

N° d'ordre NNT : 2017LYSE1225

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée au sein de

l'Université Claude Bernard Lyon 1

École Doctorale ED512
Infomath

Spécialité de doctorat : Informatique

Soutenue publiquement le 23/10/2017, par :

Ghizlane ECHBARTHI

Big Graph Processing: Partitioning and Aggregated Querying

Devant le jury composé de :

Nom Prénom, grade/qualité, établissement/entreprise	
	Président(e)
Karine Zeitouni, Professeur, Université de Versaille	Rapporteure
Raphael Couturier, Professeur, Université de Franche-Comté	Rapporteur
Angela Bonifati, Professeur, Université Lyon 1	Examinatrice
Mohand Boughanem, Professeur, Université Paul Sabatier	Examinateur
Hamamache Kheddouci, Professeur, Université Lyon 1	Directeur de thèse

Remerciements

Tout d'abord je tiens à remercier mon directeur de thèse Hamamache KHEDDOUCI, de m'avoir acceuilli au sein de l'équipe GOAL, d'avoir constamment veillé sur la qualité de mon travail et de m'avoir apporté un soutien scientifique et morale d'une grande importance. Mes sincères remerciements vont aux membres de jury, pour avoir accepté d'évaluer mon travail et de réviser ce manuscrit.

Mes remerciements chaleureux vont à l'ensemble du personnel du laboratoire LIRIS et de l'université de Lyon, et en particulier le personnel administratif que j'ai pu côtoyer, Isabelle, Brigitte, Jean-Pierre, Catherine, pour leur efficacité et leur bienveillance.

Je tiens à remercier mon amie Kaoutar Klaye, pour sa fidélité et son soutien intarissable et aussi pour l'inspiration qu'elle me donne pour toujours persévérer et donner le meilleur de moi-même.

Je remercie ma famille, mon père, ma mère, mon frère et sa petite famille, ma soeur et sa petite famille, mes deux soeurs Fati et Meriem d'être là pour moi et de m'apporter toujours un soutien très chaleureux.

Je tiens à remercier mon mari Ouadie, je lui dois beaucoup dans ce travail de thèse, je tiens à le remercier pour sa patience, son écoute, sa disponibilité, sa générosité et son précieux soutien.

Ghizlane

Abstract

The advent of "big data" has tremendous repercussions on a broad range of data related domains, and it has impelled the use of novel techniques that achieve the best tradeoff between computational cost and precision.

In the graph theory field, graphs represent a powerful tool for data and problem modeling where all kinds of problems, starting from the very simple to the very complicated, can be effectively formalized. To resolve NP-complete or NP-hard problems in this field, research is being directed to approximation algorithms and heuristic solutions rather than exact solutions, which exhibit a very high computational cost that makes them impossible to use for massive graph datasets.

In this thesis, we tackle two main problems: first, the graph partitioning problem is studied in the context of "big data", where the focus is put on large graph streaming partitioning. In fact, the graph partitioning is an important and challenging problem when performing computation tasks over large distributed graphs, the reason is that a good partitioning leads to faster computations.

We studied and proposed several streaming partitioning models and heuristics, and we studied their performances theoretically and experimentally as well.

The second problem that we tackle in this thesis, is the querying of partitioned/distributed graphs. In fact, querying graph datasets proves to be an important task as most of the real-life datasets are represented by networks of labeled entities.

In this context, we study the problem of "aggregated graph search" that aims to answer queries on several graph fragments, and has the task to build a coherent final answer such that it forms an "approximate matching" to the initial query. We proposed a new method for "aggregated graph search", and we studied its performances theoretically and experimentally on different real word graph datasets.

Key words: Balanced graph partitioning, Streaming partitioning, Streaming heuristics, Graph querying, Graph matching, Graph similarity metric, Aggregated search.

Résumé

Avec l'avènement du "big data", de nombreuses répercussions ont eu lieu dans tous les domaines de la technologie de l'information, préconisant des solutions innovantes remportant le meilleur compromis entre coûts et précision. En théorie des graphes, où les graphes constituent un support de modélisation puissant qui permet de formaliser des problèmes allant des plus simples aux plus complexes, la recherche pour des problèmes NP-complet ou NP-difficil se tourne plutôt vers des solutions approchées, mettant ainsi en avant les algorithmes d'approximation et les heuristiques alors que les solutions exactes deviennent extrêmement coûteuses et impossible d'utilisation.

Nous abordons dans cette thèse deux problématiques principales: dans un premier temps, le problème du partitionnement des graphes est abordé d'une perspective "big data", où les graphes massifs sont partitionnés en streaming. En effet, le partitionnement de graphe est une tâche cruciale et importante lors de la mise en place des graphes de données distribuées, pour la principale raison est qu'un bon partitionnement permet d'accélerer les calculs sur ces graphes de données distribuées.

Dans ce contexte, nous étudions et proposons plusieurs modèles et heuristiques de partitionnement en streaming et nous évaluons leurs performances autant sur le plan théorique qu'empirique.

