishes a global ordering for the subgraphs to remove redundant enumerations. The technique is based on three conditions that need to be satisfied for every subgraph code (list of vertices or list of edges) obtained: *(1)* valid extensions must always be greater than the first element in the subgraph code; *(2)* extensions greater than the last element in the code are always valid; *(3)* extensions less than the last element in the code are valid if,

and only if, the extension is adjacent solely to the last element of the code. The original procedure takes $O(k^2 d)$ time, where d denotes the maximum degree of the input graph, since for each edge in the neighborhood of a k-sized subgraph ($O(kd)$) the algorithm performs an $O(k)$ canonical test to determine whether the subgraph plus the extension is valid.

The idea behind these conditions is that if an extension is smaller than the last element of the code and this extension is adjacent to another element besides the last one, then this enumeration should be generated earlier and thus, we can ignore it at this position. This is equivalent to maintaining a lower bound of extensions allowed at each position i in *code*, i.e., as we iterate verifying the neighbors of the subgraph we prune the extensions outside the bounds, generating only valid extensions without symmetries. Our equivalent interpretation of a series of canonical tests allows a reduction in the cost of the algorithm: instead of repeating $O(k)$ tests for each extension Algorithm 3 generates canonical extensions directly with a one-pass extension check, i.e., in $O(kd)$ time. Since this represents an improvement over the original work [121] we adopt this algorithm as our standard method for edge- and vertex-induced extension types.

Algorithm 3 ALL-EXTENSIONS(G, S): subgraph extension method that generates all the vertex-oriented or edge-oriented extensions of a subgraph.

```
1: extensions ← ∅
2: code ← γ(S)
3: lowerBound ← FIRST(code)
4: for c in REVERSE(code) do
5:     for n in N(c) do
6:         if n > lowerBound then
7:             extensions ← extensions ∪ {n}
8:         else
9:             extensions ← extensions − {n}
10:    lowerBound ← MAX(lowerBound, c)
11: return extensions − V(S)
```

All extensions method for pattern-oriented type ($M_P(\rho)$): Algorithm 4 shows an extension method that generates all the extensions of a subgraph matching an user-defined pattern ρ. In this case, the subgraph is defined by its list of vertices and the edges among them are defined by the pattern given as input. In this case, the subgraph is defined by a list of its vertices because the connections between them are implicitly obtained from pattern ρ.

The extension process guided by a pattern ρ works as follows. First, we determine the next vertex that must be matched in the pattern to extract the pattern edges touching the next vertex. The next vertex together with its pattern edges form a star connecting potentially several existing vertices already in the subgraph. Thus, the candidates for

extension are exactly the intersection of neighbors in the subgraph adjacent to the new vertex. That way we are certain that the correct pattern is generated. In case the pattern is induced, the algorithm excludes those extensions incident to edges that should not be in the pattern. An additional step is to apply the symmetry breaking conditions [46] over the set of extensions to further extensions that would generate subgraph duplicates (Algorithm 5).

Algorithm 4 ALL-PATTERN-EXTENSIONS(G, S, ρ): subgraph extension method that generates all the pattern-oriented extensions of a subgraph.

```
1: extensions ← ∅
2: vertices ← V(S)
3: nextVertexIdx ← LENGTH(vertices)
4: patternEdges ← FIND-EDGES(ρ, nextVertexIdx)
5: edgeSrcs ← ∅
6: for e in patternEdges do
7:     src ←EDGE-SOURCE-INDEX(e)
8:     edgeSrcs ← edgeSrcs ∪ {src}
9:     if extensions = ∅ then
10:        extensions ← N(vertices[src])
11:    else
12:        extensions ← extensions  ∩ N(vertices[src])
13: if IS-INDUCED(ρ) then
14:    for src not in edgeSrcs do
15:        extensions ← extensions − N(vertices[src])
16: sbConditions ← GET-OR-COMPUTE-SYMMETRY-BREAKING(ρ)
17: return {n | n ∈ extensions and SATISFIES(n, sbConditions)}
```

4.1.3 Assembling extension types and methods

So far we discussed each aspect of the extension process separately, now we have the tools to present the extension primitive requirements as a whole. Definition 16 describes the extension primitive as an atomic GPM operation. In particular, the extension primitive is a means to obtain larger subgraphs (size[3] $k + 1$) from existing subgraphs (size k), which by induction means that we are able to reach the whole search space of subgraphs or induced subgraphs.

Definition 16. (Extension primitive) The extension primitive, denoted $E(T, M)$, is a GPM atomic operation that produces subgraphs with $k+1$ extension units of type T from

[3] we refer to "size" of a subgraph as the number of extension units it contains

Algorithm 5 SYMMETRY-BREAKING(ρ): generates a set of conditions of a pattern ρ that if applied during subgraph enumeration, it filters out subgraph symmetries, i.e., it guarantees that each subgraph instance is generated once.

```
1: sbConditions ← {}
2: SYMMETRY-BREAKING-REC(ρ, sbConditions)
3: function SYMMETRY-BREAKING-REC(ρ, sbConditions)
4:     vertexEquivalences ← VERTEXEQUIVALENCES(ρ)
5:     if |vertexEquivalences| = |V(ρ)| then
6:         return sbConditions
7:     for eq in vertexEquivalences do
8:         if |eq| > 0 then
9:             v_ρ^(i) ← POP(eq)
10:            v_ρ^(j) ← POP(eq)
11:            l' ← NEXT-UNIQUE-LABEL(ρ)
12:            l'' ← NEXT-UNIQUE-LABEL(ρ)
13:            ρ' ← UPDATE-LABELS(ρ, v_ρ^(i), l', v_ρ^(j), l'')
14:            if i < j then
15:                sbConditions' ← sbConditions ∪ {(v_ρ^(i) < v_ρ^(j))}
16:                return SYMMETRY-BREAKING-REC(ρ', sbConditions')
17:            else
18:                sbConditions' ← sbConditions ∪ {(v_ρ^(j) < v_ρ^(i))}
19:                return SYMMETRY-BREAKING-REC(ρ', sbConditions')
20: function UPDATE-LABELS(ρ, v_ρ^(i), l', v_ρ^(j), l'')
21:     ρ' ← {}
22:     for (s, l_s, d, l_d, l) in ρ do
23:         if s = v_ρ^(i) then
24:             l'_s ← l'
25:         else if s = v_ρ^(j) then
26:             l'_s ← l''
27:         else
28:             l'_s ← l_s
29:         if d = v_ρ^(i) then
30:             l'_d ← l'
31:         else if d = v_ρ^(j) then
32:             l'_d ← l''
33:         else
34:             l'_d ← l_d
35:         ρ' ← ρ' ∪ {(s, l'_s, d, l'_d, l)}
36:     return ρ'
```

n subgraphs with k extension units of type T using the extension method M. We denote as $M(S_i)|T$ the process of obtaining extension units of type T using method M over subgraph S_i. Formally, $E(T, M)\{S_1, S_2, \cdots, S_n\} = \{S_i \cup e \mid 1 \leq i \leq n \text{ and } e \in (M(S_i)|T)\}$.

Different combinations of extension types and methods produce different outcomes.

Table 4.1 show which configurations of these within an extension primitive are well-defined, and next we highlight their behavior:

Table 4.1: Compatibility matrix among general-purpose extension types and methods.

	T_V	T_E	$T_P(\rho)$
M_C	$E(T_V, M_C)$	$E(T_E, M_C)$	*undefined*
$M_P(\rho)$	*undefined*	*undefined*	$E(T_P(\rho), M_P(\rho))$

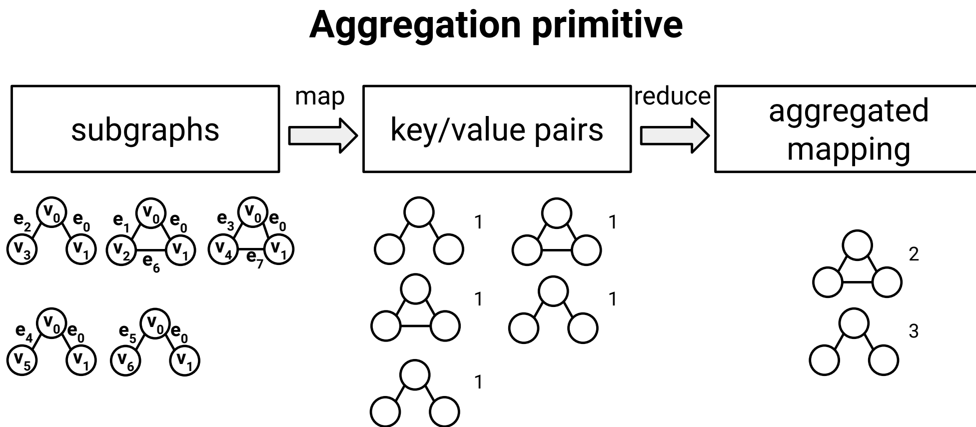
- $E(T_V, M_C)$: this primitive represent an extension process that enumerates all unique induced subgraphs (M_C) vertex-by-vertex (T_V), i.e., it generates induced subgraphs of size 2 from induced subgraphs of size 1, induced subgraphs of size 3 from induced subgraphs of size 2, and so on.
- $E(T_E, M_C)$: this primitive represent an extension process that enumerates all unique subgraphs (M_C) edge-by-edge (T_E), i.e., it generates subgraphs of size 2 from subgraphs of size 1, subgraphs of size 3 from subgraphs of size 2, and so on.
- $E(T_P(\rho), M_P(\rho))$: this primitive represent an extension process that enumerates all unique subgraphs of certain pattern ($T_P(\rho)$) vertex-by-vertex oriented by pattern ($M_P(\rho)$), i.e., the process extends one vertex of the pattern at each step, producing prefixes of the pattern until its completion.

4.2 Aggregation primitive

The output size of subgraph enumeration can quickly become intractable because of the exponential growth present in combinatorial problems such as this [103]. In practice, the impossibility of scaling the subgraph enumeration for bigger subgraphs demands a different approach for output generation in GPM algorithms. Thereby, a summarization feature for GPM algorithms is central for the applicability of the solutions. Indeed, most GPM algorithms produce as output a summarization of the solution space of subgraphs, for example: kernels like cliques listing and counting (Section 2.3.2) rely on counting to summarize the output, while kernels like motifs or FSM (Sections 2.3.1 and 2.3.4) rely on mappings to obtain the distribution of patterns in the input graph. This feature is fundamental for any GPM application that relies on frequency counts, or any application that works with aggregates (*e.g.*, sum). In this context, we adopt a GPM primitive designed to allow general-purpose summarization of large amounts of subgraphs: the *aggregation primitive*.

We adopt a map/reduce approach for aggregation, in which keys and values are first obtained from subgraphs and aggregated by key, according to a reduction function. Figure 4.4 shows the general scheme for aggregation primitive. In other words, it receives a set of subgraphs as input and maps them to key/value entries for subsequent reduction (see Definition 11). For aggregation one needs to define three functions: *(1)* a mapping function to extract a key from a subgraph; *(2)* a second mapping function to extract a value from a subgraph; and *(3)* a reduction function to reduce the values sharing the same key. Definition 17 describes the aggregation primitive as a GPM operation.

Figure 4.4: Aggregation primitive scheme.



Source: Made by the author.

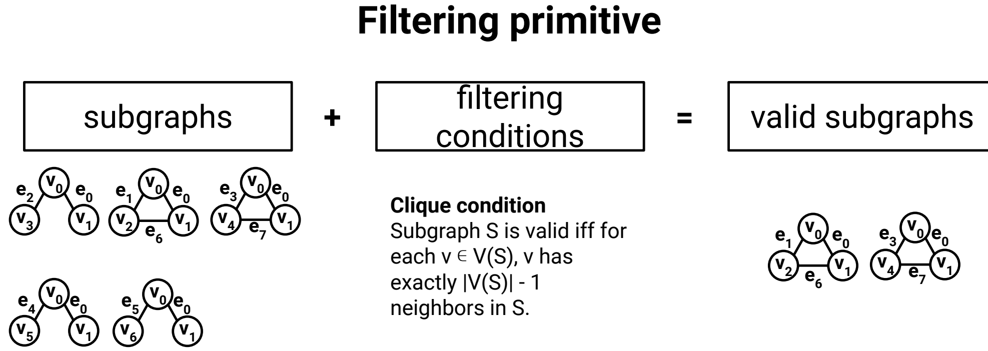
Definition 17. (Aggregation primitive) The aggregation primitive, denoted $A(g, h, r)$, is a GPM atomic operation that produces a subgraph aggregation $\mathcal{A}(\mathcal{S}, g, h, r)$ (Definition 11) from a set of subgraphs $\mathcal{S} = \{S_1, S_2, \cdots, S_n\}$. The parameters for this primitive are the mapping function for keys g, the mapping functions for values h, and a reduction function r. Formally, $A(g, h, r)\{\mathcal{S} = \{S_1, S_2, \cdots, S_n\}\} = \mathcal{A}(\mathcal{S}, g, h, r)$.

4.3 Filtering primitive

The search space of subgraphs is exponential and hence, very costly to traverse. Thus, efficient GPM methods usually consider only a portion of this huge solution space to improve the efficiency of the solutions proposed. For example, enumerating all k-cliques from an input graph is costly, but not as costly as enumerating all subgraphs from the same graph. Therefore, another operation we identify for GPM computations is the *filtering primitive*. The filtering primitive is used to prune subgraphs that do not satisfy

user-defined pruning criteria. Figure 4.5 shows the behavior of the filtering primitive. Definition 18 formalizes the filtering primitive. The only parameter this primitive requires is a boolean function to determine whether a subgraph is valid or not (predicate).

Figure 4.5: Filtering primitive scheme.



Source: Made by the author.

Definition 18. (Filtering primitive) The filtering primitive, denoted $F(p)$, is a GPM atomic operation that selects only valid instances from a set of subgraphs $\mathcal{S} = \{S_1, S_2, \cdots, S_n\}$. The parameter for this primitive is the function $p : \mathcal{S} \rightarrow \{true, false\}$, which represents a predicate with the necessary conditions of a valid subgraph. Formally, $F(p)\{\mathcal{S} = \{S_1, S_2, \cdots, S_n\}\} = \{S_i \mid S_i \in \mathcal{S} \text{ and } p(S_i) = true\}$.

Filtering primitives may also prune subgraphs by considering a source of information provided by an upstream aggregation primitive (Definition 17). This particular filter can be leveraged within algorithms like FSM, where subgraphs that do not belong to the current set of frequent subgraphs may be discarded. For instance, filtering functions can hold some output from previous aggregations and can include such information directly in its definition (e.g., as lookup tables [139]).

4.4 Design of applications

The GPM primitives *extension*, *aggregation*, and *filtering* represent the building blocks in GPM applications. Thus, our modeling strategy comes down to specifying sequences of primitives along with their respective parameters that represent the solution logic. Our approach is novel in the sense that this is the first attempt to model GPM algorithms in a way that is independent of system implementation, that is concise in terms

of the operators needed to represent different strategies, and that is modular in terms of how easily the operators can be combined to solve complex graph analytics routines.

Because our goal is to model general-purpose graph pattern mining applications, we introduce the notion of multi-step applications, in which an output (e.g. the result of an aggregation) of a previous step is used as input to the next step (e.g. as part of filtering conditions). An application step (Definition 19) is denoted by a string of primitives to be applied over an input graph G (regular expression) that starts with an extension primitive (E), proceeds with any sequence composed of extension and filtering primitives ($(E+F)$), finishing with an aggregation primitive that produces the output (A). For clarity, we omit the primitive parameters.

Definition 19. (Application step) An application step is denoted by the regular expression $GE(E+F)^*A$ composed of input graph G, and primitives extension (E), filtering (F), and aggregation (A).

Finally, a GPM application is a sequence of application steps. Steps may be related to each other. In FSM, for example, the aggregation of previous steps determines which patterns are frequent, and this information may be used in subsequent steps for growing those frequent patterns.

The specification on how these steps are deployed for execution is entirely a system's design choice and independent from this model. For instance, one may implement a GPM system that ensures that steps are executed sequentially in a single-thread machine; on the other hand, a different approach would be to parallelize the execution of a step for multi-threaded distributed environments. We explore more on how to provide an effective implementation of this model in Chapter 5. For now we discuss how to describe algorithmic solutions for GPM problems using this model as a facilitating tool.

4.4.1 GPM algorithm design

In this work we use the following following scheme to illustrate an application step, which can be composed to represent GPM applications:

GPM application step:	$\mathtt{GP_0P_1 \cdots P_n}$
	where $\mathtt{G}$ = "input graph", $\mathtt{P_0} = \mathtt{E(T,M)}$, $\mathtt{P_n} = \mathtt{A(g,h,r)}$,
	and $\mathtt{P_i} \in \{\mathtt{E(T,M)}, \mathtt{F(p)}\}$ for $1 \leq i \leq n-1$

Figure 4.6: Algorithm design: pattern-oblivious (POSE) vs. pattern-aware (PASE).

	Pattern-oblivious (POSE)	Pattern-aware (PASE)
motifs (*k-MC*)	1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_V}, \mathtt{M_C})$ 2: **output** $\mathtt{GE_1 \cdots E_k A}$	1: $\mathtt{P} \leftarrow$ VPATTERNS-IND(k) 2: **for** ρ **in** $\mathtt{P}$ **do** 3: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho), \mathtt{M_P}(\rho))$ 4: **output** $\mathtt{GE_1 \cdots E_k A}$
cliques (*k-CL*)	1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_V}, \mathtt{M_C})$ 2: $\mathtt{F} \leftarrow \mathtt{F}(\mathtt{isClique})$ 3: **output** $\mathtt{GE_1F_1 \cdots E_kF_kA}$	1: $\rho \leftarrow$ CLIQUE-PATTERN(k) 2: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho), \mathtt{M_P}(\rho))$ 3: **output** $\mathtt{GE_1 \cdots E_k A}$
pattern querying (*ρ-PQ*)	1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_E}, \mathtt{M_C})$ 2: $\mathtt{F} \leftarrow \mathtt{F}(\mathtt{isSubpatternOf}(\rho))$ 3: **output** $\mathtt{GE_1F_1 \cdots E_{\lvert E(\rho)\rvert}F_{\lvert E(\rho)\rvert}A}$	1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho), \mathtt{M_P}(\rho))$ 2: **output** $\mathtt{GE_1 \cdots E_k A}$
FSM (*k-FSM-α*)	1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_E}, \mathtt{M_C})$ 2: $\mathtt{P'} \leftarrow \mathtt{EA}$ 3: $\mathtt{P_f} \leftarrow$ FREQ-PATTS$(\mathtt{P'})$ 4: **output** $\mathtt{P_f}$ 5: **for** $i \leftarrow 2$ **to** k **do** 6: $\mathtt{F} \leftarrow \mathtt{F}(\mathtt{patternIsFreq}(\alpha, \mathtt{P_f}))$ 7: $\mathtt{out} \leftarrow \mathtt{GE_1F_1 \cdots E_{i-1}F_{i-1}E_iA}$ 8: $\mathtt{P_f} \leftarrow$ FREQ-PATT-SUPP$(\mathtt{out})$ 9: **if** $\mathtt{P_f} = \emptyset$ **then** 10: **break** 11: **output** $\mathtt{P_f}$	1: $\mathtt{P_f} \leftarrow \emptyset$ 2: **for** $\mathtt{i} \leftarrow 1$ **to** $\mathtt{k}$ **do** 3: $\mathtt{P_e} \leftarrow$ EXTEND-BY-EDGE$(\mathtt{P_f})$ 4: $\mathtt{P_f} \leftarrow \emptyset$ 5: **for** ρ **in** $\mathtt{P_e}$ **do** 6: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho), \mathtt{M_P}(\rho))$ 7: $\mathtt{P'} \leftarrow \mathtt{GE_1 \cdots E_{\lvert V(\rho)\rvert}A}$ 8: $\mathtt{P_f} \leftarrow \mathtt{P_f} \cup$ FREQ-PATT-SUPP$(\mathtt{P'})$ 9: **if** $\mathtt{P_f} = \emptyset$ **then** 10: **break** 11: **output** $\mathtt{P_f}$
quasi cliques (*k-QC-α*)	1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_V}, \mathtt{M_C})$ 2: $\mathtt{qc} \leftarrow \mathtt{canBeQuasiClique}(\mathtt{k}, \alpha)$ 3: $\mathtt{F'} \leftarrow \mathtt{F}(\mathtt{qc})$ 4: $\mathtt{F''} \leftarrow \mathtt{F}(\mathtt{isQuasiClique}(\alpha))$ 5: **output** $\mathtt{GE_1F'_1 \cdots E_kF'_kF''A}$	1: $\mathtt{P} \leftarrow$ VPATTERNS-IND(k) 2: $\mathtt{P'} \leftarrow \{\rho \in \mathtt{P} \mid$ DENSITY$(\rho) \geq \alpha\}$ 3: **for** ρ **in** $\mathtt{P'}$ **do** 4: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho), \mathtt{M_P}(\rho))$ 5: **output** $\mathtt{GE_1 \cdots E_k A}$
query specialization (*ρ-QS*)	1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_E}, \mathtt{M_C})$ 2: $n \leftarrow \lvert E(\rho)\rvert$ 3: $\mathtt{P} \leftarrow \{\rho' \supset \rho \mid \lvert \mathtt{E}(\rho')\rvert = \mathtt{n}+1\}$ 4: $\mathtt{F} \leftarrow \mathtt{F}(\mathtt{isSubpatternOf}(\mathtt{P}))$ 5: **output** $\mathtt{GE_1F_1 \cdots E_nF_nE_{n+1}A}$	1: $\mathtt{n} \leftarrow \lvert \mathtt{E}(\rho)\rvert$ 2: $\mathtt{P} \leftarrow \{\rho' \supset \rho \mid \lvert \mathtt{E}(\rho')\rvert = \mathtt{n}+1\}$ 3: **for** ρ **in** $\mathtt{P}$ **do** 4: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho), \mathtt{M_P}(\rho))$ 5: **output** $\mathtt{GE_1 \cdots E_{\lvert E(\rho')\rvert+1}A}$
label search (*k-LS-ℒ*)	1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_V}, \mathtt{M_C})$ 2: $\mathtt{F} \leftarrow \mathtt{F}(\mathtt{labelsSubsetOf}(\mathcal{L}))$ 3: **output** $\mathtt{GE_1F_1 \cdots E_kF_kA}$	1: $\mathtt{F} \leftarrow \mathtt{F}(\mathtt{labelsSubsetOf}(\mathcal{L}))$ 2: $\mathtt{P} \leftarrow$ VPATTERNS-IND(k) 3: **for** ρ **in** $\mathtt{P}$ **do** 4: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho), \mathtt{M_P}(\rho))$ 5: **output** $\mathtt{GE_1F_1 \cdots E_kF_kA}$
minimal keyword search (*k-MKS-𝒦*)	1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_V}, \mathtt{M_C})$ 2: $\mathtt{p} \leftarrow \mathtt{coveredOnceOrNone}(\mathcal{K})$ 3: $\mathtt{F'} \leftarrow \mathtt{F}(\mathtt{p})$ 4: $\mathtt{F''} \leftarrow \mathtt{F}(\mathtt{isMinimal})$ 5: **output** $\mathtt{GE_1F'_1 \cdots E_kF'_kF''A}$	1: $\mathtt{p} \leftarrow \mathtt{coveredOnceOrNone}(\mathcal{K})$ 2: $\mathtt{F'} \leftarrow \mathtt{F}(\mathtt{p})$ 3: $\mathtt{F''} \leftarrow \mathtt{F}(\mathtt{isMinimal})$ 4: $\mathtt{P} \leftarrow$ VPATTERNS-IND(k) 5: **for** ρ **in** $\mathtt{P}$ **do** 6: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho), \mathtt{M_P}(\rho))$ 7: **output** $\mathtt{GE_1F'_1 \cdots E_kF'_kF''A}$

Source: Made by the author.

Recall from Section 2.4 that pattern-aware (PASE) and pattern-oblivious (POSE) are alternative paradigms for GPM. In Table 4.6 we show how to model algorithms for the kernels considered in this work using these two approaches. Next we drilldown into important highlights of these algorithms:

Motifs (k-*MC*). For POSE, the algorithm is a sequence of k extensions, used to produce induced subgraphs with k vertices, followed by an aggregation to count by pattern (denoted as $\mathtt{A}$). Recall that $E(T_V, M_C)$ implies in enumerating induced subgraphs (T_V) using a method M_C that returns all unique candidates (see Section 4.1). For PASE, we split the computation into *multiple pattern-aware steps* (line 2), each representing the querying of a pattern. For example, if $k = 3$ we have two possible patterns: a path of size 2 or a triangle. In this case, we construct a pattern-aware modeling with two steps ($\mathtt{GE_1E_2E_3A}$): querying 2-paths (pattern ρ_1) if $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho_1), \mathtt{M_P}(\rho_1))$; and querying triangles (pattern ρ_2) if $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_P}(\rho_2), \mathtt{M_P}(\rho_2))$.

Cliques (k-*CL*). For POSE, the algorithm may be a sequence of k subsequences of extension followed by a clique filtering condition (`isClique`), ending with a counting aggregation: $\mathtt{GE_1 \cdots E_kFA}$. Although this design is correct, it is not efficient. Indeed, we miss the opportunity for pruning after each extension primitive. After all, cliques represent only a small portion of all the induced subgraphs in the input graph and this strategy delay this filtering to the last extension only, which will likely generate a huge number of invalid subgraphs. This improved design is the one described in Figure 4.6. For PASE, the algorithm only have one kind of pattern to enumerate: clique pattern (line 1). In fact, this PASE design for cliques is a special case of pattern querying when ρ is a triangle.

Pattern querying (ρ-*PQ*). The POSE algorithm enumerates subgraphs edge-by-edge and filters out the ones that are not isomorphic to some connected sub-pattern $\rho' \subseteq \rho$. A pattern $\rho' \subseteq \rho$ is a connected sub-pattern iff it can be represented by a subset of patten edges in ρ whose topology is connected (represented as predicate `isSubpatternOf`(ρ)). In PASE algorithm, valid subgraphs in this querying kernel are obtained by extending the current subgraph according to the query pattern ρ. The type of extension $T_P(\rho)$ and the method of extension $M_P(\rho)$ specifies this behavior, repeated $|V(\rho)|$ times, i.e., the number of vertices in the query pattern. While POSE has to visit many invalid subgraphs to apply the filter function, PASE design in this case is capable of reaching valid subgraph directly without any kind of subgraph filter.

Frequent subgraph mining kernel (k-*FSM*-α). In POSE algorithm, because FSM is hierarchically defined, we model this application as multiple iterations. On each it-

eration, except the first one, the execution enumerates known frequent subgraphs and extend this subgraph by one edge, generating larger subgraphs, which are used to determine the frequency of larger patterns. The first step only determines via aggregation the frequent edges in the input graph (line 2). Subsequent steps alternate between extension primitives and filtering primitives (line 7), making sure the algorithm enumerate only subgraphs with known frequent patterns (`patternIsFreq`). Note that this is possible because of the anti-monotonic property of the minimum image-based support definition [16]. The PASE algorithm is also iterative (line 2) but on each iteration multiple steps are generated, one for each pattern support computation (line 5). generates pattern candidates (pattern generation) and verifies whether those generated patterns are frequent. From the set of frequent patterns $\mathtt{P_f}$ the algorithm generates a new candidate pattern set $\mathtt{P_e}$ composed of frequent pattern plus a single edge between vertices of frequent labels. This process is repeated until no more frequent patterns are found or whenever all k-edge patterns are verified. For each pattern candidate $\rho \in \mathtt{P_e}$, a new application step for querying that pattern is created (line 7).

We highlight that both *POSE* and *PASE* alternatives for k-*FSM*-$\alpha$are multi-step. However, *POSE* induces *one step per subgraph size, i.e., k steps*, while *PASE* generates *one step per pattern candidate, i.e., patterns that are extensions of frequent patterns by one edge*. Naturally, the magnitude of steps in *PASE* is larger than in *POSE*, because the search space of patterns also grows exponentially.

Quasi cliques (k-QC-α). In POSE algorithm we consider two predicates for pruning the search space of quasi cliques: `canBeQuasiClique` keeps only subgraphs that has the potential to be considered a k-vertex quasi clique given a density measure, `isQuasiClique` checks whether a subgraph is indeed a quasi clique. The first predicate as an early pruning condition to reduce the number of spurious subgraphs generated and checked. An alternative approach used in PASE algorithm is to first determine which patterns of a given size meet the quasi clique condition *before enumerating any subgraph* (lines 1-2). After that, the algorithm generates one step per quasi clique pattern (lines 3-5).

Query specialization (ρ-QS). The POSE algorithm starts by determining which patterns with one additional edge (i.e. ρ plus one edge) exist (line 3). With that information, the subgraph enumeration may proceed and invalid ones can be pruned via `isSubpatternOf` predicate. The PASE algorithm generates the same set of possible superpatterns of ρ and queries each one individually in a multi-step application (line 5).

Label search (k-LS-$\mathcal{L}$). In this kernel we are interested in induced subgraphs with k vertices that only contain certain labels in $\mathcal{L}$. Both POSE and PASE algorithms leverage

a predicate `labelsSubsetOf` used to determine whether the subgraph's labels ($L(S)$) only contains labels of interest in $\mathcal{L}$. The main difference between the alternative paradigms remains the same: while POSE explores subgraphs of no particular pattern within a single step, PASE does so in multiple steps, one for each pattern generated (line 3).

Minimal keyword search (k-*MKS*-$\mathcal{K}$). In POSE algorithm we consider two predicates: `coveredOnceOrNone` ensures that keywords in partial subgraphs are covered once or none, which guarantees that each of subgraph's vertices is indispensable for generating a minimal set; `isMinimal` is a final predicate used to ensure that only minimal subgraphs are aggregated. This design is similar to quasi cliques', while the former predicate is applied after each subgraph extension the latter is used as a final filtering to maintain correctness. The PASE algorithm leverages the same two predicates by generates one step per pattern of a given size. What this means is that even if no minimal subgraph of a given pattern that covers the label set exists, the algorithm will submit a new step for it anyway because there is not trivial to determine this apriori.

Overall comparison: pattern-oblivious (POSE) vs. pattern-aware (PASE). The central property of POSE algorithms is that they gather in a *single application step* the enumeration of subgraphs representing *different patterns*, which tends to produce more coarse-grained execution tasks. The possible drawback of this strategy is that pruning the search space in many cases must rely only on expensive filtering predicates and moreover, many spurious subgraphs may have to be generated *before* filtering can happen. The central property of PASE algorithms is that they submit *one application step per pattern*, meaning that each step is responsible for only a particular subset of subgraphs. Unlike POSE, PASE produces more fine-grained execution tasks. Because the number of patterns grows exponentially with the subgraph size, this may be a problem for larger patterns.

4.5 The advantages of primitive-based model

This primitive-based model allows representing algorithm solutions to any GPM problem that involves the enumeration of subgraphs from the input graph and this includes search (aggregate subgraphs satisfying the predicate), optimization (find *the best* subgraph satisfying the predicate according to some objective function), and decision problems (verify whether some subgraphs satisfy the predicate). Conciseness, robustness, and interactiveness are the main requirements guiding this modeling design. This primitive-based approach is *concise* because the domain of possible operators is reduced

and each element has a well-defined purpose: extension to navigate the search space, aggregation to produce summarized output, and filtering to prune the search space. Such approach is also robust, because it is able to express algorithms to any instance of the subgraph aggregation problem (Definition 12) using both pattern-oblivious (POSE) and pattern-aware (PASE) paradigms. Finally, this modeling is also interactive, in the sense that partial results obtained from primitives can be verified during the solution design phase. Such aspect is especially important and desirable in data science exploratory scenarios where the results and hypothesis are usually built incrementally and interactively [101].

