sbbd:03

Chapter 3

Practical Graph Pattern Mining: Systems, Applications, and Challenges

Vinícius Dias, Samuel Ferraz

Abstract

Graph Pattern Mining (GPM) refers to algorithms that extract and process subgraphs from larger graphs. This course covers the GPM fundamentals, an overview of GPM systems, and practical use case applications for students and professionals with basic graph and programming skills. It features Fractal and DuMato systems, two in-house GPM systems offering good experience and performance with such algorithms. Fractal, on Spark, provides an efficient API for algorithm design, while DuMato, on CUDA, enables efficient GPM on GPUs. The challenges of GPM algorithms and performance issues are discussed, providing a solid grasp of graph processing acceleration for efficient solutions that are valuable for practitioners and the database community.

Resumo

Mineração de Padrões em Grafos (sigla em inglês: GPM) refere-se a algoritmos que extraem e processam subgrafos de grafos maiores. Este curso aborda os fundamentos de GPM, uma visão geral dos sistemas de GPM e aplicações práticas para estudantes e profissionais com habilidades básicas em grafos e programação. Ele apresenta os sistemas Fractal e DuMato, dois sistemas GPM que oferecem uma boa experiência e desempenho com tais algoritmos. O Fractal, baseado em Spark, oferece uma API eficiente para o projeto de algoritmos, enquanto o DuMato, em CUDA, possibilita uma mineração eficiente em GPUs. Os principais desafios dos algoritmos de GPM e questões de desempenho são discutidos, fornecendo uma compreensão sólida da aceleração do processamento de grafos para soluções eficientes, valioso tanto para praticantes quanto para a comunidade de banco de dados.

3.1. Introduction

Graphs are widely used to model problems in various areas, including web applications, social media, biological networks, brain networks, conceptual graphs, among others. With

the popularization of large-scale systems and the easy access to large volumes of data, the demand for graph processing has been substantial [Fan 2022]. The scope of this course proposal is on a particular class of graph algorithms, recently referred to as Graph Pattern Mining (GPM). Figure 3.1 depicts the key aspects of GPM processing. Given an input graph, GPM algorithms visit its subgraphs of interest through a combinatorial procedure (subgraph enumeration) and, using an user-defined processing on the visited subgraphs, filter the search-space of subgraph candidates and generate the result. The relevance of GPM computation is multidisciplinary, including applications such as motif extraction from biological networks [Agrawal et al. 2018], frequent subgraph mining [Elseidy et al. 2014], subgraph searching over semantic data [Elbassuoni and Blanco 2011], social media network characterization [Ugander et al. 2013], community discovery [Benson et al. 2016], periodic community discovery [Qin et al. 2019], temporal hotspot identification [Yang et al. 2016], identification of dense subgraphs in social networks [Hooi et al. 2020], link spam detection [Leon-Suematsu et al. 2011], financial fraud detection [Hoffman and Krasle 2015], recommendation systems [Zhao et al. 2019], graph learning [Meng et al. 2018], to cite a few. Since GPM algorithms are often complex to develop (graph theory involved) and also expensive in terms of systems performance (combinatorics and high memory consumption), many general-purpose GPM systems emerged in the last decade to improve the user experience in those aspects. These systems provide implementations of subgraph enumeration and facilitate the creation of user-defined processing on subgraphs.

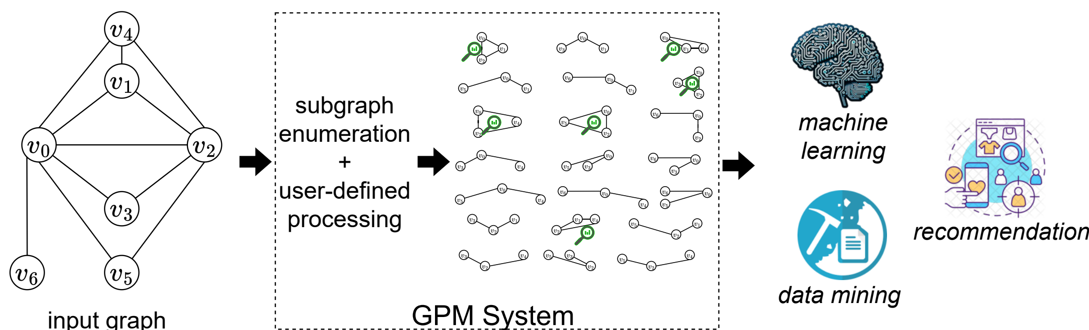


Figure 3.1. Overview of GPM processing over graph data.

In this course we are going to give an overview of GPM fundamentals, existing GPM systems, and use case scenarios that may benefit from such processing. We also discuss the main challenges that impact the design choices in state-of-the-art GPM systems. This course is intended for students and professionals interested in expanding their knowledge on algorithms and systems for mining graph data. This includes data scientists, database community students, researchers on graph processing and big-data in general, among others. Since this course is going to address graph algorithms, basic knowledge on graphs and basic programming experience is required.

3.2. Background on Graph Pattern Mining

We adopt in this course an input graph model with vertices[1] and non-directed edges. Vertices and edges also may contain a label, which captures application-specific semantics

[1] In this work, the terms "vertex" and "node" are used interchangeably

in real data. Formally, 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 f_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. The labels of a vertex or an edge are defined according the function $f_L : V(G) \cup E(G) \to L(G)$. Figure 3.1 shows an example of input graph considered, in this case the graph is undirected and unlabeled, although colors could be used to represent vertex/edge labels.

The most fundamental routine in GPM algorithms concerns the processing of subgraphs extracted from a larger input graph G. A **subgraph** (a.k.a. general subgraphs) 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 $E(S)$. If not otherwise specified, when we mention the word "subgraph" in this course we actually mean *connected subgraph*. Hence, a *subgraph* refers to a subgraph *instance* in an input graph G. Some application-specific semantics may require a more strict definition of a subgraph, namely a *connected and induced subgraph*. A subgraph S is **induced** in case it can be obtained from the input graph G from a set of vertices, and consequently its edges comprises all existing edges from G among those vertices. Note that a result of these definitions is that general not induced subgraphs are more fine-grained than induced subgraphs, since many connected not induced subgraphs may be extracted from an induced subgraph. Consider, for instance, a complete induced subgraph with 4 vertices (a.k.a. a clique) – if any of its edges is removed, we would have produced a different still connected subgraph and many of these exist given the same 4 vertices.

Subgraphs can be mapped to a naive representation of its structural and labeling information, referred simply as **pattern**. Note that patterns represent multiple subgraphs, thus they discard the identification of individual vertices and edges from the input graph. Figure 3.2 provides an example of these concepts. Colors represent vertex labels and numbers represent vertex unique identifiers.

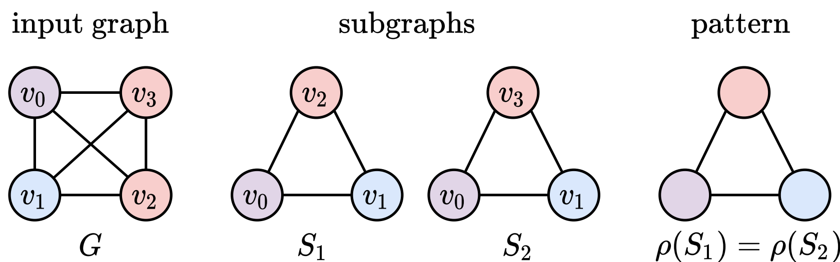


Figure 3.2. Example of input graph, subgraphs, and pattern [dos Santos Dias 2023]. Vertex colors denote labels. A single edge label is illustrated in this Figure: solid black lines.

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** is the problem of verifying whether two (sub)graphs have an identical structure (topology), being fundamental to a variety of GPM applications in graph data mining and machine learning. Although the

definition of pattern is useful to visit subgraphs with specific properties, there are different ways to represent the same pattern. For example, the triangle pattern in Figure 3.2 can be represented by a list of its edges, $[(0,1),(1,2),(2,0)]$, considering indexes starting from 0. However, an alternative representation to the same pattern can be $[(0,1),(0,2),(1,2)]$. 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 define **canonical pattern** to be a common and unique representation for each pattern. Canonical labeling algorithms exists with the purpose of mapping a subgraph to its canonical pattern, so they can be compared by equality operators (string comparison, for example) [Kuramochi and Karypis 2005, Huan et al. 2003, Yan and Han 2002, Borgelt 2007, Junttila and Kaski 2007]. It is out of the scope of this work to dive into these algorithms, but it is important to notice its importance within GPM systems, as it allows handling isomorphism issues that may arise from pattern and or subgraph extraction.

The visitation of subgraphs from a graph is also related to the concept of isomorphism. Specifically, subgraphs are built by the visitation of connected vertices and edges from G in a specific order, incrementally. Therefore, any permutation of vertices and edges represents a visitation ordered code for the *same* subgraph instance in G (Figure 3.3). We refer to these codes as subgraph codes.

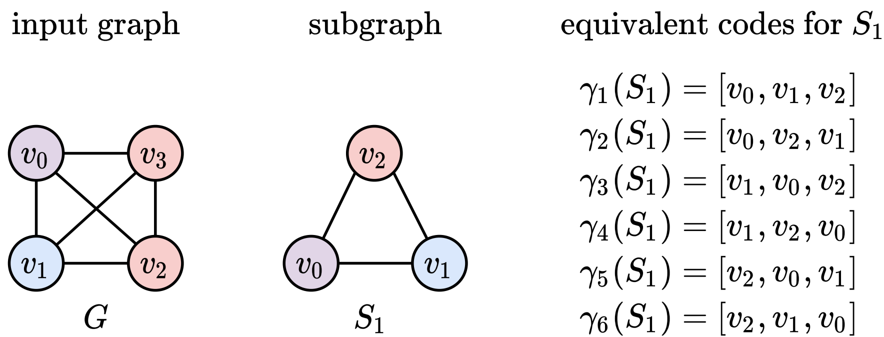


Figure 3.3. Example of input graph, subgraph, and equivalent codes [dos Santos Dias 2023]. In the provided example it is sufficient to specify vertex ordering, as each vertex in the ordering induces new edges connecting it to previous vertices in the ordering.

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. To prevent redundancy in computation and information, GPM systems usually strict themselves to visiting only a single **canonical representative code**[2] for each subgraph to prevent redundant and unnecessary work. Again, the details about enumeration algorithms is out of the scope of this course, more details can be found in [Dias et al. 2019]. Instead, in this text we are going to highlight the two main subgraph exploration paradigms used in subgraph enumeration and adopted by most GPM systems. In a **pattern-oblivious** subgraph enumeration, subgraphs (induced or not) of no particular pattern are extracted from the

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

input graph and hence, the subgraph codes produced can be of two types: a sequence of vertices in case of induced subgraphs or a sequence of edges in case of general subgraphs. Figures 3.4b and 3.4c illustrate two enumeration trees from the input graph of Figure 3.4a. The first represents the enumeration of *induced subgraphs* with 3 vertices and hence, each root-to-leaf path is a subgraph code used to represent a particular subgraph in the input graph. The second tree represents the enumeration of *general subgraphs* and hence, the subgraph codes depicted in the figure are sequences of edges instead of vertices. Nevertheless, both enumeration trees are pattern-oblivious since the subgraphs obtained are of possibly different patterns. In a **pattern-aware** subgraph enumeration, only subgraphs of a particular given pattern are extracted. This process is also known as *pattern/subgraph matching*. Figure 3.4d shows an enumeration tree conditioned by a 4-clique pattern, i.e., each path represent an enumerated subgraph that in the input graph has the structure of a complete subgraph with 4 vertices.