Dans un second temps, nous nous intéressons au requêtage des graphes distribués ou partitionnés. En effet, une grande majorité de données est modélisée sous formes d'entités étiquetées et interconnectées, ce qui fait du requêtage des graphes distribués une tâche importante dans une large liste d'applications.

Dans ce cadre, nous étudions la problématique de la "recherche agrégative dans les graphes" qui a pour but de répondre à des requêtes interrogeant plusieurs fragments de graphes et qui se charge de la reconstruction de la réponse finale tel que l'on obtient un "matching approché" avec la requête initiale.

Mots clés: Requête de graphes, Matching de graphes, Mesure de similarité dans les graphes, Recherche agrégative dans les graphes, Partitionnement des graphes, Partitionnement en streaming, Heuristiques de streaming, Partitionnement équilibré des graphes.

Contents

Remerciements ii

Abstract iii

Résumé iv

List of Figures ix

List of Tables xi

1 Introduction 1
1.1 Scope of the thesis and contributions 2

I Massive Graph Partitioning 4

2 Graph Partitioning Problem 6
2.1 Graph Partitioning: Problem Definition and Application 6
2.1.1 Notations 6
2.1.2 Problem Definition 7
2.1.2.1 Balanced & Unbalanced Graph Partitioning 7
2.1.2.2 Hardness Results and Approximation 8
2.1.3 Hypergraph Partitioning 8
2.1.4 Graph Partitioning Applications 9
2.1.4.1 Parallel Processing 10
2.1.4.2 Complex networks 10
2.1.4.3 Road networks 11
2.1.4.4 Image Segmentation 11
2.1.4.5 VLSI Physical Design 11
2.2 Offline Partitioning 12
2.2.1 Spectral Partitioning 12
2.2.2 Multilevel Partitioning 14
2.2.2.1 Coarsening phase 15
2.2.2.2 Partitioning phase 17
2.2.2.3 Uncoarsening and refinement phase 18
2.2.3 Geometric Partitioning 19
2.2.3.1 Geometric partitioning: Linear Embedding based Graph Partitioning 20
2.3 Online Partitioning: Streaming Graph Partitioning 21

2.3.1 Problem Setting & Definition . . . 21
2.3.1.1 Streaming Model . . . 21
2.3.1.2 Problem Definition . . . 23
2.3.2 One Pass Streaming Partitioning . . . 23
2.3.2.1 Linear Deterministic Greedy heuristic . . . 24
2.3.2.2 FENNEL heuristic . . . 24
2.3.2.3 Fractional Greedy heuristic . . . 25
2.3.3 Restreaming Partitioning . . . 25
2.3.3.1 Restreaming the one-pass streaming partitioning heurtistics . . . 26
2.4 Chapter Summary . . . 27

3 Fractional Greedy Heuristic & Partial Restreaming Partitioning 28
3.1 Fractional Greedy: a proposed streaming partitioning heuristic for large graphs . . . 29
3.1.1 Preliminaries . . . 29
3.1.1.1 Streaming Model . . . 29
3.1.1.2 Motivation . . . 30
3.1.2 Fractional Greedy Heuristic . . . 31
3.1.2.1 Objective function . . . 31
3.1.2.2 Fractional Greedy algorithm . . . 32
3.1.3 Experimental Evaluation . . . 33
3.1.3.1 Experimental Set up . . . 33
3.1.3.2 Experimental results . . . 34
3.2 Partial Restreaming Partitioning . . . 35
3.2.1 Restreaming Partitioning: a brief introduction . . . 36
3.2.2 Simple Partial Restreaming Partitioning . . . 36
3.2.3 Selective Partial Restreaming Partitioning . . . 38
3.2.4 Experimental Evaluation . . . 39
3.2.4.1 Experimental Set up . . . 39
3.2.4.2 Experimental results . . . 40
3.3 Chapter summary . . . 42

4 Streaming METIS Partitioning: an Online version of METIS Heuristic for Big Graph Partitioning 44
4.1 Introduction & Context . . . 45
4.2 METIS: Multilevel Graph Partitioning Heuristic . . . 46
4.2.1 Coarsening phase . . . 47
4.2.2 Partitioning phase . . . 47
4.2.3 Uncoarsening and refinement phase . . . 47
4.3 Streaming METIS Partitioning Approach . . . 48
4.3.1 Notations . . . 48
4.3.2 Overview of the method . . . 48
4.3.3 Streaming METIS Partitioning Approach . . . 49
4.3.3.1 *Atomic graph construction* . . . 49
4.3.3.2 Partitioning . . . 50
4.3.3.3 Complexity Analysis . . . 53

4.4 Experimental Evaluation . . . 54
4.4.1 Settings . . . 54
4.4.2 Performance Metrics . . . 55
4.4.3 Experimental Results . . . 56
4.5 Chapter Summary . . . 60