Other interesting aspect of this design concerns the opportunities it brings in terms of exploring *alternative solution strategies for the same problem*. Indeed, the equivalence between the GPM paradigms along with the modeling support for effortless expressiveness of applications using both approaches allow better reasoning about execution plans and optimization in general-purpose systems. Such decoupling between problem and alternative solution strategies opens a whole new front of opportunities: to bring GPM computations closer to optimization-based systems with well-defined operators and execution plans [7]; and to improve the performance diagnosis of execution bottlenecks [100, 29].

Additional discussion on the advantages of the modeling proposed in this chapter is provided in Chapter 6, where we anticipate novel alternative abstractions that emerges from it. Meanwhile, in Chapter 5, we proceed in showing how this model may be designed and implemented into a general purpose distributed and parallel system for GPM programming and computation.

Chapter 5

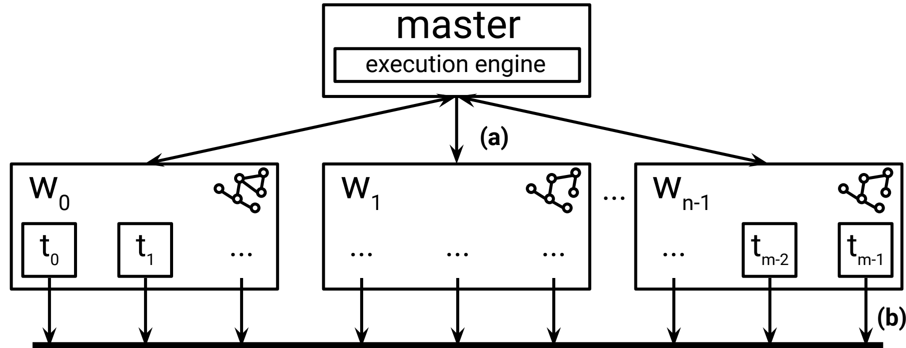
Implementing graph pattern mining systems

The first step when dealing with performance optimization in systems is to determine the scope of the execution environment considered. Thus, we must specify what kind of computing resources are available for processing before proposing optimizations for improving the execution performance. Because GPM-related tasks are so time consuming and challenging, we decide to focus on the scalability aspect of such methods and consequently, a distributed environment to accommodate our GPM application modeling (see Section 4).

Specifically, we consider a distributed and parallel environment for GPM processing, organized with a single application master and multiple workers. As illustrated in Figure 5.1, the master contains the execution engine, which manages the underlying cluster resources and coordinates the execution of chunks of work (a). Workers represent instances that perform the actual processing. Specifically, each worker $(w_0, \ldots, w_{n-1})$ is a process running on multiple execution threads $(t_0, \ldots, t_{m-1})$ over a shared-memory model. Communication occurs between master and workers for scheduling, but also among workers via a message passing paradigm. Such design eases the initialization and the asynchronous assignment of work among workers. Other advantages of this model become clear as we discuss our execution design and optimizations in the following sections. Finally, we assume that all workers have access to the whole input graph G, i.e., the input graph G is replicated in-memory on each worker. Such choice is simplification design since the subgraph enumeration cost dominates the cost of maintaining the input graph in memory. Also, our design can be extended to leverage distributed efficient graph storing technologies [72].

We highlight two main challenges concerning graph pattern mining computations in this distributed environment: (1) to develop an efficient subgraph enumeration method; and (2) to ensure an adequate load balanced parallel execution. The first aspect (Section 5.1) concerns the proper design of the subgraph enumeration algorithm for the GPM primitives both in terms of memory and time. The second aspect (Section 5.2) arises in any massively parallel environment, especially in skewed scale-free graphs.

Figure 5.1: Execution environment. (a) The master communicates with workers for scheduling. (b) Workers execute the tasks assigned to them.



Source: Made by the author.

5.1 Efficient subgraph enumeration

Many graph pattern mining (GPM) algorithms suffer from a combinatorial explosion of the search space, often requiring the maintenance of large intermediate state (memory) and thereby overloading the underlying system. As a motivating example to illustrate this issue, we estimate the amount of memory necessary to keep all induced subgraphs occurring in the medium-sized *Microsoft Collaboration Network* [34], composed of 100K vertices and 1.08M edges. We present our estimates in Table 5.1. We consider that each subgraph can be represented solely by its vertices (with no memory overheads), *i.e.*, `NumberOfVertices` × `BytesPerVertex`, where `BytesPerVertex` is set to a standard integer of 4 *bytes*. As a result, the memory requirements quickly become unbearable for subgraphs with four or five vertices, resulting in demands of 163.27GB and 46.37TB, respectively. Moreover, because those estimates do not account for the aggregation space requirements, the problem will only be exacerbated. This also supports our assumption that relatively medium-sized input graphs that easily can be accommodated in main memory of modern architectures may generate extremely intractable workloads very quickly due to the exponential growth: in conclusion, the bottleneck is clearly on the subgraphs enumerated instead of being equivalent to the input size.

At this point we make a distinction between the *memory demands coming from the enumeration cost* versus the *demands that accrue due to the aggregation cost*. Because the former tend to be higher than the latter and moreover the latter is domain-specific, we decide to focus on optimizing the memory demands of the extension primitive.

Before presenting our solution to this issue, let us first review the subgraph enumeration strategy adopted by existing GPM systems like Arabesque [121] and NScale [106]: the *breadth-first subgraph enumeration* strategy. Then, we are going to argue why this may not be a good choice for large scale graph pattern mining workloads. This sub-

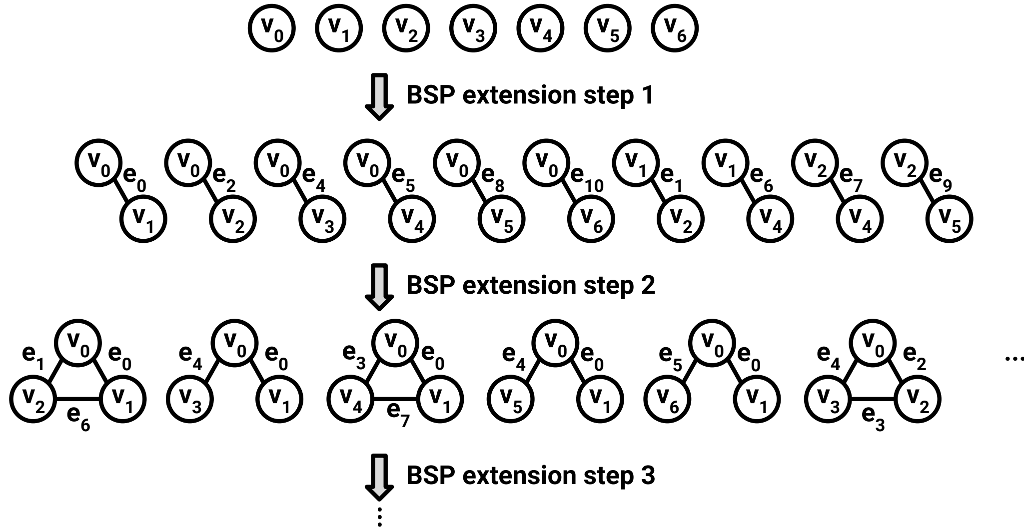
Table 5.1: Estimated memory demands - the number of induced subgraphs grows exponentially, and so the space required to store them.

# vertices	# subgraphs	estimated memory
1	100,000	390.62 *KB*
2	1,080,156	8.24 *MB*
3	66,081,419	756.24 *MB*
4	10,957,439,024	163.27 *GB*
5	2,549,490,788,644	46.37 *TB*

graph enumeration strategy is usually deployed using a Bulk Synchronous Parallel (BSP) paradigm [125]. In this case, subgraphs are extended hierarchically, using a breadth-first search algorithm. Figure 5.2 illustrates the basic idea and its relevance in a large-scale distributed execution platform. Without loss of generality, let us assume the enumeration of induced subgraphs. At each enumeration level, an additional vertex extension is considered. Thereby, the runtime generates subgraphs with two vertices from subgraphs with one vertex, subgraphs with three vertices from subgraphs with two vertices, and so on. This strategy has one main advantage and one major drawback. The main purpose and advantage of such strategy is to allow the work to be repartitioned at the end of each synchronization step, i.e., at the end of each subgraph enumeration depth. In this context, roughly equal amounts of work can be reassigned for a partitioned parallel execution. The major drawback of this strategy concerns the space cost of maintaining the subgraphs enumerated on each step. Moreover, the space cost of this approach is bounded by the number of subgraph candidates, which quickly becomes infeasible as discussed in the beginning of this section. Finally, because the time efficiency of this method relies on the load balancing and the load balancing strategy assumes a redistribution of work at the end of each enumeration step, this space cost is inevitable, as the subgraph candidates must be materialized in memory or in secondary storage at some point.

We take a different approach for subgraph enumeration. Instead of enumerating using a breadth-first strategy and experiencing huge space inefficiency, we propose a subgraph processing methodology that avoids the need to maintain intermediate state by (1) enumerating subgraphs using depth-first search algorithm based on primitives (Section 5.1.1) and (2) recomputing the subgraphs from scratch on each application step (Section 5.1.2). This is possible due to our computational model, designed to support multiple primitive components in a single application step. As we discuss in Section 5.1.3, our method is capable of delivering a reliable, space-bounded subgraph enumeration for general-purpose graph pattern mining.

Figure 5.2: Breadth-first subgraph enumeration for induced subgraphs from the input graph in Figure 4.2.



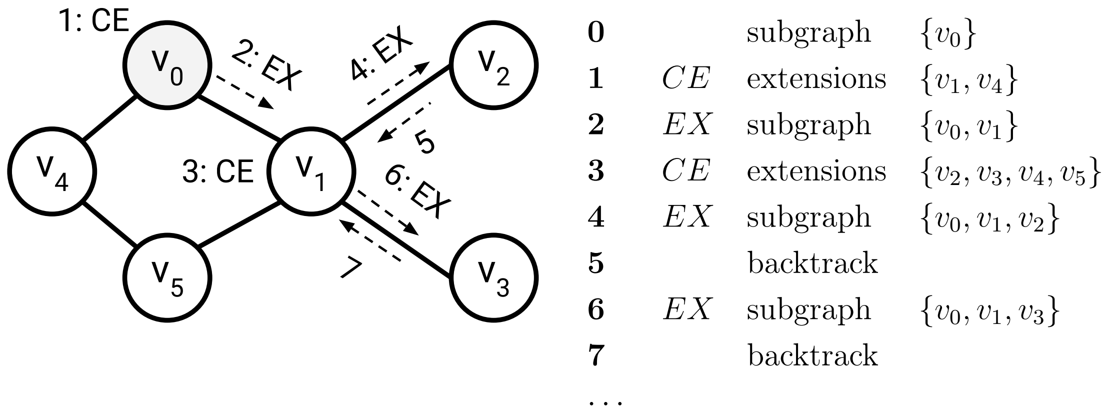
Source: Made by the author.

5.1.1 Depth-first subgraph enumeration

In a naïve depth-first subgraph exploration, subgraphs are generated recursively considering as possible extensions the neighborhood of the current subgraph. Such a simple approach has a major issue: subgraphs may need to be verified several times, because this extension unit can be connected to the current subgraph from multiple sites. In this case, the amount of spurious work for processing the generated subgraphs has the potential to be exacerbated beyond the necessary. In this context, we propose a modified in-depth subgraph exploration that combines two phases: a subgraph expansion phase followed by an in-depth exploration phase. Thus, new subgraphs are generated by computing the possible extensions of the current subgraph (`computeExtensions` – CE), immediately extending the subgraph to one of them in depth until the enumeration depth targeted (`extend` – EX), and backtracking for consuming remaining extensions. In particular, the expansion phase is used to reduce the redundant work experienced by the naïve approach, as the valid subgraph extensions are unique and generated once per subgraph to be consumed after. Specifically, the algorithm starts with an extension unit (e.g. a vertex for induced subgraphs) representing the root of the enumeration tree, generates valid extensions (vertices) connecting to this root, add the first valid extension to the current subgraph, and proceeds the processing with this larger subgraph composed of two vertices. The algorithm repeats this process up to a certain enumeration depth, determined based on the size of the subgraphs of interest and while still remains unexplored computed extensions.

Figure 5.3 presents an example of this method. In this example, we consider the enumeration of induced subgraphs with three vertices, starting from enumeration root v_0. The execution starts with root v_0 (step 0), computes the valid extensions for subgraph $\{v_0\}$ (step 1), consumes extension v_1 and extends the current subgraph obtaining a new subgraph $\{v_o, v_1\}$ (step 2), computes the extensions of subgraph $\{v_0, v_1\}$ (step 3), consumes the first extension v_2 yielding the first subgraph with three vertices $\{v_0, v_1, v_2\}$ (step 4), backtracks to subgraph $\{v_0, v_1\}$ (step 5), consumes the next extension v_3 from step 3 yielding the second subgraph with three vertices $\{v_0, v_1, v_2\}$ (step 6), and so on.

Figure 5.3: Depth-first exploration for subgraph enumeration. In this example, the target is subgraphs with three vertices.



Source: Made by the author.

To make this process transparent and extensible, we propose a new data structure specifically designed to represent this two phase in-depth enumeration method. We call this structure *subgraph enumerator* because it represents a checkpoint for the subgraph enumeration process. Figure 5.4 shows the structure of a subgraph enumerator.

Figure 5.4: Subgraph enumerator abstraction.

```
subgraph-enumerator {
  currentSubgraph; // current subgraph
  computeExtensions(); // compute the valid extensions of the current subgraph
  extend(); // consume next extension
  next(); // returns the subgraph enumerator in the next enumeration depth
}
```

Source: Made by the author.

Each subgraph enumerator is identified by a current subgraph under extension process. Extensions candidates of this subgraph are generated with `computeExtensions()` , i.e., it implements the first phase of subgraph expansion (CE). In case the current subgraph is empty, this function generates the set of initial extension units of the input graph. For example, for edge-oriented extension type (T_E) it generates the set of edges of

the input graph; for vertex-oriented extension type (T_V) it generates the set of vertices of the input graph; and for pattern-oriented extension type ($T_P(\rho)$) it also generates the set of vertices of the input graph. Subgraph enumerators work as hierarchical work queues, where a `computeExtensions()` call produces the work items (i.e. extension units) based on the current subgraph and an `extend()` call consumes a work item from this queue. The consuming process via `extend()` generates a new state in the subgraph enumerator of the next enumeration level (`next()`), sharing the current subgraph extended by one extension unit consumed for expansion. This strategy allows the enumeration engine to maintain an in-place current subgraph that grows whenever it includes new extension units and that shrinks whenever it backtracks.

Algorithm 6 describes this two-phase depth-first subgraph enumeration method. Its input is an application step, *i.e.*, a sequence (array) of primitives to be executed (Definition 19). The algorithm initiates by creating an empty subgraph enumerator, which is given as parameter to the function *process* (lines 1-2). This function (lines 3-12) applies the primitives over the subgraph enumerators recursively. For example, the first primitive is indexed by zero in the *step* array. In case of extension (lines 4-8), the algorithm invokes the method recursively for each possible extension of the current subgraph. In case of filtering (lines 9-10), we only call the process function pointing to the next primitive if subgraph passes the filter. Finally, the algorithm handles the aggregation according to the user's reduction function (lines 11-12), which marks the end of the recursive call.

Algorithm 6 DFS-PROCESSING(*step*)

```
 1: se ← CREATE-SUBGRAPH-ENUMERATOR()
 2: PROCESS(se, step, 0)
 3: function PROCESS(se, step, idx)
 4:     P ← step[idx]
 5:     S ← se.currentSubgraph
 6:     if IS-EXTENSION(P) then                                  ▷ E(T, M)
 7:         se.computeExtensions()
 8:         while se.extend() do
 9:             PROCESS(se.next(), step, idx + 1)
10:     else if IS-FILTER(P) and FILTER(S) then                   ▷ F(p)
11:         PROCESS(se.next(), step, idx + 1)
12:     else if IS-AGGREGATION(P) then                            ▷ A(g, h, r)
13:         AGGREGATE(KEY(S), VALUE(S))
```

The main advantage of this method is to allow a reduced space cost for application step processing. Specifically, in this strategy we only need to maintain one *subgraph enumerator per enumeration level at a time*. In particular, considering induced subgraphs, each execution thread has to maintain at most $O(kn)$ words representing computed extensions on each enumeration depth, where n is the number of vertices in the input graph and k is the target size for subgraphs. However, we notice that k is usually orders of

magnitude smaller than n and moreover, for sparse graphs (which is usually the case for real-world datasets) we observe that n is an overestimation since most vertices have low degree ($N(v) \ll n$ for $v \in V(G)$).

This is not true for breadth-first approaches, where *all subgraphs of certain size must be materialized for the subsequent level*. We highlight that this in-depth enumeration method is targeted for the processing of single application steps. Indeed, Algorithm 6 assumes as input a single step represented by its array of sequential primitives. A remaining question is how to model multi-step applications, issue that we address next in Section 5.1.2.

5.1.2 From-scratch step execution

Our subgraph enumeration strategy so far is memory-efficient due to its depth-first mechanism, which allows a reduced space cost during subgraph enumeration. However, many GPM applications, including the FSM kernel, can only be efficiently implemented with multiple application steps, making incomplete the use of Algorithm 6 alone. In such scenarios, we need to design the policy for the design of multi-step applications.

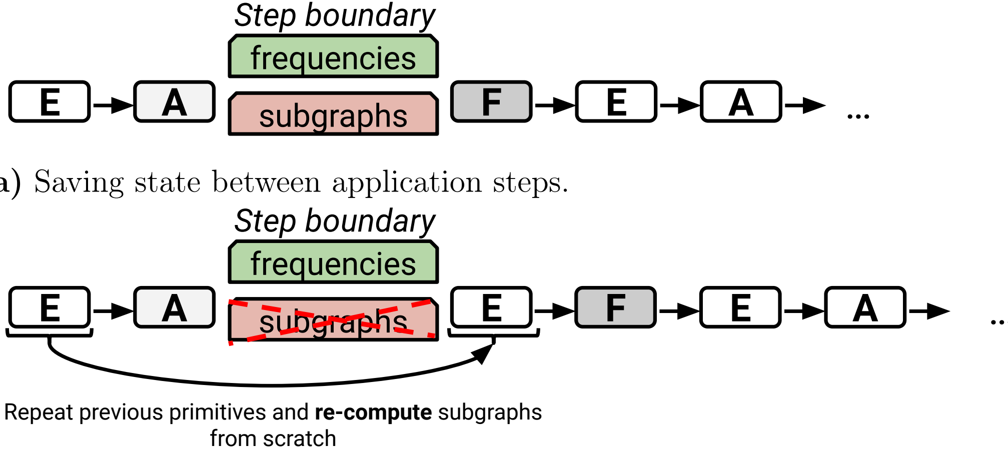
Figure 5.5 shows two alternatives for modeling such applications, using as example the two first steps of the FSM kernel. FSM is a multi-step application because it has to synchronize globally (across all partitions) to obtain patterns and their respective support frequencies in order to determine which ones meet the user-defined minimum support threshold[1]. In Figure 5.5a we see the strategy equivalent to what is used in existing general-purpose approaches such as Arabesque [121] and NScale [106]: store subgraphs enumerated at each step for quick access in the next step. In Figure 5.5b we describe our approach of not storing subgraphs enumerated between steps. In the first step, FSM kernel seeks to generate and aggregate patterns of size one (step 1: `EA-`[2]), but without keeping the generated subgraphs in memory (or disk) for the next iteration. Instead, in the second step, the method generates on demand the subgraphs of size two, but only those that were extended from frequent patterns of size one, obtained in the first step (step 2: `EFEA-`). In the third step, we have again the composition of the two former steps before the new computations (step 3: `EFEFEA-`), and this process continues while frequent patterns exists. Thereby, the computations become cumulative and we only need to store the aggregations, i.e., the actual output to the user. We denote this strategy *from-scratch processing* and next, we argue why this may be a more reliable option for graph pattern

[1] refer to Section 4.4 to review the modeling of the FSM kernel in terms of the primitives

[2] we omit primitive parameters in this notation and represent the end of steps with "-"

mining at scale.

Figure 5.5: Multi-step application design alternatives for FSM kernel. Our approach adopts a from-scratch computation to ensure a space-bounded processing.



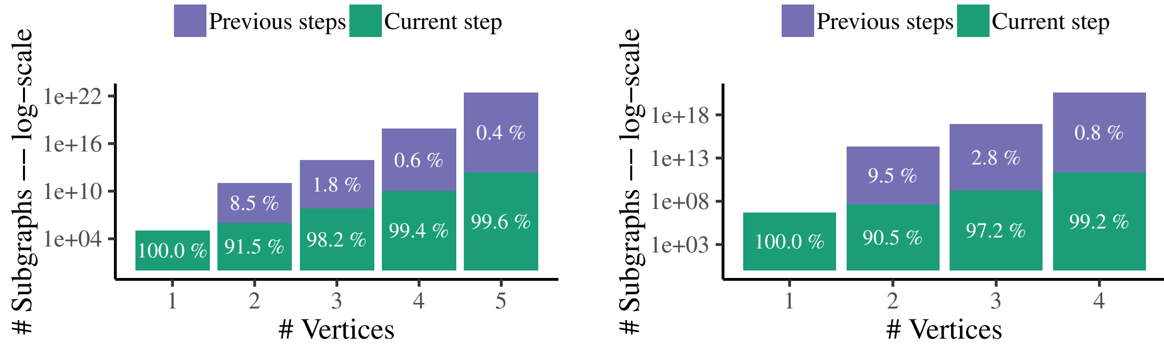
(a) Saving state between application steps.

(b) From-scratch processing: steps are recomputed on demand.
Source: Made by the author.

We propose to recompute the primitives from scratch to reduce the intermediate state of the GPM applications. This strategy essentially trades off memory for redundant processing, since we need to recompute the subgraphs from scratch. However, given the scales of subgraphs involved, (see Table 5.1), this may not be a bad trade-off since the cost of accessing precomputed subgraphs (including potentially out-of-core access) may dominate re-computation costs. In fact, we verified in the experiments and we anticipate that the cost of enumerating subgraphs (during the combinatorial explosion phase of the algorithm) dominates the execution time of a GPM task for representative real-world input graphs. In Figure 5.6 we demonstrates experimentally that usually the cost of cumulative previous steps in comparison with the cost of current step is negligible, which suggests that our strategy is indeed appropriate in this context: we get minimal benefits from optimizing previous steps.

Algorithm 7 shows how Fractal splits a sequence of primitives into steps and submits them for execution. The input of FROM-SCRATCH-EXECUTION is an array of primitives, representing the application. Primitives are assigned to the current step until a step boundary, which is marked by an aggregation primitive (line 6). Whenever a primitive matches a step boundary, the algorithm adds a copy of the current step to the pool of steps and proceeds accumulating other primitives (line 9). Thus, steps always accumulate computation from their ancestors: primitives in steps $\{0, \ldots i-1\}$ also belong to step i with the exception of the aggregation primitive. Finally, the algorithm calls the enumeration procedure (Algorithm 6) for each step built previously (lines 12 - 13). We highlight that aggregation results are not recomputed: the execution engine reuses their results on every subsequent step if necessary and once they are computed. Despite reusing

Figure 5.6: Enumeration cost of each step against the cumulative enumeration of previous steps for an exemplar GPM application - it suggests trading off memory for redundant computation can be effective.



(a) *Microsoft Collaboration Network* [34]
100K vertices and 1.08M edges

(b) *Youtube Network* [23]
4.58M vertices and 43.96M edges

Source: Made by the author.

aggregation results, this is safer than reusing subgraphs enumerated, as aggregations are supposed to summarize the results of a subgraph enumeration.

Algorithm 7 FROM-SCRATCH-EXECUTION(*primitives*)

1: *steps* ← ARRAY()
2: *step* ← ARRAY()
3: *idx* ← 0
4: **while** *idx* < LEN(*primitives*) **do**
5:     *p* ← *primitives*[*idx*++]
6:     **if** IS-AGGREGATION(*p*) **then**
7:         *step*′ ← COPY(*step*)
8:         ADD(*step*′, *p*)
9:         ADD(*steps*, *step*′)
10:     **else**
11:         ADD(*step*, *p*)
12: **for** *step* **in** *steps* **do**
13:     DFS-PROCESSING(*step*) ▷ Algorithm 6

From-scratch execution combined with the two phase depth-first subgraph enumeration provide a reliable method for designing the execution of general-purpose graph pattern mining applications. The simplicity and the effectiveness of the re-computation part may not be clear at first glance, but given the limits of modern computation models in terms of memory and processing, this strategy arises as a safer solution for large scale GPM execution, as we are going to discuss next in Section 5.1.3.

5.1.3 Space bounded parallel and distributed subgraph processing

The method proposed can be easily extended to parallel and distributed environments. Considering the architecture in Figure 5.1, a parallel implementation distributes the initial extension units among each execution thread t_j of each worker w_j, which serves as a bootstrap for the root subgraph enumerators on each thread in parallel. Thereby, Algorithm 6 can be adapted to accommodate multiple subgraph enumeration trees at once: one for each extension unit used as root.

An important advantage of the proposed solution – depth-first subgraph enumeration + from-scratch step processing – is that the execution will not crash due to *out-of-memory* errors and, hence, more memory is available for the user-defined aggregations. Moreover, the execution cost tends to be dominated by the CPU time because the memory footprint of this solution is minimal. Also, we avoid the cost of accessing precomputed subgraphs, including potentially out-of-core accesses. However, two potential concerns regarding our solution are: (1) the cost of recomputing the subgraphs from scratch and (2) it can lead to imbalance among workers, since the cost among enumeration roots tends to be skewed. In fact, trading off memory for redundant processing is beneficial since the cost of enumerating subgraphs (during the combinatorial explosion phase of the algorithm) will dominate the execution time of a GPM task. The load imbalance among workers is addressed next in Section 5.2.

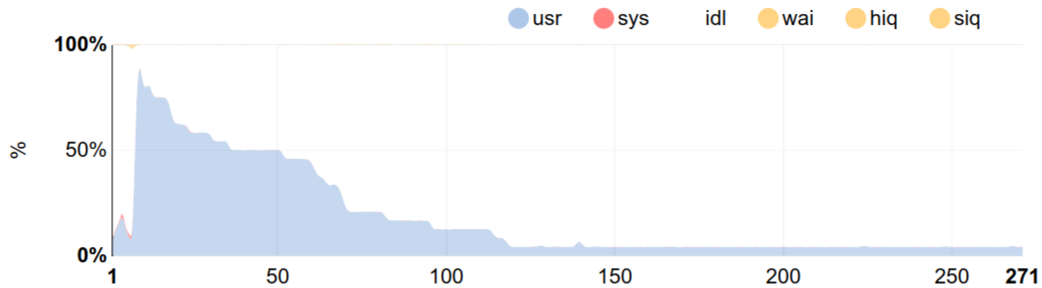
5.2 Load balancing

Our subgraph processing is efficient in terms of space and is easily parallelizable, as discussed in Section 5.1. However, graph pattern mining in practice can be tricky due to the irregular nature of degree distributions in real graphs. Therefore, naively applying the subgraph enumeration strategies introduced in Section 5.1 in real scenarios can result in serious performance bottlenecks concerning load imbalance. Considering our computation model, we would observe some worker cores finishing their part of the work almost instantly and other worker cores taking too long to complete. Furthermore, high-degree nodes tend to lie in dense regions of the graph, which exacerbates the problem (skew) as we deepen into the enumeration process in a recursive manner. In such scenarios, load-balancing becomes key to achieve both good performance and resource utilization

since standard pipelining leverages task independence to coordinate parallelism.

Figure 5.7 shows the resource utilization (CPU) when we employ the depth-first from-scratch strategy on a single machine with 28 cores for the cliques kernel with $k = 4$. Each core initially takes a partition of the graph vertices and enumerates all 4-cliques rooted by vertices in their respective partition, as we described in Section 5.1.3. We observe a critical scalability issue: the resource utilization drops very quickly as some cores finish their work early, while others keep running as stragglers for a long time (long tail).

Figure 5.7: Subgraph enumeration without any work balancing. The x-axis represents relative utilization of the worker, considering all available cores. The y-axis represents the timeline of application execution. In this case, CPU is not well utilized due to skewness: utilization drops very quickly within a few seconds of execution.



Source: Made by the author.

5.2.1 Hierarchical work stealing

In this work, we propose a *hierarchical work stealing* strategy for dynamic work balancing, improving the resource utilization of the underlying system. Our strategy is composed of two levels the first one focuses on communication within execution threads of the same worker (*internal work stealing*, or WS_{int}), and the second focuses on coordination across threads of different workers (*external work stealing*, or WS_{ext}). Naturally, thread communication within a single worker (internal) is more efficient, since access to the memory is (often) shared. On the other hand, inter-process communication is expensive because it involves serializing, sending, receiving and deserializing data structures, as we will see next. Thus, WS_{int} is always preferred to WS_{ext}.

Algorithm 8 details our work stealing strategy, executed in parallel by each execution thread t_i. We assume each thread is already initialized with the *workerId* it belongs to, references to work queues from all local cores (*threadQueues*) used for shared memory communication and references to remote workers (*workerRefs*) used for message passing

communication. The execution proceeds in trials, while unprocessed work exists (lines 4 - 6). Internal work stealing is preferred and performed whenever other threads have work to share (lines 7 - 14). Otherwise, the current thread schedules a work request to traverse other workers until some work is successfully stolen, or every worker has been queried, indicating remote termination (lines 15 - 20). Finally, the algorithm checks if some remote request is fulfilled and ready to be processed (lines 21 - 29).