3.2.1. GPM systems: efficient abstractions for GPM

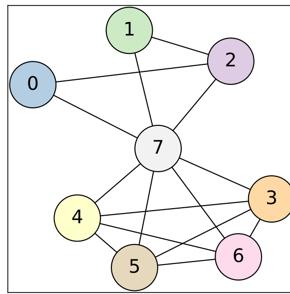
The main goal of a GPM system is to provide abstractions that properly navigate the trade-off between performance and programming experience concerning the modeling and development of subgraph enumeration and user-defined processing of visited subgraphs. The performance in this context is central because often GPM tasks include combinatorial algorithms that require efficient system optimizations to scale in parallel and/or distributed architectures. The programming experience is also important since they guarantee the applicability of the system in real-world data analysis pipelines, such as machine learning and data mining. Thus, important features of GPM systems include:

1. High-level programming abstractions to cope with the complexity of GPM theory (e.g. isomorphism, enumeration algorithms, canonical definitions and so on);
2. General-purpose programming design to enable handling user-defined application semantics;
3. System optimization strategies capable of scaling general-purpose applications in parallel/distributed architectures.

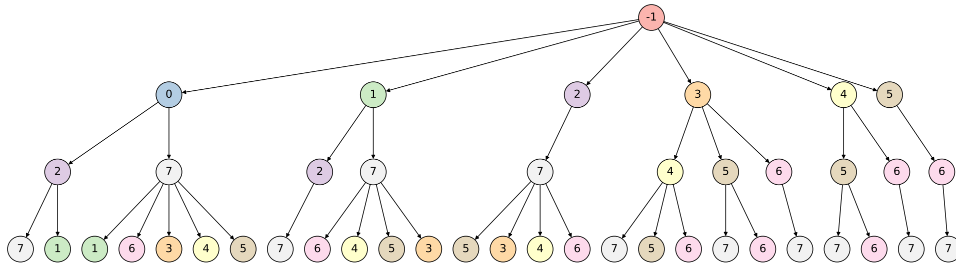
Properly providing all these wanted features simultaneously is not an easy task since systems performance and abstractions are often handled as a compromise. Next we review the literature of GPM systems and give an overview of how seminal works approach this trade-off.

3.3. GPM tools and Related work

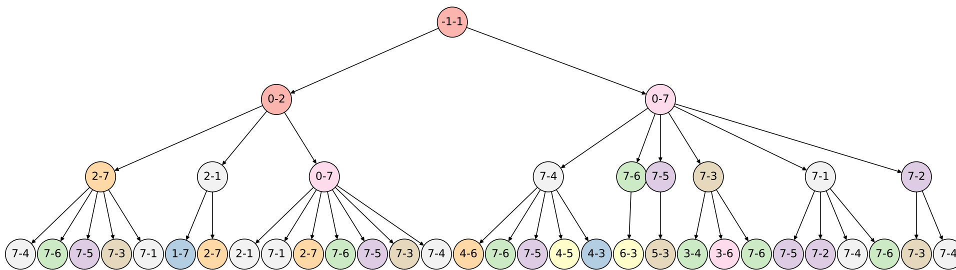
Table 3.1 briefly reviews the most seminal GPM systems in the literature. We classify the existing systems according to the important features established in Section 3.2.1. Concerning column "Productivity", a GPM system gets a HIGH in this criterion if, and only, if it can be programmed with a handful of lines and also allows easy integration with other systems; we consider a GPM system to have a FAIR productivity when it supports concise programming but fails in providing integration with other systems; finally, LOW productivity systems are both challenging to deploy and to integrate with other systems. Two



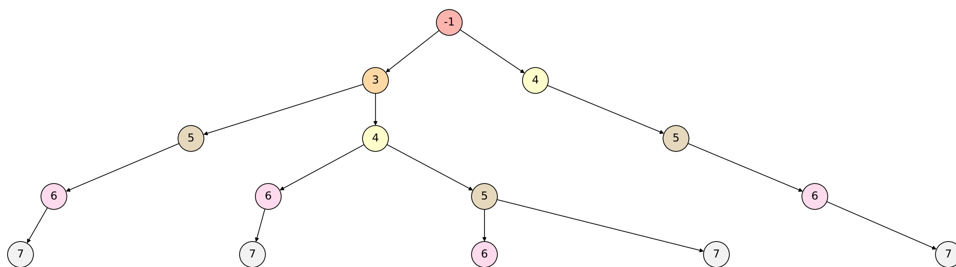
(a) Input graph example: vertex colors represent individual vertices.



(b) Induced subgraphs with 3 vertices via pattern-oblivious exploration.



(c) Subgraphs with 3 edges via pattern-oblivious exploration (entire tree has 182 leaves, and each node represents an edge).



(d) Pattern-aware exploration given a 4-clique pattern.

Figure 3.4. Enumeration trees given different target subgraph types. Each node in this tree represent one item in the subgraph code (vertex or edge), each edge in this tree indicates the order in which the subgraph code is extracted, hence each *root*→*leaf* path represent one unique subgraph from the graph of Figure 3.4a.

systems, Fractal [Dias et al. 2019] (CPU) and DuMato [Ferraz et al. 2024] (GPU), are selected to give in this chapter a practical overview of applications and system challenges.

Next we detail the related work of GPM systems.

Table 3.1. Selected related works on GPM systems: Fractal and DuMato are used as tools in this work to for practical examples.

	Proc. Unit	Parallel	Distrib.	Productivity
Fractal [Dias et al. 2019]	CPU	YES	YES	HIGH
DuMato [Ferraz et al. 2024]	GPU	YES	NO	LOW
Peregrine [Jamshidi et al. 2020]	CPU	YES	NO	FAIR
Pangolin [Chen and Arvind 2022]	GPU	YES	NO	LOW
G2Miner [Chen and Arvind 2022]	GPU	YES	NO	LOW
Tesseract [Bindschaedler et al. 2021]	CPU	YES	YES	FAIR
Arabesque [Teixeira et al. 2015]	CPU	YES	YES	FAIR
RStream [Wang et al. 2018]	CPU	YES	NO	FAIR
AutoMine [Mawhirter and Wu 2019]	CPU	YES	NO	FAIR
GraphZero [Mawhirter et al. 2021]	CPU	YES	NO	FAIR

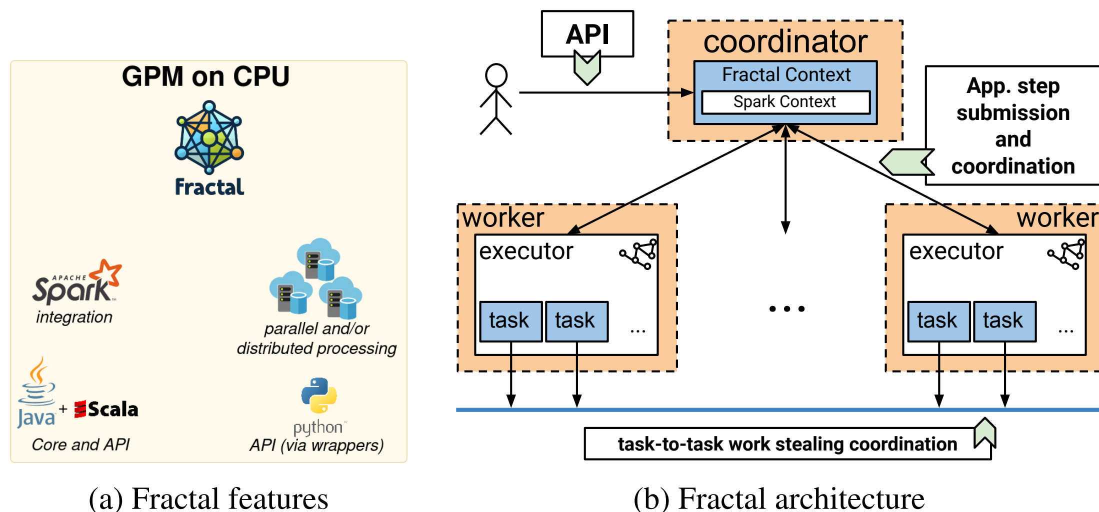
GPM systems for CPU. *Arabesque* [Teixeira et al. 2015] is one of the first GPM systems targeting distributed memory machines. The enumeration engine of Arabesque is known as pattern-oblivious, as the subgraphs are visited with no pattern information, i.e., vertex- and edge-induced only. Arabesque also proposes a data structure to compress subgraphs in-memory and to mitigate the memory demands of the BFS-style exploration while it also employs load balancing. *RStream* [Wang et al. 2018] is a relational GPM system that relies on expensive join operations to perform subgraph enumeration. It presents limitations caused by high memory consumption as the length of enumerated subgraphs increases. *Fractal* [Dias et al. 2019] is a distributed memory CPU-based GPM system that uses a DFS exploration strategy to reduce memory demands. Fractal proposes and implements a hierarchical work-stealing mechanism to mitigate load imbalance. *AutoMine* [Mawhirter and Wu 2019] proposes an automated code generation for GPM algorithms on CPU. It employs efficient scheduling of intersect/subtract operations to automate code generation for custom patterns. *Peregrine* [Jamshidi et al. 2020] is a parallel GPM system designed for shared-memory CPU machines. GraphZero [Mawhirter et al. 2021] is a compilation-based GPM system which improves AutoMine's schedule generation and symmetry breaking. Both Peregrine, AutoMine and GraphZero use an exploration strategy known as pattern-aware, where canonical pattern representatives are used to guide the subgraph enumeration by leveraging specialized execution plans (i.e. pattern-induced subgraphs are produced). Although pattern-aware exploration is efficient for enumerating small subgraphs, it has limitations whenever the application searches for a large number of canonical representatives (e.g., counting large motifs). In general, CPU based GPM systems offer a limited scaling capability since the number of execution units is reduced but, on the other hand, can be more easily integrated into data analysis pipeline due to simplified memory management and programming interfaces.