II Aggregated Graph Search 61

5 Aggregated search: General definition and overview 63
5.1 Aggregated search: Motivation and definition . . . 64
5.1.1 Motivation . . . 65
5.1.2 Aggregated search definition . . . 65
5.2 Aggregated search: related research and IR disciplines . . . 68
5.2.1 Federated search . . . 69
5.2.2 Question answering . . . 69
5.2.3 Natural language generation . . . 70
5.2.4 Composite retrieval . . . 70
5.3 Graph aggregation: Miscellaneous approaches around "aggregation" in graphs . . . 71
5.4 Relational Aggregated Search . . . 71
5.4.1 Relational Aggregated search framework . . . 72
5.4.1.1 Query Dispatching . . . 72
5.4.1.2 Relation Retrieval . . . 73
5.4.1.3 Result Aggregation . . . 73
5.5 Chapter summary . . . 74

6 Graph search methods 76
6.1 Preliminaries . . . 76
6.1.1 Graph search . . . 77
6.1.1.1 Graph Indexing . . . 77
6.1.1.2 Graph Matching . . . 78
6.2 Filtering and verification methods . . . 82
6.3 Graph querying methods based on graph matching . . . 83
6.3.1 (Sub)graph matching . . . 83
6.3.2 Graph matching based on similarity metrics . . . 84
6.3.2.1 SAGA: a subgraph matching tool for biological graphs . . . 84
6.3.2.2 NEMA . . . 87
6.4 Querying semi-structured data (RDF graphs) . . . 91
6.5 Aggregated search in graph databases. . . . 91
6.6 A taxonomy on graph search methods and discussion . . . 92
6.7 Chapter summary . . . 93

7 LaSaS: A new Approximate Graph Matching framework based on Aggregated Search 94
7.1 Graph search using Aggregated Search . . . 95
7.1.1 Aggregated search in graph databases: Definition . . . 95
7.1.2 Aggregated search in graph databases: Motivation . . . 96

7.2 Preliminaries . . . 97
7.2.1 Query, Target graph, Answer . . . 98
7.2.2 Objective function . . . 98
7.2.2.1 Label similarity component . . . 98
7.2.2.2 Structure similarity component . . . 99
7.2.3 Problem Formulation . . . 100
7.3 Query Processing Algorithm . . . 100
7.3.1 Selection step . . . 100
7.3.2 Aggregation step . . . 103
7.3.3 Refinement step . . . 104
7.4 Experimental evaluation . . . 105
7.4.1 Experimental set up . . . 105
7.4.2 Experimental results . . . 107
7.5 Chapter summary . . . 110

8 Conclusion and Perspectives **111**

Bibliography **113**

List of Figures

1.1 An example of directed and undirected graphs. 1

2.1 Example of a hypergraph [1]. 9
2.2 Multilevel Graph Partitioning [2]. 15
2.3 The streaming model used in the streaming graph partitioning problem. . 22

3.1 Streaming model used in the streaming graph partitioning problem. . . . 30
3.2 Cost function g. The x-axis represents the size of a part and the y-axis represents the cost value. 32
3.3 Average fraction of edge cut λ for $FG, LDG, Fennel$ and $METIS$ over ten graph datasets. 35
3.4 Variation of λ with the growing size of the portion being restreamed represented by β for Webgoogle graph and LiveJournal. On the left $PartFennel$ and on the right $PartLDG$ and at the bottom $PartFG$. . . . 41

4.1 Constructing the *atomic graph* at iteration t. 51
4.2 Coarsening phase: an example with a focus on nodes d, M_2 and M_3. . . . 53
4.3 SMP method: Variation of λ for different values of p for Enron and Slashdot graph datasets. 59

5.1 Example of an Aggregated search result in Google web search engine. . . 66
5.2 Aggregated search framework. 68
5.3 A taxonomy of Data Aggregation. 75

6.1 A taxonomy of Graph Search methods. 92

7.1 Aggregated Graph Search vs. Classical Graph Search. 96
7.2 Vertex weight update vs. Maximum Common Subgraph. 102
7.3 Pruning process of the aggregate. 105
7.4 F1-measure and Runtime (in seconds) reported on DBpedia graph, upon four different query sets with 100 query within each set while varying the label noise, having the structural noise set to 10%. 107
7.5 F1-measure and Runtime (in seconds) reported on DBpedia upon four different query sets with 100 query within each set while varying the structural noise, having the label noise set to 10%. 108
7.6 F1-measure and Runtime (in seconds) reported upon four different query sets with 100 query within each set while varying the matching nodes ratio in the B set. 108

7.7 F1-measure in terms of node matches, graphs matches and runtime reported upon four different query sets with 100 query within each set while varying the parameter λ. 109
7.8 F1-measure in terms of node matches and runtime of LaSaS, NEMA, SAGA and BLINKS over all datasets. Results are reported for query set 3 (n=7,d=3), where $k = 1$ and label and structural noise were set to 10%. 109
7.9 Runtime reported upon four different query sets for all three datasets while varying the parameter k (the number of expected aggregates). . . . 110