Algorithm 8 Hierarchical work stealing

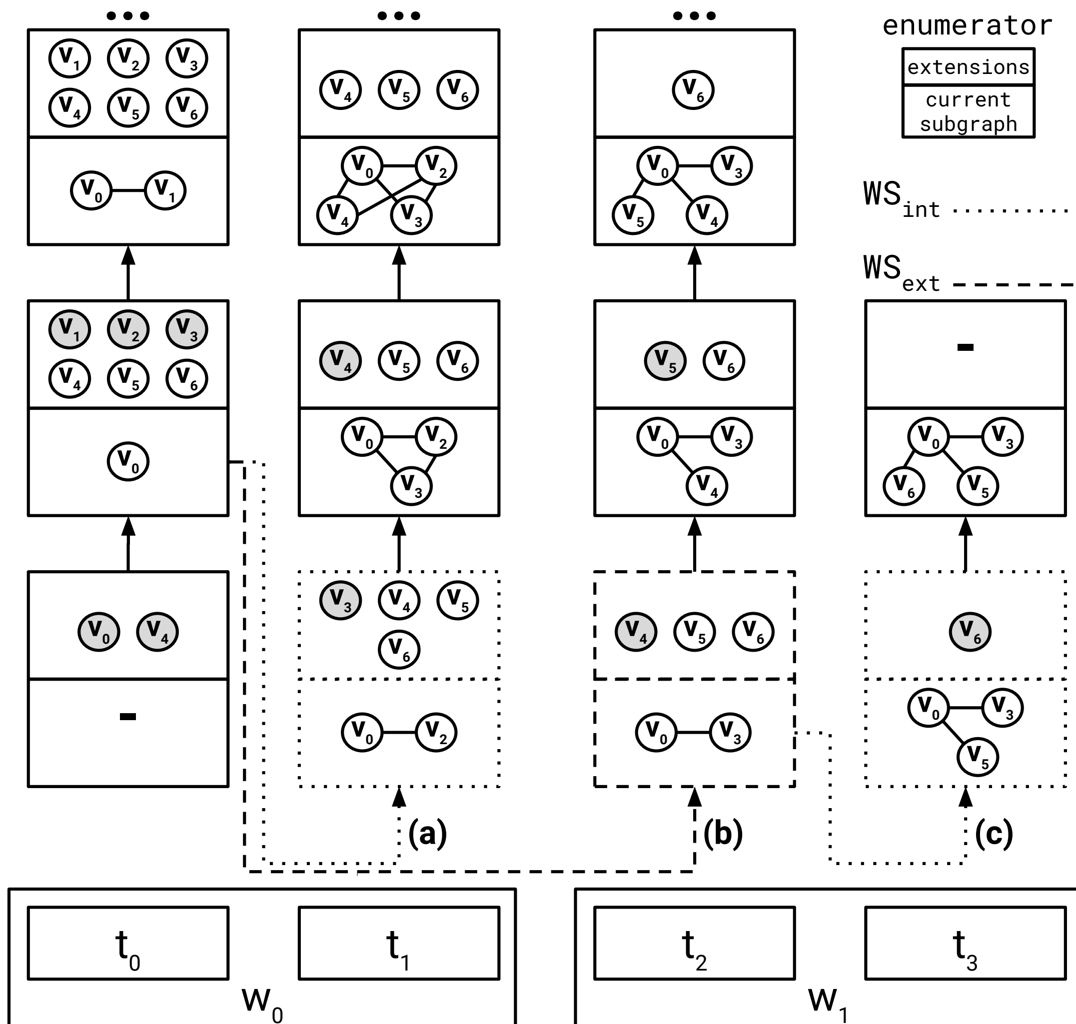
Input: *workerId* local worker ID; *threadQueues* local thread work queues; *workerRefs* worker references

```
 1: remoteResponseQueue ← FIFO-QUEUE()
 2: inFinished ← false; exFinished ← false
 3: requestsOnTheFly ← 0
 4: while not EMPTY(remoteResponseQueue) or
 5:       not inFinished or not exFinished or
 6:       requestsOnTheFly > 0 do

 7:     ▷ INTERNAL WORK STEALING
 8:     enumerator ← WORK-STEAL-LOCAL(threadQueues)
 9:     if not EMPTY(e) then
10:         PROCESS(enumerator)
11:         inFinished ← false; exFinished ← false
12:         continue
13:     else
14:         inFinished ← true

15:     ▷ EXTERNAL WORK STEALING (REQUEST)
16:     if not exFinished and requestsOnTheFly = 0 then
17:         WORK-STEAL-REMOTE(workerId, workerRefs)
18:         requestsOnTheFly ← requestsOnTheFly + 1
19:         inFinished ← false; exFinished ← false
20:         continue

21:     ▷ EXTERNAL WORK STEALING (CONSUME)
22:     enumerator ← POP(remoteResponseQueue)
23:     if enumerator ≠ NULL then
24:         requestsOnTheFly ← requestsOnTheFly − 1
25:         if not EMPTY(enumerator) then
26:             PROCESS(enumerator)
27:             inFinished ← false; exFinished ← false
28:         else
29:             exFinished ← true
```

For example, consider the execution state presented in Figure 5.8, where induced subgraphs are enumerated from the graph of Figure 4.2 in parallel. The four available threads (t's) are organized in two workers (w's) and all cores finished their original as-

signed work, except for t_0, which is busy with its own initial work. In case (a), t_1 can accommodate a WS_{int} by extending the second subgraph enumerator from t_0, since both belong to w_0. Hence, such operation generates a new subgraph enumerator with the current subgraph composed of edge (v_0, v_2) in t_1, ceasing its idleness and mitigating imbalance. In case (b), t_2 triggers a WS_{ext} because no thread in w_1 has work to share. Thus, t_2 sends a work stealing request to w_0, which in turn forwards the request to t_0. A separate thread in w_0 is then responsible for extending the first non-empty t_0 enumerator and shipping this piece of work back to the requester t_2 at w_1. Such procedure fills t_2 with a new enumerator with a current subgraph composed by edge (v_0, v_3). Finally, in case (c), t_3 leverages the previous external request to perform a low-cost WS_{int} with t_2. The result is a new enumerator with current subgraph composed of path (v_0, v_3, v_5).

Figure 5.8: Work stealing happens (a)(c) internally among threads of the same worker or (b) externally among threads of different workers. To reduce network communication, we use remote work stealing only when no other local thread has work to share.



Source: Made by the author.

Detailed operation of internal and external work stealing strategies are described in

Algorithm 9. Moreover, detailed description of how remote messages are handled among different workers is presented in Algorithm 10. Nevertheless, our method is adaptive to different workload characteristics and does not assume anything about the input distribution. Also, we verify empirically the importance of the two levels of work stealing and show significant gains on CPU utilization and system's performance.

Algorithm 9 Work stealing auxiliary functions

```
 1: function WORK-STEAL-LOCAL(threadQueues)
 2:     for ref in threadQueues do
 3:         consumer ← GET-CONSUMER(ref)
 4:         if not EMPTY(consumer) then
 5:             return consumer
 6:     return EMPTY-CONSUMER()
 7: function WORK-STEAL-REMOTE(workerId, workerRefs)
 8:     callerId ← workerId
 9:     callerRef ← workerRefs[callerId]
10:     nextWorkerId ← (callerId + 1)% LENGTH(workerRefs)
11:     nextWorkerRef ← workerRefs[nextWorkerId]
12:     SEND-REQUEST(callerId, callerRef, nextWorkerRef)
```

Algorithm 10 Work stealing message handler

Input:

workerId	local worker ID
threadQueues	local cores references
workerRefs	worker references
remoteResponseQueue	remote work waiting to be processed
msg	current message

```
 1: switch msg do
 2:     case REQUEST(callerId, callerRef)
 3:         if callerId = workerId then
 4:             SEND-RESPONSE(callerRef, EMPTY-CONSUMER())
 5:             return
 6:         consumer ← WORK-STEAL-LOCAL(threadQueues)
 7:         if not EMPTY(consumer) then
 8:             SEND-RESPONSE(callerRef, consumer)
 9:         else
10:             nextWorkerId ← (workerId + 1)% LEN(workerRefs)
11:             nextWorkerRef ← workerRefs[nextWorkerId]
12:             SEND-REQUEST(nextWorkerRef, callerId, callerRef)
13:     case RESPONSE(consumer)
14:         PUSH(remoteResponseQueue, consumer)
```

5.3 A generalized framework for static graph pattern mining

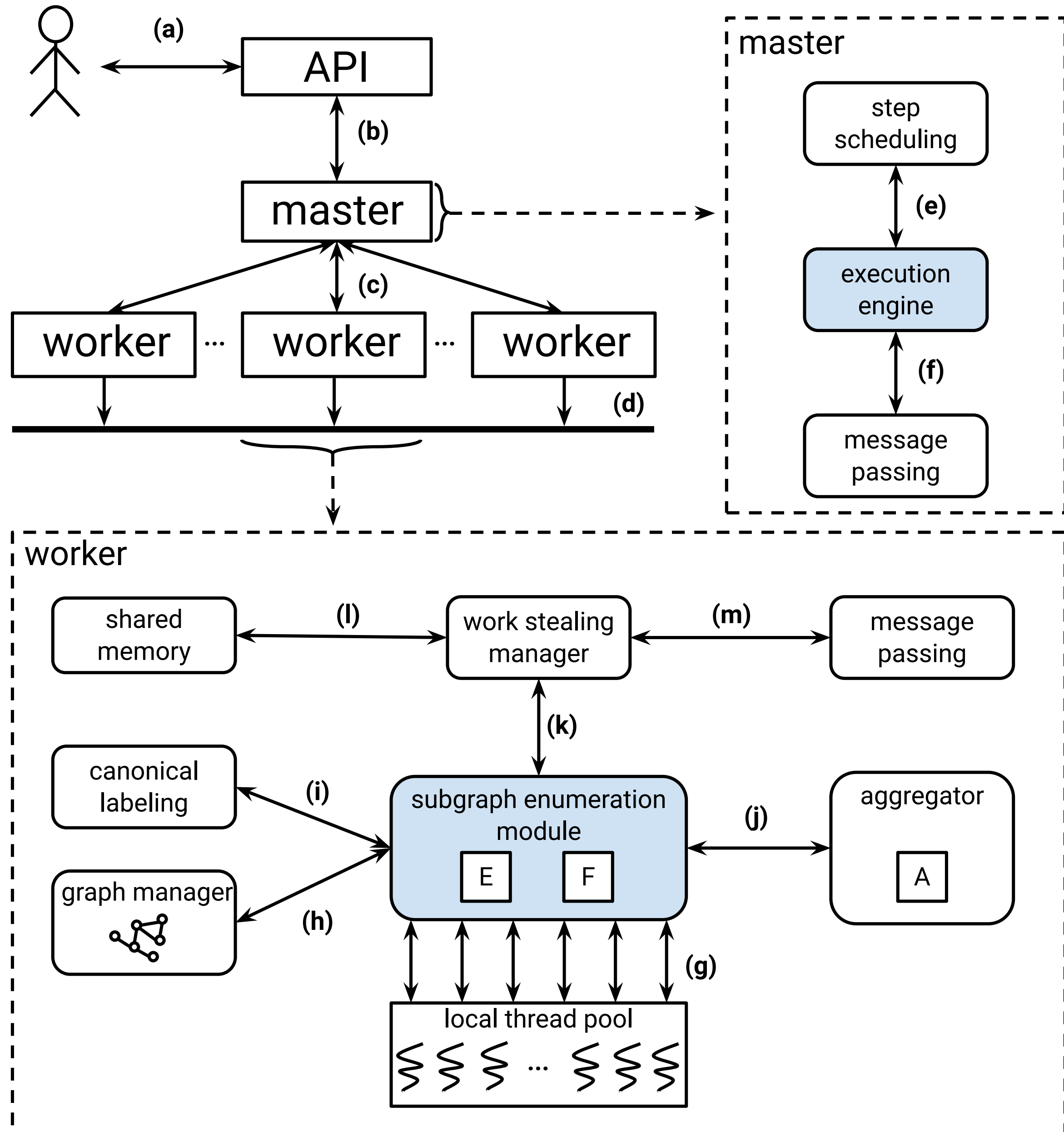
In this section we describe the design and the implementation of a generalized framework for static graph pattern mining. Our goal is to organize the modeling and optimization ideas discussed so far into a framework that facilitates graph pattern mining tasks at scale. A few features are especially important for a general-purpose framework for GPM: (1) the framework must be scalable; (2) the framework must be easy to use and provide robust abstractions to the user; (3) the framework must be portable to different languages and platforms.

Because our goal is general-purpose extensible graph pattern mining, the framework must be flexible to accommodate the execution environment considered in this work (Figure 5.1 and be robust enough to support the execution model and optimizations proposed. Figure 5.9 shows the architecture of the proposed framework. Users interact with the framework via an application programming interface (API) that enables application modeling using primitives, submission of applications, and handling of results **(a)**. A user GPM program contains two important specifications: *the input graph path and the sequence of primitives with their respective parameters*. The API translates user commands to GPM programs that can be handled and scheduled by the master **(b)**. The master is responsible for organizing the GPM programs to execution, for splitting the programs into application steps, and for submitting steps for execution in parallel on each worker **(c)**. Each worker receives a program specification including the input graph path and the sequence of primitives for execution. Each worker is responsible for reading the input graph and for dispatching a portion of work among local threads. Workers also communicate among themselves for efficient work coordination **(d)**. Next we detail the design of the master and workers.

The master in this architecture has the *execution engine* as the core element. The execution engine coordinates the initialization, execution, and termination of workers using an inter-process communication module of *message passing*. Details of which and how this module works goes beyond this discussion, and interested readers are encouraged to read Section 5.3.1 for details on how one could implement this. The execution engine receives commands from the API, calls the *step scheduling* module to split generate from-scratch execution steps **(e)**, serializes the specification and finally, uses the *message passing* module to ship user programs for execution in the workers **(f)**.

Workers have a more complex design, as they need to handle the actual GPM processing and coordination. The core of each worker is the *subgraph enumeration module*, responsible for generation subgraph candidates of interest according to the extension

Figure 5.9: Generalized graph pattern mining framework.



Source: Made by the author.

and filtering primitives that compose the current application step. The primitives executor has direct access to the worker *local thread pool*, used for GPM work tasks such as subgraph enumeration, filtering, aggregation, graph reading, but also management tasks such as inter/intra-process communication **(g)**. Before executing primitives, the worker must ensure the input graph is available in memory. The *graph manager* is responsible for reading the input graph **(h)**. During subgraph aggregation through the aggregation primitive it may be the case where the canonical form of subgraph patterns must be determined for proper pattern aggregation. The *canonical labeling* module is responsible for converting enumerated subgraphs into their canonical form for aggregation – this usually

means leveraging existing canonical labeling software such as Bliss [64] or Nauty [88] **(i)**. The actual aggregation of subgraph keys and values is performed by the *aggregator*, which reads the aggregation specifications and maintain the ongoing local aggregation store **(j)**. The aggregator provide means to generate output from a GPM execution: a key value mapping representing the aggregation primitive $A(g, h, r)$. Policies on how to maintain these outputs, i.e., in memory or in secondary storage or in distributed stores are main concerns of different implementation strategies also discussed in Section 5.3.1. Finally, the *work stealing manager* ensures that the local execution is balanced and thus, it seeks to minimize execution thread idleness and communication cost of work sharing **(k)**. Work stealing among local threads is enabled by a *shared memory* mechanism where threads can steal subgraph enumeration branches from each other **(l)**. Work stealing among different workers is enabled by *message passing* paradigm for inter-process communication, which typically only happens among processes of different workers/machines **(m)**.

Next we describe the *Fractal system*: a proof of concept of this proposed framework in the context of large-scale cloud computing platforms. Fractal is built on top of Spark [139] and can be easily deployed in distributed environments, such as a commodity cluster [27].

5.3.1 Fractal: A General-Purpose Graph Pattern Mining System

Fractal [30] is a distributed and parallel system implementation of the generalized framework for static graph pattern mining. The reference to *Fractal* as the system's name comes from the resemblance of geometric fractals with the recursive nature of subgraph enumeration algorithms. It is organized with a single application master and multiple workers, as in Figure 5.1.

The Fractal project is open-source, implemented on Scala/Java OpenJDK (JVM) 8, and its source code is publicly available[3]. Although its current version is built on top of Spark 2.0[4], the design is independent enough to fit other platforms: the only requirement is a distributed platform where workers are organized in a shared-memory model and a inter-process communication module is available among workers and master. Additionally, we extend Spark's execution model with an actor model provided by Akka 2.5.3[5], so we can support communication between workers (Fig. 5.9d). Nevertheless, we highlight

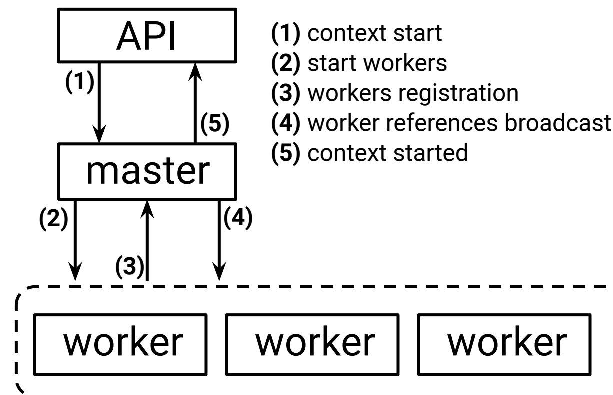
[3] http://github.com/dccspeed/fractal
[4] https://spark.apache.org/docs/2.0.0/
[5] https://akka.io

a stronger reason for leveraging Spark's model: integration and resiliency. Integration allows the system to be included in the Spark ecosystem composed of distributed storage solutions [114] and resource scheduling and multi-tenancy in large-scale processing on clusters [51, 126]. We also leverage the resiliency features of Spark, especially the *Resilient Distributed Datasets* (RDD) [139] abstraction, to export Fractal results: subgraph lists and aggregation results. In Section 5.3.1.3 we present in detail the resiliency aspect that Spark's execution models brings to Fractal. Next, we discuss the details and challenges concerning Fractal's implementation and operation: system initialization – Section 5.3.1.1, scheduling and execution – Section 5.3.1.2, primitives execution – Section 5.3.1.3, work stealing – Section 5.3.1.4, and programming interface – Section 5.3.1.6.

5.3.1.1 System initialization

Figure 5.10 shows the initialization steps of Fractal. The master is the first to start, upon a *context start* call from the API **(1)**. In this step, all the resource negotiation parameters necessary to setup the distributed environment are provided: number of workers, number of execution threads per worker, master memory upper bound, and workers memory upper bound. The next step is a start message from the master to the workers, containing all the resource parameter specifications configured during the context start **(2)**. Worker initialization includes the creation of internal structures – including a global identifier for the worker and each one of its execution threads. Because the master acts as a central point for hand-shaking among workers, each worker sends a registration message to the master, marking the end of the initialization of workers **(3)**. The remaining information each worker needs is the Akka actor reference for message passing communication with other workers. This is accomplished by a broadcast message from the master to the workers, containing all the worker references **(4)**. Thus, when the master acknowledges the registration of each expected worker, it broadcasts their addresses. After this, the broadcast resumes and the context is marked as started **(5)** Thus, every worker knows how to reach the others, in addition to the master.

Figure 5.10: Fractal initialization.



Source: Made by the author.

5.3.1.2 Scheduling and execution

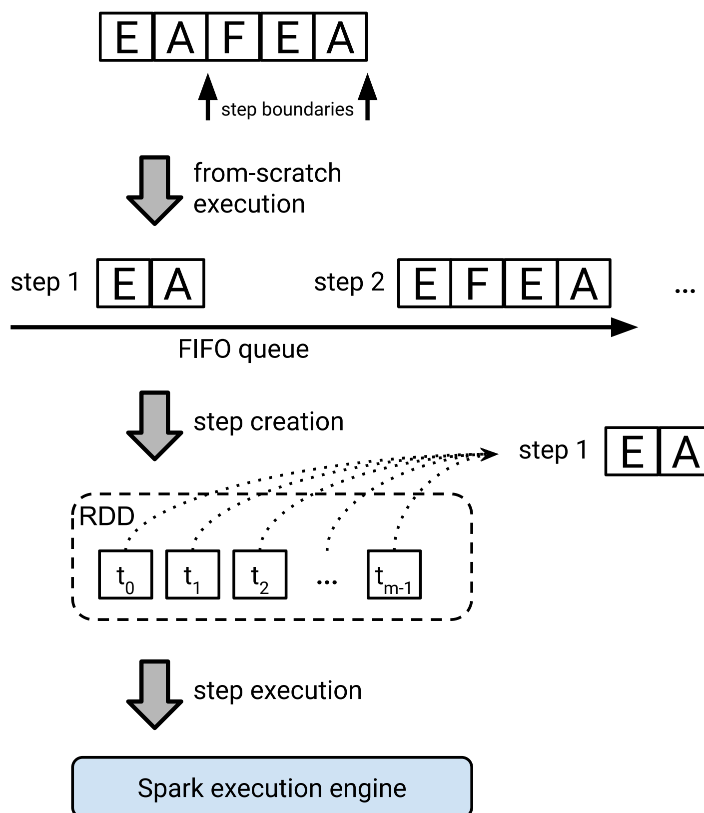
Spark's batch computation model is used to represent and to schedule application steps in Fractal. Thus, each application step corresponds to a Spark job, and the computation performed by each execution thread corresponds to the processing of a Spark partition in the underlying system. Figure 5.11 shows the scheduling operation of an application step. The input for this process is a pair of (1) input graph and (2) sequence of primitives for execution.

First, the scheduling manager determines the step boundaries of the workflow of primitives submitted. Step boundaries are determined by the from-scratch execution module, which marks as boundary where there is aggregation primitive (see Algorithm 7). In the Spark computation model, this is transparently implemented using the notion of pipelined operators [139], and in our case, pipelined primitives. Inspired by the notion of wide and narrow dependencies between distributed collections [139], this pipelined design , which allows two or more computations be executed in a single step, offers an advantage for long-running applications: it enforces global synchronization only when such event is inevitable, i.e., during *aggregations*.

Second, the from-scratch execution module creates the cumulative steps to ensure a space-bounded subgraph processing in later steps. Then, pairs of input graph and application steps are scheduled for execution in the Spark execution engine. For each execution step, we create an RDD composed of m partitions, one for each available execution thread in the environment (Figure 5.1). Thereby, partition identifiers in Spark correspond to the thread identifiers in Fractal. This part covers all communication requirements between master and workers (Figure 5.9c).

A step execution in the workers starts by checking whether the input graph is

Figure 5.11: Fractal step scheduling.



Source: Made by the author.

already in-memory, in case it is not the system asks the local graph manager start reading the input graph. Input graphs may be stored on the local file system or on HDFS [114]. After that, the worker is ready to start executing the sequence of primitives submitted as an application step. Thus, each execution thread in the system executes the same instruction pipeline, over different portions of the data.

5.3.1.3 Implementing GPM primitives in Fractal

Primitives in Fractal are implemented by pipelining Scala functions, which are combined, serialized and shipped for execution in the workers. To accomplish the proper execution ordering, Fractal makes sure the output of an earlier primitive is used as input to the next primitive. For instance, an extension (E) followed by a filtering (F) means that, each valid subgraph produced by the extension is verified by the downstream fil-

tering function. In a distributed setting, each execution thread of every worker starts the execution over an empty subgraph enumerator and an initial partition of extension units from the input graph, determined on-the-fly using its unique thread identifier. For both vertex- and pattern-oriented extension types (T_V or $T_P(\rho)$), the initial extensions are single vertices, while edge-induced extension type use single edges (T_E, in Figure 4.2).

The aggregation primitive is a bit more elaborated than that, because it has the potential of becoming the performance bottleneck in case heavy or skewed aggregations. Fractal adopts the *two-level pattern map* introduced in Arabesque [121], but specifically designed to work on any type of keys and values. Specifically, local aggregation (Definition 10) is performed by each execution thread in the system. Then, partial aggregations from each execution thread are flushed down to Spark shuffling system for final aggregation as distributed datasets of pairs composed of keys and values. More discussion on this reliable downstream processing is given in Section 5.3.1.5.

5.3.1.4 Implementing work-stealing in Fractal

We implement work stealing directly over the subgraph enumerator abstraction (see Figure 5.4). In particular, we make the extension function (`extend()`) thread-safe and efficient, to allow a fine-grained work sharing among execution cores. Upon an `extend()` call, Fractal copies the subgraph prefix, consumes an extension (thread-safe), and adds the new extension to the prefix of the new enumerator. Because we end up with a very short critical section (consumption of extensions), work stealing in Fractal comes with a small overhead and little contention among execution threads. Indeed, the depth-first enumeration maintains one enumerator per extension level, which can be locked and consumed independently. Subgraph enumerators also facilitate work sharing among distributed workers: a subgraph enumerator (prefix) represents a unique independent piece of work that can be shipped to any worker for processing.

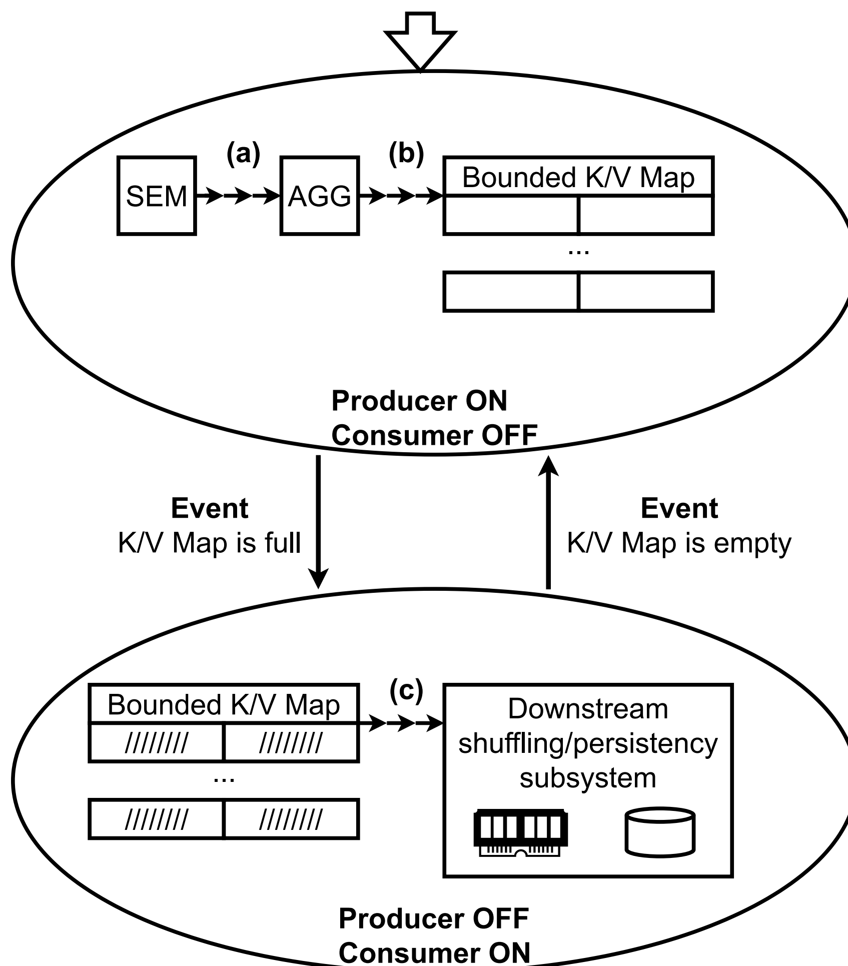
We highlight that work stealing messages are completely asynchronous and do not interfere with the ongoing execution: extra threads are dedicated to the coordination process.

5.3.1.5 Enabling reliable downstream processing in Spark

Reliable aggregation is an important feature for distributed systems that handle huge amounts of data. Specifically, one must be able to reduce the working set of the processing pipeline as soon as possible to mitigate performance bottlenecks concerning memory management and network communication [29]. In general, filtering primitives in subgraph enumeration applications can be used for that purpose. However, aggregation primitives may still suffer from huge working sets in cases where the selectiveness of filtering is suboptimal. In this scenario, the cost of materializing working sets (subgraphs) in-memory for downstream aggregation and processing in complex data analytics pipelines can easily become impractical. On the other hand, data-parallel systems usually adopt streaming immutable models [139] that prevent user's application from reusing objects or even performing inplace computations for conscious use of available memory. Actually, such computation model limitation is the main reason for performance overheads in subgraph enumeration applications running over standard data-parallel models [26].

Given this challenge, we defend that any distributed subgraph enumeration system should enable both: (1) efficient subgraph enumeration; and (2) efficient subgraph aggregation. The first feature is assessed via memory-efficient subgraph enumeration primitives (Section 5.1) along with optimized resource utilization (Section 5.2). The second feature is implemented using a reliable subgraph aggregation routine that alternates between a producing phase, where aggregation buffers are filled; and a consuming phase, where aggregation buffers are consumed and dispatched for downstream processing by a data-parallel framework (in Fractal we leverage Spark for this). Figure 5.12 describes this operation. The operation starts with a producer thread running and a consumer thread waiting for produced key/value items from aggregation. Specifically, the subgraph enumeration module (SEM) generates subgraph candidates one by one as a stream **(a)**. For each subgraph generated, the aggregator module (AGG) applies the aggregation primitive to the current subgraph and forwards a key/value aggregation item to a *bounded key/-value map* **(b)**. The boundness of this map is essential in our design since our goal is to prevent this map from growing indefinitely and causing out of memory crashes. Thereby, whenever this map fills entirely a change of state is triggered where the producer stops generating new subgraph candidates, goes to a waiting state, and immediately after that the consumer thread starts running. The consumer is then responsible for taking each key value item produced and aggregated in the intermediate bounded map and for forwarding this item to downstream shuffling or persistency subsystem **(c)**. This module is implemented by data-parallel systems such as Spark, which leverages primary and secondary storage to produce outputs (e.g. for persisting a key/value distributed datasets) or to reorganize key/value items for a distributed data shuffling [139].

Figure 5.12: Reliable downstream processing of subgraph aggregations



Source: Made by the author.

An important advantage of this approach is that fixed-sized buffers prevent the aggregation process from out-of-memory errors. Also, because Fractal alternates between producing and consuming states in batches, it is feasible to tune the size of the bounded key value map to an adequate value given the amount of memory available for execution. The outcome is that, from the point of view of a practitioner, the output of a subgraph aggregation is produced reliably and totally partitioned as distributed datasets, which can be used as input to other data analytics pipelines including machine learning algorithms [91], real-time streaming processing [6], and SQL-like distributed processing [7].

5.3.1.6 Fractal programming interface

Designing a flexible and expressive API for distributed GPM applications is challenging. An API is expressive when it is easily readable and interpretable, and it is flexible

when it is capable of representing a wide range of applications. Fractal's API is *subgraph-centric* [121, 106, 17] in that it exposes a small set of intuitive and modular operators to construct complicated GPM applications. All operators act on a state object, called a *fractoid*.

A fractoid holds the state of a Fractal application during the execution process. Such state includes an array of primitives representing the user workflow and any aggregation result required for computation. One can derive a fractoid from either another fractoid or from the input graph. Fractal supports three types of fractoids – edge-induced, vertex-induced and pattern-induced – matching that correspond to the extension type of interest (T_E or T_V or $T_P(\rho)$). The choice of which type to use is application dependent. and, therefore, must be chosen by the data analyst at the beginning of a Fractal application. which must be chosen by the user according with her GPM application. Since the type of a fractoid also determines which extension type the system will employ to enumerate the subgraphs, it must be defined by the user at the beginning of her application. By default, fractoids also carry default implementations of extension methods for generating all canonical extensions from a search space (M_C or $M_P(\rho)$). Figure 5.13 shows the standard Fractal API.

The entry point to an application is the `FractalContext`, responsible for configuring and initializing all the required resources to build and run Fractal routines. Since our current implementation runs on top of Spark [139], we instantiate a `FractalContext` (`[FCTX-INIT]`) directly from a `SparkContext`. In order to obtain the first fractoid, the user must first create a fractal graph from the context by passing an input path (`[FCTX-IG]`) and then ask for a vertex-induced fractoid (`[FG-V]`), an edge-induced fractoid (`[FG-E]`) or a pattern-induced fractoid (`[FG-P]`) (see Listing 5.1 for a vertex-induced example).

Listing 5.1: Initialization of a vertex-oriented Fractal application.

```
1 val sc = new SparkContext(..)
2 val fctx = new FractalContext(sc)
3 val graph = fctx.inputGraph(graphPath)
4 val vfrac = graph.vfractoid()
```

Workflow operators are used to describe the processing performed over fractoids in order to explore the space of solutions (subgraphs) or must be explored in the input graph. The `expand` function (`[FRAC-E]`) represents the extension primitive (E) and enumerates subgraphs by extending the subgraphs given as input. Considering a GPM application that uses the edge-induced extension method, an m-expansion over k-edge subgraphs generates all (and unique) subgraphs of size $(k + m)$ edges. The filtering primitive (`[FRAC-F]`) is used to prune the search space based on local information (e.g. clique kernel) or input from previous steps (e.g. frequent subgraph mining kernel) – invalid subgraphs are not further expanded and explored. Our next workflow operator (`[FRAC-REPEAT]`) is used

Figure 5.13: Fractal: Standard API

```
// [FCTX-INIT]
new FractalContext(sc)

FractalContext {
    // [FCTX-IG]
    def inputGraph(path: String): FractalGraph
}

FractalGraph {
  // [FG-V]
  def vfractoid(): Fractoid // T_V, M_C

  // [FG-E]
  def efractoid(): Fractoid // T_E, M_C

  // [FG-P]
  def pfractoid(p: Pattern): Fractoid // T_P(ρ), M_P(ρ)
}


Fractoid {
    // [FRAC-E]
    def expand(n: Int): Fractoid // E(T, M)

    // [FRAC-F]
    def filter(s: Subgraph): Fractoid // F(p)

    // [FRAC-REPEAT]
    def repeat(n: Int): Fractoid

    // [FRAC-A]
    def aggregate(key: Subgraph => K,
                  value: Subgraph => V,
                  reduce: (V,V) => V): RDD[(K,V)] // A(g, h, r)
}
```

Source: Made by the author.

to keep applications clean and concise. This operator chains a workflow fragment n times, simplifying the implementation of iterative algorithms.