GPM systems for GPU. *Pangolin* [Chen et al. 2020] is a GPM system designed for GPU and follows the pattern-oblivious enumeration using the BFS exploration strategy. Pangolin's design enables execution optimizations by pruning the search-space of subgraphs and by reducing the amount of isomorphism tests required. Materialized intermediate states generated by the BFS exploration facilitate the runtime to leverage BSP (*Bulk Synchronous Parallel*) load balancing schemes. However, the BFS high memory demand limits its applicability to enumerate small subgraphs. Besides, Pangolin does not leverage optimizations to handle irregularity of parallel GPM algorithms on GPU and relies on CPU frameworks to perform isomorphism tests. *G2Miner* [Chen and Arvind 2022] improves Pangolin by supporting multi-GPU executions for speedup and a DFS pattern-aware subgraph enumeration engine. *DuMato* [Ferraz et al. 2024] is a pattern-oblivious GPM system with a warp-centric execution model and an automated work redistribution scheme, allowing improved memory access on GPU and more balanced executions. In general, GPU systems have a greater potential for scaling GPM tasks due to its massive number of processing units. However, this potential is challenging to fully leverage since real-world data is often skewed, which implies in very poor resource utilization if not employed together with coalesced memory access and load balancing mechanisms.

3.4. Fractal system: GPM processing on Big-Data stack

Fractal [Dias et al. 2019] is a high-performance and high-productivity system implemented over Spark [Zaharia et al. 2012] that provides a compositional API that facilitates the design of GPM algorithms while ensuring competitive parallel performance. The system (both design and implementation) is publicly available as an open-source software. Fractal core implements the main features, such as subgraph enumeration and aggregation engine and Spark integration (Figure 3.5a). Programming in Fractal may be with Scala API (core implementation) or with Python wrappers, which expose some of the main features via Py4J[3] interfaces. Figure 3.5b shows the architecture of the system. The user interacts with a local coordinator via an API that abstracts GPM processing and facilitates the application design. This program is automatically parallelized and distributed among worker nodes, each responsible for some portion of the original work. Each task in this architecture represents such unit of work that can be done in parallel. Besides this transparent parallelization provided by Spark, Fractal also extend the existing design to enable task-to-task communication for load balancing. We highlight that this load balancing protocol is central to the scalability of the system, as GPM workload is often very skewed (scale-free networks). Therefore, two points summarize the main contributions of Fractal: (1) programming abstractions that improve user experience with complex GPM algorithms and that allows them to be easily integrated into existing data analysis pipelines; and (2) extension of Spark's execution model to accommodate dynamic load balancing via work-stealing during parallel subgraph enumeration and aggregation routines. Although we do not further discuss runtime performance matters within Fractal, in Section 3.5 we address some of the main challenges in implementing efficient GPM systems.

[3] `https://www.py4j.org/`



(a) Fractal features (b) Fractal architecture

Figure 3.5. Fractal: a high productivity GPM system for parallel/distributed GPM.

3.4.1. Fractal installation and execution environment initialization

Fractal is essentially an extension over Spark Core JVM (*Java Virtual Machine*) programming interface and thus, Fractal programs are executed as Spark applications. This means that, provided that we have Fractal's package (`.jar` in this case) and dependencies, a Fractal application can be submitted via any of the following methods: (1) via Spark's `spark-submit` tool for batch executions; or (2) via some interactive engine such as Spark's `spark-shell` and other programming notebooks. Due to the practical approach of this material, we are going to address only how to compile the project and use the interactive alternatives for interfacing with Fractal applications. Nevertheless we point the curious reader to the official page of the project[4], which contains detailed information for deploying Fractal for batched executions as well.

The most straightforward approach to setup an interactive Fractal application is through the use of Python notebooks, for instance, Google Colab[5] or Jupyter[6]. The following Python notebook command is the only requirement to setup a local running environment for Fractal applications. This set of commands are expected to (1) download Fractal and Spark dependencies, and (2) compile and install any dependencies, including Spark pre-built library.

```
!git clone -b sbbd2024_course https://github.com/dccspeed/fractal.git
!cd fractal/python/ && make && make install
!pip install -q networkx tabulate matplotlib pygraphviz
```

To confirm that the installation were successful, the following python libraries should be available for importing:

```
from pyspark.sql import SparkSession  # Spark entrypoint for apps
import pyfractal                      # Fractal python wrapper
```

[4]Fractal project page: https://github.com/dccspeed/fractal
[5]Google Colab Cloud-hosted Notebooks: https://colab.research.google.com/
[6]Jupyter notebooks: https://jupyter.org/

```
import networkx as nx                  # Graph visualization
import torch                           # Tensor API and PyG models
```

A Spark session is the entrypoint in a Spark application that abstract all the underlying parallel and distributed computation. Fractal wrapper provides a default configuration builder for Spark, including important configurations such as dependencies and default parameters. At this point that the underlying execution environment of Spark can be configured to make use of parallelism and/or a cluster of machines. Details on how to create a Spark session on a distributed environment are available on the official Spark documentation[7], and are out of the scope of this practical review. From a Spark session, a Fractal context can be created – as we may see, a Fractal context is the basis for designing and executing GPM applications.

```
builder = pyfractal.DefaultSparkBuilder
builder = builder.master("local[8]")                   # 8 local threads
builder = builder.config("spark.driver.memory", "2g")  # 2GB memory
builder = builder.appName("FractalQuickstartApp")
spark = spark_builder.getOrCreate()                    # spark session
fc = FractalContext(spark)                             # fractal context
```

3.4.2. Preparing the graph data

A Fractal context object allows the creation of graph objects that can be used as input to GPM tasks. These graph objects, as we may see, are built from a directory containing the graph data. The format supported for graph data is illustrated in Figure 3.6 for a toy example. The graph data must include the following mandatory files: `metadata`, containing the number of vertices and edges of the graph; `adjlists`, containing the adjacency lists of the graph. Additionally, vertex and/or edge labeling information can be included via files `vlabels` and `elabels`, respectively. In this course, due to brevity, we only address unlabeled and vertex labeled graphs.

The following code snippet can be used to create a Fractal graphs from the directory path for the toy example in Figure 3.6. Note that the same graph data can be loaded as a vertex labeled graph (Figure 3.6d) or unlabeled graph (Figure 3.6c) – in the latter the system just ignores file `vlabels` and assigns a default single label to each vertex.

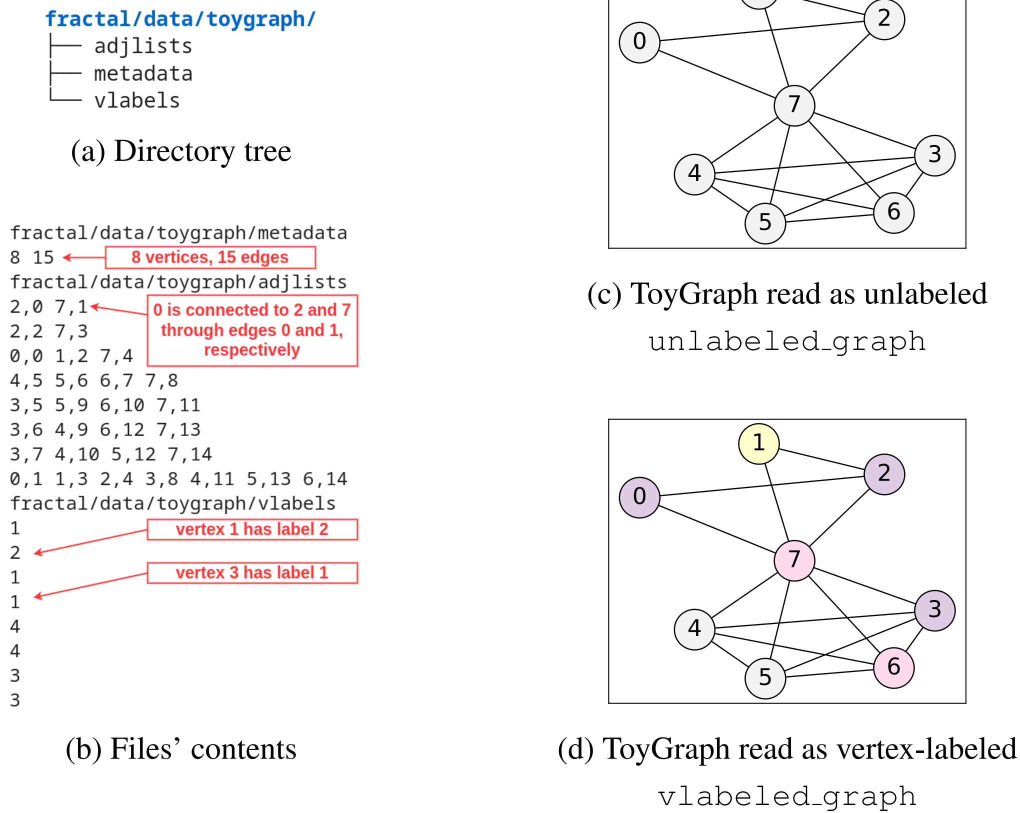
```
unlabeled_graph = fc.unlabeled_graph("fractal/data/toygraph/")
vlabeled_graph = fc.vertex_labeled_graph("fractal/data/toygraph/")
```

The Fractal graph object is the entry point for building GPM tasks. We highlight also that in case the system is deployed in a distributed setting, the graph data is loaded on each worker machine independently (i.e. the graph data is replicated).

3.4.3. Fractal programming model and library

Table 3.2 summarizes the Fractal functions discussed in this course. These are the basic calls that allows the practitioner to build customized GPM applications or to access existing optimized implementations.

[7] Spark deploy: https://spark.apache.org/docs/3.5.0/cluster-overview.html



(a) Directory tree

(b) Files' contents

(c) ToyGraph read as unlabeled `unlabeled_graph`

(d) ToyGraph read as vertex-labeled `vlabeled_graph`

Figure 3.6. ToyGraph: Fractal default graph input format. Colors in this figures represent vertex labels.

Fractal low-level API for custom applications

Fractal low-level API allows building custom GPM applications, with user-defined semantics. Next we drilldown into the main operators of Fractal API, accessed via the Python Wrapper. As we may see, these building blocks allow the enumeration of different types of subgraphs/paradigms [VI,EI,PI], the filtering of the search-space [EX,FI], and the efficient output retrieval via Spark's RDDs [SN]. Spark RDD stands for *Resilient Distributed Datasets*, and allows efficient, parallelized, distributed and fault-tolerant processing of datasets, especially useful in settings where the amount of data is massive (BigData). The subgraph enumeration settings are configured by three alternative function calls: `vfractoid()` indicates that enumeration must produce induced subgraphs via a pattern-oblivious paradigm, such as depicted in Figure 3.4b; `efractoid()` indicates that enumeration must produce not induced (general) subgraphs via a pattern-oblivious paradigm, such as depicted in Figure 3.4c; `pfractoid(pattern, induced)` indicates that enumeration must produce subgraphs of a particular `pattern`, and subgraphs are induced or not depending on the flag `induced`.

[VI,EX,FI,SN] Induced subgraphs via a pattern-oblivious exploration: Suppose a GPM routine whose goal is to extract induced subgraphs from the input graph that contain

Table 3.2. Fractal Python Wrapper API and built-in library. Only the functions addressed in this course are listed in this table, please refer to the official documentation for a complete view.