List of Tables

3.1 Graph Datasets used for our tests. 34
3.2 Fraction of edge cut λ and maximum load normalized ρ for 3 streaming heuristics LDG and $FENNEL$ and FG and METIS,(1.001) indicates that the slackness allowed is 001. Results are obtained for 10 graph datasets where $k = 40$ 34
3.3 Comparison of the fraction of edge cut λ and the normalized maximum load ρ for ReFG and PartFG, ReLDG and PartReLDG, ReFennel and PartFennel. $k = 40$ and $s = 10$ and $\beta = \frac{k}{2}$. 41
3.4 Comparison of the fraction of edge cut λ and the normalized maximum load ρ for Partial restreaming methods and Selective partial restreaming methods $PartFG$ *vs* $PSelectFG$, $PartLDG$ *vs* $PSelectLDG$, $PartFennel$ *vs* $PSelectFennel$. Results are obtained for 7 graph datasets where $k = 40$ 42
3.5 Runtime gain computed for $PartLDG$ and $PartFennel$ and $PartFG$ over executions on ten graphs with $k = 40$ and $s = 10$. 42

4.1 Graph Datasets used for our tests. 55
4.2 Fraction of edge cut λ given by SMP vs METIS. The balance ρ is 1.001 for both methods. 56
4.3 Fraction of edge cut λ and normalized maximum load ρ given by SMP vs. online competitors: LDG, FENNEL and FG. 57
4.4 Fraction of edge cut λ and normalized maximum load ρ given by SMP vs. restreaming methods: ReLDG, ReFENNEL and ReFG. 58
4.5 Fraction of edge cut λ given by SMP in the BFS, DFS and Random orders. 58
4.6 Fraction of edge cut λ given by SMP vs. online competitors: LDG, FENNEL and FG under the DFS streaming order. 59
4.7 Execution runtime T_{exec} of SMP and METIS given in seconds. 60

Chapter 1

Introduction

Graphs are the core subject of study in graph theory. The trivial structure of a graph, which consists of nodes and edges, was first used by J. J. Sylvester in 1878 [3].

An inclusive definition of graphs is found on Wikipedia: *"a graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge (also called an arc or line). Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges."* [4]. Two major types of graphs exist: the undirected and directed graphs (see Figure 1.1). The undirected graphs represent nodes that share mutual relations represented by edges, whereas the directed graphs represent nodes with directed relations, *i.e.*, not reciprocal. A simple example of a directed graph is a graph where nodes represent father and son, and the relation "is the father of "is represented by a directed edge, from the father node to the son node. An example for undirected graphs is where nodes represent siblings, and two nodes representing sister and brother have a mutual relation "is the sibling of" represented by an undirected edge.

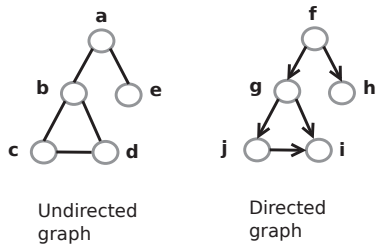


FIGURE 1.1: An example of directed and undirected graphs.

Formally, graphs are usually defined as follows: we consider a graph $G = (V, E)$, where V denotes the vertex set (or the nodes set), and E denotes the edge set, where each edge is represented by two vertices and we have $E = V \times V$.

After this brief introduction to graphs, let us discuss on how graphs became so popular and what are the main surrounding challenges in the field.

Since the 19th century, graphs have attracted the interest of many researchers, and a rich literature has arisen throughout the years. In fact, graphs have been proved as an effective way of representing objects and modeling structured data [5]. Moreover, they represent a pervasive and flexible representation formalism suitable for a broad range of problems in intelligent information processing.

With the advent of big data, recent years have witnessed a sheer increase in the size of graph datasets emerging in different applications such as social networks. For instance, Facebook amounts to 1.284 billion of daily active users [6] and Twitter totals up to 328 millions of active users [7], not to mention the world wide web consisting of 4.61 billion of hyperlinks [8].

1.1 Scope of the thesis and contributions

The scope of this thesis could be seen as the intersection of the graph theory with big data in two main problems: the *Graph Partitioning* problem and the distributed graph querying problem, also known as *Aggregated Graph Search*.

The actual sheer increase in data has led to a plethora of graph datasets, which has attracted the interest in creating new frameworks for big graph processing: Graphlab [9], Microsoft's Trinity [10], Pregel [11] and its open-source version Giraph [12], to name but a few. These platforms usually perform a partitioning of the graph as a preprocessing step, *i.e.*, it distributes the graph dataset across several machines and performs parallel computation afterward. However, graph processing frameworks use a hash function to partition the graph, which gives balanced partitions but incurs a big communication volume causing the computations to slow down.

The graph partitioning problem tackles this issue, it aims to divide the graph data into several distinct sets of equal sizes with the constraint of minimizing the number of edges crossing these sets. In fact, balancing the partition ensures that each machine is given the same workload, and the minimized number of crossing edges reduces the network overhead which ultimately makes the graph partitioning an essential preprocessing step aiming to speed up the computations on graph datasets.

In parallel, data proliferation favors the use of graphs as a storage support, and as a result, graphs become a popular data model that enables efficient data processing. In considering this matter, graph querying has attracted the interest of many researchers as it represents a crucial task to explore the knowledge in these datasets.