The operator `[FRAC-A]` is used to get the result of the current workflow according to a given aggregation plan: `key`, a function to obtain an aggregation key from the processed subgraph; `value`, a function to obtain an aggregation value from the processed subgraph; and `reduce`, a function used to combine values sharing the same key. The result of an aggregation is exported as a Spark key-value RDD, which enables downstream processing in an integrated data-parallel environment.

Fractal's API allows an intuitive and interactive experience since every partial

result of a workflow (*fractoids*) can be easily executed and verified separately. Users can combine any sequence of primitive components and perform successive refinements in their analysis. Existing systems lack such support since they view applications as atomic jobs waiting to be executed in batch mode [106, 59]. Next we present the implementation of a few kernel implementations.

Example 5.3.1. *(POSE k-MC)* Listing 5.2 shows the code excerpt for motif counting using a pattern-oblivious paradigm. The algorithm establishes that the subgraphs will be induced by vertices, by calling `vfractoid`. Next, we expand the initial empty fractoid by generating all unique subgraphs with k vertices using the call `expand(k)`. Then, we use the `aggregate` operator to configure an aggregation represented by pairs of *pattern::count*. More specifically, we must specify the aggregation key (`s.pattern()`), the initial value one (`1L`) and the reduction summing function. The output is an RDD of pairs containing patterns (motifs) and their respective frequencies in the input graph.

Listing 5.2: Motifs application.

```
1 val fc = new FractalContext(sc)
2 val fg = fc.inputGraph(path)
3 val n = fg
4   .vfractoid() // vertex-induced subgraphs
5   .extend(k) // k-vertex induced subgraphs
6   .aggregate(s => s.pattern(), s => 1L, _ + _) // count by pattern
7   .reduceByKey(_ + _) // Spark API: finish aggregation
```

Example 5.3.2. *(POSE k-CL)* Listing 5.3 shows the code except for clique counting using a pattern-oblivious paradigm. A vertex-induced workflow is built with primitives $(EF)^k$: k extensions, each one of those followed by a cliques checking, ending with an aggregation. In this case, all values `1L` for counting have the same arbitrary key `0L` because we are interested in the absolute counting. By having the partial counts as pairs `(0,count)`, we are able to finish counting using the standard Spark API that finishes aggregation of partial counts.

Listing 5.3: Motifs application.

```
1 val fc = new FractalContext(sc)
2 val fc = new FractalContext(sc)
3 val fg = fc.inputGraph(path)
4 val n = fg
5   .vfractoid() // vertex-induced subgraphs
6   .extend(1).filter(isClique)
7   .repeat(k) // k-cliques
8   .aggregate(s => 0L, s => 1L, _ + _) // (0,count)
9   .values.sum() // Spark API: sum all values
```

5.4 Evaluation

In this section we evaluate Fractal, a proof of concept distributed and parallel system that implements our GPM modeling proposal (Chapter 4). We evaluate Fractal from a perspective of system's runtime performance and we do not consider in our study other important evaluation aspects such as energy consumption in parallel architectures, which we highlight as promising future work in Section 7.1. In particular, our goal is to show that: (Section 5.4.1) Fractal is competitive in terms of performance efficiency compared to general-purpose systems and specialized algorithm solutions; and (Section 5.4.2) Fractal's design features are effective in providing a reliable, resource efficient, and scalable GPM executions.

Hardware. All experiments, unless otherwise specified, were run on a cluster with 10 machines, each one having an Intel Xeon E52680 with hyperthreading (14 cores, 28 execution threads) and 25 MB cache, 500 GB RAM, running CentOS Linux 3.10. The machines were connected by Gigabit Ethernet.

Datasets. In Table 5.2 we describe the graphs used in our evaluation. Note that such datasets were also used in previous works in order to evaluate graph mining algorithms and systems [34, 1, 121]. In Mico [34], vertices are authors (labeled with their research field) and edges represent co-authorship. Patents [48] has patents published in US as vertices and their citations as edges; the labels on vertices are given by the year in which the patents were released. Youtube [23] contains videos posted from February 2007 to July 2008. In this graph, there is an edge between two vertices if their videos are related. The label of a vertex is computed by combining the video's rating and length. Orkut [134] is a single-labeled network with vertices representing users, and edges, friendships among them. Throughout this section we refer to these graphs by their name followed by a suffix indicating whether that specific version is single-labeled (-SL) (i.e. effectively unlabeled graphs) or multi-labeled (-ML). Confidence intervals are presented for a confidence of 95%.

Table 5.2: Fractal evaluation: real-world datasets used in the experiments.

	$\|V(G)\|$	$\|E(G)\|$	max.deg.	avg.deg.	labels
Mico (MI) [34]	100K	1M	1.3K	22.3	29
Patents (PA) [48]	2.7M	13.9M	789	10.1	37
Youtube (YO) [23]	4.5M	43.9M	2.5K	19.1	108
Orkut (OR) [134]	3.07M	117.1M	33.3K	76.2	-

JVM-based Baselines. We compare Fractal (a JVM-based system) with several specialized JVM-based distributed algorithms including those for Motifs (MRSUB [110]), subgraph querying (SEED [76]) and Cliques (QKCount [39]). We also compare Fractal with general-purpose JVM-based systems, such as Arabesque [121], and GraphFrames [26] and GraphX [45] where possible. The GPM kernels used are implementations of problems described in Section 2.3.

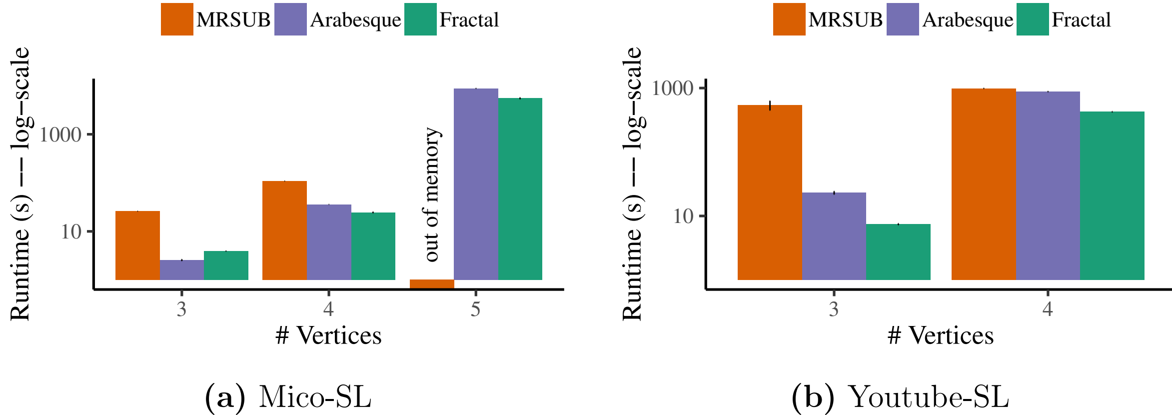
5.4.1 Fractal: Comparative Performance

Fractal supports multiple GPM kernels at once, while most existing GPM systems are kernel-specific or lack the programming flexibility and improved performance of Fractal. In this section, we evaluate four kernels – motifs, cliques, FSM, and subgraph querying – considering a mixture of general-purpose GPM systems and kernel-specific distributed implementations in each scenario. In this section we leverage available baseline systems to compare against implementations in Fractal.

Unlabeled motifs (k-MC). Figure 5.14 compares the performance of Fractal with baselines (Arabesque and MRSUB) on the Motifs benchmark. Fractal results consider a $POSE$ solution to k-MC. Considering the single-labeled input graphs (Mico-SL and Youtube-SL), we observe that when the amount of work is small, Arabesque outperforms Fractal (see Mico-SL for 3-MC). Fractal pays a small *setup overhead* to support its work stealing environment and such overhead becomes significant when the amount of work is small. However, Fractal becomes more efficient as we target larger subgraphs (4-MC and 5-MC) or when a larger network is involved (Youtube-SL), obtaining a speedups of up to 1.6$\times$ for Mico-SL and 3.10$\times$ for Youtube-SL. MRSUB, a recent specialized approach performs worse than the other two methods across the board (running out of memory in one instance).

Labeled motifs (k-MC). For labeled graphs, we compare the same $POSE$ implementation of motifs kernel in Fractal against Arabesque system that supports multi-labeled motifs. Despite runtime comparison we highlight in this experiment how labeled graphs may increase the number of patterns for counting. Figure 5.3 shows the results. Fractal's load balancing and from scratch computation contribute to an efficient usage of resources, allowing more available memory for mining patterns (up to $855,010$) and improving the overall runtime by up to three orders of magnitude. When we have Mico-ML as input, we see an interesting behavior. The number of mined patterns is huge due to the num-

Figure 5.14: Motifs runtime on Mico-SL and Youtube-SL.



(a) Mico-SL **(b)** Youtube-SL

Source: Made by the author.

ber of labels in the network, which significantly hurts Arabesque's performance. More specifically, Arabesque took 2.02× longer to finish for size 3 motifs and failed due *out-of-memory* error for motifs with 4 vertices and larger, while Fractal completes in 78.12 seconds. Indeed, Arabesque has to keep one ODAG for each one of the 855,010 patterns with 4 vertices, which overwhelms the available memory causing the observed error (bottleneck). Even when the available space is enough to keep the ODAGs in memory, the cost to broadcast this intermediate state between every superstep dominates Arabesque's performance. This is exemplified by Fractal's speedup of 570.57× on Youtube-ML. We drill down further on such memory issues in Section 5.4.2.1. Such gain is an outcome of our reliable subgraph processing with bounded space required (Section 5.1) along with efficient aggregation design in Fractal (Section 5.3.1.5).

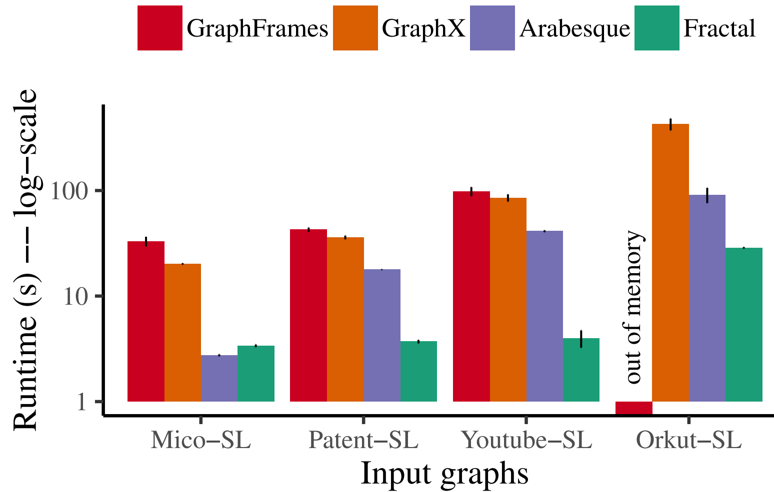
Table 5.3: Motifs runtime on Mico-ML and Youtube-ML.

Graph	Instance	# Patterns	Arabesque (seconds)	Fractal (seconds)
Mico-ML	3-*MC*	16,593	13.46 ± 0.07	6.65 ± 0.28
	4-*MC*	855,010	*out-of-memory*	78.12 ± 0.49
Youtube-ML	3-*MC*	636,947	$17,613.57 \pm 390.81$	30.87 ± 0.47

Triangles (3-*CL*). We next examine the performance of the triangle counting application on Fractal, Arabesque, GraphFrames and GraphX in Figure 5.15. Fractal implementation for triangles (3-*CL*) considers a *POSE* paradigm implementation. Fractal significantly outperforms the competing methods on three of the four datasets (up to an order of magnitude better), while being slightly slower than Arabesque on the smallest dataset due to setup overhead.

Cliques (*k*-*CL*). We evaluate the cliques application ($k > 3$) on Fractal and baselines (Arabesque, GraphFrames and QKCount) in Figure 5.16. Fractal's implementation uses

Figure 5.15: Performance on Triangles benchmark. GraphFrames ran out of memory for Orkut-SL.



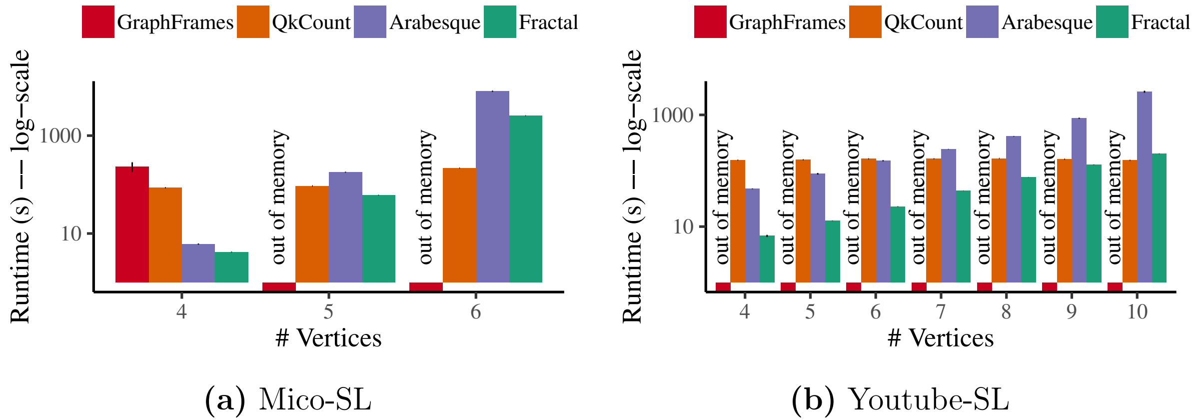
Source: Made by the author.

the same *POSE* paradigm approach as used for triangles. Fractal outperforms Arabesque in almost every scenario. On Youtube-SL the performance gains are even more obvious (see Figure 5.16b). Fractal obtains speedups that range from 5.19× to 12.87× against Arabesque in all configurations considered. On this larger dataset, since Arabesque has to keep the subgraphs (compressed in special data structures called ODAGs) from one step to another, this imposes extra memory and network costs to maintain that information consistent among workers. Arabesque, however, is able to mine 3-cliques faster than Fractal on Mico-SL (again due to the setup overhead).

Fractal competes well with the state-of-the-art, QKCount (a distributed algorithm for clique counting implemented over MapReduce/Hadoop [27]), outperforming it on many settings, while being slower on Mico-SL for cliques of size six. Fractal's mechanism to control memory pressure, efficient work stealing and its method to leverage pipelined computations and extension primitives (described in Chapter 4) allow it to compute cliques efficiently, without the need to keep any intermediate state. QKCount, on the other hand, must generate a huge intermediate state of subgraphs to perform enumeration using a MapReduce computing framework, which introduces substantial overhead of external memory accesses. Such design simplifications lead to performance competitive with the state-of-the-art.

FSM (k-*FSM*-α). We evaluate the performance of the FSM application implemented over Fractal, Arabesque, and ScaleMine [1], a high-performance specialized implementation. In this experiment Fractal's implementation approach is *POSE*. Scalemine relies on a two phase approach: in the first phase it estimates search-space loads and uses that information for load balancing in the second phase. While Scalemine produces exactly

Figure 5.16: Cliques runtime on Mico-SL and Youtube-SL. GraphFrames often ran out of memory.

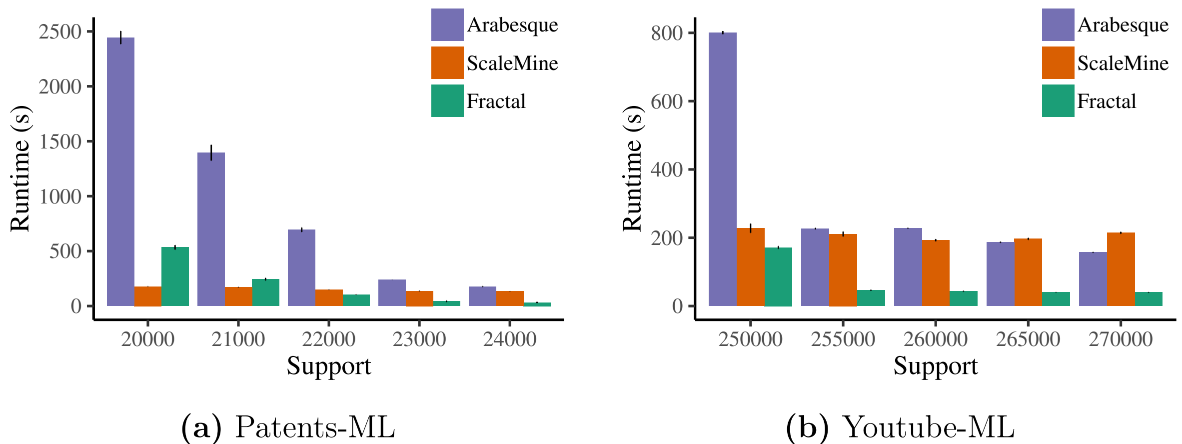


(a) Mico-SL (b) Youtube-SL

Source: Made by the author.

the same set of frequent patterns, as Fractal and Arabesque, it does not visit all the subgraphs of a given pattern (*i.e.*, the frequency counts cannot be exact). For this set of experiments, we consider two labeled graphs and we vary the minimum support of the algorithm (see Figure 5.17).

Figure 5.17: FSM performance. Fractal's stateless characteristic allows competitive scalability with Scalemine.



(a) Patents-ML (b) Youtube-ML

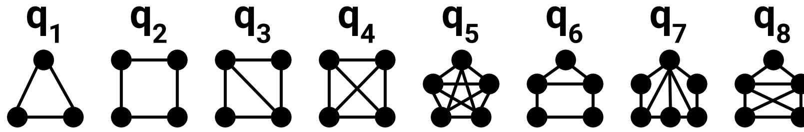
Source: Made by the author.

While Fractal has the initial setup overhead (for work stealing), Scalemine's first phase (also used for load estimation) can be quite expensive especially when there is less overall work [1]. Fractal's stateless operation w.r.t. intermediate state provides a better scalability against Arabesque, showing speedups of up to 4.57× (when the support is 20k). For higher values of support, it outperforms ScaleMine (in spite of being an exact algorithm) due its fast and balanced enumeration strategy, achieving a speedup of 4.12× when the support is 24k. For lower values of support, Scalemine outperforms Fractal. This is a surprisingly good result for Fractal, since Scalemine is implemented on C++

with MPI, demonstrating the effectiveness of work stealing, limiting memory pressure and graph reduction optimizations within Fractal.

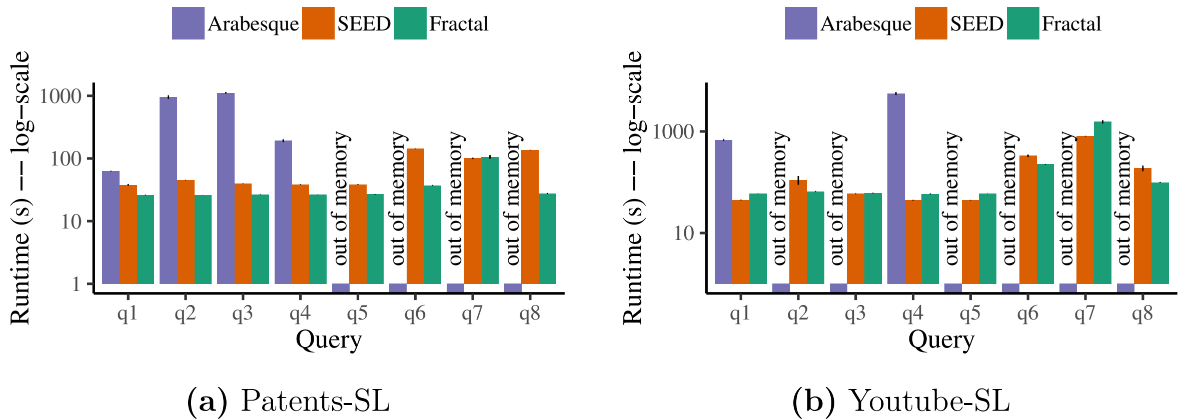
Pattern Querying (ρ-PQ). In Figure 5.19, we evaluate the performance of the pattern querying application on Fractal, SEED and Arabesque. SEED is the state-of-the-art specialized subgraph enumeration system implemented over Hadoop, which computes larger subgraphs by joining smaller ones. We use the same queries supported by SEED [76] to evaluate our system (see Fig. 5.18). We implemented the same queries in Arabesque, for comparison with a general purpose approach for GPM. Fractal's implementation in this experiment leverage the $PASE$ paradigm since a pattern query is easily expressed as a subgraph enumeration routine that lists subgraphs isomorphic to a given unique reference pattern.

Figure 5.18: Queries for pattern querying evaluation.



Source: Made by the author.

Figure 5.19: Fractal outperforms SEED in most configurations, except for those where SEED's execution plan leverages the overlapping substructures (e.g., q_7).



(a) Patents-SL **(b)** Youtube-SL

Source: Made by the author.

Considering the Patents-SL graph (Figure 5.19a), SEED outperforms Fractal only for q_7, because the execution plan generated in this case is most effective. Specifically, SEED computes the matches of the pattern $q3$ and joins them to obtain q_7, reducing significantly the subgraph enumeration cost. Meanwhile, Arabesque executions finish successfully only for queries that are easier to enumerate (q_1 and q_4) or have fewer edges (q_2 and q_3). The other executions fail with out-of-memory errors since the number of subgraphs and their sizes start to pressure the memory, even for compressed representations like Arabesque's ODAGs. Fractal's pattern-aware subgraph exploration (PASE)

and stateless enumeration allow a more efficient pattern querying, specially compared to edge-induced approaches like Arabesque, across the board.

On Youtube-SL (Figure 5.19b), SEED also performs best when the query allows an execution joining plan with overlapping structures. Indeed, SEED outperforms Fractal for cliques (q_1, q_4, and q_5) and for q_7, because of that pattern's symmetry. In the remaining configurations, Fractal outperforms SEED (q_2, q_6, and q_8) or remains competitive (q_3). Overall, again Fractal demonstrates competitive performance with a state-of-the-art specialized baseline.

5.4.2 Fractal Drilldown

Fractal's key systemic contributions (memory demand reduction, hierarchical work stealing and graph reduction) were found to be useful across a range of GPM kernels. We next drill down on some of those in turn, with specific kernels.

5.4.2.1 Memory footprint analysis

In this section we drill down on the memory costs of a couple of applications w.r.t the memory optimization facilitated by Fractal's computation model. Specifically, our goal is to evaluate how an efficient subgraph processing (Section 5.1) and a carefully designed subgraph aggregation and downstream processing (Section 5.3.1.5) are essential to high-performance distributed GPM computation. Our metric is the *average memory usage* among all workers in the execution. A lower value of this metric indicates a better memory footprint, *i.e.*, less prone to *out of memory* errors or long garbage collection pauses (which may cause performance degradation and unpredictability).

We consider the following applications for this experiment: *(1) k-CL*, representing applications in which the enumeration phase is the bottleneck; and *(2) k-MC*, an application that not only enumerates all subgraphs up to a given depth but has to perform expensive isomorphic checks and to aggregate pattern counts. Both Fractal implementations are *POSE*. We use Arabesque as baseline for this analysis since it is the only distributed system for general-purpose GPM with source code available. Table 5.4 summarizes our results.

For the first scenario, we measure the memory footprint of the cliques application

Table 5.4: Memory per worker.

Dataset	Instance	Arabesque (GB)	Fractal (GB)	×
Youtube-SL	3-*CL*	8.5 ± 1.9	10.5 ± 0.2	0.8×
	4-*CL*	10.6 ± 1.8	10.5 ± 0.2	1.0×
	5-*CL*	16.7 ± 2.6	10.9 ± 0.2	1.5×
	6-*CL*	18.9 ± 2.3	10.9 ± 0.2	1.7×
Youtube-ML	3-*CL*	22.9 ± 1.2	10.9 ± 0.1	2.1×
	4-*CL*	57.5 ± 1.4	12.8 ± 0.1	4.5×
	5-*CL*	117.4 ± 1.4	11.8 ± 0.1	10.0×
	6-*CL*	204.3 ± 1.1	11.6 ± 0.0	17.6×
Mico-ML	3-*MC*	0.2 ± 0.0	0.4 ± 0.0	0.6×
	4-*MC*	1.8 ± 0.3	0.4 ± 0.0	4.9×
	5-*MC*	46.9 ± 1.0	0.9 ± 0.3	49.9×

over a single-labeled version of the Youtube graph (Youtube-SL). We see that initially Arabesque performs better than Fractal (0.80× Fractal's memory requirements), attributed to the additional structures used by Fractal to coordinate the execution cores such as work-stealing routines. However, while Fractal exhibits a fairly constant memory cost across all evaluation depths (between 10.48 and 10.87 GB), Arabesque significantly increases its memory requirements when executions require a deeper exploration level and, then, more subgraphs (ODAGs) are kept across computation steps, which imposes significant additional cost for the executions regarding memory management and data shuffling (massive network communication). Fractal achieves reduction factors of 1.01× to 1.74× w.r.t Arabesque.

Next we consider the multi-labeled network, Youtube-ML. Fractal is able to keep the memory requirements relatively constant (range from 10.88 and 12.84 GB). Some variation is expected due to the non-deterministic behavior of the Java Virtual Machine (*e.g.*, garbage collection) in multi-threaded environments (range from 10.88 and 12.84 GB). Meanwhile, in Arabesque, another GPM system that levers JVM, we see a significant increase in the memory used by workers, which is a direct outcome of how the system keeps its intermediate state across enumeration depths. Specifically, subgraphs are kept in the memory of each worker in a compressed data structure (ODAG) per pattern [121]. As there are more patterns templates in a multi-labeled network, Arabesque must keep more ODAGs in memory, increasing the working memory of the workers. In particular, the workers of the baseline system require an average of 204.28 GB of memory in the enumeration depth of five, while Fractal needs only 11.58 GB. This represents a reduction factor of 17.64× regarding memory.

Finally we consider the Motifs application. In this example, we show that the intermediate state of workers significantly grows in the baseline system as we increase the

exploration depth (even for moderately sized) graphs. Indeed, the amount of memory used by Arabesque increases 49.86×. On the other hand, Fractal's executions require no more than 0.94 GB of memory per worker (on average).

5.4.2.2 Hierarchical Work Stealing

We start by revisiting the 4-cliques use-case of Section 5.2 to show how resource utilization (CPU) can be significantly improved by employing the work stealing strategy described in this section. Figure 5.20 presents the new CPU utilization of Fractal, as a counterpoint to the naive strategy described previously and evaluated in Figure 5.7. We can see that the CPU utilization remains high during the whole execution, which results in a new total runtime of 73.8 seconds, representing 3.58× speedup compared to the naive solution.

Figure 5.20: 4-*CL* with and without dynamic work balancing. Good CPU utilization throughout the whole execution. Runtime: 73.8*s*.

(a) Work stealing OFF

(b) Work stealing ON

Source: Made by the author.

Next we detail the benefits of *hierarchical work stealing* environment within Fractal. We focus on the FSM algorithm, which is a multi-step application and, consequently, has the potential to exhibit a richer per-level behavior. The input graph considered is Patents-

ML and we set the support to $20k$ for this drilldown experiment.

Since our work stealing strategy is composed of two levels of balancing (*internal* and *external*), our evaluation consider four configurations: *1.Disabled*, where we disable both levels; *2.Internal*, where we enable only the internal work stealing (WS_{int}); *3.External*, where we enable only the external work stealing (WS_{ext}); and *4.Internal+External*, where we enable both levels (WS_{int} + WS_{ext}), representing our complete strategy. We seek to evaluate the effectiveness of each of them in mitigating imbalance. Figure 5.21 presents the execution times of the parallel tasks discriminated by step and scenario. The rows represent the five fractal steps and columns represent the four working stealing configurations. The y-axis is the individual runtime of each task (x-axis).

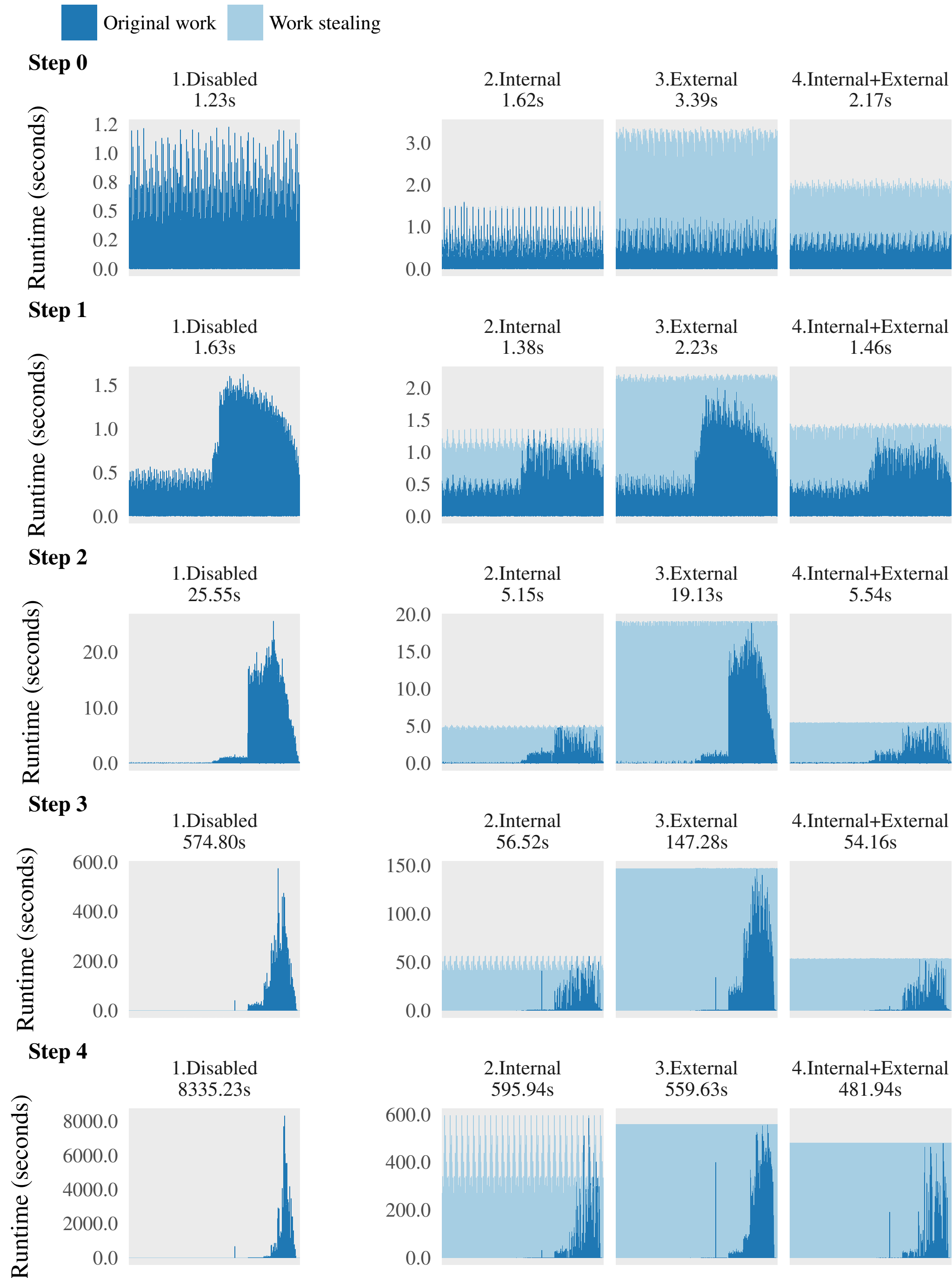
In the first configuration (*1.Disabled*), we can see the raw imbalance in load. As expected, the execution becomes more skewed for later steps, as we are enumerating bigger subgraphs (step 4 is a extreme case, for example). However, we observe a significant improvement in skew reduction across all steps when the internal work stealing is enabled (*2.Internal*). Note that, in this case, some imbalance across workers still exists since the original work is only allowed to be shared among threads in the same process. In the next configuration, we enabled only the external work stealing (*3.External*). We may see a better load balancing among the tasks, as each one has more options to steal work, but the communication overhead of work requests increases the execution time in comparison with the internal work stealing alone (*2.Internal*). Finally, the configuration which combines both strategies (*4.Internal+External*) results in near perfect load balancing as well as reduces the communication overhead in Fractal.