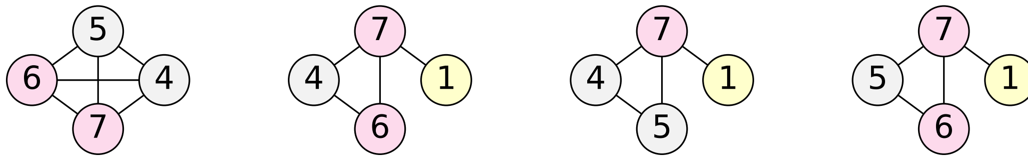
Low-level API (Purpose: to build custom applications)
[VI]:`vfractoid()` [EI]:`efractoid()` [PI]:`pfractoid(pattern, induced)`
[EX]:`extend(k)` [FI]:`filter(subgraph_filter)` [SN]:`subgraphs_networkx()`

Built-in Library (Purpose: to leverage optimized built-in applications)
[MC]:`motif_counting(k)` [PQ]:`pattern_querying(pattern, induced)`
[CC]:`cliques(k)` [QC]:`quasi_cliques(k, min_density)`
[KH]:`khop_induced_subgraphs(k)` [GV]:`graphlet_degree_vectors(k)`
[FSM]:`frequent_subgraph_mining(k, min_support)`

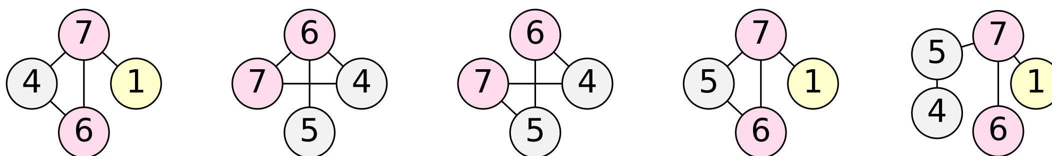
a certain set of vertex labels. The following example depicts an application that extracts induced subgraphs with 4 vertices, but only those in which each vertex have some label of interest. In this example, this predicate condition is enforced through the user-defined function `filter_func`, which returns True iff each vertex has label in the set $\{2,4,3\}$. Note also that the application is build sequentially from the input graph: first the intention to mine vertex induced subgraphs, followed by the configuration or size it is wanted for the subgraphs, and finally a label filtering function is applied to the set of subgraphs. The output is returned as a Spark RDD of NetworkX graphs, so they can be used for downstream processing tasks.

```
labels_of_interest = set([2,4,3])

# check whether a subgraph contains only labels of interest
def filter_func(s):
    for vlabel in nx.get_node_attributes(s, 'label').values():
        if vlabel not in labels_of_interest: return False
    return True

subgraphs_app = vlabeled_graph.vfractoid()          # induced subgraphs
subgraphs_app = subgraphs_app.extend(4)             # add 4 vertices
subgraphs_app = subgraphs_app.filter(filter_func) # user-def filter
subgraphs = subgraphs_app.subgraphs_networkx()      # output: nx graphs
```

Next we illustrate a few outputs of this code. Notice how every induced subgraph has 4 vertices and none contains vertex with labels in $\{1\}$ (represented as purple in Figure 3.6).

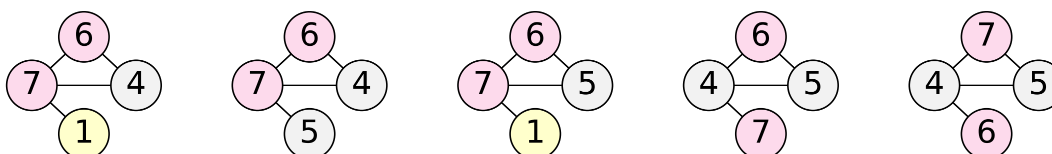


[EI] Not induced subgraphs via pattern-oblivious exploration: Also, the following output would be the result if `efractoid()` were used instead of `vfractoid()`. In this case, only 4 edges are allowed, which produces general subgraphs with at least 4 vertices and at most 5 vertices.



[PI] Subgraphs via pattern-aware paradigm: Fractal API also allows exploring the search space of subgraph of a particular predetermined pattern of interest, in this case `pfractoid()` must be set before adding vertices. The following code snippet generates only subgraphs whose pattern is a not induced "tailed triangle".

```
tailed_triangle = nx.from_edgelist([(0,1),(0,2),(0,3),(1,2)])
subgraphs_app = vlabeled_graph.pfractoid(tailed_triangle, induced=False)
# ... the rest is exactly the same as the first example above
```



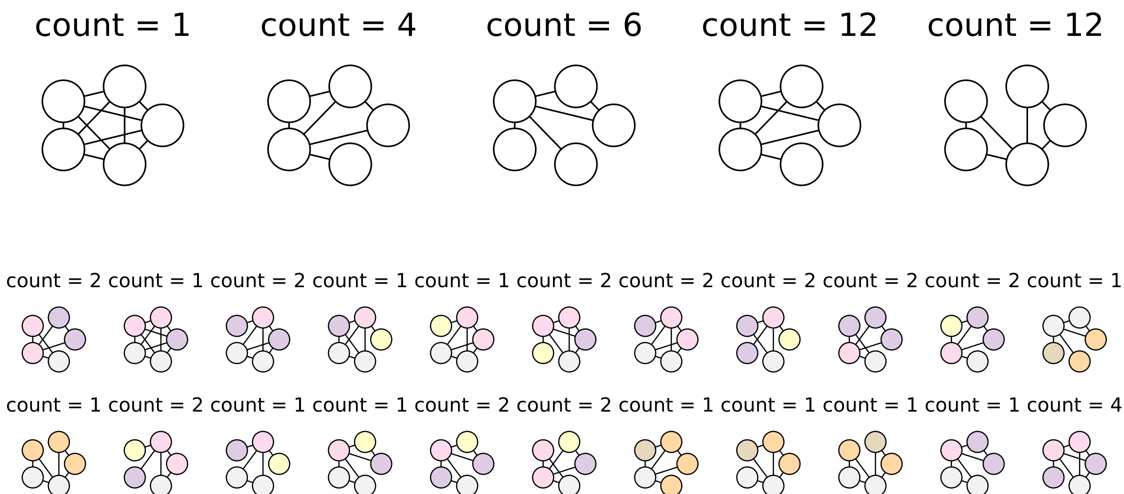
Fractal built-in library

For the traditional GPM tasks, Fractal offers off the shelf standard implementations. In this section we cover some of these applications, which can be accessed directly from a Fractal graph object.

[MC] Motif Counting (parameters: integer k): Given a number of vertices k and an input graph G, the goal of Motif counting is to output how many *unique* induced subgraphs of each different pattern exist in G. Motif counting is often used to characterize

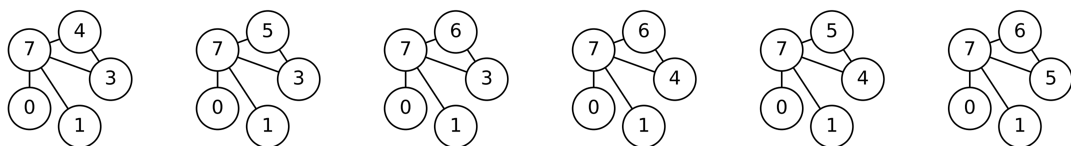
complex networks [Ribeiro et al. 2021] and the its computation allows the extraction of *Graph Degree Vectors* (a.k.a. GDV vectors) to be used as node features [Pržulj et al. 2004] in graph learning tasks. Next we show the code and output of counting motifs with 5 vertices in both the unalabeled and vertex-labeled version of the input graph (Fig. 3.6).

```
motif_count_unlabeled = unlabeled_graph.motif_counting(5)
motif_count_vlabeled = vlabeled_graph.motif_counting(5)
```



[PQ] Pattern Querying (parameters: pattern p): Pattern querying (a.k.a. pattern matching or subgraph matching) receives as input a pattern p and must list all the unique subgraphs (induced or not, depending on the goal) of such pattern in G. This computation is somewhat related to Neoj's[8] `MATCH` clause, however subgraph uniqueness during enumeration is not explicitly enforced in these transactional database systems. GPM systems, on the other hand, are more specialized and prepared to deal with such isomorphism-related issues. In the following example we see examples of subgraphs of a specific pattern (triangle with two tails). Notice also that induced flag is set to true, which indicates that induced subgraphs are to be considered (i.e. exactly the same 6 subgraphs counted in the motif counting example above). In case of flag induced set to false, the number of subgraphs of this particular pattern would be much larger, as this pattern is contained in other patterns, for example, a clique with 5 vertices.

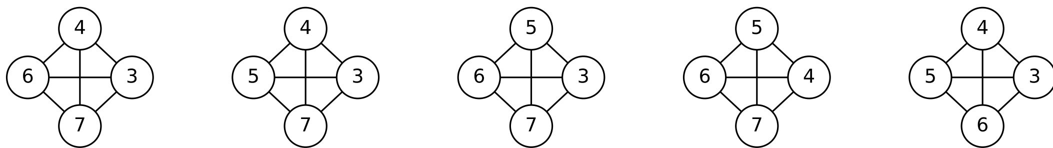
```
pattern_5 = nx.from_edgelist([(0,1),(0,2),(0,3),(0,4),(1,2)])
subgraphs = unlabeled_graph.pattern_querying(pattern_5, induced=True)
```



[8] https://neo4j.com/

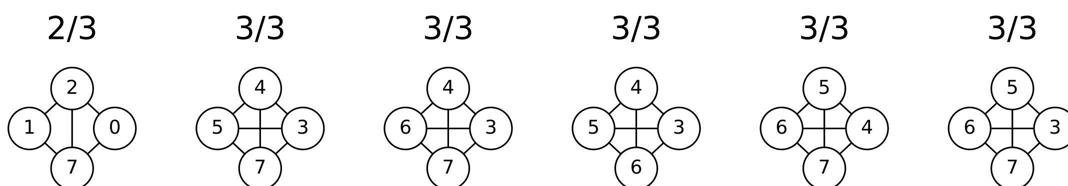
[CC] Clique Listing (parameters: integer k): Clique listing is a problem whose goal is to count/list all cliques with k vertices in the graph G. A k-clique is a complete subgraph of G with exactly k vertices. If the density condition is relaxed, more diverse output can be generated through *quasi clique listing*. Next a small example of clique enumeration, code followed by output.

```
cliques = unlabeled_graph.cliques(4)
```



[QC] Quasi-clique Listing (parameters: integer k, density threshold α): In Quasi clique listing the goal is to find in G induced subgraphs satisfying a minimum density measure threshold $0 < \alpha < 1$. Many density measures exist and for the purpose of this work, a subgraph S is considered dense iff for each one of its vertices u, its degree in the subgraph is at least $\lceil \alpha * (k-1) \rceil$. Dense subgraphs are used to extract communities [Dourisboure et al. 2009], to identify surprising groups in complex networks [Hooi et al. 2020], to improve graph compression [Buehrer and Chellapilla 2008], among others. Next we show the code and the output (with densities) for a quasi clique finding call requesting subgraphs with 4 vertices and minimum density of $2/3$, i.e., each subgraph vertex must be connected to at least other two in the subgraph.