Personal contributions during this thesis are as follows: we introduce a new heuristic for streaming partitioning that brings significant improvements over the state-of-the-art heuristics. We also introduce the *partial restreaming partitioning* which is a hybrid streaming model that improves the overall performances and lowers the computational costs.

In addition, we present a novel streaming heuristic for graph partitioning, termed Streaming METIS Partitioning (SMP), based on an adaptation of the well-known offline METIS [2]. The new heuristic extends METIS to online setting and brings about significant benefits to streaming graph partitioning.

Last but not least, we propose a framework for approximate graph matching called Label and Structure Similarity Search (LaSaS). The proposed framework enables an effective RDF graph querying using the *aggregated search* paradigm in the context of graphs.

Organization of the thesis. The first part of this thesis, from Chapter 2 to 4 is about massive graph partitioning. In Chapter 2, we present the graph partitioning problem, introduce many of the underlying concepts, and give an overview of the literature on the Graph Partitioning problem as well.

In Chapter 3, we present our proposed heuristic for streaming graph partitioning named Fractional Greedy, and present our proposed *partial restreaming partitioning* model.

In Chapter 4, we present our proposed approach called Streaming METIS Partitioning (SMP), intended for the streaming graph partitioning problem.

The second part of the thesis, from Chapter 5 through 7, is about the Aggregated Graph Search. In Chapter 5, we give a general overview of the general *aggregated search* concept. In Chapter 6, we review foremost algorithms and frameworks for the graph querying or the graph search task. Chapter 7 presents our proposed framework for performing *aggregated graph search*. Finally, in Chapter 8, we give a summary of the thesis and raise important future work directions and perspectives.

Part I

Massive Graph Partitioning

Graph partitioning (GP) is a crucial task as a preprocessing step on large-scale graph processing frameworks. There is a broad range of applications for the graph partitioning problem, and the well-known is in parallel computing, where GP plays a key optimization role by accelerating computations and balancing the workload among parallel processors or machines. Another reason for the GP to have gained such popularity is the advent of big data in the last decades, which gave rise to a plethora of datasets that are being stored and exploited as graphs.

In this part of the thesis, we will focus on the graph partitioning problem for massive graphs. The first chapter focuses on introducing the graph partitioning problem along with surrounding notions and concepts. We also give definitions and reviews on foremost works on the problem.

Chapter 3 and 4 present personal contributions on the field of large graph partitioning. Precisely, in Chapter 3, we present our proposed heuristic and model for the streaming graph partitioning, the latter being a partitioning setting that is suitable for processing massive graphs.

In Chapter 4, our proposed method called Streaming Metis Partitioning in presented. The novelty of this approach is that it has a hybrid feature, from both the offline and the online setting in graph partitioning, which makes it a well-performing heuristic for GP solving.

Chapter 2

Graph Partitioning Problem

This chapter is dedicated to introducing the graph partitioning problem, and give definitions about underlying concepts, along with a synoptic list of foremost works and methods for solving the graph partitioning problem. Section 2.1 introduces the graph partitioning, and gives a formal definition of the problem. Section 2.2 reviews related work on graph partitioning within the offline class of methods, while in Section 2.3, we review related works belonging to the online class of methods. Section 2.4 summarizes the chapter and outlines important arisen ideas.

2.1 Graph Partitioning: Problem Definition and Application

This section is dedicated to introduce and present the graph partitioning problem, and related notions as well. We identify different variants of the problem, give definitions of related notions, and we finally give important applications of the problem of graph partitioning.

2.1.1 Notations

We will be using the following notations throughout the chapter. Let G be a graph, and we have $G = (V, E)$, where V is the vertex set and E the edge set with $|V| = n$ (the number of vertices in G) and $|E| = m$ (the number of edges in G). Let k be a positive integer that represents the number of parts into which the graph G is partitioned. We define the partition P of graph G as the partition of V into k subsets of equal size, ($i.e. \{V_1, V_2, \cdots, V_k\}$ such that $V_i \bigcap V_j = \emptyset$ for $i \neq j$ and $\bigcup V_i = V$) .

We refer to the k sets as parts, clusters or machines interchangeably. Let $[k]$ denotes the set of positive integers $\{1, \cdots, k\}$.

In the partition P, the *cut* or *edge cut* refers to the number of edges in E whose incident vertices belong to different subsets V_i and V_j with $i \neq j$. We denote by λ the *cut ratio*, which is the fraction of the edge cut over the total number of edges in the graph.

2.1.2 Problem Definition

Graph partitioning is a well-studied problem, and it is commonly known as the k-way graph partitioning. It asks to divide a given graph into k balanced parts while minimizing the number of edges running across these parts.

In other words, given a graph $G = (V, E)$, the balanced k-way partitioning aims to partition V into k subsets $(\{V_1, V_2, \cdots, V_k\})$ of equal size such that they form a *partition* P of V, which minimizes the *edge cut*.

Cut Objective function. The partition that is often sought in the graph partitioning problem, adheres to an objective function that should be either minimized or maximized. The prominent and mostly used objective function is the *edge cut*.