5.4.2.3 COST analysis

Motivated by [89], we evaluate Fractal against state-of-the-art single-thread graph mining algorithms in terms of the COST metric. The COST is defined as the number of execution threads a system needs to outperform an efficient single-thread implementation. We lever recent single threaded JVM based implementations for this purpose. For the Motifs (k-MC), Cliques (k-CL) and Pattern Querying (q_2 and q_3) kernels (ρ-PQ) we use Gtries [32, 108]. We consider queries q_2 and q_3 for Pattern Querying, as their amount of work is satisfactory to our analysis in both Fractal and baseline. For FSM we use Grami [34] and for triangle counting, we use Neo4j as baseline, which is a standard open-source single-node graph processing system. Fractal's implementations are $POSE$ approaches, except for pattern querying, which adopts a $PASE$ paradigm.

Representative results for motifs, cliques, FSM, and pattern querying are reported

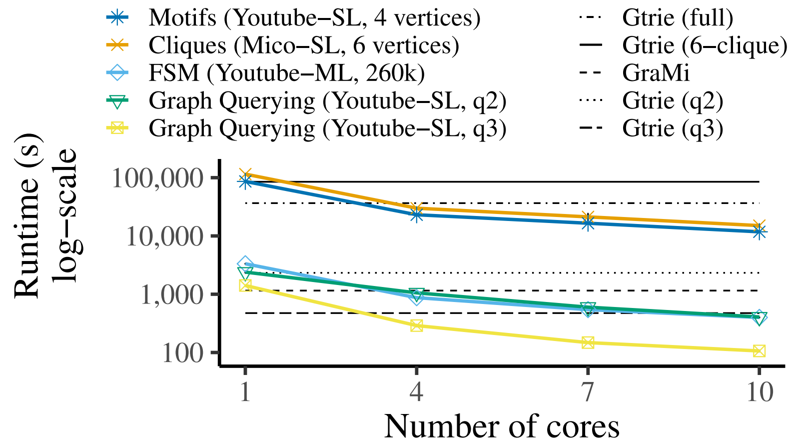
Figure 5.21: Work stealing evaluation. Imbalance becomes evident with load balancing strategies disabled *(1.Disabled)*. Internal work stealing allows a good intra-worker load balancing at low communication cost *(2.Internal)*. External work stealing allows an efficient load balancing at a higher communication overhead *(3.External)*. Applying both strategies gives the best trade-off between load balancing and communication overhead *(4.Internal+External)*. Times of each step are noted on top of each chart.



Source: Made by the author.

in Figure 5.22. One may observe that the COST typically range from 3-4 threads. For instance, Fractal beats Gtries for motifs (36.5k seconds) with 3 cores (≈ 30k seconds). Fractal outperforms both Gtries for cliques (2416s) and Grami (1154s) when using 4 cores,

Figure 5.22: COST analysis: number of cores that Fractal needs to reach state-of-the-art single-thread methods.



Source: Made by the author.

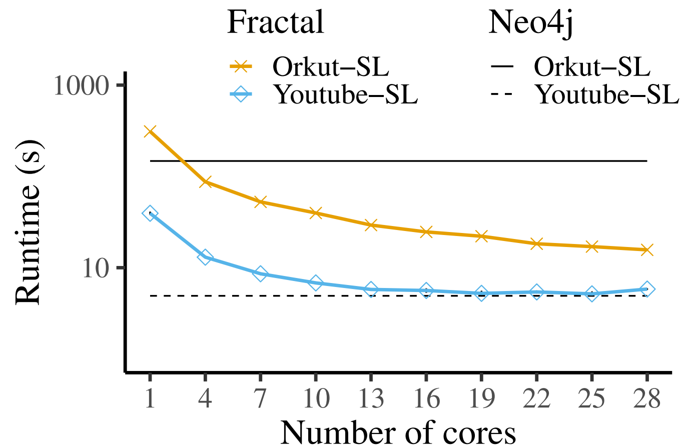
taking 844s and 872s respectively. Finally, Fractal outperforms the baseline for Graph Querying in both queries: the baseline evaluates q_2 in 2328s against 1055s for Fractal with 4 threads and q_3 in 474s against 289s for Fractal also with 4 threads. These numbers are representative for a large majority of our experimental settings. The positive exceptions to this (lower COST) arise in long-running tasks dominated by enumeration computations – here we see COST values as low as 2 threads (*e.g.* motifs on Mico). The negative exceptions (higher COST) arise when overheads dominate due to short duration tasks. For example, with the 3-cliques counting application on Youtube the COST value blows up to 16 threads (of Fractal). The overheads associated with initialization, actor set up and thread management cause this blowup.

Figure 5.23 shows additional results for triangle counting (3-CL). In a scenario such as Orkut-SL, we observe that Fractal is able to outperform Neo4j built-in implementation with four threads, continuing to scale up to 28 execution threads. In other datasets with a reduced number of triangles (and consequently, reduced number of general cliques), Fractal does not scale below the single-thread line – this is the expected case for Youtube-SL.

5.4.2.4 Scalability

We review the strong scalability of Fractal on four of our most time-consuming kernels (see Figure 5.24). We observe that if sufficient work exists the efficiency of Fractal is reasonable when compared to a single node (28-thread) implementation. For motifs

Figure 5.23: Triangles COST analysis.



Source: Made by the author.

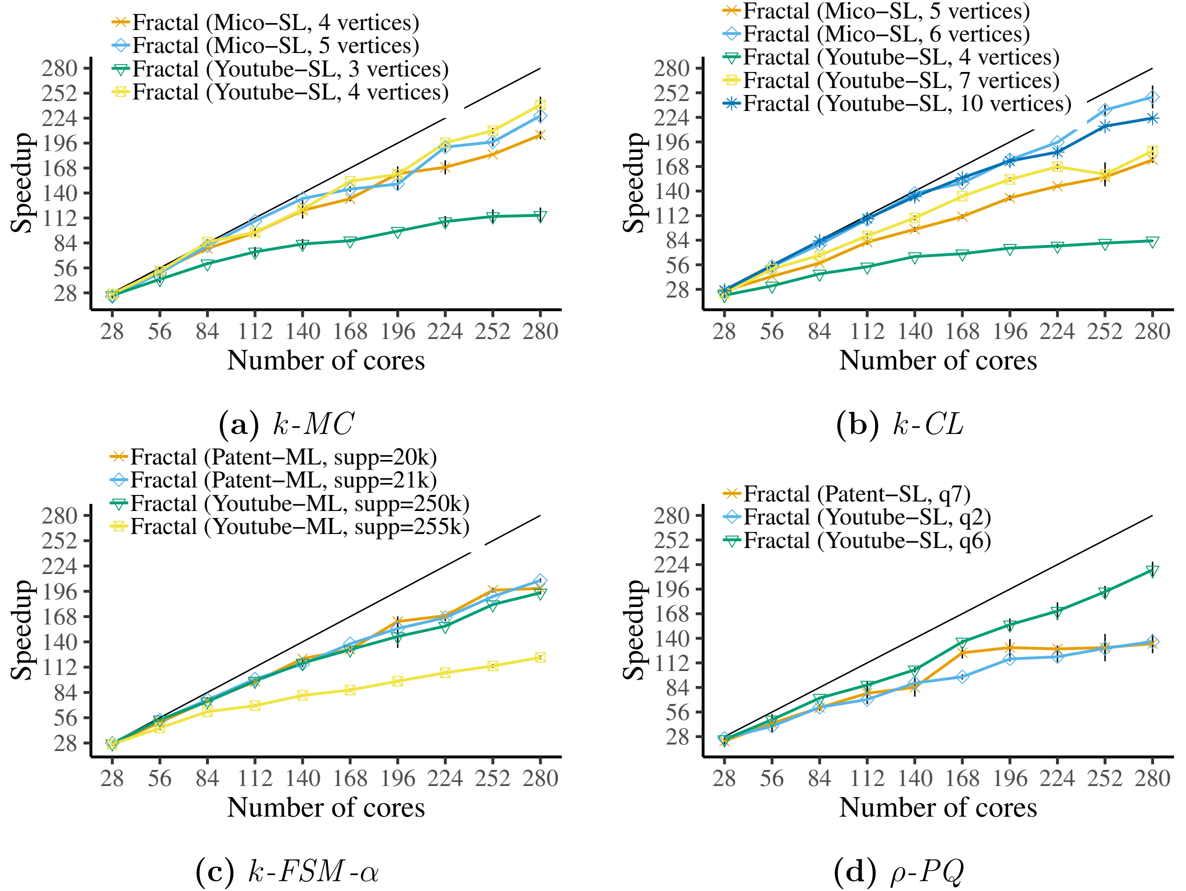
kernel, Fractal achieves around 85% efficiency. For cliques kernel, Fractal achieves around 90% parallel efficiency on Mico-SL and Youtube-SL. The efficiency for motifs and cliques is higher because enumeration dominates the cost of these applications. For FSM, a challenging task for most graph systems (due to the number of aggregations and data transfers required), we observe around 75% parallel efficiency except when there is insufficient work (Youtube-ML,support:255K). For subgraph querying, the efficiency depends on the query: patterns that are harder to enumerate (q_6) achieves better performance in Fractal – around 80% efficiency – than more symmetric and dense ones (q_2, q_7) – around 65% efficiency. Aggregations in the latter two applications lead to increased data movement costs, limiting parallel efficiency.

5.5 Understanding limitations associated with design choices

In this section we review some of the design assumptions and where each optimization may not pay off, including overhead costs.

The assumption that the input graph is replicated on each worker. In this work we consider only exact GPM problems and thus, the combinatorial explosion of subgraph enumeration can be observed even for small graphs (with hundreds of vertices). Therefore, we argue that the real bottleneck in this setting is not the memory demand for storing the input graph itself but the demand to manage the subgraphs, which is

Figure 5.24: Fractal scalability.



(a) *k-MC* **(b)** *k-CL* **(c)** *k-FSM-α* **(d)** *ρ-PQ*

Source: Made by the author.

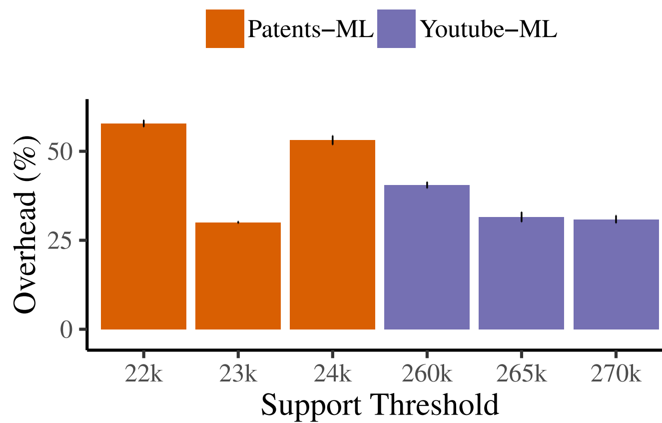
handled in Fractal by our depth-first subgraph enumeration and recomputation of steps. Nevertheless, extensions of this design to partitioned graphs can be proposed by adopting a global memory space for the input graph that can optimize accesses to graph data displaying the power-law property [36, 78], which is a reasonable assumption in most application scenarios for GPM. Our design could also benefit from these global memory spaces to implement more elaborate queueing policies for our hierarchical work stealing scheme.

Overheads associated with reliable subgraph enumeration. This optimization does not pay off when available memory on the machine exceeds the needs of the application. In such cases there is no need to re-compute as one may process directly by maintaining relevant embedding lists. We observed this in several short-duration tasks. For example, small motifs (Figure 5.14), and triangles on smaller datasets (Figure 5.15). Note that the overhead cost associated with enabling this optimization is negligible but it can lead to significant load imbalance (see Sec. 5.2). We next discuss the overheads associated with the work stealing component of Fractal which seeks to alleviate this problem.

We evaluate the overhead introduced by the bounded memory demand optimiza-

tion for the FSM application, as it is the only multi-step kernel from the selected applications. Specifically, we consider the worst case scenario where the redundant computation of previous steps and load balancing comes for free, i.e., memory is sufficient to accommodate all subgraph instances and moreover, we assume zero cost for load balancing at the end of each step (e.g. re-shuffling data). Figure 5.25 shows the results for Patents-ML and Youtube-ML. For most configurations, the overhead stays around 30-40%, which is a fair price for the reliability of a bounded memory system. For other configurations, such as Patents-ML with support threshold of 22k, we observe an overhead around 60%, representing scenarios where the later steps starts to become cheaper than the cumulative runtime cost of previous steps (shrinking lattice).

Figure 5.25: Worst case runtime overhead of keeping memory bounded with re-computation if redundant computation and load balancing come for free.



Source: Made by the author.

Overheads associated with work stealing. We use a low-overhead profiler AsyncProfiler[6] to monitor Fractal executions and measure the time spent on work stealing related code. Our experiments consider several algorithms and number of workers. We find that the overhead of work stealing is about 1.05%, with standard error of 0.44%. In terms of the initialization cost of the actor system, we observe this typically takes about one to two seconds. Such overheads impact especially the performance of executions with reduced amount of work. For instance, see Fig. 5.14 the 3-node motifs case.

[6]https://github.com/jvm-profiling-tools/async-profiler

Chapter 6

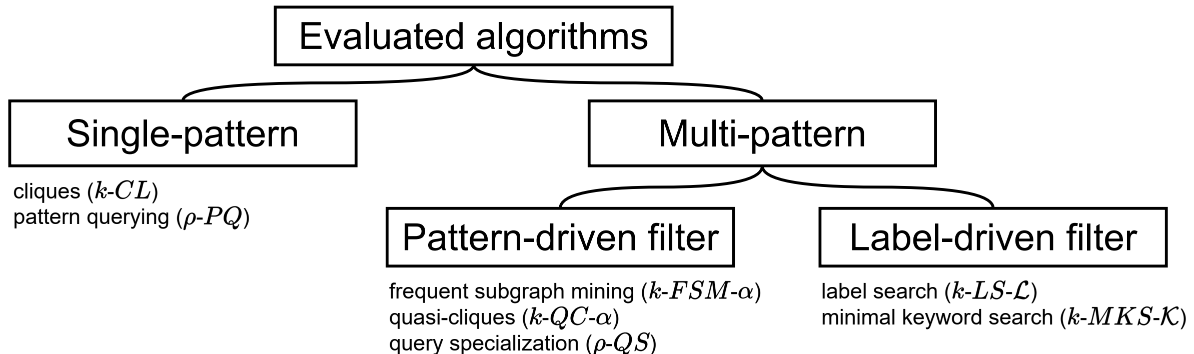
Consolidating graph pattern mining paradigms and abstractions

Graph pattern mining paradigms are central for the design and implementation of GPM systems. Moreover, because the search space of subgraphs is usually exponential, the engineering of these systems should benefit even from the most modest performance gains: especially on distributed systems where there is room for scaling processing for massively parallel environments. In this context, GPM systems usually decide the paradigm that provide the most scalable results [121] or the one that provide the most work efficient solutions [60]. Despite the fact that many works have been considering pattern-awareness as the state-of-the-art paradigm for general-purpose GPM systems [60, 21], in this chapter we work towards a more conservative statement: *no single paradigm is best for every application scenario.* The major challenge in showing these trade-offs is that most existing GPM systems support only a particular paradigm for application design (POSE or PASE, not both) and to complicate things even more, we observe a wide variety of targeted computing architectures and parallelization approaches in these systems. These aspects all together hide the real reason for reported improvements and negatively interfere with the performance diagnosis. Therefore, we argue that any experimental work with the goal of unveiling these paradigm trade-offs should do so starting from the same application model and implementation: only this way we may be more certain that the performance differences may be indeed explained by the alternative paradigms.

Our proposed model (Chapter 4) and its proof of concept implementation (Chapter 5) help building complex general-purpose GPM pipelines in a concise and productive way but also it meets the desired requirements for studying GPM paradigms from a experimental perspective, namely, being able to support multiple paradigms for building algorithms in different scenarios. In this chapter we present an extensive experimental evaluation of GPM paradigms on a wide range of application scenarios and along the way we also propose new perspectives and renewed bearing for systems abstractions targeted on GPM processing. As we may see, although many ideas presented here are not mandatory for correctness, they may guide practitioners towards more efficient and versatile GPM systems.

Before describing evaluated algorithms we make an effort to organize the considered application scenarios into categories. These categories are useful for two main reasons: (i) they help identifying which kernels can accommodate which kind of alternative algorithm design; and (ii) they represent a first effort in the community towards delimiting what constitutes a GPM problem. Figure 6.1 illustrates the categories and kernels within them. *Single-pattern* algorithms represent GPM routines in which the subgraphs of interest share a same structural pattern (e.g., enumeration of k-cliques), whereas *multi-pattern* algorithms represent GPM routines in which subgraphs of interest may span multiple structural patterns. Among multi-pattern algorithms we also consider whether the subgraphs of interest are selected based on a *pattern-driven filter* (e.g., finding subgraphs meeting a density threshold) or based on a *label-driven filter* (e.g., searching for subgraphs containing a few labels of interest). In our experimentation study we do not evaluate the motif kernel (k-MC) w.r.t. GPM paradigms because this problem unconditionally generates all subgraphs of a given size (i.e. no pruning whatsoever), which limits how subgraph enumeration can be improved, and also because this problem is extensively studied in previous work [138, 107].

Figure 6.1: Taxonomy: categories represent problems with similar algorithm design requirements.



Source: Made by the author.

Throughout these categories and for each problem we consider three algorithm variants, two of which represent the paradigms evaluated in this study (POSE and PASE) and the third representing alternatives used to capture how standard paradigms may be extended or combined in new algorithms. Thus, the first two algorithm variants represent the vanilla approach of directly applying a given paradigm to a GPM problem, while the third represents new perspectives on how certain categories of problems may allow alternative (and possibly improved) algorithms. PASE and POSE algorithms for each kernel are described in Figure 4.6. The other algorithms are presented next, as each category in Figure 6.1 allows a different set of alternative designs for algorithms.

In Section 6.1 we discuss opportunities to leverage optimized extension methods in the proposed model, which can be applied to *single-pattern algorithms*. In Section 6.2, we

propose a new GPM primitive to enhance our model and to allow multi-paradigm hybrid algorithms, which can be applied in the context of *multi-pattern algorithms with pattern-driven filter*. Finally, in Section 6.3 we illustrate how effective graph filtering procedures may be included in our model, which can be applied to *multi-pattern algorithms with label-driven filter*.

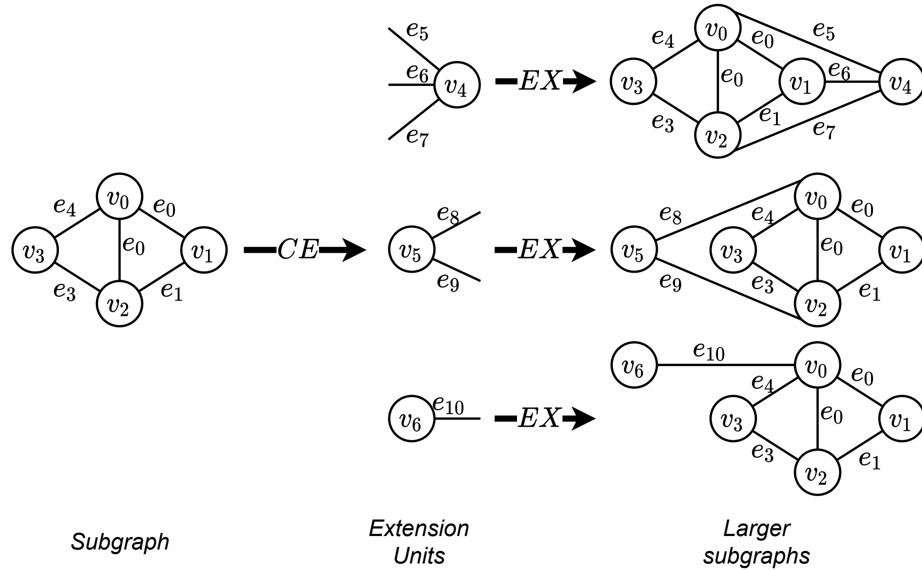
6.1 Enabling customized extension methods

As detailed in Chapter 4, the extension primitive is described by two parameters: the extension type, and the extension method. In straightforward application modeling, the extension method is set to be one of the default and generic all-extensions approach: M_C for edge-oriented and vertex-oriented extension types, or $M_P(\rho)$ for pattern-oriented extension type. Single-pattern category admits a more optimized enumeration strategy because with the information of which pattern to enumerate one can deploy specialized algorithms [35, 25, 67]. In this case we can incorporate pruning strategies that normally would be ensured via filtering primitives into the extension primitive method itself, which reduces the number of primitives of a program and consequently, the number of calls to filtering functions.

From the point of view of a subgraph being extended, two fundamental operations describe the process of exploring new subgraphs: an operation that computes candidate extension units (CE); and an operation that effectively adds extension units to the subgraph to obtain larger subgraphs (EX). Figure 6.2 illustrates these two operations over the example of Figure 4.2 and considering vertex-oriented extension type. In particular, the operation that computes candidate extensions (CE) corresponds to the extension method, defined as a parameter for the extension primitive.

Novel extension methods through custom implementations of the CE operation enables new semantically equivalent applications that are capable of reaching subgraphs of interest using a more fine-tuned approach. Indeed, such abstraction may be used for prioritizing some extension units over others, for maintaining ad-hoc data structures for quick subgraph neighborhood access, for maintaining a reduced local graph speeding up extension units selection, among others. Another important aspect is that this customization is made per extension primitive and thus, such modeling allows the combination of multiple customized extension methods, each one related to an extension primitive in the application string representation. Next we show how this may be accomplished for single-pattern kernels such as clique finding and pattern querying.

Figure 6.2: Extension primitive operations. The example assumes a vertex-oriented extension type.



Source: Made by the author.

Figure 6.3: Custom: algorithms with specialized extension methods.

(a) cliques (k-CL)

1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_V}, \mathtt{M_{KCL}})$ ▷ $\mathtt{M_{KCL}}$: Algorithm 11
2: **output** $\mathtt{GE_1 \cdots E_k A}$

(b) pattern querying (ρ-PQ)

1: $\mathtt{E} \leftarrow \mathtt{E}(\mathtt{T_V}, \mathtt{M_{MCVC}})$ ▷ $\mathtt{M_{MCVC}}$: Algorithm 12
2: **output** $\mathtt{GE_1 \cdots E_{|V(\rho)|} A}$

Source: Made by the author.

Custom algorithm for cliques (k-CL) (Figure 6.3a). KClist [25] is an optimized clique listing algorithm that reduces the clique search space by using special views of the input graph on each enumeration depth. A state is maintained during KClist enumeration which is a DAG (directed acyclic graph) extracted from the induced subgraph in the neighborhood of each extension candidate. The idea of this algorithm is to maintain a DAG composed of only valid clique extensions (which added to a clique becomes a larger clique) and to refine this DAG as we include new extensions to the current clique. Specifically, if we add extension unit e to a k-clique S, then we produce a new $(k+1)$-clique and most important, the new DAG only need to include extension units e' that are adjacent to e in the current DAG and moreover, the neighbors of e' that must be in the new DAG are only those that are also in the neighborhood of the added extension unit e. If any of these two conditions do not hold, the extension unit cannot be part of a larger clique. The intuition is that this strategy progressively reduces the input graph as larger cliques are discovered and, consequently, it reduces the cost of generating new cliques by disregarding vertices and edges that will never be part of a larger clique. Algorithm 11 details this method, denoted by M_{KCL}. The extension method has two main subroutines: (i) ensure that the neighbors of e are valid extensions in terms of cliques; and (ii) pre-

compute the refined DAG G' for subsequent calls for the larger clique. We highlight that this method is more efficient than the all extension method for vertex-oriented extension (M_C) presented as default to the subgraph aggregation problem because in this case we skip numerous spurious checks of whether a visited subgraph represents a clique. In some ways, we enumerate cliques directly without filtering primitives.

Algorithm 11 KClist's extension method $\mathtt{M}_{\mathtt{KCL}}(G, S)$: subgraph extension method that generates only clique extension units from a DAG G and that generates a new DAG G' for subsequent calls.

1: $e \leftarrow \text{LAST}(\gamma(S))$
2: $V' \leftarrow N_G(e)$
3: **let** $G'(V', E')$ be a new DAG
4: **for** e' **in** $N(e)$ **do**
5: $\quad N_{G'}(e') \leftarrow N_G(e') \cap N_G(e)$
6: **return** $V(G')$ ▷ Extension units are obtained from the DAG vertices

Custom algorithm for pattern querying (ρ-PQ) (Figure 6.3b). The custom extension method may also be used to reduce the cost of pattern querying. Instead of matching one vertex at a time via expensive recursive calls ($M_P(\rho)$ in Algorithm 4), the *Minimum Connected Vertex Cover* strategy [67, 60] matches a connected subset of ρ (core) such that no edge connects two vertices outside this core. Thus, once the core is matched the remaining of the pattern can be safely obtained from the neighborhood of the core itself, which tends to improve performance by avoiding the re-computation of extension candidates in expensive recursive calls. Algorithm 12 shows how this strategy may be modeled as a custom extension method M_{MCVC}. This method receives the input graph G, the current subgraph S, the target pattern ρ, and additionally, it maintains a mapping structure $\mathcal{E}$ that stores extension candidates for vertices not in the core. In order to match the core the algorithm simply returns the standard all pattern extensions method ($M_P(\rho)$) described in Algorithm 4. Whenever the vertex cover size is reached (line 6) the vertex candidate sets for the remaining vertices are computed and subsequent calls to compute extensions for larger subgraphs are going to quickly access this pre-computed structure (line 9). Finally, the method returns the subset of these candidates that satisfies the symmetry breaking conditions. Figure 6.3b shows how to incorporate this custom extension method in an application design for pattern querying (ρ-PQ): instead of using the standard $M_P(\rho)$ as method for the extension primitives we just plug this optimized method M_{MCVC} on each extension level instead.

Algorithm 12 MCVC's extension method $\mathtt{M_{MCVC}}(G, S, \rho, \mathcal{E})$

1: $c \leftarrow \text{MCVC}(\rho)$ ▷ Pre-processed
2: $mcvcSize \leftarrow |V(c)|$
3: $numVertices \leftarrow |V(S)|$
4: **if** $|V(S)| < mcvcSize$ **then**
5: **return** ALL-PATTERN-EXTENSIONS(G, S, ρ) ▷ fall back to $\mathtt{M_P}(\rho)$: Algorithm 4
6: **else if** $|V(S)| = mcvcSize$ **then**
7: **for** $i \leftarrow mcvcSize + 1$ **to** $|V(\rho)|$ **do**
8: $\mathcal{E}[i] \leftarrow$ COMPUTE-VERTEX-CANDIDATES(S, G, ρ, i)
9: $extensions \leftarrow \mathcal{E}[numVertices + 1]$
10: $sbConditions \leftarrow$ GET-OR-COMPUTE-SYMMETRY-BREAKING(ρ)
11: **return** $\{n \mid n \in extensions$ and SATISFIES$(n, sbConditions)\}$

6.1.1 Incorporating extension abstractions in Fractal

Fractal implements the process of customizing primitive operations (Section 6.1) by giving access to its subgraph enumeration engine through a data structure called *subgraph enumerator*. This data allows the primitive operation to be transparent and extensible. Subgraph enumerators represent a checkpoint for the subgraph enumeration process. Figure 6.4 shows the programming interface for the subgraph enumerator abstraction. During subgraph exploration, extensions for the current subgraph (`[SE-SG]`) are computed using the `[SE-CE]` function. Next computed extensions are consumed, one at a time, by the `[SE-EX]` function. Such process is repeated for each extension primitive encountered in the application and thus, an additional function for specifying an extension primitive with a custom subgraph enumerator is provided by `[FRAC-E-SE]`.

By default, it implements algorithms for vertex-induced (M_C), edge-induced (M_C), or pattern-induced (M_P) extension methods that explore all valid subgraphs from the input graph. By overriding these functions, a practitioner may implement a specific subgraph exploration behavior for customized and optimized approaches of solving a GPM problem.

Example 6.1.1. *(KClist in Fractal)* The user may implement a custom subgraph enumerator and pass it as an additional parameter to the customized expand operator (`[FRAC-E-SE]`, Figure 6.4) to support complex GPM implementations. Listing 6.1 shows the implementation of a KClist subgraph enumerator for Fractal applications. The implementation follows the same steps discussed in Figure 6.3a. Listing 6.2 shows how to incorporate this enumerator in a Fractal workload.

Figure 6.4: Fractal: the subgraph enumerator API.

```
SubgraphEnumerator {
  // [SE-SG]
  def currentSubgraph: Subgraph

  // [SE-CE]
  def computeExtensions(): Unit

  // [SE-EX]
  def extend(): Boolean
}

Fractoid {
  // [FRAC-E-SE]
  def expand(n: Int, seClass: Class[_ <: SubgraphEnumerator]): Fractoid
}
```

Source: Made by the author.

Listing 6.1: KClist subgraph enumerator.

```
 1 class KClistEnum extends SubgraphEnumerator {
 2   val dag: Map[Int,IntList] // DAG adj. lists
 3
 4   override def computeExtensions(): Unit = {
 5     val newDag = new Map[Int,IntArrayList]()
 6
 7     val u = currentSubgraph().getVertices().getLast()
 8     val uneighborhood = dag.get(u)
 9     for (v <- uneighborhood) {
10       val vneighborhood = dag.get(v)
11       val intersectionNeighborhood = uneighborhood.
12         intersection(vneighborhood)
13       newDag.put(v, intersectionNeighborhood)
14     }
15
16     val nextEnumerator = getNextSubgraphEnumerator()
17     nextEnumerator.setDag(newDag)
18   }
19 }
```

6.2 Enabling multi-paradigm algorithms via mapping primitive

The existence of different GPM paradigms (Section 2.4) for solving the same problem raises important questions concerning how they compare to each other in different

Listing 6.2: KClist cliques application.

```
1 val kclist = graph
2   .vfractoid() // vertex-oriented extension type T_V
3   .expand(1, classOf[KClistEnum]) // extension method M_KCL
4   .repeat(k) // repeat towards k-cliques
5 val numCliques = kclist
6   .aggregate(s => 0L, s => 1L, _ + _) // aggregation
7   .values.sum() // Spark API: sum all values
```

workload scenarios. We discussed trade-offs between paradigms enable a whole new opportunity for combining extension types within the same GPM application. By default, existing implementations for graph pattern mining applications follow one of the GPM paradigms: pattern-aware (PASE) or pattern-oblivious (POSE). If on one hand PASE produces an exponential number of steps, its subgraph exploration allows more efficient implementations because we may rely on fast set intersection routines to speedup the generation of extension candidates [60]. POSE is the opposite: we generate fewer application steps but at the cost of a less effective subgraph enumeration. Thereby, we explore in this work a hybrid design that could capture the best features of both: fewer steps plus more effective subgraph enumeration. We explore this idea for *multi-pattern algorithms with pattern-driven filter* as they admit reasoning about different ways one could reach subgraphs of various patterns.