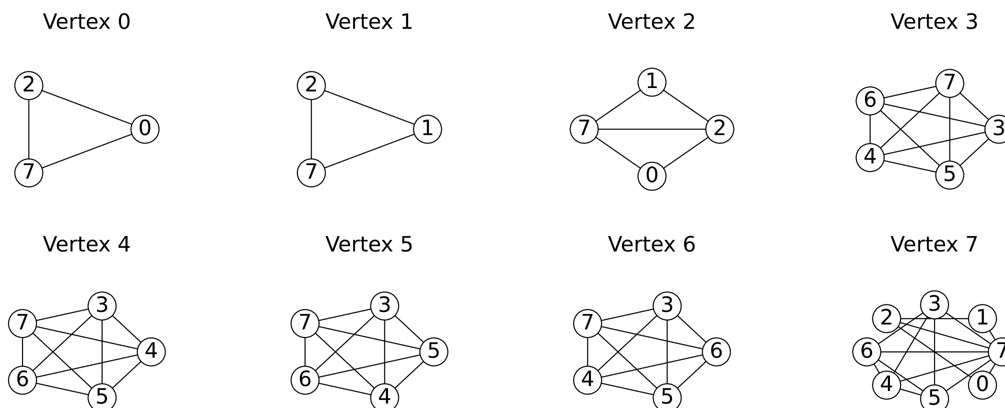
```
quasi_cliques = unlabeled_graph.quasi_cliques(4, 2/3)
```



[KH] K-hop subgraph (a.k.a. ego network) extraction (parameters: integer k): The goal of this procedure is to extract k-hop induced subgraphs around each vertex of graph G. This task is often used in machine learning pipelines as a mean to extract local node features., i.e., the surroundings of a node in a graph is expected to provide important insights on its role in the graph. Ego-Networks are widely used in graph learning tasks, for instance, k-hop can be used to map node-level tasks (e.g. node classification) into graph-level tasks (e.g. graph classification of k-hop subgraphs extracted from a graph) [Sun et al. 2023]. Also, reachability queries may benefit from such computation, limiting the search space that needs to be considered [Jin et al. 2009]. The following code and Figure shows the 2-hop induced subgraphs of the graph in Figure 3.6. Notice how ego networks

have an interesting capability of distinguishing similar node neighborhoods, fact that we explore in more details for a real-world graph in Section 3.4.5.

```
khop_subgraphs = unlabeled_graph.khop_induced_subgraphs(2)
```



[GV] Graphlet-Degree Vectors (parameters: integer k): For machine learning tasks on graphs in which the structure of the input graph is relevant to the target goal, capturing such information as node-features may substantially improve the overall quality of the models. One approach for doing this is through *Graphlet-Degree Vectors* extraction. This feature extraction strategy consists of building a counting vector per vertex/node in the graph. Each vector contains the frequency counts of all possible motif/graphlet orbits up to a pattern size k. In general terms, graphlet orbits represent non-equivalent positions in a set of motifs up to size k. Figure 3.7 shows an example of the GDV of vertex u in a graph – notice how four distinct positions exists in motifs up to 3 vertices and hence, the GDV is composed of these four dimensions. This GPM task is implemented by enumerating all induced subgraphs of each possible motif and counting each subgraph position individually according to the unique positions determined as graphlet orbits. The output is one vector per vertex in the graph. Below we illustrate the API call and result for the unlabeled ToyGraph. In this case, graphlet orbits are extracted from motifs of up to 4 vertices, resulting a 15-dimension feature vector for each one of the 8 vertices in the graph.

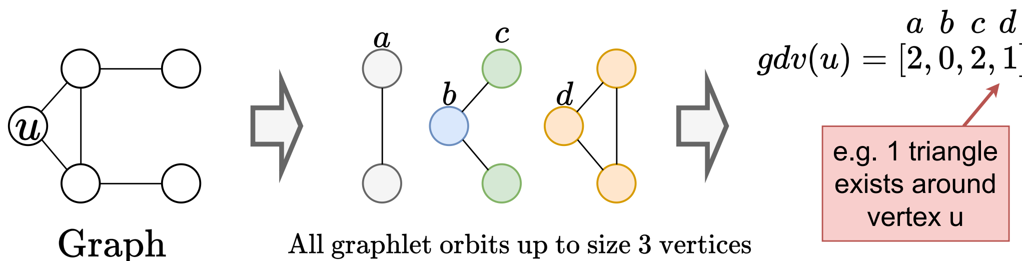


Figure 3.7. Graphlet-Degree Vector example: extracting node-features via GPM tasks.

```
gdvs = unlabeled_graph.graphlet_degree_vectors(4)
```

```
tensor([[ 2.,  8.,  1.,  1., 20.,  4., 16.,  0.,  1.,  6.,  0.,  3.,  1.,  0.,  0.],
        [ 2.,  6.,  1.,  1., 18.,  4., 10.,  0.,  1.,  6.,  0.,  3.,  1.,  0.,  0.],
        [ 1.,  6.,  1.,  1., 20.,  5.,  6.,  1.,  1.,  5.,  1.,  3.,  0.,  1.,  0.],
        [ 4., 12.,  1.,  6., 24.,  4., 12.,  1., 12., 15.,  2.,  2.,  6.,  2.,  4.],
        [ 3.,  9.,  1.,  3.,  4.,  4.,  6.,  1.,  3.,  2.,  2.,  2., 11.,  1.,  1.],
        [ 2.,  6.,  1.,  1., 12.,  4.,  2.,  1.,  3.,  2.,  2.,  2.,  5.,  1.,  1.],
        [ 1.,  3.,  1.,  1.,  6.,  4.,  2.,  1.,  3.,  6.,  6.,  4.,  2.,  2.,  1.],
        [ 1.,  3.,  1.,  1.,  6.,  4.,  2.,  1.,  3.,  6., 15.,  1.,  2.,  1.,  1.]])
```

GDV of vertex 0

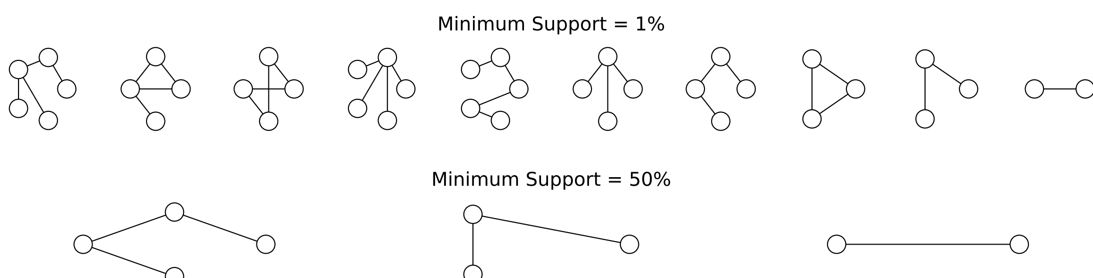
GDV of vertex 3

...

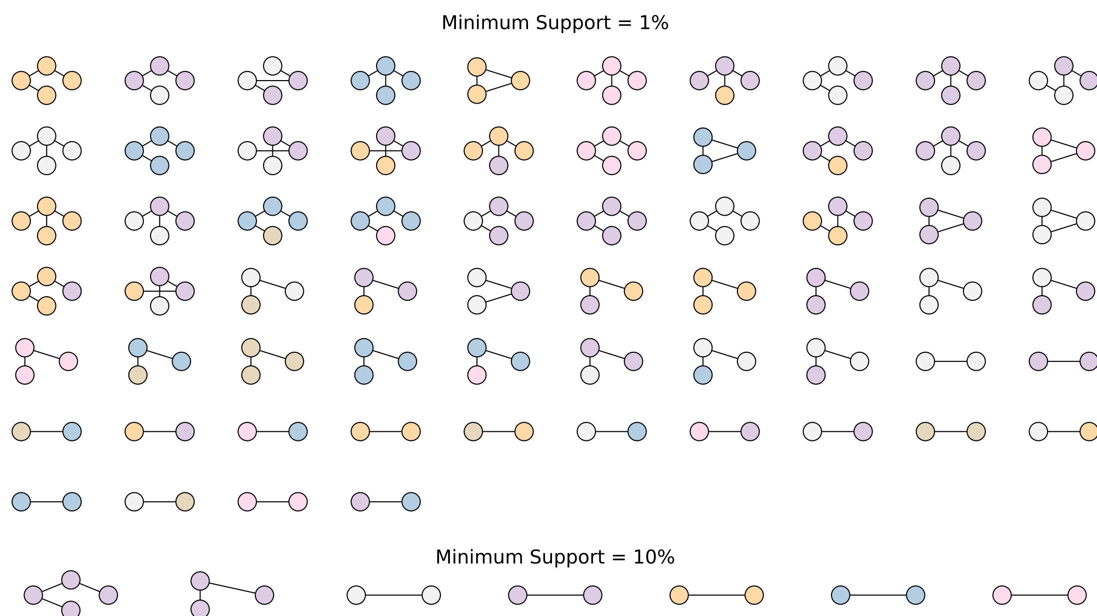
[FSM] Frequent Subgraph Mining (parameters: integer k, frequency threshold α): Frequent Subgraph Mining (FSM) leverages some frequency measure to extract from G those patterns with k edges meeting a minimum frequency threshold α. This frequency measure usually is defined to enforce the anti-monotonic property, i.e., every sub-pattern of a frequent pattern must also be frequent. Thus, if the input is a set of graphs, the frequency of a pattern can be simply the proportion of graphs containing the pattern [Yan and Han 2002]. For the single large graph setting, however, such counting frequency is not anti-monotonic. For this reason, for a single large graph, the frequency of a pattern is determined by other measures such as the *minimum image support* [Bringmann and Nijssen 2008]. Next we show an example of the usage of FSM algorithm over the real-world graph Citeseer [Elseidy et al. 2014], composed of Computer Science articles (total of 3312 nodes) and citations among them (total of 4732 edges). Vertex labels indicate the related area of the article (total of 6). In this case, the frequency is represented in terms of a percentage over the number of vertices in the graph. Thus, 1% minimum support indicates that only patterns whose minimum image support of at least "1% of the number of nodes in the graph" are considered frequent.

```python
citeseer = fc.unlabeled_graph("fractal/data/citeseer")
frequent_patterns_by_support = dict()
for min_support in [0.01,0.5]:
    frequent_patterns = citeseer.frequent_subgraph_mining(4, min_support)
    frequent_patterns_by_support[min_support] = frequent_patterns
```

Below we see the result of this code for different support thresholds – larger minimum supports are more selective to the space of patterns. Also in this illustration we show the results for the unlabeled version of Citeseer graph.



If the graph is loaded as vertex-labeled, more candidate patterns are possible and consequently, it becomes more difficult for a pattern to be considered frequent. In the output below we show the frequent patterns for two other minimum support thresholds – in this case, a minimum support of 50% returns no patterns (not shown).