Let P be a partition of V. The edge cut is the number of edges in E whose incident vertices belong to different subsets of the partition P. Let $e(V_i, V - V_i), i \in [k]$ be the set of edges with ends belonging to different clusters. We define $\lambda = \sum_{i=1}^{k} \frac{|e(V_i, V-V_i)|}{m}, i \in [k]$ as the fraction of edge cut or cut ratio. In most cases, λ should be minimized during partitioning.

Another formulation of the graph partitioning problem is (k, ν)-balanced graph partitioning, where ν represents the imbalance factor and we have $\nu = 1 + \epsilon$, where ϵ is a positive real number. For a partition to be balanced, each part among the k ones, has to be of size $\nu \times \lceil \frac{n}{k} \rceil$. Moreover, the edge cut should be minimized.

2.1.2.1 Balanced & Unbalanced Graph Partitioning

Also known as the constrained and unconstrained graph partitioning, the balanced and the unbalanced graph partitioning problems are two variants of the original problem. The main difference between these two problem variants, is that the balanced graph partitioning adheres to a strict constraint regarding the balance, *i.e.*, the parts should be of equal sizes and imbalances are tolerated to a tight and firm extent.

On the contrary, the unbalanced graph partitioning relaxes the constraint of strict balance, and aims to solely minimize or maximize an objective function. In a way, unbalanced graph partitioning problem can be seen as a clustering problem, where the priority is given to minimizing (or maximizing) an objective function regardless of the clusters' sizes to be balanced or not.

2.1.2.2 Hardness Results and Approximation

As a decision problem, the k-way graph partitioning was proved to be NP-Complete [13, 14]. Moreover, the balanced graph partitioning was proven to be NP-Hard in [15], as an optimization problem. In fact, if we consider the setting of (k, ν)-balanced graph partitioning, when $k = 2$ and $\nu = 1$ (every part has to be of size $\nu \times \lceil \frac{n}{k} \rceil$), then balanced graph partitioning problem is reduced to the problem of minimum bisection, which is also an NP-hard problem [16]. Due to the hardness of the problem, research works are mostly directed toward approximation algorithms and mainly toward heuristic solutions.

Andreev *et al.* have shown, in [16], that there is no constant-factor approximation for the exactly balanced version (when $\nu = 1 + \epsilon$ and $\epsilon = 0$) of this problem on general graphs. If the balance constrained is slightly relaxed by tolerating some imbalance, particularly if $\epsilon \in (0, 1]$, then an $O(log^2 n)$ factor approximation can be achieved. If the imbalance is larger, *i.e.* $\epsilon > 1$, an approximation ratio of $O(log n)$ is possible [17].

Several studies have suggested algorithms with approximation guarantees [16, 18, 19]. In [18], authors present an approximation algorithm with a polylogarithmic approximation guarantee.

Other contributions inspired from tasks settings propose more effective approximations: In [16], an $O(\epsilon^{-2} log^{1.5} n)$ approximation is proposed for a chosen and fixed $\epsilon > 0$. Besides, another $O(\sqrt{log(k) log(n)})$ approximation algorithm is proposed based on semidefinite programming [19].

In the practical point of view, most of the approximation algorithms are not implemented, and if so, they show poor performances and are particularly too slow when used for large graphs. Therefore, mostly heuristic solutions are used in practice.

2.1.3 Hypergraph Partitioning

Hypergraphs represent a generalization of simple graphs, where an edge can connect not only two nodes, but a group of them. Let $H = (V', E')$ be a hypergraph, then V' is the set of nodes, and E' is the set of *hyperedges* (also called *net*). In Figure 2.1, we

present an example of a hypergraph, where $e_1, \cdots, e5$ represent the hyperedges, and they connect several groups of vertices.

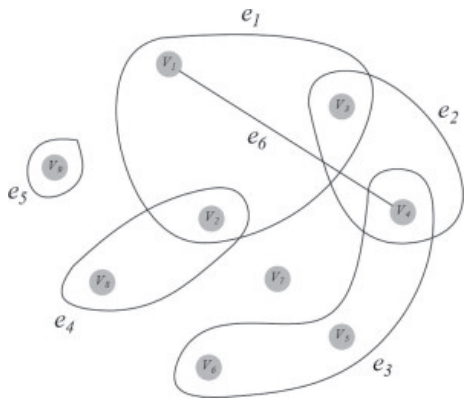


FIGURE 2.1: Example of a hypergraph [1].

Similarly to simple graph partitioning, hypergraph partitioning aims to find a partition of nodes into different subsets of equal size. However, the corresponding objective function expression is different. The edge cut in hypergraphs is called the hyperedge cut, and it represents the number of hyperedges connecting different subsets of nodes. A metric used to solve the hypergraph partitioning is called $(\lambda - 1)$ metric, where $(\lambda - 1)$ metric $= \sum_{e \in E'} \lambda_e$ where λ_e is the number of subsets connected by the hyperedge e.

Compared to simple graph partitioning, hypergraph partitioning has a major drawback of using complex algorithms due to the model complexity. Therefore, hypergraph partitioning should be used solely when the hypergraph model significantly benefits the underlying application.