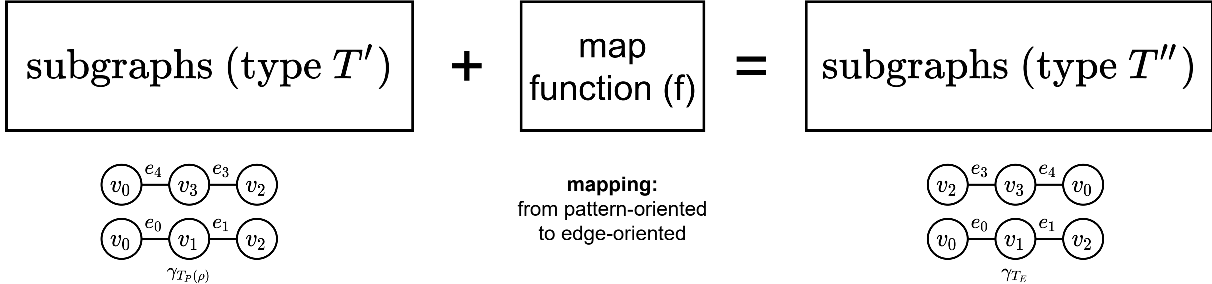
The main challenge concerning hybrid approaches is that canonical codes for subgraphs differ depending on the paradigm. In a PASE algorithm, canonical subgraph codes are given by symmetry breaking conditions generated specifically for a pattern prior to the subgraph enumeration process. In other hand, in POSE, the canonical subgraph codes are not subject to a given pattern and thus, not related to symmetry breaking conditions (see Section 4.1.2). In order to support switching among paradigms, and more specifically among extension types, we propose extending our model with a new mapping primitive abstraction.

Definition 20. (Mapping primitive) The mapping primitive is denoted by $M(T', T'', f)$, where T' is an input extension type, T'' is an output extension type, and f is a function that maps a subgraph code of type T' into one or more subgraph codes of type T''. Specifically, given a subgraph code $\gamma_{T'}(S_i)$, the mapping primitive produces n subgraph codes $f(\gamma_{T'}(S_i)) = \{\gamma^{(i)}_{T''}(S_i) \mid 1 \leq i \leq n\}$, each serving as input for the subsequent primitives of the application step.

A *mapping primitive* is an atomic GPM operation that allows mapping a subgraph code into other subgraph codes (Definition 20), for instance from $\gamma_{T_{P(\rho)}}$ to γ_{T_E}, with the purpose of ensuring correctness among switched GPM paradigms. Figure 6.5 illustrates its general operation. Application steps can accommodate this additional primitive and

the regex denoting them is updated to $\mathtt{GE}(\mathtt{E}+\mathtt{F}+\mathtt{M})^*\mathtt{A}$, where M represents the mapping primitive.

Figure 6.5: Mapping primitive: mapping subgraph codes.

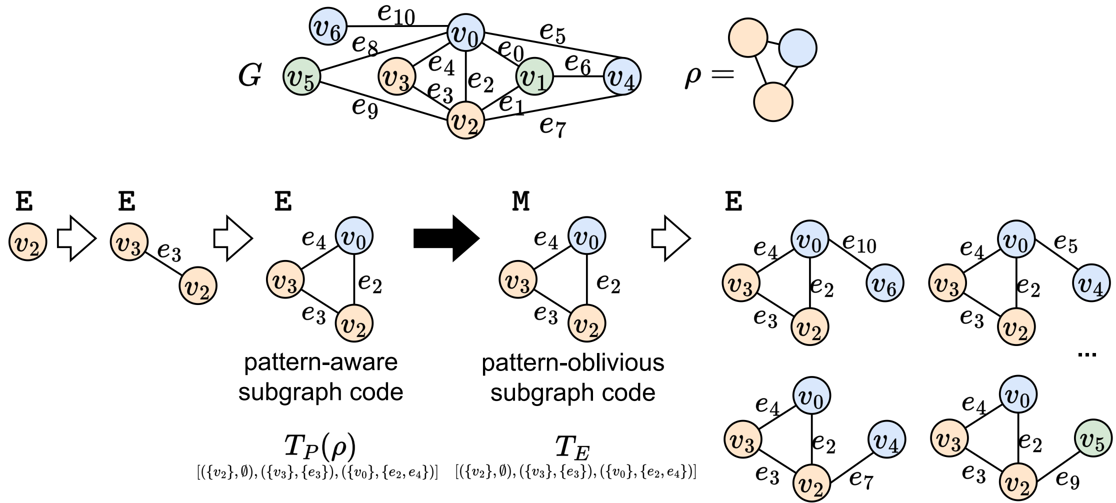


Source: Made by the author.

To show the potential of such abstraction, in this work we provide an alternative design for multi-pattern algorithms with pattern-driven filter. In this alternative design, referred as PASE+POSE, the algorithm ensures a pattern-aware paradigm (PASE) in the first extension primitives and switches to a pattern-oblivious (POSE) on the fly, which allows an unconditional exploration of larger subgraphs. This alternative algorithm design is possible whenever the problem involves multiple patterns in the output and the predicate for selecting valid subgraphs is a function of the subgraph structure (its pattern). The main challenge concerning this hybrid approach is that canonical codes for subgraphs differ depending on the GPM paradigm: symmetry breaking [46] is used for pattern-aware paradigm while canonical filtering [121] is used for pattern-oblivious pattern-oblivious. Thus, function `f` denotes a mapping process that transforms a subgraph represented through a pattern-aware code (obtained by $\mathtt{M_P}(\rho)$ method) into the same subgraph but represented through an equivalent pattern-oblivious representation (obtained by $\mathtt{M_C}$ method). In Figure 6.6 we show how to enumerate subgraphs with four edges that contain a tailed triangle ρ by (i) matching a triangle subgraph isomorphic to ρ and (ii) adding the fourth edge to the matched subgraph resulting in multiple tailed triangles. In this case, up to three vertices, the subgraph is being enumerated using a PASE paradigm with extension type $T_P(\rho)$ and, afterwards, the last edge is included using a POSE paradigm with extension type T_E. This is only correct because of the mapping primitive that includes function f responsible for *translating* a $T_P(\rho)$ subgraph code into a T_E subgraph code, which introduces consistency between canonical subgraph codes.

For instance, Algorithm 13 pattern-oriented subgraph codes (PASE paradigm) can be mapped to edge-oriented subgraph codes (POSE paradigm). An edge-oriented (T_E) subgraph code starts with the smallest edge of subgraph S (line 1) and this is sufficient to determine the first extension unit in the converted code c. The next edges are determined by the smallest edge not yet in the target code c that also maintains the code connected (line 3).

Figure 6.6: Hybrid PASE+POSE via mapping primitive (M).



Source: Made by the author.

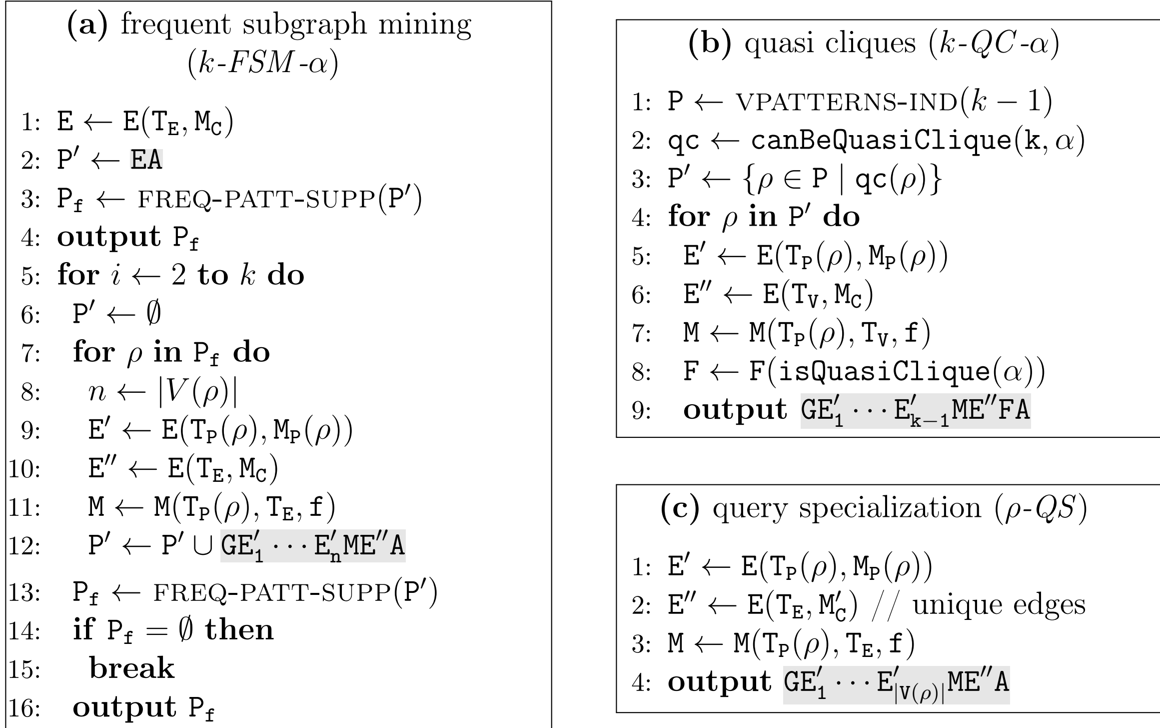
Algorithm 13 MAPPER-PO-TO-EO($\gamma_{T_P(\rho)}(S)$):

1: $c \leftarrow [(\{u, v\}, \{e\})] \mid \forall e' \in E(S), e \leq e'$
2: **for** $i = 2$ **to** $|E(S)|$ **do**
3: $\quad e \leftarrow (u, v) \in E(S) \mid (u \in V(c) \vee v \in V(c)) \wedge \forall e' \in (E(S) - E(c)), e \leq e'$
4: $\quad V' \leftarrow \{u, v\} - V(c)$
5: $\quad c[i] \leftarrow (V', \{e\})$
6: $\gamma_{T_E}(S) \leftarrow c$
7: **return** $\gamma_{T_E}(S)$

PASE+POSE algorithm for frequent subgraph mining (k-FSM-α) (Figure 6.7a). The hybrid strategy depicted in Figure 6.6 can be applied to FSM for each frequent pattern found in the previous iteration (lines 7-12) to determine which extended patterns are also frequent, i.e., which patterns with an additional edge reaches the minimum support. This is accomplished in line 12 where frequent patterns are matched via PASE paradigm, mapped to a edge-oriented representation (mapping primitive M), and extended by one edge. This approach generates *one step per frequent pattern* on each iteration, instead of *one step per extended pattern candidate* as observed of vanilla PASE. Thus, this may be used to tame the exponential growth of submitted steps of pattern-aware paradigm.

PASE+POSE algorithm for quasi cliques (k-QC-α) (Figure 6.7b). We leverage this hybrid strategy for reducing the number of steps in PASE design. First, we generate patterns with $k-1$ vertices that may be extended to become a k-vertex quasi clique (line 3). Next the algorithm matches each of these pattern candidates, adds one additional vertex to subgraphs found, and selects the ones meeting the minimum density requirement (line 9).

Figure 6.7: PASE+POSE: hybrid multi-paradigm algorithms.



Source: Made by the author.

PASE+POSE algorithm for query specialization (ρ-QS) (Figure 6.7c). The algorithm matches the query pattern ρ using pattern-aware ($\mathtt{E}'$), switches to pattern-oblivious (primitive $\mathtt{M}$), and finishes enumeration by adding an additional edge to the subgraph ($\mathtt{E}''$) – producing subgraphs containing pattern ρ (line 5). Notice that this hybrid design is very fit to query specialization since the kernel requires the specializations to contain the input pattern ρ as subpattern.

6.2.1 Incorporating mapping primitive in Fractal

Figure 6.8 shows the programming interface for the subgraph mapping abstraction in Fractal. Additional functions `FRAC-V`, `FRAC-E`, and `FRAC-P` for the *fractoid* interface allow mapping the current subgraph processing to vertex-, edge-, or pattern-oriented extension types. The input for these operators is a subgraph mapper that returns nothing. Several mapped subgraph codes may be processed by a `newSubgraph` call. Such application design is intended to encourage *in-place* subgraph processing instead of *materializing* each map result into a new subgraph object. This implementation design is known to improve the memory management efficiency in garbage collected languages (such as Scala/-Java) [84], especially in our context of a potentially exponential subgraph search-space to

be explored. Next we explore an example of how this may be accomplished in Fractal.

Figure 6.8: Fractal: the subgraph mapping API.

```
SubgraphMapper {
  abstract def map(s: Subgraph): Unit // to be implemented
  def newSubgraph(s: Subgraph): Unit // final
}

Fractoid {

  // [FRAC-V]
  def vfractoid(m: SubgraphMapper): Fractoid

  // [FRAC-E]
  def efractoid(m: SubgraphMapper): Fractoid

  // [FRAC-P]
  def pfractoid(p: Pattern, m: SubgraphMapper): Fractoid

}
```

Source: Made by the author.

Example 6.2.1. *(Hybrid k-FSM-α subroutine)* Listing 6.3 overviews the implementation of the hybrid exploration strategy introduced ina Figure 6.6 and applied for FSM. The subgraph mapper implemented for this particular example produces a single mapped subgraph code according to Algorithm 13, which is marked by a single `newSubgraph` call. A one-to-many mapping could be accomplished by several calls of `newSubgraph` with modified version of the given subgraph, i.e., several subgraph codes derived from the same input subgraph.

6.3 Enabling effective pruning via graph filtering

Subgraph pruning is important for reducing the exponential growth of the exploration process and in certain scenarios, filtering functions may be applied directly to the input graph. Specifically, some filtering primitives applied in the context of subgraphs for GPM applications can be pushed down directly to the data source, i.e., the input graph. Exploratory routines over graph data often exhibit locality during processing: the working set of visited vertices and edges is reduced or it shrinks as the algorithm progresses. Also, because the cost of graph processing engines is directly related to the input size, being able to work with just the essential regions of the graph can significantly reduce the

Listing 6.3: Hybrid k-*FSM*-αsubroutine: enumerating subgraph candidates from a known frequent pattern.

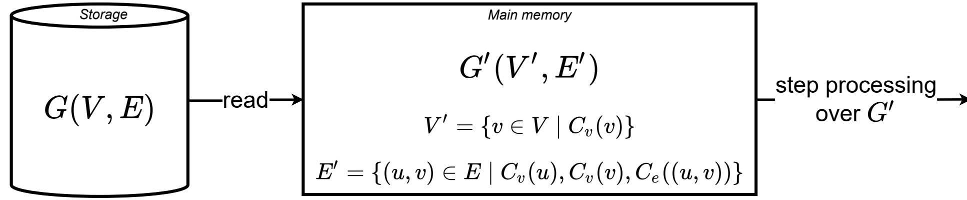
```
1  val mapperPoToEo = new SubgraphMapper {
2    override def map(s: Subgraph): Unit = {
3      val mappedSubgraph = // ... mapped subgraph according to Algorithm 13
4      newSubgraph(mappedSubgraph)
5    }
6  }
7
8  val frequentPattern = // ... known frequent pattern ρ
9
10 val ls = graph
11   .pfractoid(frequentPattern) // pattern-oriented extension type T_P(ρ)
12   .expand(pattern.numVertices()) // extension method M_P(ρ)
13   .efractoid(mapperPoToEo) // edge-oriented extension type (T_E)
14   .expand(1) // extension method M_C
15
16 // .. do whatever aggregation is intended for 'ls'
```

overall cost of such computations. During the enumeration process, extension candidates are generated from the edges connecting the current subgraph with extension units not in the current subgraph. Moreover, this is a repeating process for each subgraph being extended, which usually means an exponential search-space. Thus, reducing the input graph may be beneficial for filtering conditions that can be applied directly to the input graph once, instead of being applied for each subgraph found.

The graph filtering (GF) enables such optimizations by skipping edges and vertices that are not of interest before the enumeration process starts. This local filtering on graph has the potential of a great reduction in the number of invalid subgraph extensions, improving the overall performance. While this can be implemented as a preprocessing in the input graph, doing so becomes infeasible if one considers arbitrary filtering predicates that may unfold from graph pattern mining tasks – store and manage that many views of the input graph is not trivial, considering the scale of the input and the general-purpose requirement of the system. Figure 6.9 shows the operation of graph filtering. Given an input graph G, a user-defined vertex predicate C_v, and a user-defined edge predicate C_e, the step processing originally over graph G is performed over a reduced view of G, G', composed only by vertices and edges satisfying predicates C_v and C_e, respectively.

We consider applying graph filtering (GF) in the context of *multi-pattern algorithms with label-driven filter*. In this case, subgraph predicates based on some label property may induce in local conditions that each vertex or edge should meet. Thus, if the predicate is known to be anti-monotonic we conclude that is safe to remove them from the input graph before subgraph enumeration begins. Next we show how to incorporate such optimization into pattern-oblivious algorithms. We refer to these optimized

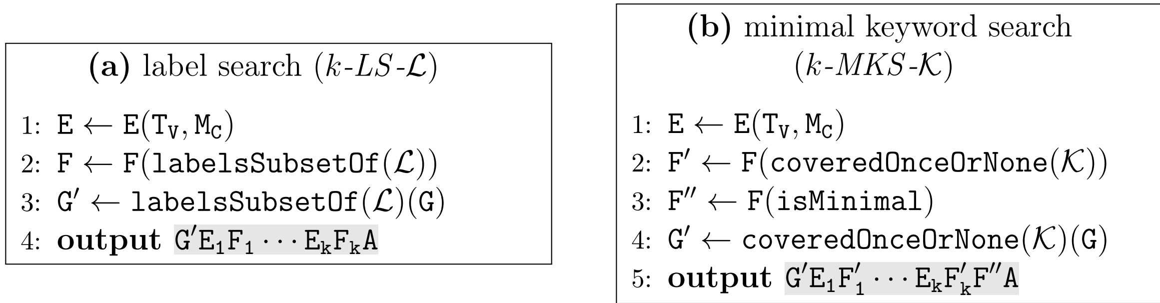
Figure 6.9: Abstraction: graph filtering.



Source: Made by the author.

pattern-oblivious algorithms as POSE+GF.

Figure 6.10: POSE+GF: effective pruning via graph filtering (GF).

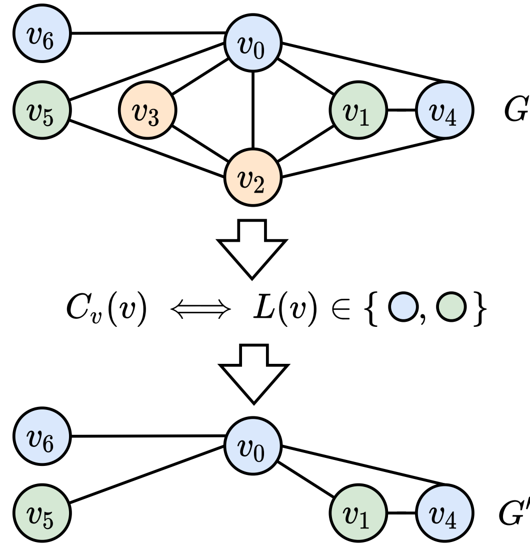


Source: Made by the author.

POSE+GF algorithm for label search (k-LS-$\mathcal{L}$) (Figure 6.10a) The predicate `labelsSubsetOf`, the same initially defined with the purpose of being the argument for filtering primitives on subgraphs, is applied directly to the input graph (line 3). Figure 6.11 shows an example of graph filtering with a predicate that selects only vertices having blue or green labels. Such filtering can be used for solving specific instances of the subgraph search application. In this case, instead of applying exhaustive filters on each subgraph enumerated, a reduced graph G' (instead of G) is passed to downstream step processing concerning the enumeration and aggregation of subgraph. The outcome is a reduction in the number of invalid extensions generated while expanding subgraph candidates.

POSE+GF algorithm for minimal keyword search (k-MKS-$\mathcal{K}$) (Figure 6.10b) Minimal keyword search admits a similar operation but with predicate `coveredOnceOrNone`. Because this predicate is applied after each extension primitive originally, we may assume that this is in fact an anti-monotonic filter applied to subgraphs in this kernel. Notice that the application step is now applied over a filtered version of the input graph ($\mathtt{G'}$) in line 5.

Figure 6.11: Graph filtering example: selecting vertices having specific labels.



Source: Made by the author.

6.3.1 Incorporating filtering abstractions in Fractal

Figure 6.12 shows the programming interface for the graph filtering abstraction in Fractal. Edge filtering can be applied over input graphs (`FractalGraph`) and its only parameter is a function representing an edge predicate for removing unwanted edges while reading the input graph. The description of an edge has five elements: (1) the source vertex identifier u; (2) the labels of vertex u; (3) the destination vertex identifier v; (4) the labels of vertex v; and (5) the labels of edge (u, v). Thus, filtering criteria may include any local reasoning over an edge and its properties. Vertex filtering is similar and selects vertices satisfying a given predicate considering the vertex and its labels.

Example 6.3.1. *(k-LS-$\mathcal{L}$ with graph filtering on Fractal)* The user may apply the filtering condition operator directly on the input graph to keep only vertices that have the target labels of the query (`[FG-VF]`, Figure 6.12). Listing 6.4 shows the implementation of label-based subgraph search with graph filtering in Fractal.

Figure 6.12: Fractal: the graph filtering API.

```
FractalGraph {

  // [FG-EP]
  typedef EdgePredicate: (
    Int, // vertex u
    IntArrayList, // labels of u
    Int, // vertex v
    IntArrayList, // labels of v
    IntArrayList // labels of (u,v)
  ) => Boolean

  // [FG-VP]
  typedef VertexPredicate: (
    Int, // vertex u
    IntArrayList, // labels of u
  ) => Boolean

  // [FG-EF]
  def efilter(pred: EdgePredicate): FractalGraph

  // [FG-VF]
  def vfilter(pred: VertexPredicate): FractalGraph

}
```

Source: Made by the author.

Listing 6.4: k-LS-$\mathcal{L}$with graph filtering application.

```
 1  val labelSet = // ... input label set for querying
 2  val filteredGraph = graph
 3     .vfilter((u, uLabels) => labelSet.containsAll(uLabels))
 4  val ls = filteredGraph
 5    .vfractoid() // vertex-oriented extension type T_V
 6    .expand(1) // extension method M_C
 7    .repeat(k) // repeat towards k-vertex induced subgraphs
 8  val numSubgraphs = ls
 9    .aggregate(s => 0L, s => 1L, _ + _) // aggregation
10    .values.sum() // Spark API: sum all values
```

6.4 Evaluating GPM paradigms: consolidation and renewed bearing

The alternative paradigms for GPM (PASE and POSE) exhibit very particular characteristics concerning how they approach subgraph enumeration. In fact, their be-

havior in terms of the number of steps necessary to a GPM task or in terms of the effectiveness of subgraph extension poses important questions regarding these trade-offs and how they manifest in real-world application scenarios. Because we consider that there is this gap between algorithmic strategies and a proper landscape of their limitations, we conclude this text with a wide evaluation of GPM paradigms and application scenarios. We expect to go beyond reporting performance results but also be able to consolidate the existing knowledge about GPM paradigms and to highlight promising directions for future work.

As with any experimental work, we must be careful with how we approach the implementation and the deployment of such algorithms. A naïve methodology would be to leverage some existing PASE frameworks [60, 17, 87] and other POSE frameworks [121, 12] and compare their performance directly. In this case, however, we would be comparing *system implementations* and not particularly *paradigms*. For instance, performance differences reported in the literature may be explained by multiple factors not related to GPM paradigms whatsoever: parallelization strategies, underlying computing architecture, fine tuned implementations, programming languages and virtual machines, and so on. We argue that a proper evaluation of GPM paradigms should be provided over the same general-purpose system model and implementation. Fortunately, our primitive-based model for GPM (Chapter 4) implemented over Fractal system (Chapter 5) admits multiple paradigms and application scenarios via a general-purpose programming interface. We believe that such deep experimental evaluation is a proper closure for this work because it exercises our proposed model and tools but also provides additional knowledge and important insights for the GPM community.

In this chapter we compare implementations of GPM algorithms in our model and within the same system (Fractal). Because we provide results obtained from the same system and architecture, we are able to draw more expressive conclusions regarding which *paradigms* are more effective instead of which *implementation* are more effective. We highlight that a detailed comparison of our model against competing systems is presented earlier in Section 5.4.1. Besides PASE and POSE (Figure 4.6), we also consider evaluating the alternative algorithm designs Custom (Figure 6.3), PASE+POSE (Figure 6.7), and POSE+GF (Figure 6.10). With those we expect to provide more concrete directions on how default algorithms may be modified towards alternative and more powerful approaches. Kernels considered, algorithms and categories are summarized in Table 6.1.

Hardware. All experiments were run on a cluster with 5 machines, each one having two CPU Intel Xeon E5-2695v2 Ivy Bridge 2,4GHZ (12 cores each, 24 per machine), 64GB RAM, running RedHat Linux 7.6. The machines are connected by Infiniband FDR (56Gb/s).

Table 6.1: Summary of algorithms evaluated. *Categories* represent problems with similar requirements concerning algorithm design. *Algorithms* represent standard paradigms (POSE and PASE in Figure 4.6) and derived strategies for solution (Custom in Figure 6.3, PASE+POSE in Figure 6.7 , and POSE+GF in Figure 6.10).

Problems	Categories		Algorithms
cliques (k-CL) pattern querying (ρ-PQ)	Single pattern		POSE PASE Custom
frequent subgraph mining (k-FSM-α) quasi cliques (k-QC-α) query specialization (ρ-QS)	Multi-pattern	Pattern-driven filter	POSE PASE PASE+POSE
label search (k-LS-$\mathcal{L}$) keyword search (k-MKS-$\mathcal{K}$)		Label-driven filter	POSE PASE POSE+GF

Datasets (Table 6.2). The datasets used in our evaluation have been widely used previously to evaluate graph mining algorithms and systems [34, 1, 121, 60, 17, 87]. Mico [34] is a co-authorship network, Patents [48] models the citations of patents published in the US, LiveJournal and Orkut [134] model friendship in social networks, and Youtube [23] models posted videos and how they are related. In our experiments all graphs are loaded into the memory of workers using a compressed sparse row graph representation (CSR).

Table 6.2: Real-world datasets used in the experiments.

	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	max.deg.	avg.deg.	labels
Mico (MI) [34]	100K	1M	1.3K	22.3	29
Patents (PA) [48]	2.7M	13.9M	789	10.1	37
LiveJournal (LJ) [134]	3.9M	34.6M	14.8K	17.3	-
Youtube (YO) [23]	4.5M	43.9M	2.5K	19.1	108
Orkut (OR) [134]	3.07M	117.1M	33.3K	76.2	-

Performance evaluation measures. We consider a time budget of 5 hours for each execution, which allows us to study larger and more diverse GPM paradigms. Within each execution, to be fair in our comparison, we divide the time budget amongst application steps in a way that lighter steps (concerning more dense and infrequent patterns in scale-free networks) are scheduled first so more time budget is left for heavier steps. We consider two evaluation metrics: (1) the *runtime* representing elapsed time of executions whenever it is not trivial to measure the application throughput (particularly, for FSM); and (2) the *normalized throughput* of aggregated subgraphs – in this case, we indicate whether an execution reached the time limit with an asterisk ($*$). Throughput allows us to determine

the most efficient strategy given a time budget. These metrics are widely used to evaluate the performance of subgraph querying systems under time constraints [49] and also in the context of streaming analytics where aggregated results are continuously consumed via a publish-subscribe framework [12].

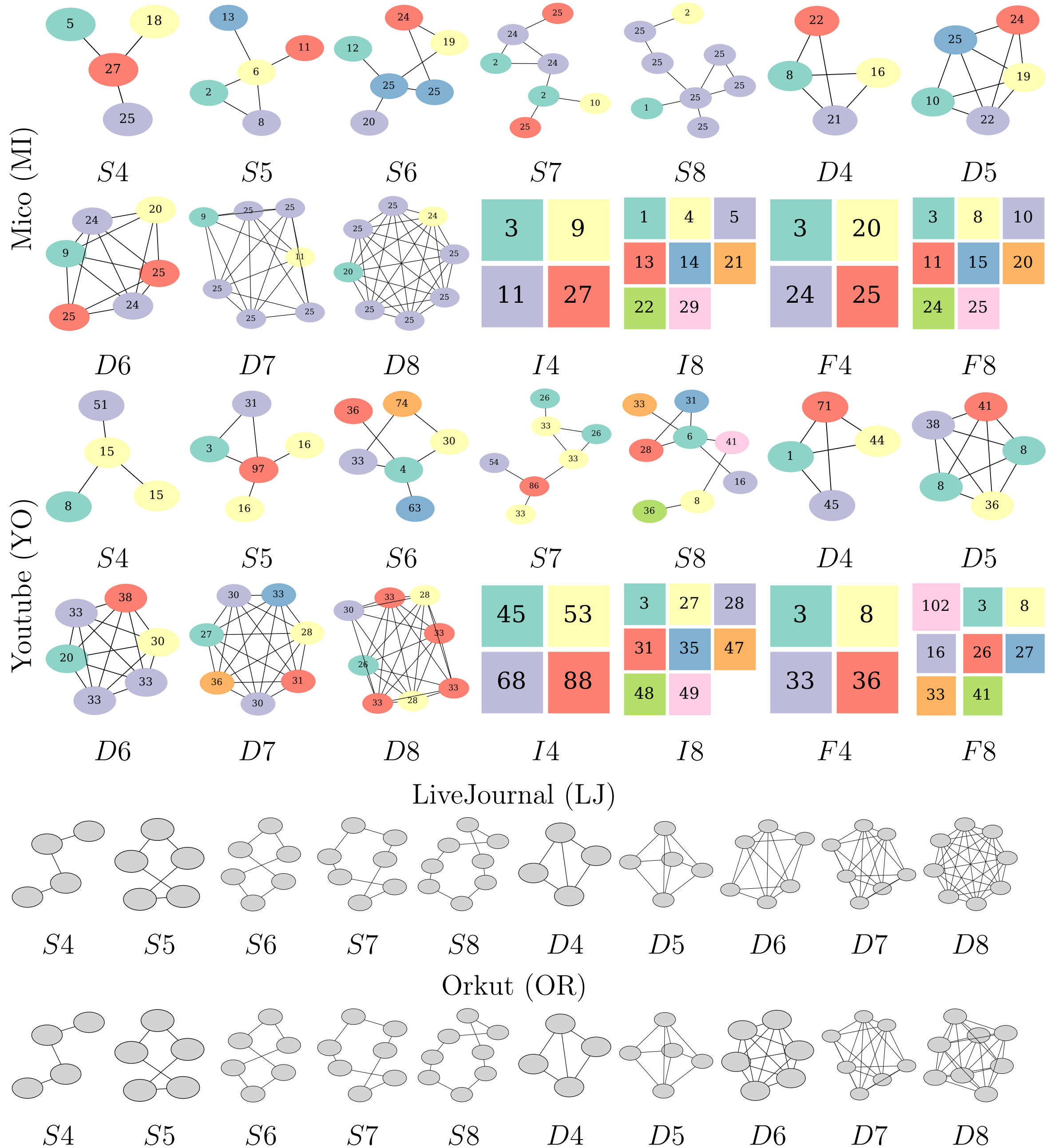
Pattern and label query generation (Figure 6.13). For GPM problems such as query specialization (ρ-QS) and pattern querying (ρ-PQ), we generate input patterns for each dataset based on their density. Specifically, we implement a widely known unbiased sampling method for subgraphs [130] to extract representative patterns of a given size k: (1) Sk, sparse pattern with k vertices; and (2) Dk, dense pattern with k vertices. For problems such as label search and minimal keyword search, we take a similar approach to produce label sets occurring together in a given dataset: (1) Ik, representing k infrequent labels that rarely occur together within a subgraph; (2) Fk, representing k labels that most frequently appearing together within a subgraph.