3.4.4. Use Case: Extracting node-level features for graph learning tasks

GPM can be used in the context of graph learning, enabling the extraction of meaningful structural features for machine learning. Node-level graph learning tasks is an example that can benefit from GPM computation. In a graph-level classification task, a database of graphs is given as input. Each graph has an associated class and the goal is to train a model that is competent in guessing the class of graphs given their structural information.

In MUTAG dataset, derived from a study of chemical compounds [Kriege and Mutzel 2012], each compound is a graph, and vertices represent atoms and edges indicate chemical bonds between a pair of atoms. Overall, MUTAG dataset consists of 188 chemical compounds divided into two classes according to their mutagenic effect on a bacterium. In this use case we are going to consider the following graph-level learning task over this dataset: to predict the class of a chemical compound given its structure. One possible approach to accomplishing this is to generated Graphlet Degree Vector (GDV) features for the vertices in the dataset and forward this modified dataset to a classification model of choice.

PyTorch Geometric (PyG)[9] is a graph learning framework focused in deep models on graphs (e.g. Graph Neural Networks and Shallow Embedding). PyG also facilitates the access to public datasets such as MUTAG and hence, in this use case we show how to generate GDV feature vectors for the MUTAG dataset and use this modified dataset in a PyG classification model for the task described above. Figure 3.8 illustrates the workflow of how to integrate Fractal output into Machine Learning Systems such as PyG.

The following code implements method `get_dataset_with_gdv` from Figure 3.8. The overall operation is to merge the whole dataset as a single Fractal graph and compute the GDV features for each vertex using Fractal built-in API [GV].The method returns a copy of the dataset with GDV as vertex features, represented in PyG as attribute `data.x`.

[9] https://pytorch-geometric.readthedocs.io/en/latest/index.html

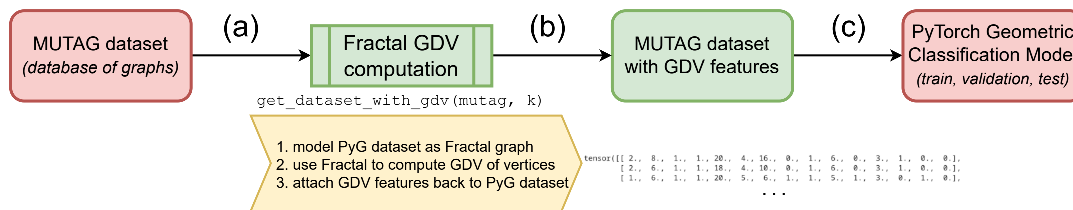


Figure 3.8. Workflow of use-case: Using Fractal to extract GPM features to enrich graph data for classification. (a) and (b) PyG MUTAG dataset is used as input for Fractal's computation of GDV features; (c) the modified dataset is forwarded to a machine learning pipeline of training/validation/testing.

```
def get_dataset_with_gdv(dataset, k):
    # compute graphlet degree vectors
    loader = DataLoader(dataset, batch_size=len(dataset), shuffle=False)
    full_batched_data = next(iter(loader))
    fg = fc.unlabeled_graph_from_pyg_data(full_batched_data)
    full_batched_data_x = fg.graphlet_degree_vectors(k)

    # apply GDV features to a copy of the dataset
    dataset_with_gdv_features = []
    for i in range(len(dataset)):
        data = dataset[i].clone()
        from_idx = full_batched_data.ptr[i]
        to_idx = full_batched_data.ptr[i + 1]
        x = full_batched_data_x[from_idx:to_idx]
        data.x = x
        dataset_with_gdv_features.append(data)
    return dataset_with_gdv_features
```

We omit the whole code since details on machine learning pipelines on graphs is out of the scope of this course and because our focus is on showing how GPM algorithms may be useful as preprocessing steps in such scenarios. Nevertheless, below we show the output of this workflow for a PyG Classification Model composed of three Graph Convolution Network [Kipf and Welling 2017] layers followed by a pooling mechanism for graph classification. The output is obtained from random dataset splits. While these results depend on specific hyperparameter configurations and may vary, stills holds that GPM features can be useful for graph learning tasks and hence, this integration is beneficial. Another important observation is that GDV features include *only structural information* concerning the surroundings of the vertices, which alone is enough to provide high accuracy results.

```
== Model with default features (atom attributes) ==
[01] Train Loss: 0.50 Validation Accuracy: 0.79 Test Accuracy: 0.68
[02] Train Loss: 0.49 Validation Accuracy: 0.79 Test Accuracy: 0.74
[03] Train Loss: 0.52 Validation Accuracy: 0.74 Test Accuracy: 0.68
[04] Train Loss: 0.54 Validation Accuracy: 0.84 Test Accuracy: 0.68
[05] Train Loss: 0.56 Validation Accuracy: 0.74 Test Accuracy: 0.79
Average Test Accuracy: 0.79

== Model with Graphlet Degree Vector Features ==
```

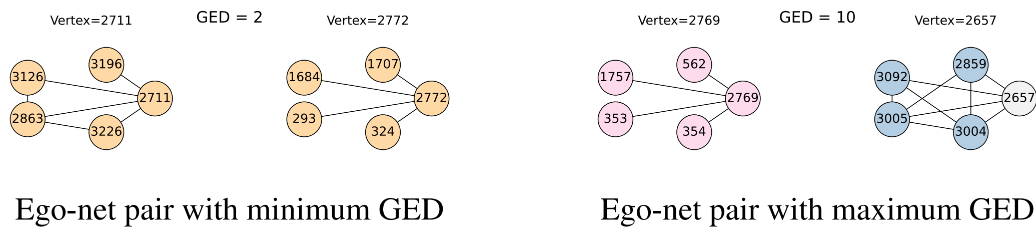
```
[01] Train Loss: 0.23 Validation Accuracy: 0.84 Test Accuracy: 0.79
[02] Train Loss: 0.28 Validation Accuracy: 0.84 Test Accuracy: 0.95
[03] Train Loss: 0.30 Validation Accuracy: 0.63 Test Accuracy: 0.79
[04] Train Loss: 0.32 Validation Accuracy: 0.74 Test Accuracy: 0.74
[05] Train Loss: 0.33 Validation Accuracy: 0.89 Test Accuracy: 0.89
Average Test Accuracy: 0.89
```

3.4.5. Use Case: Node similarity via Ego-Networks

In this use case we illustrate how to leverage k-hop subgraphs (a.k.a. ego networks) to compare nodes in a labeled graph. We consider the real-world graph Citeseer [Elseidy et al. 2014], the same graph used in the FSM example above. An alternative to compute the similarity between nodes in a networks is as follows: (1) generate k-hop networks for each node; (2) compare nodes' networks by Graph Edit Distance (GED); (3) the output measure is an estimate on how similar two nodes are with respect to structure and labeling. The GED of a pair of graphs (G_1, G_2) is defined as the number of graph operations required to transform G_1 into G_2. The graph operations include adding/deleting nodes/edges and changing node labels. Therefore, a GED distance of 0 indicates that both graphs are exactly the same, while a GED of k indicate that this amount of graph operations is required to both be equal. This measure is included in the NetworkX package and thus, we can generate ego-nets using Fractal and compute pairwise similarities. The following code accomplishes this workflow – notice how we choose $k = 2$ to be the radius of the ego networks and it is defined that two nodes are equal whenever they share the same label.

```
citeseer = fc.vertex_labeled_graph('fractal/data/citeseer')
ego_nets = citeseer.khop_induced_subgraphs(2)

def node_match(n1, n2): return n1['label'] == n2['label']

for en1 in ego_nets:
    for en2 in ego_nets:
        ged = nx.graph_edit_distance(en1, en2, node_match=node_match)
        yield en1, en2, ged
```

From the output of the code above, we may select two pairs of graphs to exemplify this similarity computation. In the left side of the Figure below we see the most similar non-equal ego networks with 5 nodes in the Citeseer graph. These networks correspond to nodes 2711 and 2772, respectively, with a GED of 2 operations. Notice how both graphs share the same unique label for each node, so the 2 operations necessary are just edge additions to the first one. In the right side of the Figure below, we see the opposite: the least similar pair of ego networks with 5 nodes. In this case, the GED is equal to 10 because 5 label transformations and 5 edge additions are necessary to make both graphs equal.



Ego-net pair with minimum GED Ego-net pair with maximum GED

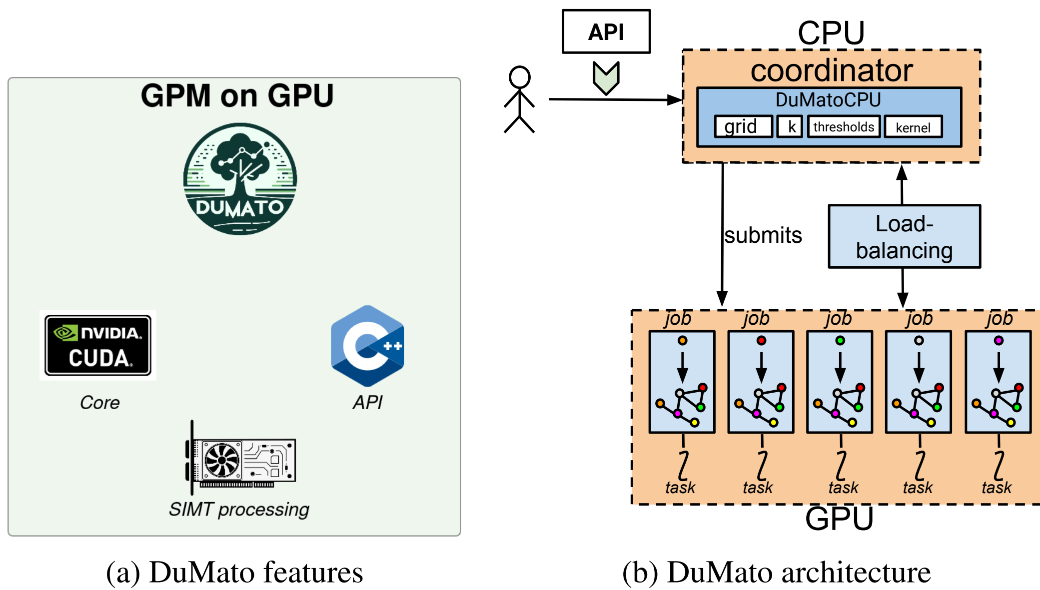
3.5. DuMato system: massively parallel GPM on GPUs

The CUDA architecture is the market-leading GPU architecture proposed by NVIDIA [Corporation 2024], and other GPU vendors such as AMD follow the same architectural concepts but using a distinct nomenclature. In this course we will adopt the CUDA nomenclature to define the standard GPU concepts.