In this thesis, we focus on the simple graph partitioning problem and therefore, omit giving more detailed information about the hypergraph partitioning. However, it is noteworthy to mention that many methods for simple graph partitioning have been translated to hypergraph partitioning [20–23], and the main application domain for hypergraph partitioning is VLSI design (Subsection 2.1.4.5).

2.1.4 Graph Partitioning Applications

The graph partitioning problem has many applications in different domains. In the following, we list some of the important and well-known applications of the problem.

2.1.4.1 Parallel Processing

The k-way graph partitioning has major importance in distributed computation systems as it directly affects their performance: the k-way partitioning aims to reduce the overall runtime of the application, by assigning to each processor (or machine) the same amount of data while reducing the communication overhead by minimizing the edge cut.

2.1.4.2 Complex networks

Processing involving complex networks that arise in many domains benefit significantly from the graph partitioning problem. In the following, we focus on three well-known types of complex networks: social networks, power grids and biological networks.

Social Networks. A trivial problem that is studied in social network domain is the identification of community structure, also known as the community detection problem. In the context of these kinds of problems, the number of desired clusters is not specified as an input, unlike the graph partitioning problem, where the number of desired parts or clusters is fixed a priori. Even though, graph partitioning technics have largely inspired the community detection algorithms [24], and graph partitioning methods are usually used to give a first approximation of them. Readers interested in more examples of graph partitioning methods used for the community detection problem are directed to [25].

Power grids. In the realm of power grids, two major concerns are cascading failures and disturbances, which if controlled, can avoid calamitous blackouts. To prevent the propagation of failures in power grids, there is an approach that consists in partitioning the grid into independent subsets [26]. This partitioning uses an objective function that is combined with other optimization components such as the load shedding to intensify robustness and lessen the impact of cascading failures [27].

Biological Networks. A wide range of complex biological systems are modeled by graph representations, for instance, we can cite the protein-protein interactions network and gene co-expression network. In graph representation of a biological network, nodes correspond to biological entities (proteins, genes), and edges to a common feature in some biological process. The partitioning of these networks can have numerous goals. For example, the partitioning gathers, in one cluster, nodes that exhibit identical behavior to each other, which can be used for data reduction. Moreover, partitioning can help in the detection of some biological processes by identifying clusters of nodes participating in the process. Additional information about the biological networks could be found in [28].

2.1.4.3 Road networks

In such networks, nodes usually correspond to places, and edges represent road segments. In this context, graph partitioning is widely used for route planning speed up [29–33]. An algorithm called arc-flags was proposed in [29], which performs geometric partitioning as a preprocessing step to reduce the search space of Dijkstra's algorithm. In fact, finding good partitions in road networks plays a crucial role in reducing the post-processing costs.

2.1.4.4 Image Segmentation

In the field of image processing, image segmentation corresponds to the task of partitioning the pixels of an image into several groups that correspond to distinct objects, where the graph partitioning have been widely used as solution technique.

Since early 90's, representing images as graph structures was very popular, and many cut-based methods for image processing have arisen in this context. The representation of an image as a graph is such that a vertex corresponds to a pixel or group of pixels, and edges, usually weighted, represent similarities between them. Usually, the edge weights quantify a similarity or dissimilarity degree between nodes, like the difference of color intensity between the two pixels. Hence, the objective function used in graph partitioning in this context can be expressed differently depending on the applications and the explicitness needed to distinguish segments. Additional information about graph partitioning and image segmentation could be found in [34, 35].

2.1.4.5 VLSI Physical Design

Physical design of digital circuits for very large-scale integration (VLSI) systems are known for having intensively used graph partitioning technics for achieving general performance optimization. The graph representation of such circuits is such that nodes represent the cells, which correspond to atomic units of the circuit (gates), and edges represent the wires (cables). The main goal of partitioning these circuits is to reduce the VLSI design complexity, by partitioning it into smaller components and keeping the overall length of all the wires (cables) short. Moreover, there are additional constraints for the circuits partitioning to ensure even better results such as: fixing the set of nodes that should end up in the same part (cluster), and fixing a firm threshold on the maximum cut size between parts (clusters). However, it is noteworthy to point out that hypergraphs (see 2.1.3) model the circuit more precisely, as the gates (nodes) are

connected with wires (edges) with more than two endpoints. Further information about VLSI circuits partitioning can be found in [36].

2.2 Offline Partitioning

The offline partitioning designates a typical feature of all early proposed methods for graph partitioning. This feature refers to the fact that in order to partition a graph, the latter should be entirely memory resident. In other words, traditional or classical methods for graph partitioning necessitate complete information of the whole graph to be partitioned, which is justifiable since graph datasets used at the time were relatively of small sizes. However, with the advent of the internet and big data, using such methods for graph partitioning becomes extremely challenging, which has led to design new methods and technics that we refer to as the "online partitioning" (see the following section).