Matching order for pattern-aware (PASE). Additionally, we adopt the following heuristic for determining the order in which patterns are matched when using the pattern-aware paradigm. First, we match vertices having infrequent labels first, if there is a tie match vertices with more symmetry breaking conditions [46]. Finally, if there is still a tie, we match vertices with more backward edges in the pattern so that denser subpatterns are matched first. In our experiments this heuristic is sufficient to ensure that no arbitrarily low quality matching order penalizes the pattern-aware approach. An in-depth study on various subgraph matching approaches including orders can be found at Sun et al. [119] and is out of the scope of this work.

6.4.1 Single-pattern algorithms

Cliques (k-CL, Figure 6.14). Overall, the algorithm representing the PASE strategy exhibits superior performance compared to its POSE counterpart. This is because both strategies require a single application step with PASE having the advantage of avoiding the expensive filtering calls of POSE that are applied to an increasing number of invalid subgraphs. Finally, since the number of clique candidates may be progressively pruned as the recursive subgraph enumeration advances, the Custom algorithm (see Figure 6.3a) is able to outperform the standard approaches in almost all configurations, whenever the amount of work is sufficient to, justify maintaining local views of the input graph on each execution thread - as required by the parallel version of KClist [25].

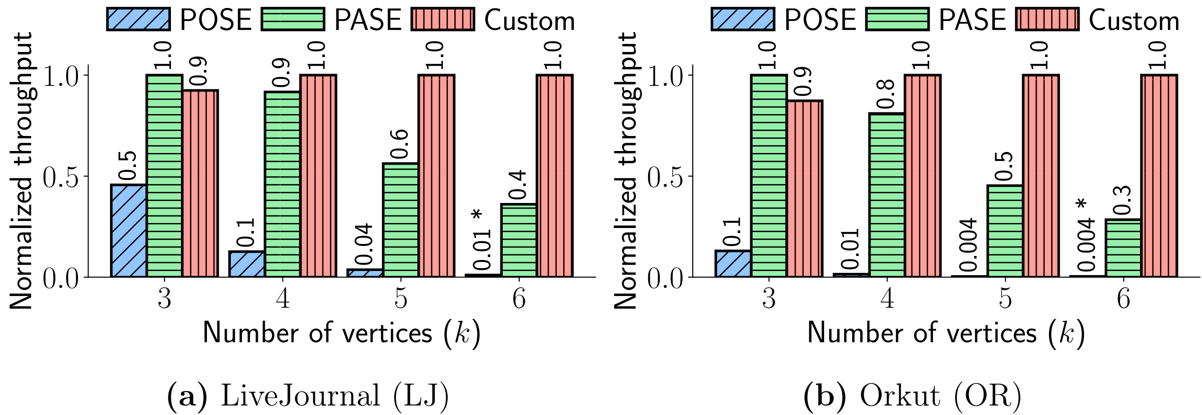
Figure 6.13: Pattern and label queries used to evaluate algorithms.



Source: Made by the author.

Pattern querying (ρ-PQ, Figure 6.15). Overall the POSE strategy is inferior for all configurations considered, with many executions that do not even found one single output subgraph within the timeout. This confirms the natural hypothesis that pattern-aware computation is best whenever the target patterns are known and generated apriori. Regarding the two remaining alternatives - PASE and Custom - that leverage pattern information during subgraph enumeration, we observe that the latter outperforms the former in almost all configurations. The exceptions to this rule happens in $D8$ for Live-Journal, and in $D6$ for Orkut. A careful examination of these results offers the nuanced but interesting finding that the frequency and density of the target pattern alone is insuffi-

Figure 6.14: Throughput: Cliques (k-CL).

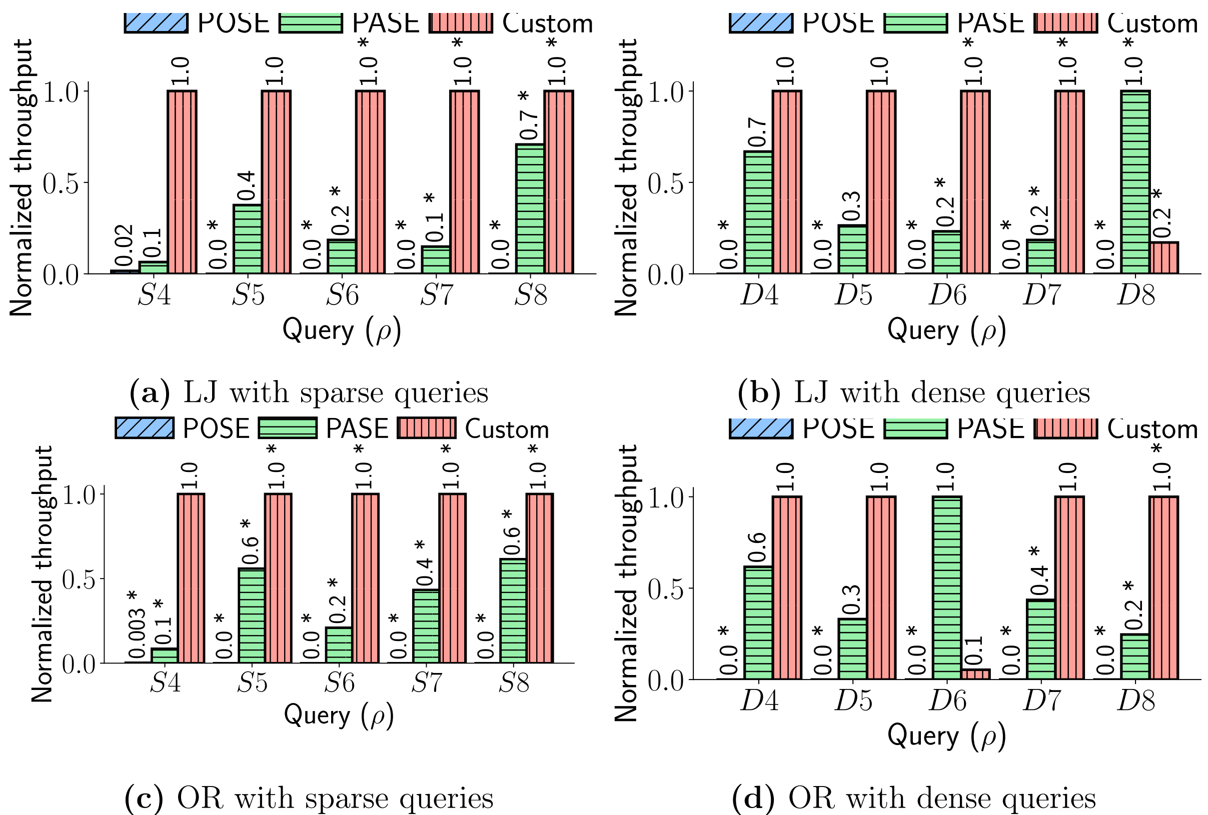


(a) LiveJournal (LJ) **(b)** Orkut (OR)

Source: Made by the author.

cient to explain the best alternative. Essentially one must also consider the the minimum vertex cover for the input pattern (i.e. the optimized algorithm considered for pattern querying). In fact, both of these exceptions represent large patterns that are cliques and in this case, the minimum vertex cover is always all vertices but one. This makes the custom strategy least effective because matching the cover is almost the same as matching the entire pattern.

Figure 6.15: Throughput: Pattern querying (ρ-PQ).



(a) LJ with sparse queries **(b)** LJ with dense queries

(c) OR with sparse queries **(d)** OR with dense queries

Source: Made by the author.

Discussion. In general, PASE outperforms POSE in this single-pattern setting thanks to its pre-computed execution plan that avoid spurious subgraph visits. Specifically, PASE's overhead of generating and matching each pattern individually pays off since these problems concern only one pattern. On the other hand, Custom algorithms tend to be more work-efficient than competitors in most cases but in some nuanced cases can be inefficient. This means that as we aim towards more specialized algorithms for a given application scenario, especially for GPM, it may be a good practice to fully understand their limitations first. For instance, we observed that for some configurations (especially the ones concerning more dense query patterns) the Custom strategy provided by MCVC [67, 60] is not as efficient as PASE, mostly because in these cases the minimum connected vertex cover tends to be of almost the same size as the query pattern.

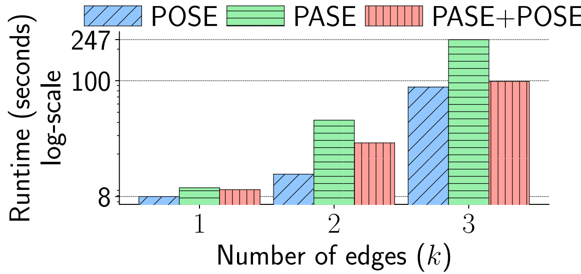
Thereby, we see an opportunity for the development of adaptable search strategies within GPM systems capable of understanding and learning the most appropriate subgraph exploration paradigm conditioned on pattern input and other related contextual features of the graph. In particular, some application scenarios could really benefit from switching between available exploration strategies at runtime to cope with unpredictable input workloads. For instance, an automated GPM system should be able to determine that POSE paradigm is not adequate for single-pattern problems and moreover, that depending on the characteristics of the query data for pattern querying PASE may be a more accurate design choice compared to MCVC.

6.4.2 Multi-pattern algorithms with pattern-driven filter

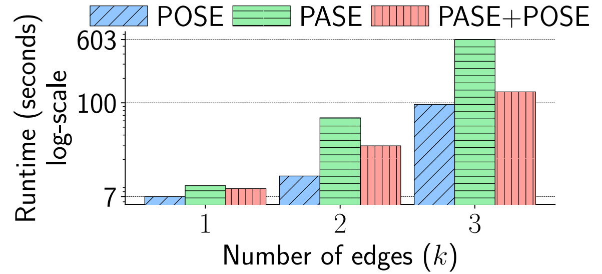
Frequent subgraph mining (k-FSM-α, Figure 6.16). As the output of FSM is the set of frequent patterns and supports, no trivial measure of execution progress exist and we report the runtime of configurations in which at least one of the alternative algorithms finished within the time limit. Overall, POSE is more effective in the majority of configurations, especially for Mico and Patents datasets. An exception occurs in k-FSM-$200K$ on Youtube in which PASE exhibits better performance (Figure 6.16f). The reason for that is directly related to the number of steps required by the PASE strategy, on those scenarios. For instance, while 3-FSM-$20K$ and 4-FSM-$20K$ on Patents (Figure 6.16d) require 1529 and 4463 steps, respectively, 3-FSM-$200K$ and 4-FSM-$200K$ on Youtube require much less: 34 and 115 respectively. Not unexpectedly, PASE+POSE represents a middle ground in terms of the number of steps required and its performance lies between the two baseline strategies. Additionally, the worst case scenario in terms of how many steps a particular algorithm requires can be determined at runtime for PASE and

PASE+POSE from the set of already known frequent patterns and possible extensions to them.

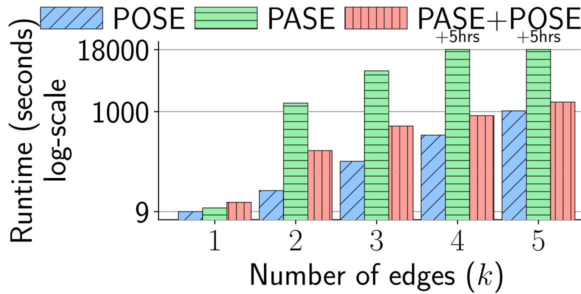
Figure 6.16: Runtime: Frequent subgraph mining (k-*FSM*-α).



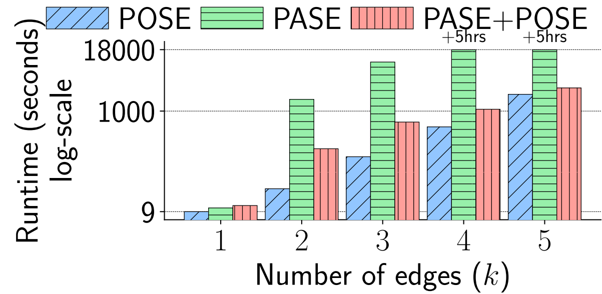
(a) k-*FSM*-4K on MI



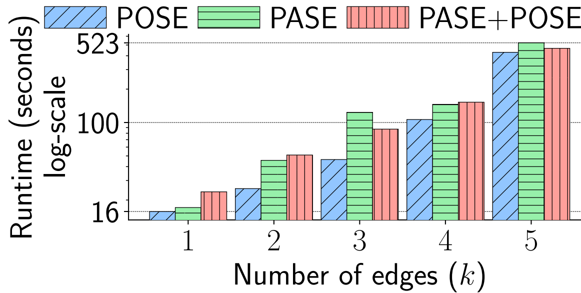
(b) k-*FSM*-3K on MI



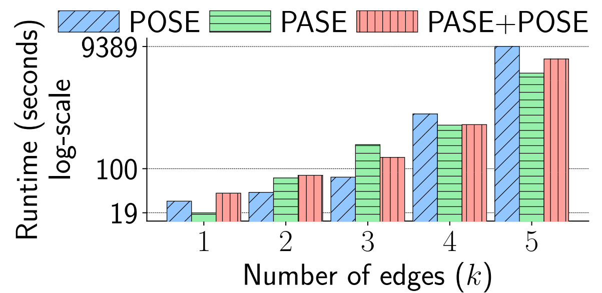
(c) k-*FSM*-21K on PA



(d) k-*FSM*-20K on PA



(e) k-*FSM*-250K on YO

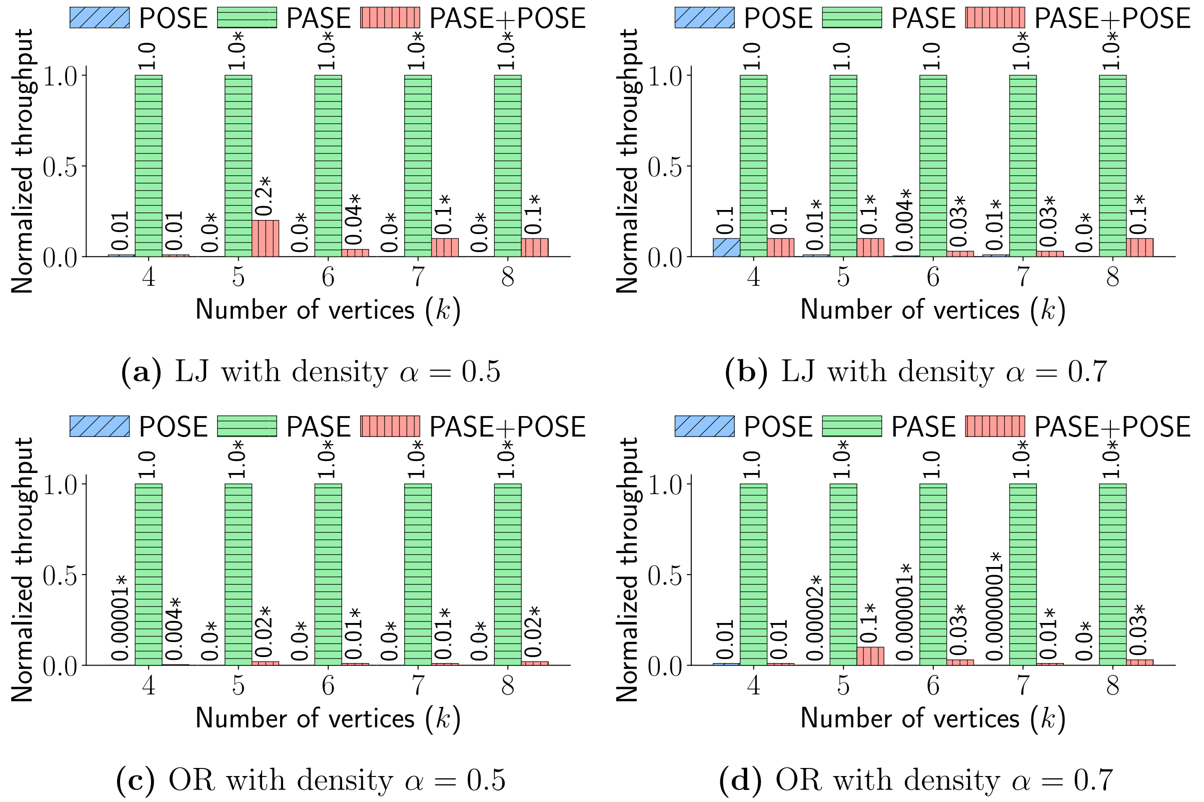


(f) k-*FSM*-200K on YO

Source: Made by the author.

Quasi cliques (k-*QC*-α, Figure 6.17). For the configurations considered, we see a vast superiority of the PASE strategy. Compared to POSE and PASE+POSE, the PASE approach is more efficient in these settings since most queried patterns exist in the input graph, which makes the overhead of PASE to payoff. However, as the size of subgraphs increases and the number of pattern candidates in PASE increases exponentially (increasing overhead) and one starts evaluating patterns that are not found in the input graph. For instance, in Orkut, PASE submits 5 application steps with no output for 7-QC-0.5 and 41 for 8-QC-0.5, indicating that this quantity tends to become greater and incurs in higher overhead for pattern-aware paradigm, even for unlabeled applications such as quasi clique finding on such problems.

Figure 6.17: Throughput: Quasi cliques (k-QC-α).



(a) LJ with density $\alpha = 0.5$

(b) LJ with density $\alpha = 0.7$

(c) OR with density $\alpha = 0.5$

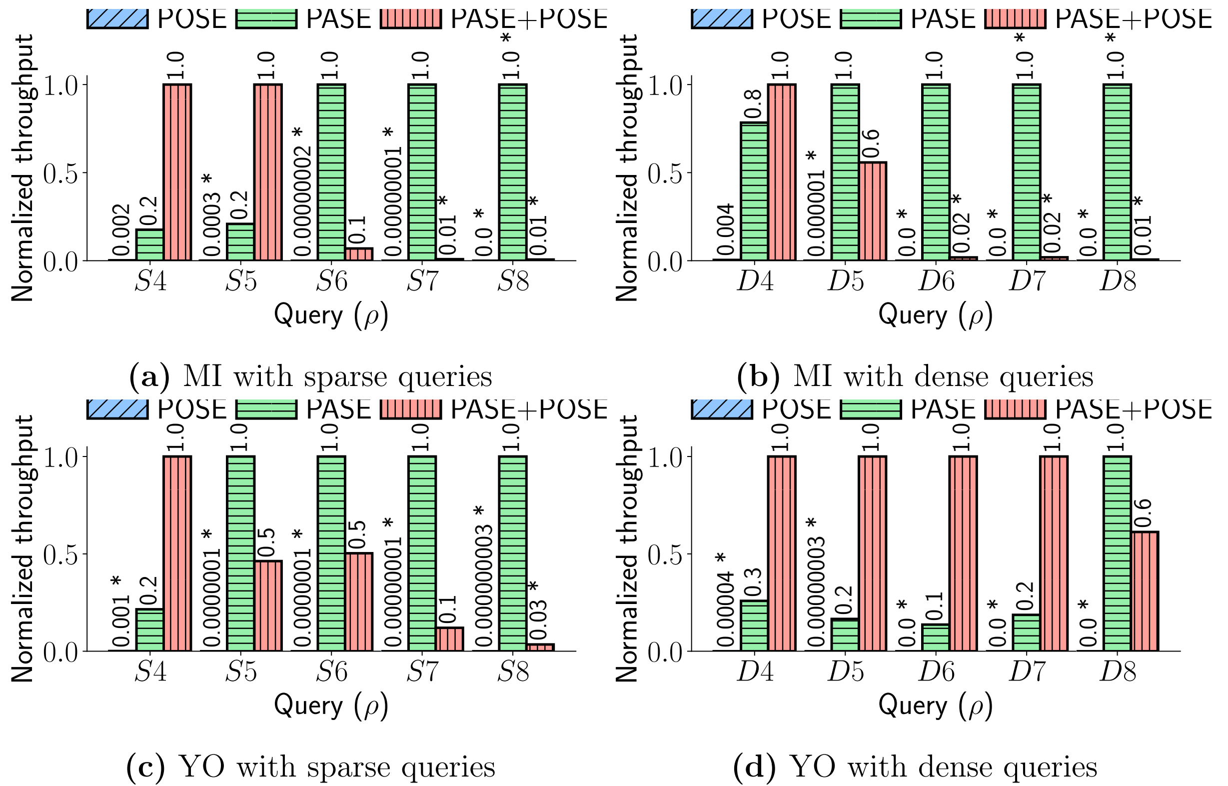
(d) OR with density $\alpha = 0.7$

Source: Made by the author.

Query specialization (ρ-QS, Figure 6.18). The POSE strategy - which relies on expensive visit/filter operations in order to ensure that the input pattern is contained in the enumerated subgraph - performs poorly on this task. The PASE approach is superior in almost all the scenarios, especially in Mico, which is a very dense dataset with overrepresented labels and patterns. Drilling into the comparative analysis of PASE vs. PASE+POSE (hybrid variant), let us consider the Youtube results for $D7$-QS and $D8$-QS, respectively. In the former, the hybrid variant is more efficient and we observe that 261 of 648 ($\approx$ 40%) steps in PASE finishes with zero output. In the latter, PASE is slightly more efficient and 119 of 542 ($\approx$ 22%) steps in PASE finishes with zero output. A similar behavior can be observed for the other scenarios, which leads us to conclude that there is a correlation between spurious steps (the ones that return zero output) and PASE performance. The hybrid approach, on the other hand, is not prone to this because it matches pattern ρ *first* (i.e. the subpattern), and then extends it with one additional edge, which prevents the runtime from spurious querying steps. This behavior depends on the occurrence of each pattern in the underlying network, which explains why Mico (dense and with most patterns occurring in the dataset) exhibit the best results for PASE.

Discussion. POSE is not an adequate exploration strategy when the pruning conditions are subject to subgraph's pattern structure *and* particularly if they are known apriori, such

Figure 6.18: Throughput: Query specialization (ρ-QS).



(a) MI with sparse queries

(b) MI with dense queries

(c) YO with sparse queries

(d) YO with dense queries

Source: Made by the author.

as quasi cliques and query specialization. However, for larger subgraph sizes and especially in a labeled context, PASE may incur in substantial spurious work by matching patterns that do not occur in the graph and hybrid approaches can be an interesting alternative to redress this issue. We observe that this PASE overhead does not affect so much the performance of quasi cliques, for example. This is actually because, by definition, the predicate condition that selects quasi cliques is able to provide a large reduction in the number of patterns that must be matched on each application step, mitigating PASE's overhead. There are also some applications in which POSE can be more efficient, such as FSM. Although a multi-pattern algorithm with pattern-driven filter, this kernel exhibit one defining difference: its filtering condition based on pattern structure (pattern-driven) must be determined at runtime, as FSM requires the computation of supports to back up threshold pruning. Thus, an FSM algorithm is unable to determine a priori which patterns are *certain* to be frequent and should be extended/matched, resulting in many spurious application steps for PASE.

Hence, we see significant potential in hybrid subgraph exploration paradigms (such as PASE+POSE) towards mitigating PASE overheads of spurious querying of patterns, especially for larger spaces where the number of patterns becomes exponentially larger. It remains a challenge to determine prior to execution which alternative may be the best on each case. Determining whether the overhead of multiple application steps pays off may be even more challenging considering that in real-world sparse graphs, the distribution

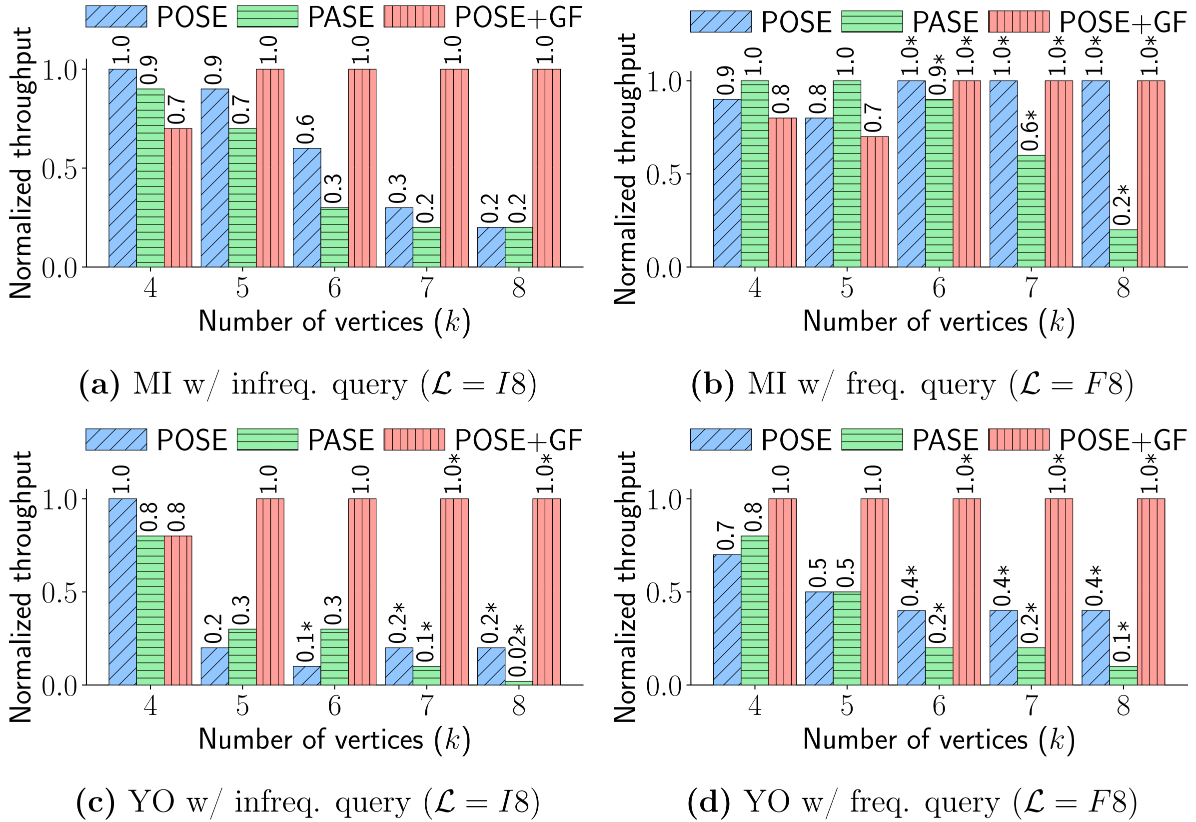
of subgraphs per patterns is very skewed (characteristic that we demonstrate empirically in the next Section), which implies in even more skewed application steps for PASE. We anticipate that GPM systems would benefit from pragmatic (contextual) knowledge of the input data in order to automatically determine the most appropriate paradigm (or combination of them) for a particular task and moreover, to better schedule an increasingly skewed set of application steps towards mitigating submission overheads. Additionally, such adaptive choice of algorithm may also be applied at runtime, as long as these this contextual knowledge and estimates are captured from each application step.

6.4.3 Multi-pattern algorithms with label-driven filter

Label search (k-LS-$\mathcal{L}$, Figure 6.19). For Mico, the PASE strategy exhibits larger overhead for small subgraph sizes since it is modeled as a multi-step application, especially when the amount of work is not substantial (e.g. 4-LS-$I8$). For larger values of k this overhead starts to payoff and the performance becomes roughly equivalent to the POSE approach. PASE also tends to become more inefficient for larger subgraph sizes (e.g. [4-8]-LS-$F8$ in Youtube). Such behavior is explained by the skewness of patterns w.r.t. their frequency – in particular, the algorithm spends much of the assigned time enumerating subgraphs of an individual pattern that represents a very small portion of the total output, affecting the rate at which output is generated. To support this claim we show in Figure 6.20 the number of subgraphs (output) per pattern found in PASE: greater skew indicates that many small application steps sum up increasingly larger submission overhead and consequently, more substantial drops in performance for larger subgraph sizes. The POSE+GF algorithm is most effective for frequent labels ($F8$), since a larger valid subgraph space (output) tends to exacerbate the overhead and redundancy of filtering routines, applied on every enumeration level. More stable results can be observed for Youtube (lower skew in labels compared to Mico). In this case the gap between POSE and PASE in this scenario is small. Finally, the hybrid (POSE+GF) approach exhibit substantial improvement over the alternatives. For example, on Youtube - a larger dataset - graph reduction produces a very interesting effect of reducing the working set at runtime, improving memory efficiency and caching.

Minimal keyword search (k-MKS-$\mathcal{K}$, Figure 6.21). We do not see a clear winning strategy for this task - in keyword search, arbitrary labels are allowed to appear between query labels, making it non-trivial to determine *which patterns* map to valid subgraphs in the input graph (explains the lack of trend in the results, for example, for $7, 8$-MKS-$F4$ on

Figure 6.19: Throughput: Label search (k-LS-$\mathcal{L}$).



(a) MI w/ infreq. query ($\mathcal{L} = I8$)

(b) MI w/ freq. query ($\mathcal{L} = F8$)

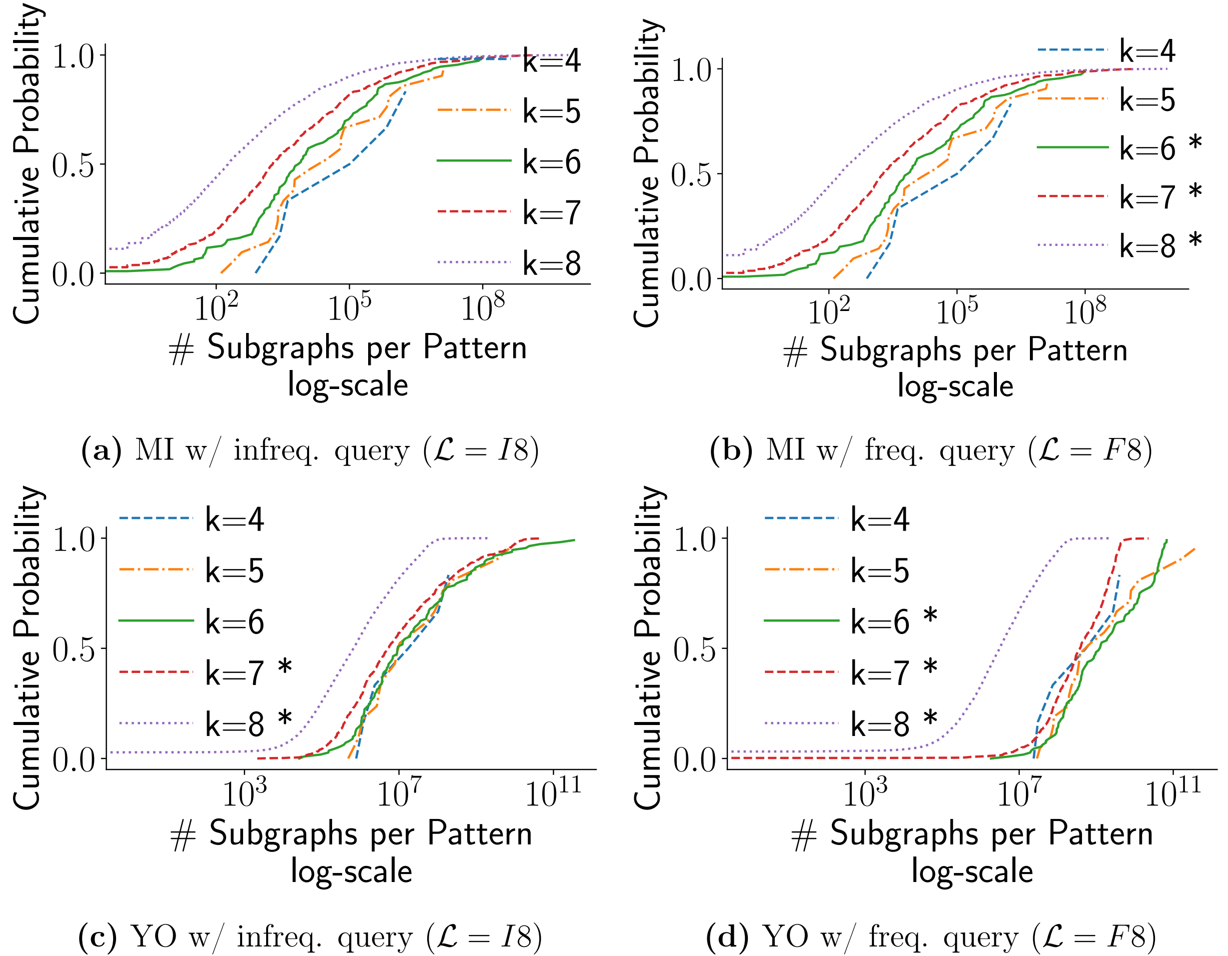
(c) YO w/ infreq. query ($\mathcal{L} = I8$)

(d) YO w/ freq. query ($\mathcal{L} = F8$)

Source: Made by the author.