A GPU is a massively parallel device that follows the SIMT (Single-Instruction Multiple-Thread) computing scheme. In this scheme, a parallel task is executed by the GPU by a group of 32 cores, and this execution unit is called a *warp*. Warps execute independently, but threads belonging to the same warp are supposed to execute the same instruction in lockstep to achieve maximum parallel efficiency. Besides, a warp shall follow a regular memory access pattern to reduce the amount of memory transactions needed to answer memory requests. When threads belonging to the same warp need to execute different instructions, we say there is a *divergence* and the parallel performance deteriorates. In the same sense, when threads belonging to the same warp follow irregular memory access patterns, we say there is *memory uncoalescence* and memory performance deteriorates. The modern GPUs provide distinct physical groups of cores (multiple of warp size) to execute a pool of warps, and these groups are named *Streaming Multiprocessors* (SM).

DuMato [Ferraz et al. 2022, Ferraz et al. 2024] is a graph pattern mining system implemented over the CUDA architecture, and supports efficient implementations of GPM algorithms on GPU through a high-level API and an well-defined execution workflow. It is also publicly available as a open-source software, and requires C/C++ 11 and CUDA 12. DuMato's core implements pattern-oblivious subgraph enumeration and provides high-level programming interfaces to allow an user to create custom routines to visit only specific induced subgraphs and generate the desired output of the GPM algorithm. DuMato uses a graph traversal named *DFS-wide*, which allows a predictable memory consumption and improves memory locality. One may develop a GPM algorithm with DuMato using C/C++ on host-side and CUDA C++ on device-side (Figure 3.9a).

The design and implementation of Graph Pattern Mining in any parallel computing environment must deal with two challenges: **irregularity** and **load imbalance**. The *irregularity* is related to the dynamism and unpredictability of the access pattern regarding the data structures (mainly the adjacency lists) during the visitation of subgraphs, generating overheads concerning memory uncoalescence and caching. Assuming different threads process different regions of the input graph, those threads tend to visit different amounts of subgraphs, incurring a *load imbalance* that may deteriorate parallel performance. Take Figure 3.4b for example, 7 induced subgraphs with 3 vertices starting at vertex 0 exist, whereas only 1 exists starting at vertex 5. This imbalance can only be exacerbated for large graphs, leading to poor resource utilization and runtime performance. In Section 3.5.2 we show real data to support the importance of a well-balanced GPM



(a) DuMato features (b) DuMato architecture

Figure 3.9. DuMato system for GPU-accelerated GPM.

execution and the performance benefits in incorporating such optimization into the runtime, meanwhile we provide further details on DuMato's design choices and programming overview.

3.5.1. DuMato architecture and programming overview

Figure 3.9b shows the architecture of the system. The user creates a *DuMatoCPU* object on CPU to define the grid size to be launched on GPU, the size k of the subgraphs visited, load-balancing thresholds and a function pointer to the GPU kernel for the desired GPM algorithm (implemented using DuMato API). The *DuMatoCPU* object submits GPM jobs to be executed by parallel tasks on the GPU. Given an input graph G and a subgraph S of G, a GPM job is responsible for visiting the desired subgraphs with k vertices of the GPM algorithm starting from S. In Figure 3.9b, each parallel task received one job that visits all desired subgraphs of the GPM algorithms starting from a specific vertex of the input graph. A parallel task may receive more than one job, depending on the load-balancing thresholds.

After submitting the job on GPU, the *DuMatoCPU* object starts the load-balancing layer on CPU, depicted by Figure 3.10. Each task on GPU carries a flag indicating whether its enumeration is active, and the load-balancing layer reads all GPU flags asynchronously. Once the load-balancing layer detects that the amount of idle GPU tasks is higher than a threshold, it stops the kernel execution on GPU, reads the current enumeration data, redistributes the jobs among the tasks in the grid, and resubmits the job to the GPU. The load-balancing layer also uses another parameter to indicate the amount of jobs each parallel task is supposed to receive after redistribution.

A DuMato parallel task depicted in Figure 3.9b may have two different granularities: *thread-level* and *warp-level*, named as *DM_DFS* and *DM_WC* is DuMato's specification, respectively. In the thread-level granularity, each GPM job is allocated to a single

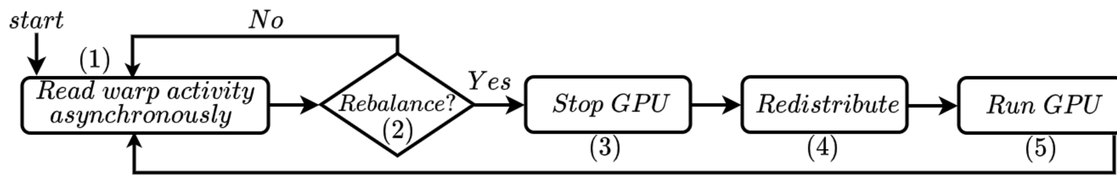


Figure 3.10. Load-balancing layer proposed by DuMato [Aquino 2023].

GPU thread. On the other hand, in the warp-level granularity each GPM job is allocated to a single GPU warp. The warp-level granularity was implemented using the warp-centric programming model [Hong et al. 2011], and is an optimization proposed by DuMato to improve GPU's parallel efficiency. This optimization allowed all threads within a warp to execute in lockstep to minimize GPU divergences (reducing the amount of individual instructions executed by GPU), as well as to facilitate memory coalescence and improve memory locality (reducing the amount of memory transactions needed to answer parallel memory requests). DuMato also provides a version named *DM_WCLB*, which extends the version *DM_WC* with the load-balancing performed by the CPU.

Table 3.3 shows DuMato's API used to build GPM algorithms on GPU. The function *control* is responsible for receiving load-balancing information from CPU, and function *move* goes forward and backward in the enumeration tree. Function *extend* creates new subgraphs from the current subgraph, and function *filter* eliminates subgraphs according to user-defined criteria defined by the function pointer parameter *P*. The aggregate functions are used to generate outputs for the algorithms, which can be a total counting (*aggregate_counter*), a counting per pattern (*aggregate_pattern*) and a buffer containing the subgraph codes of all visited subgraphs (*aggregate_store*). DuMato provides an skeleton code that can be used as the starting point to implement any GPM algorithm using the API.

Table 3.3. DuMato API. [Aquino 2023]

Functions	Scope
`[CT]` *control(TE)* `[MV]` *move(TE, genedges)*	Algorithm-independent
`[EX]` *extend(TE, begin, size)* `[FL]` *filter(TE, P, args)* `[A1]` *aggregate_counter(TE)* `[A2]` *aggregate_pattern(TE)* `[A3]` *aggregate_store(TE)*	Algorithm-specific

In the next section we present results of DuMato execution to understand the performance challenges associated with the parallel design and implementation of GPM algorithms on GPU.

3.5.2. Performance Challenges of GPM on GPU.

The experiments executed for this section were performed using Google Colab. Our goal is to demonstrate empirically the importance of load balancing in GPM systems, and in particular within GPU. Once a GPU environment is chosen, CUDA Toolkit is available and you may type the following commands to download and compile DuMato:

```
!git clone https://github.com/samuelbferraz/DuMato.git
!cd DuMato && make sm=75   # compile all built-in applications
```

All executable files will be available under the `DuMato/exec` folder after the compilation process. We executed the motif counting application provided by DuMato to evaluate the performance impacts of parallel granularity. The command lines executed are the following:

```
!./motifs_DFS ../datasets/citeseer.edgelist 3 102400 256
!./motifs_DFS ../datasets/citeseer.edgelist 4 102400 256
!./motifs_DFS ../datasets/citeseer.edgelist 5 102400 256
!./motifs_DFS ../datasets/citeseer.edgelist 6 102400 256
!./motifs_HAND_WC ../datasets/citeseer.edgelist 3 102400 256
!./motifs_HAND_WC ../datasets/citeseer.edgelist 4 102400 256
!./motifs_HAND_WC ../datasets/citeseer.edgelist 5 102400 256
!./motifs_HAND_WC ../datasets/citeseer.edgelist 6 102400 256
```

The executable `motifs_DFS` contains the *thread-level* implementation, while the executable `motifs_HAND_WC` contains the *warp-level* implementation. These two versions receive the same parameters, which are: the input graph (using the standard edge list format), the size of the visited subgraphs, the number of parallel threads and the block size. All executions were performed using the same amount of threads (102400) and the same block size (256), whose values are the default adopted in DuMato.

Table 3.4 shows the execution time of DuMato using the *thread-level* and *warp-level* granularity. For the small subgraph sizes (3 and 4), the executions are fast and the performance difference between the versions are not clear. However, starting from subgraph size 5, we clearly see that the version with the warp-level granularity provides speedups from 11x to 21x, showing that warp-level lockstep execution associated with good memory locality/coalescence are essential to keep performance as the size of visited subgraphs increases.

Table 3.4. Execution time (seconds) varying the parallel granularity.

Granularity	***Subgraph size (k)***			
	3	***4***	***5***	***6***
Thread-level	0.04	0.63	10.95	339.08
Warp-level	0.01	0.08	0.96	16.15

We also executed the motif counting application along with the load-balancing layer turned on, in order to evaluate the performance impacts of load-balancing. The command lines executed are the following:

```
!./motifs_DM_WCLB ../datasets/citeseer.edgelist 3 102400 256 8 30
!./motifs_DM_WCLB ../datasets/citeseer.edgelist 4 102400 256 8 30
!./motifs_DM_WCLB ../datasets/citeseer.edgelist 5 102400 256 8 30
!./motifs_DM_WCLB ../datasets/citeseer.edgelist 6 102400 256 8 30
```

The executable `motifs_DM_WCLB` implements the warp-level granularity along with the load-balancing layer on CPU. This executable receives two additional parameters: the amount of jobs assigned per task during the load-balancing phase (8, for all executions), and the upper-bound percentage of threads allowed to be idle without the need of calling the load-balancing layer (30%, for all executions). Table 3.5 compares the result of warp-level granularity without and with the load-balancing layer.

Table 3.5. Execution time (seconds) using the load-balancing layer.

Granularity	***Subgraph size (k)***			
	3	***4***	***5***	***6***
Warp-level	0.01	0.08	0.96	16.15
Warp-level with load-balancing	0.11	0.12	0.35	0.91

For the small subgraph sizes (up to 4), the version with load-balancing is a little worse than the standard warp-level version. This happens because the inclusion of the load-balancing layer generates an overhead that is not worth paying for short executions. The importance of load-balancing becomes clear after subgraph size 5, when we start seeing an increasing speedup. This happens because, as the size of the visited subgraphs increases, some regions of the input graph tend to concentrate most of the computation, and when the load-balancing is enabled, this data skewness is mitigated. This shows the importance of effective load-balancing on GPUs as we increase the size of visited subgraphs for any GPM algorithm, as GPUs are massive parallel environments which rely on active parallel threads throughout the entire execution to obtain valuable speedups.

Figure 3.11 shows the average SM occupancy (number of active warps per SM divided by the maximum amount of active warps per SM) through the execution of motif counting application for the Citeseer dataset, using subgraph size 8 and load-balancing threshold 30%. In other words, when the average SM occupancy is lower than 30%, the load-balancing layer is called to perform job redistribution. This behavior is depicted by the red line in the Figure, and every time we see a peak in the plot, it means the load-balancing layer was called. Although all SMs are active through the execution (green plot), the amount of active warps per SM varies. The choice of the appropriate load-balancing threshold is challenging, as the load imbalance varies depending on the dataset and the algorithm. An extensive evaluation of the load-balancing is beyond of the scope of this course, but you may check additional references that explain this trade-off deeply [Ferraz et al. 2024, Ferraz et al. 2022, Aquino 2023].

3.6. Conclusion

Graph Pattern Mining (GPM) refers to the processing of graph data that involves the extraction of subgraphs, being of especial interest to the data mining and machine learning

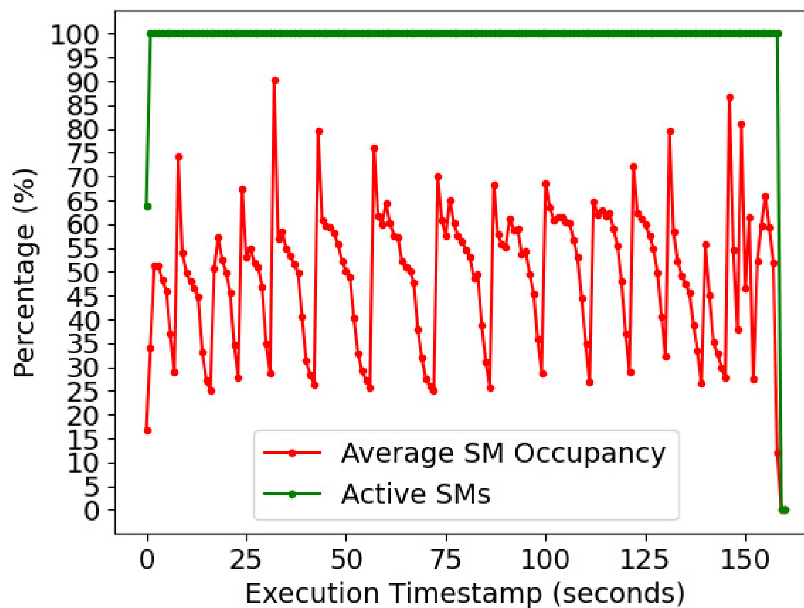


Figure 3.11. Load-balancing rounds of DuMato with warp-level parallelism and load balancing activated. In case, load balancing was turned off, the occupancy would progressively decay and hurt the overall performance.

communities. GPM systems emerged in the last decade as a solution to improve the user experience with GPM algorithms and also their runtime performance. Thus, GPM systems offer strong abstractions for programming that hide much of the complexity of algorithm design while ensuring performance scalability via system optimizations tailored for graph data processing. In general, GPM algorithms are often used as a preprocessing step to extract relevant knowledge from graph data. In this chapter, we walked through the main concepts, applications and challenges concerning GPM systems. Our practical approach leverages two recent GPM systems published as scientific work in the area: Fractal and DuMato, both open-source and public available.

Through the use of Fractal via Python Wrappers we are able to illustrate the main concepts and paradigms for enumerating and processing subgraphs. We give an overview of the programming interface which allows, with a few lines of code and inexpensive effort, to build custom user-defined applications and also to leverage existing built-in optimized application implementations. We also show use case scenarios where Fractal processing is integrated with other data analysis frameworks via Spark's resilient datasets (RDD) and, making complex data pipelines relying on GPM processing more natural and straightforward to implement and deploy. Through the use of DuMato system, we demonstrate how GPM processing can be accelerated using GPUs. Our examples offer a general picture on the main challenges in ensuring a proper parallel performance of GPM processing on GPUs: irregularity of real-world graph data and constant load imbalance. We take an experimental approach to understanding these challenges and show runtime performance that highlight the importance of the proposed optimizations for dealing with the aforementioned challenges.

Overall we believe this chapter gives a straightforward and concise overview of the topic, being useful for students, researchers, and other practitioners interested in learning interesting and efficient alternatives for extracting knowledge from graph data.

References

[Agrawal et al. 2018] Agrawal, M., Zitnik, M., and Leskovec, J. (2018). Large-scale analysis of disease pathways in the human interactome. *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing*, 23:111–122.

[Aquino 2023] Aquino, S. B. F. (2023). *Strategies for efficient subgraph enumeration on GPUs*. Phd thesis, Federal University of Minas Gerais. Available at http://hdl.handle.net/1843/62443.

[Benson et al. 2016] Benson, A. R., Gleich, D. F., and Leskovec, J. (2016). Higher-order organization of complex networks. *Science*.

[Bindschaedler et al. 2021] Bindschaedler, L., Malicevic, J., Lepers, B., Goel, A., and Zwaenepoel, W. (2021). *Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs*, page 458–473. Association for Computing Machinery, New York, NY, USA.

[Borgelt 2007] Borgelt, C. (2007). Canonical forms for frequent graph mining. In Decker, R. and Lenz, H. J., editors, *Advances in Data Analysis*, pages 337–349, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Bringmann and Nijssen 2008] Bringmann, B. and Nijssen, S. (2008). 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. Springer-Verlag.

[Buehrer and Chellapilla 2008] Buehrer, G. and Chellapilla, K. (2008). A scalable pattern mining approach to web graph compression with communities. In *Proceedings of the 2008 International Conference on Web Search and Data Mining*, WSDM '08, page 95–106, New York, NY, USA. Association for Computing Machinery.

[Chen and Arvind 2022] Chen, X. and Arvind (2022). 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. USENIX Association.

[Chen et al. 2020] Chen, X., Dathathri, R., Gill, G., and Pingali, K. (2020). Pangolin: An efficient and flexible graph mining system on cpu and gpu. *Proc. VLDB Endow.*, 13(8):1190–1205.

[Corporation 2024] Corporation, N. (2024). NVIDIA Website. https://www.nvidia.com/. [Online; accessed 5-August-2024].

[Dias et al. 2019] Dias, V., Teixeira, C. H. C., Guedes, D., Meira Jr., W., and Parthasarathy, S. (2019). Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*.

[dos Santos Dias 2023] dos Santos Dias, V. V. (2023). *Graph pattern mining: consolidating models, systems, and abstractions*. Phd thesis, Federal University of Minas Gerais. Available at http://hdl.handle.net/1843/51806.

[Dourisboure et al. 2009] Dourisboure, Y., Geraci, F., and Pellegrini, M. (2009). Extraction and classification of dense implicit communities in the web graph. *ACM Trans. Web*, 3(2).

[Elbassuoni and Blanco 2011] Elbassuoni, S. and Blanco, R. (2011). 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. ACM.

[Elseidy et al. 2014] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. (2014). Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.*, 7(7):517–528.

[Fan 2022] Fan, W. (2022). Big graphs: Challenges and opportunities. *Proc. VLDB Endow.*, 15(12):3782–3797.

[Ferraz et al. 2024] Ferraz, S., Dias, V., Teixeira, C. H., Parthasarathy, S., Teodoro, G., and Meira, W. (2024). Dumato: An efficient warp-centric subgraph enumeration system for gpu. *Journal of Parallel and Distributed Computing*, 191:104903.

[Ferraz et al. 2022] Ferraz, S., Dias, V., Teixeira, C. H., Teodoro, G., and Meira, W. (2022). 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.

[Hoffman and Krasle 2015] Hoffman, F. and Krasle, D. (2015). Fraud detection using network analysis. Patent No. EP2884418A1, Filed September 1st., 2014, Issued June 17th., 2015.

[Hong et al. 2011] Hong, S., Kim, S. K., Oguntebi, T., and Olukotun, K. (2011). Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 267–276, New York, NY, USA. Association for Computing Machinery.

[Hooi et al. 2020] Hooi, B., Shin, K., Lamba, H., and Faloutsos, C. (2020). Telltail: Fast scoring and detection of dense subgraphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):4150–4157.

[Huan et al. 2003] Huan, J., Wang, W., and Prins, J. (2003). 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. IEEE Computer Society.

[Jamshidi et al. 2020] Jamshidi, K., Mahadasa, R., and Vora, K. (2020). Peregrine: A pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA. Association for Computing Machinery.

[Jin et al. 2009] Jin, R., Xiang, Y., Ruan, N., and Fuhry, D. (2009). 3-hop: a high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 813–826, New York, NY, USA. Association for Computing Machinery.

[Junttila and Kaski 2007] Junttila, T. and Kaski, P. (2007). 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. Society for Industrial and Applied Mathematics.

[Kipf and Welling 2017] Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks.

[Kriege and Mutzel 2012] Kriege, N. and Mutzel, P. (2012). Subgraph matching kernels for attributed graphs. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML'12, page 291–298, Madison, WI, USA. Omnipress.

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

[Leon-Suematsu et al. 2011] Leon-Suematsu, Y. I., Inui, K., Kurohashi, S., and Kidawara, Y. (2011). 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.

[Mawhirter et al. 2021] Mawhirter, D., Reinehr, S., Holmes, C., Liu, T., and Wu, B. (2021). Graphzero: A high-performance subgraph matching system. *SIGOPS Oper. Syst. Rev.*, 55(1):21–37.

[Mawhirter and Wu 2019] Mawhirter, D. and Wu, B. (2019). 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. ACM.

[Meng et al. 2018] Meng, C., Mouli, S. C., Ribeiro, B., and Neville, J. (2018). Subgraph pattern neural networks for high-order graph evolution prediction.

[Pržulj et al. 2004] Pržulj, N., Corneil, D. G., and Jurisica, I. (2004). Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515.

[Qin et al. 2019] Qin, H., Li, R.-H., Wang, G., Qin, L., Cheng, Y., and Yuan, Y. (2019). Mining periodic cliques in temporal networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1130–1141.

[Ribeiro et al. 2021] Ribeiro, P., Paredes, P., Silva, M. E. P., Aparicio, D., and Silva, F. (2021). A survey on subgraph counting: Concepts, algorithms, and applications to network motifs and graphlets. *ACM Comput. Surv.*, 54(2).

[Sun et al. 2023] Sun, X., Cheng, H., Li, J., Liu, B., and Guan, J. (2023). All in one: Multi-task prompting for graph neural networks. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, page 2120–2131, New York, NY, USA. Association for Computing Machinery.

[Teixeira et al. 2015] Teixeira, C. H. C., Fonseca, A. J., Serafini, M., Siganos, G., Zaki, M. J., and Aboulnaga, A. (2015). Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15. ACM.

[Ugander et al. 2013] Ugander, J., Backstrom, L., and Kleinberg, J. (2013). Subgraph frequencies: mapping the empirical and extremal geography of large graph collections. In *WWW*.

[Wang et al. 2018] Wang, K., Zuo, Z., Thorpe, J., Nguyen, T. Q., and Xu, G. H. (2018). 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. USENIX Association.

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

[Yang et al. 2016] Yang, Y., Yan, D., Wu, H., Cheng, J., Zhou, S., and Lui, J. C. (2016). 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. Association for Computing Machinery.

[Zaharia et al. 2012] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*.

[Zhao et al. 2019] Zhao, H., Zhou, Y., Song, Y., and Lee, D. L. (2019). 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. ACM.