Mico, Figure 6.21(b)). Overall POSE+GF is prone to produce better results, confirming the results concerning label search (Figure 6.19). Indeed, graph filtering (GF) improves POSE to be more efficient than PASE in some cases where this was not true initially (e.g. 5-MKS-$I4$ on Figure 6.21a). For larger subgraph sizes ($k = 8$), PASE can be less efficient due to the number of application steps that must be submitted in the algorithm – spurious querying in PASE may even make generated results to drop to zero (e.g. $k = 8$ on Figures 6.21c and 6.21d). For small subgraph sizes, PASE vs. POSE performance really depends on the query and input graph. A careful examination of these executions (not shown) indicates that good scenarios for PASE are those in which each submitted step has significant contributions in terms of the output. The POSE approach is not limited in enumerating patterns one at a time and combined with the fact that it is more coarse-grained in terms of submitted application steps results in a more efficient subgraph finding approach for such tasks. This behavior of PASE vs. POSE may indicate that for some applications there may be hidden relationships between label-based filtering conditions and pattern-based filtering conditions; and because vanilla PASE is not able to exploit those, it suffers from unpredictable performance issues in such settings.

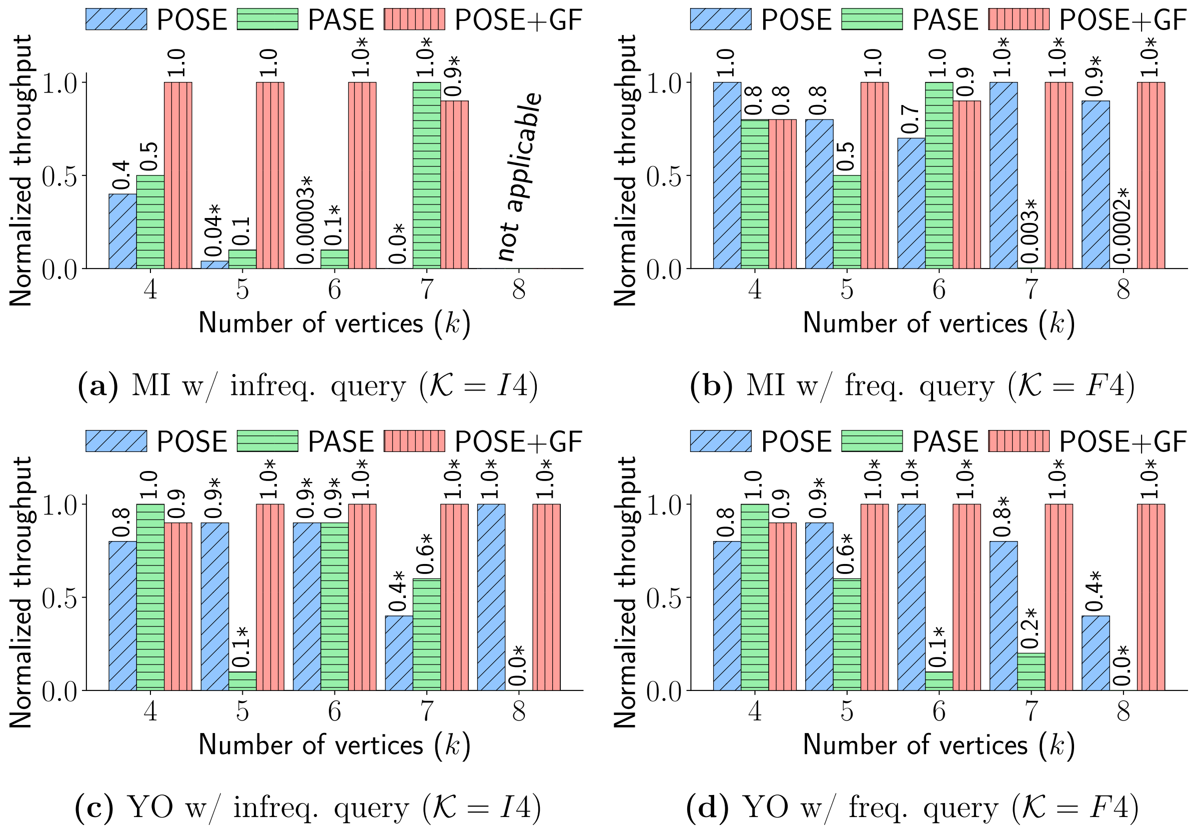
Figure 6.20: Number of Subgraphs per Pattern concerning label search results (Fig. 6.19): pattern-aware (PASE) is penalized by skewness of application steps. On Mico dataset (MI) this skewness is more expressive, resulting in larger drops of performance as subgraph size k increases. A similar behavior can be observed on Youtube dataset (YO) but at a lower scale.



Source: Made by the author.

Discussion. In general, POSE and PASE remain competitive with one another in such tasks: PASE is not particularly suited for label-driven filtering however if the number of patterns queried is small, its improved subgraph enumeration method ($\mathtt{M_P}(\rho)$) can be more efficient by avoiding expensive calls to filtering primitives. The operation of PASE may generate an exponentially increasing number of steps of very skewed sizes in scale-free graphs (illustrated in Figure 6.20), which may impose constraints and complexities with respect to distributed scaling and performance – label-driven filters only exacerbate this disadvantage and adopting a pattern-aware paradigm may be challenging in such settings. On a different matter, for local filters such as label-driven, the graph optimization that pushes down filtering conditions to the input graph can provide substantial performance improvement as a reduced working set may imply in better memory efficiency (caching). Although we applied this optimization for POSE only, this could easily help to improve the performance of PASE algorithms or any other algorithm relying on anti-monotonic

Figure 6.21: Throughput: Minimal keyword search (k-MKS-$\mathcal{K}$). "not applicable" indicates that none of the algorithms produced output for that particular problem instance.



(a) MI w/ infreq. query ($\mathcal{K} = I4$)

(b) MI w/ freq. query ($\mathcal{K} = F4$)

(c) YO w/ infreq. query ($\mathcal{K} = I4$)

(d) YO w/ freq. query ($\mathcal{K} = F4$)

Source: Made by the author.

filters on subgraphs.

We see room for several novel strategies. First scaling PASE for larger subgraph sizes. Second, taming the exponentially growth of patterns through improved scheduling and/or hybrid approaches combining PASE and POSE to reduce the number of submitted tasks and their skewness. Third, we believe that understanding the relationships between label-driven filters and underlying subgraph patterns may be used to develop hybrid strategies to improve pruning efficiency and cost within such GPM systems. For instance, we observed many configurations where some labels of interest do not occur in many specific pattern structures. In this case, PASE could be substantially improved if with that information we determine a priori *which patterns should not be matched via an application step simply because no output is going to be generated from this.* The other way around, POSE could also benefit from it by early pruning subgraphs of (sub)patterns that are certain to not contain a specific set of labels.

Chapter 7

Final Remarks

This work is built upon an increasing interest in Graph Pattern Mining (GPM) algorithms that have important applications in data analytics on graphs. We identify an absence of proper formal model for describing general-purpose graph pattern mining algorithms. In particular, we observe that existing approaches are too limited in representing multiple graph pattern mining paradigms (pattern-aware and pattern-oblivious), or they fail in modularity and conciseness for application design. In this work we propose a primitive-based model that can be used to express GPM algorithms using different paradigms (Chapter 4), a proof of concept distributed implementation of that model (Chapter 5), and an extensive experimental analysis of GPM paradigms in the context of general-purpose GPM systems.

From a systems perspective, we implement the primitive-based model in an open-source system called Fractal that enables improved productivity in designing and integrating complex GPM algorithms. We demonstrate empirically the effectiveness of our model implementation in proving competitive performance against existing approaches without sacrificing the expressiveness of algorithms or the composability of operators.

Finally, we leverage our system implementation of the primitive-based model to close some gaps concerning the trade-offs among GPM paradigms. In this context, we present an experimental study that considers a wide range of application scenarios and compares multiple algorithmic solutions: ranging from standard pattern-aware and pattern-oblivious algorithms to promising optimizations and hybrid alternatives. We believe that because we study these algorithms under the same execution settings and underlying application model, our findings are more reliable and useful for the GPM community. Our experiments reveal that there is no silver bullet when it comes to choosing subgraph enumeration paradigms, be pattern-aware or pattern-oblivious. In particular, we confirm that pattern-oblivious is inferior whenever the search space is pruned based on pattern structure, but, on the other hand, pattern-aware suffers from an exponentially increasing overhead of querying individual patterns one at a time, especially in distributed large-scale environments. Our findings go beyond performance comparison and show promising directions for future work, which we discuss next.

7.1 Limitations and Future work

This work can be improved in a few fronts. On the systems front, we see opportunity for exploring new parallelization strategies, graph storage models, and hybrid computing architectures. First, adaptive load balancing schemes such as the hierarchical work stealing proposed in this work come with nontrivial thread synchronization costs. We anticipate that parallel work efficiency could be improved if more coarse-grained load balancing strategies could be used [123], especially considering the skewness of work in GPM workloads. Naturally the challenge in doing this is to predict how coarse should the parallel tasks be and we believe that prior knowledge about the underlying input graph is a promising direction to handle this. Second, we assume in this work that the input graph is replicated on each worker node, which may be an issue considering that real-world huge graphs were never so ubiquitous. We highlight at this point that GPM is an expensive task even for small graphs because of its intractable nature. However, we see big opportunities to develop subgraph enumeration engines that work independently on different parts of a graph and are able to combine results with little effort. This is somewhat an well studied problem in graph partitioning community [47, 118], but we highlight that as far as we know this has not been explored in the context of subgraph enumeration algorithms that have the potential to be even more challenging. Third, considering the rise of GPU processing applied in GPM context [21, 17, 38] we observe that there is some interesting directions for future work concerning hybrid system architectures that combine CPU(s) and GPU(s) processing to accomplish the same GPM task. We understand that there are specifics to GPM tasks that may favor GPU processing (more dense and regular computation on dense regions of the input graph) or the contrary to favor CPU processing (on sparse/irregular computation on sparse regions of the input graph). Given this intuition about the heterogeneous nature of GPM algorithms, we believe hybrid approaches could be further explored. Fourth, we also see opportunity for developing better performance diagnosis tools for GPM, capable of identifying bottlenecks of GPM programs at low cost. Other interesting research direction for systems is characterizing and optimizing energy consumption of GPM workloads, especially if we consider relaxed versions of these problems where partial output is allowed.

On the applications and models front, we believe that this work may allow some interesting directions concerning multi-paradigm GPM algorithm design and automated GPM. First, we consider that this work is a first step towards automated, cost-based optimization GPM processing. Indeed, when the operators of some model are very well defined we may start reasoning about which combinations of operators or which optimization may work best given some contextual information about the underlying data. Notice that this is a very well established research area in the database community, how-

ever, we are not aware of any application model that accomplishes that specifically for GPM. The closest to this are Neo4j[1] and similar systems [63] from the graph database community that try to express querying on graph using some methodology similar to relational algebra to optimize execution plans. However, we highlight that the focus of these graph databases are not on subgraph enumeration and consequently, not directly applied to GPM. Second, many insights that we provide in Chapter 6 represent promising directions for future work. For instance, we show consistent opportunity to explore hybrid multi-paradigm approaches to GPM algorithm design and pruning strategies that could explore latent implicit conditions for a given pair of input data and program that could be applied directly to the input graph or inferred based on pattern and/or labeling information. Third, because trade-offs exist among alternative paradigms and they interact differently depending on the characteristics of the input data, we plan to explore even higher levels of abstraction capable of choosing algorithm designs based on problem constraints that are most likely to produce better performance results. This aspect may even facilitate the proposal of improved declarative languages [63, 6, 13] for GPM or even low-code/no-code tools for practitioners with less expertise on programming.

[1] https://neo4j.com

References

[1] 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 61:1–61:12, Piscataway, NJ, USA, 2016. IEEE Press.

[2] Foto N. Afrati, Dimitris Fotakis, and Jeffrey D. Ullman. Enumerating subgraph instances using map-reduce. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 62–73, Washington, DC, USA, 2013. IEEE Computer Society.

[3] Monica Agrawal, Marinka Zitnik, and Jure Leskovec. Large-scale analysis of disease pathways in the human interactome. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, 23:111–122, 2018. 29218874[pmid].

[4] Rakesh Agrawal, Tomasz Imieliundefinedski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 207–216, New York, NY, USA, 1993. Association for Computing Machinery.

[5] E. G. Allan Jr., W. H. Turkett, and E. W. Fulp. Using network motifs to identify application protocols. In *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*, pages 1–7, Nov 2009.

[6] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[8] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Appl. Math.*, 65(1–3):21–46, March 1996.

[9] Austin R. Benson, David F. Gleich, and Jure Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[10] M. Besta, Z. Vonarburg-Shmaria, Y. Schaffner, L. Schwarz, G. Kwasniewski, L.Gianinazzi, J. Beránek, K. Janda, T. Holenstein, S. Leisinger, P. Tatkowski, E. Ozdemir, A. Balla, M. Copik, P. Lindenberger, P. Kalvoda, M. Konieczny, O. Mutlu, and T. Hoefler. Graphminesuite: Enabling high-performance and programmable graph mining algorithms with set algebra. In *Proceedings of the 47th International Conference on Very Large Data Bases*, 2021.

[11] M. A. Bhuiyan and M. Al Hasan. An iterative mapreduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 27(3):608–620, March 2015.

[12] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. *Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs*, page 458–473. Association for Computing Machinery, New York, NY, USA, 2021.

[13] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.*, 9(13):1425–1436, September 2016.

[14] Christian Borgelt. Canonical forms for frequent graph mining. In Reinhold Decker and Hans J. Lenz, editors, *Advances in Data Analysis*, pages 337–349, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[15] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[16] Björn Bringmann and Siegfried Nijssen. What is frequent in a single graph? In *Proceedings of the 12th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, PAKDD'08, pages 858–863, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] 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*

Thirteenth EuroSys Conference, EuroSys '18, pages 32:1–32:12, New York, NY, USA, 2018. ACM.

[18] 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 2022, page 43–55, New York, NY, USA, 2022. Association for Computing Machinery.

[19] 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.

[20] 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 ACM International Conference on Supercomputing*, ICS '21, page 378–391, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(8):1190–1205, April 2020.

[22] 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.

[23] Xu Cheng, Cameron Dale, and Jiangchuan Liu. Dataset for "statistics and social network of youtube videos". http://netsg.cs.sfu.ca/youtubedata/, 2008.

[24] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. Local search of communities in large graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 991–1002, New York, NY, USA, 2014. Association for Computing Machinery.

[25] 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, Republic and Canton of Geneva, Switzerland, 2018. International World Wide Web Conferences Steering Committee.

[26] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. Graphframes: An integrated api for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management*

Experiences and Systems, GRADES '16, pages 2:1–2:8, New York, NY, USA, 2016. ACM.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[28] Imre Derényi, Gergely Palla, and Tamás Vicsek. Clique Percolation in Random Networks. *Physical Review Letters*, 94(16):160202, April 2005.

[29] Vinicius Dias, Rubens Moreira, Wagner Meira Jr., and Dorgival Guedes. Diagnosing performance bottlenecks in massive data parallel programs. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 273–276, May 2016.

[30] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira Jr., 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, New York, NY, USA, 2019. ACM.

[31] Chao-Yi Dong, Dongkwan Shin, Sunghoon Joo, YoonKey Nam, and Kwang-Hyun Cho. Identification of feedback loops in neural networks based on multi-step granger causality. *Bioinformatics*, 28(16):2146–2153, August 2012.

[32] Ahmad Naser eddin and Pedro Ribeiro. Scalable subgraph counting using mapreduce. In *Proceedings of the Symposium on Applied Computing*, SAC '17, pages 1574–1581, New York, NY, USA, 2017. ACM.

[33] Shady Elbassuoni and Roi Blanco. Keyword search over rdf graphs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 237–242, New York, NY, USA, 2011. ACM.

[34] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528, March 2014.

[35] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in large sparse real-world graphs. *ACM J. Exp. Algorithmics*, 18, November 2013.

[36] S. M. Faisal, Srinivasan Parthasarathy, and P. Sadayappan. Global graphs: A middleware for large scale graph processing. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 33–40, 2014.

[37] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. Association rules with graph patterns. *Proc. VLDB Endow.*, 8(12):1502–1513, August 2015.

[38] Samuel Ferraz, Vinicius Dias, Carlos H.C. Teixeira, George Teodoro, and Wagner Meira. Efficient strategies for graph pattern mining algorithms on gpus. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 110–119, Nov 2022.

[39] Irene Finocchi, Marco Finocchi, and Emanuele G. Fusco. Clique counting in mapreduce: Algorithms and experiments. *J. Exp. Algorithmics*, 20:1.7:1–1.7:20, October 2015.

[40] Ming Gao, Ee-Peng Lim, David Lo, and Philips Kokoh Prasetyo. On detecting maximal quasi antagonistic communities in signed graphs. *Data Mining and Knowledge Discovery*, 30(1):99–146, 2016.

[41] P.-L. Giscard, P. Rochet, and R. C. Wilson. Evaluating balance on social networks from their simple cycles. *Journal of Complex Networks*, 5(5):750–775, 2017.

[42] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingan, Derek Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 2–2, Berkeley, CA, USA, 2015. USENIX Association.

[43] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[44] 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, Berkeley, CA, USA, 2012. USENIX Association.

[45] 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, Berkeley, CA, USA, 2014. USENIX Association.

[46] Joshua A. Grochow and Manolis Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Proceedings of the 11th Annual International Conference on Research in Computational Molecular Biology*, RECOMB'07, pages 92–106, Berlin, Heidelberg, 2007. Springer-Verlag.

[47] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. *GPU-Accelerated Subgraph Enumeration on Partitioned Graphs*, page 1067–1082. Association for Computing Machinery, New York, NY, USA, 2020.

[48] Bronwyn H Hall, Adam B Jaffe, and Manuel Trajtenberg. The nber patent citation data file: Lessons, insights and methodological tools. Working Paper 8498, National Bureau of Economic Research, October 2001.

[49] 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, page 1429–1446, New York, NY, USA, 2019. Association for Computing Machinery.

[50] Steven Hill, Bismita Srichandan, and Rajshekhar Sunderraman. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, BCB '12, pages 661–666, New York, NY, USA, 2012. ACM.

[51] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[52] Florian Hoffman and Darin Krasle. Fraud detection using network analysis, sep 2015. Patent No. EP2884418A1, Filed September 1st., 2014, Issued June 17th., 2015.

[53] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: A dsl for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM.

[54] Bryan Hooi, Kijung Shin, Hemank Lamba, and Christos Faloutsos. Telltail: Fast scoring and detection of dense subgraphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):4150–4157, Apr. 2020.

[55] Bryan Hooi, Hyun Ah Song, Alex Beutel, Neil Shah, Kijung Shin, and Christos Faloutsos. Fraudar: Bounding graph fraud in the face of camouflage. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 895–904, New York, NY, USA, 2016. Association for Computing Machinery.

[56] Jun Huan, Wei Wang, and Jan Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, pages 549–, Washington, DC, USA, 2003. IEEE Computer Society.

[57] J. Huang, Q. Zhu, M. Wang, L. Zhou, Z. Zhang, and D. Zhang. Coherent pattern in multi-layer brain networks: Application to epilepsy identification. *IEEE Journal of Biomedical and Health Informatics*, pages 1–1, 2019.

[58] Jiewen Huang, Daniel J Abadi, and Kun Ren. Scalable sparql querying of large rdf graphs. *Proceedings of the VLDB Endowment*, 2011.

[59] Eslam Hussein, Abdurrahman Ghanem, Vinicius Vitor dos Santos Dias, Carlos H.C. Teixeira, Ghadeer AbuOda, Marco Serafini, Georgos Siganos, Gianmarco De Francisci Morales, Ashraf Aboulnaga, and Mohammed Zaki. Graph data mining with arabesque. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1647–1650, New York, NY, USA, 2017. ACM.

[60] 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, New York, NY, USA, 2020. Association for Computing Machinery.

[61] Kasra Jamshidi and Keval Vora. A deeper dive into pattern-aware subgraph exploration with peregrine. *SIGOPS Oper. Syst. Rev.*, 55(1):1–10, June 2021.

[62] Xing Jiang, Hui Xiong, Chen Wang, and Ah-Hwee Tan. Mining globally distributed frequent subgraphs in a single labeled graph. *Data Knowl. Eng.*, 68(10):1034–1058, oct 2009.

[63] Martin Junghanns, Max Kießling, Niklas Teichmann, Kevin Gómez, André Petermann, and Erhard Rahm. Declarative and distributed graph analytics with gradoop. *Proc. VLDB Endow.*, 11(12):2006–2009, August 2018.

[64] 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, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[65] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. Gbase: An efficient analysis platform for large graphs. *The VLDB Journal*, 21(5):637–650, October 2012.

[66] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[67] 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, page 1231–1245, New York, NY, USA, 2016. Association for Computing Machinery.

[68] Hyunjoon Kim, Yunyoung Choi, Kunsoo Park, Xuemin Lin, Seok-Hee Hong, and Wook-Shin Han. Fast subgraph query processing and subgraph matching via static and dynamic equivalences. *The VLDB Journal*, 2022.

[69] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, September 1999.

[70] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, nov 2011.

[71] Lauri Kovanen, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs reveal homophily, gender-specific patterns, and group talk in call sequences. *Proceedings of the National Academy of Sciences*, 2013.

[72] Pradeep Kumar and H. Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 249–263, Boston, MA, February 2019. USENIX Association.

[73] Michihiro Kuramochi and George Karypis. Finding frequent patterns in a large sparse graph*. *Data Min. Knowl. Discov.*, 11(3):243–271, November 2005.

[74] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[75] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce. *Proc. VLDB Endow.*, 8(10):974–985, June 2015.

[76] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. Scalable distributed subgraph enumeration. *Proc. VLDB Endow.*, 10(3):217–228, November 2016.

[77] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow.*, 12(10):1099–1112, June 2019.

[78] D. Brian Larkins, James Dinan, Sriram Krishnamoorthy, Srinivasan Parthasarathy, Atanas Rountev, and P. Sadayappan. Global trees: A framework for linked data structures on distributed memory parallel systems. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–13, 2008.

[79] Y. I. Leon-Suematsu, K. Inui, S. Kurohashi, and Y. Kidawara. Web Spam Detection by Exploring Densely Connected Subgraphs. In *2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, volume 1, pages 124–129, August 2011. bibtex*[number=] ISSN:.

[80] Rong-Hua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. Ordering heuristics for <i>k</i>-clique listing. *Proc. VLDB Endow.*, 13(12):2536–2548, July 2020.

[81] David Liben-Nowell and Jon Kleinberg. The link-prediction problem for social networks. *J. Am. Soc. Inf. Sci. Technol.*, 58(7):1019–1031, May 2007.

[82] W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in mapreduce. In *2014 IEEE 30th International Conference on Data Engineering*, pages 844–855, March 2014.

[83] W. Lu, G. Chen, A. K. H. Tung, and F. Zhao. Efficiently extracting frequent subgraphs using mapreduce. In *2013 IEEE International Conference on Big Data*, pages 639–647, Oct 2013.

[84] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 1–1, Berkeley, CA, USA, 2015. USENIX Association.

[85] 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 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[86] Daniel Mawhirter, Sam Reinehr, Wei Han, Noah Fields, Miles Claver, Connor Holmes, Jedidiah McClurg, Tongping Liu, and Bo Wu 0002. Dryadic: Flexible and

fast graph pattern matching at scale. In Jaejin Lee and Albert Cohen, editors, *30th International Conference on Parallel Architectures and Compilation Techniques, PACT 2021, Atlanta, GA, USA, September 26-29, 2021*, pages 289–303. IEEE, 2021.

[87] 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, New York, NY, USA, 2019. ACM.

[88] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

[89] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 14–14, Berkeley, CA, USA, 2015. USENIX Association.

[90] Changping Meng, S Chandra Mouli, Bruno Ribeiro, and Jennifer Neville. Subgraph pattern neural networks for high-order graph evolution prediction, 2018.

[91] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *J. Mach. Learn. Res.*, 17(1):1235–1241, January 2016.

[92] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[93] Soheila Molaei, Nima Ghanbari Bousejin, Hadi Zare, Mahdi Jalili, and Shirui Pan. Learning graph representations with maximal cliques. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–8, 2021.

[94] Davide Mottin, Francesco Bonchi, and Francesco Gullo. Graph query reformulation with diversity. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, page 825–834, New York, NY, USA, 2015. Association for Computing Machinery.

[95] Mohammad Hossein Namaki, Yinghui Wu, Qi Song, Peng Lin, and Tingjian Ge. Discovering graph temporal association rules. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, pages 1697–1706, New York, NY, USA, 2017. ACM.

[96] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[97] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 349–365, Berkeley, CA, USA, 2016. USENIX Association.

[98] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 675–690, New York, NY, USA, 2015. ACM.

[99] Fan Ou and Jian Xu. S3feature: A static sensitive subgraph-based feature for android malware detection. *Computers and Security*, 112:102513, 2022.

[100] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 184–200, New York, NY, USA, 2017. Association for Computing Machinery.

[101] F. Perez and B. E. Granger. Ipython: A system for interactive scientific computing. *Computing in Science Engineering*, 9(3):21–29, May 2007.

[102] Nataša Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 26(6):853–854, March 2010.

[103] Miao Qiao, Hao Zhang, and Hong Cheng. Subgraph matching: On compression and computation. *Proc. VLDB Endow.*, 11(2):176–188, October 2017.

[104] Hongchao Qin, Rong-Hua Li, Guoren Wang, Lu Qin, Yurong Cheng, and Ye Yuan. Mining periodic cliques in temporal networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1130–1141, 2019.

[105] Z. Qiu, J. Wu, W. Hu, B. Du, G. Yuan, and P. Yu. Temporal link prediction with motifs for social networks. *IEEE Transactions on Knowledge and Data Engineering*, (01):1–1, aug 2021.

[106] Abdul Quamar, Amol Deshpande, and Jimmy Lin. Nscale: Neighborhood-centric large-scale graph analytics in the cloud. *The VLDB Journal*, 25(2):125–150, April 2016.

[107] Pedro Ribeiro, Pedro Paredes, Miguel E. P. Silva, David Aparicio, and Fernando Silva. A survey on subgraph counting: Concepts, algorithms, and applications to network motifs and graphlets. *ACM Comput. Surv.*, 54(2), March 2021.

[108] Pedro Ribeiro and Fernando Silva. G-tries: A data structure for storing and finding subgraphs. *Data Min. Knowl. Discov.*, 28(2):337–377, March 2014.

[109] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[110] Saeed Shahrivari and Saeed Jalili. Distributed discovery of frequent subgraphs of a network using mapreduce. *Computing*, 97(11):1101–1120, November 2015.

[111] Ping Shao, Yang Yang, Shengyao Xu, and Chunping Wang. Network embedding via motifs. *ACM Trans. Knowl. Discov. Data*, 16(3), oct 2021.

[112] 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 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 625–636, New York, NY, USA, 2014. ACM.

[113] 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, New York, NY, USA, 2013. ACM.

[114] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[115] Arlei Silva, Wagner Meira, and Mohammed J. Zaki. Mining attribute-structure correlated patterns in large attributed graphs. *Proc. VLDB Endow.*, 5(5):466–477, January 2012.

[116] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In Fernando Silva, Inês Dutra, and Vítor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, pages 451–462, Cham, 2014. Springer International Publishing.

[117] George M. Slota and Kamesh Madduri. Complex network analysis using parallel approximate motif counting. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 405–414, Washington, DC, USA, 2014. IEEE Computer Society.

[118] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, page 1222–1230, New York, NY, USA, 2012. Association for Computing Machinery.

[119] Shixuan Sun and Qiong Luo. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1083–1098, New York, NY, USA, 2020. Association for Computing Machinery.

[120] 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, New York, NY, USA, 2022. Association for Computing Machinery.

[121] 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, New York, NY, USA, 2015. ACM.

[122] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, November 2013.

[123] Ehsan Totoni, Subramanya R. Dulloor, and Amitabha Roy. A case against tiny tasks in iterative analytics. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 144–149, New York, NY, USA, 2017. Association for Computing Machinery.

[124] Johan Ugander, Lars Backstrom, and Jon Kleinberg. Subgraph frequencies: mapping the empirical and extremal geography of large graph collections. In *Proceedings of the 22nd international conference on World Wide Web - WWW '13*, pages 1307–1318, Rio de Janeiro, Brazil, 2013. ACM Press.

[125] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[126] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[127] Haixun Wang and Charu C. Aggarwal. *A Survey of Algorithms for Keyword Search on Graph Data*, pages 249–273. Springer US, Boston, MA, 2010.

[128] 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 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 763–782, Berkeley, CA, USA, 2018. USENIX Association.

[129] Z. Wang, R. Gu, W. Hu, C. Yuan, and Y. Huang. BENU: Distributed subgraph enumeration with backtracking-based framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 136–147, April 2019.

[130] Sebastian Wernicke. A faster algorithm for detecting network motifs. In Rita Casadio and Gene Myers, editors, *Algorithms in Bioinformatics*, pages 165–177, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[131] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.*, 7(14):1981–1992, October 2014.

[132] Da Yan, Guimu Guo, Jalal Khalil, M. Tamer Özsu, Wei-Shinn Ku, and John C. S. Lui. G-thinker: a general distributed framework for finding qualified subgraphs in a big graph with load balancing. *The VLDB Journal*, Aug 2021.

[133] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 721–, Washington, DC, USA, 2002. IEEE Computer Society.

[134] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowl. Inf. Syst.*, 42(1):181–213, January 2015.

[135] Yi Yang, Da Yan, Huanhuan Wu, James Cheng, Shuigeng Zhou, and John C.S. Lui. Diversified temporal subgraph pattern mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 1965–1974, New York, NY, USA, 2016. Association for Computing Machinery.

[136] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. A locality-aware energy-efficient accelerator for graph mining applications. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 895–907, 2020.

[137] Peipei Yi, Byron Choi, Sourav S. Bhowmick, and Jianliang Xu. Autog: a visual query autocompletion framework for graph databases. *The VLDB Journal*, 26(3):347–372, 2017.

[138] S. Yu, J. Xu, C. Zhang, F. Xia, Z. Almakhadmeh, and A. Tolba. Motifs in big networks: Methods and applications. *IEEE Access*, 7:183322–183338, 2019.

[139] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[140] M.J. Zaki, W. Meira, and W. Meira. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press, 2014.

[141] Huan Zhao, Xiaogang Xu, Yangqiu Song, Dik Lun Lee, Zhao Chen, and Han Gao. Ranking users in social networks with higher-order structures. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*, AAAI'18. AAAI Press, 2018.

[142] Huan Zhao, Yingqi Zhou, Yangqiu Song, and Dik Lun Lee. Motif enhanced recommendation over heterogeneous information network. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, pages 2189–2192, New York, NY, USA, 2019. ACM.

[143] Zhao Zhao, Guanying Wang, Ali R. Butt, Maleq Khan, V. S. Anil Kumar, and Madhav V. Marathe. Sahad: Subgraph analysis in massive networks using hadoop. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 390–401, Washington, DC, USA, 2012. IEEE Computer Society.