The practical importance and NP-hardness of the graph partitioning problem have caused many heuristics and methods to be proposed: the spectral methods, known to produce excellent partitions for a wide class of problems despite their high expense. Geometric methods which use geometric information about the graph to partition it. These algorithms are known to be fast but usually produce partitions of worse quality compared to spectral methods. There is also the local spectral partitioning methods, *e.g.*, EvoCut [37], but it still requires information about large portions of the graph and performs large computations after loading the graph data. Multilevel methods such as METIS [2], are known to be fast and achieve high-quality partitioning.

2.2.1 Spectral Partitioning

The spectral method is an ancient way of partitioning a graph. It was first used by W. Donath and A. Hoffman [38, 39]. By this time, the spectral method was very popular, and was widely used to solve graph partitioning problems before multilevel methods took over.

The spectral method uses the Laplacian matrix, the Fiedler vector, eigenvalues to achieve a partitioning of the graph. We define in the following each of the aforementioned elements.

Definition 2.1. Laplacian matrix. To define the Laplacian matrix, let us first define the adjacency and the degree matrices of a graph G.

Adjacency matrix. Let A be the adjacency matrix of $G = (V, E)$. Let $w(i, j)$ be the weight on edge (i, j) in E, for a simple graph, we have $w(i, j) = 1$. Then A is a square matrix $n \times n$, and for every $i, j \in \{1, \cdots, n\}^2$, we have the element $a_{i,j}$ in the i^{th} line and j^{th} column in the matrix:

$$a_{i,j} = \begin{cases} 0 & \text{if } i = j \\ w(i, j) & \text{else} \end{cases}$$

Degree matrix. The degree matrix D of the graph G is defined as follows:

$$d_{i,j} = \begin{cases} deg(i) = \sum_{l=1}^{n} w(i, l) & \text{if } i = j \\ 0 & \text{else} \end{cases}$$

Where $deg(i)$ is the degree of the i^{th} node.

Laplacian matrix. Finally, the Laplacian matrix L of G is defined as follows:

$$L = D - A$$

In order to partition a graph, the spectral method uses the eigenvectors of the Laplacian matrix along with their corresponding eigenvalues. Let us define the eigenvectors and eigenvalues.

Definition 2.2. Eigenvector and Eigenvalue. Let E be a vector space on a division ring K'. Let u be an endomorphism of E. Assume a vector x of E and θ in R such that $u(x) = \theta x$, then $u(x) = \theta x$ is called eigenvector of u. And θ is called eigenvalue of u.

Let consider a matrix M of an endomorphism on real numbers; then we call eigenvector of M every vector x (column matrix) of real numbers that verifies $\exists \theta \in \mathbb{R}, Mx = \theta x$.

The spectral partitioning method uses the *Fiedler eigenvector and eigenvalue* of the Laplacian matrix L to find a bisection of G. Nevertheless; the spectral method can perform k-way partitioning through recursive bisections, where $k = 2^i$.

Fiedler eigenvectors and eigenvalues. M. Fiedler was the first one to study the properties of these eigenvectors and values [40]. Therefore, this group of eigenvectors is usually called the Fiedler vectors. The first Fiedler vector is the unit vector, and its eigenvalue is 0. The second Fiedler vector has the smallest non-zero eigenvalue of this eigenvalues group. Particularly, it is the second Fiedler vector that is used to find a bisection of the graph.

There are several methods that can be used to find the eigenvectors of a matrix, we cite: the Lanczos iterative algorithm, the Rayleigh quotient iteration method, the Jacobi iterative algorithm and the QR decomposition algorithm. All these methods are presented in [41]. The Lanczos iterative algorithm was widely used [42–44]. However, the Rayleigh quotient iteration method is also used [45], and has gained some popularity when it is coupled with a multilevel method, which turns out to accelerate the computations [46].

Spectral method. The spectral partitioning aims to find a 2^i-partition (with $i \geq 1$) for a graph using the eigenvalues and eigenvectors of the laplacian matrix of the graph. Actually, we focus on the second smallest eigenvalue and the corresponding eigenvector, called Fiedler vector. The fiedler vector is of size n (the degree of the graph), and we take the median m' of the vector and use it as a splitting value to get a bisection of the graph, such that: Let $v(i)$ be the i^{th} component of the Fiedler vector v, then we have:

- If $v(i) \leq m'$ then place the ith vertex in V_1
- Else place the ith vertex in V_2

Where V_1 and V_2 are the two parts of the bisection.

For partitioning into 2^i parts where $i > 1$, the spectral method recursively partitions the graphs into successive bisections. For instance, when $i = 2$, *i.e.*, we want to partition a graph into 4 parts, then the graph is first partitioned into 2 parts, then each obtained part is considered a graph, where the laplacian matrix is computed to find the second fiedler vector and obtain the second 2 parts.

Though they are known for achieving good quality results, the spectral methods suffer the major problem of high expense, the reason is that the algorithms used for finding eigenvectors are computationally very expensive. Even though the process is faster when coupled with a multilevel method, they are still too expensive to use for large actual graphs.

2.2.2 Multilevel Partitioning

Multilevel graph partitioning approaches are considered as the most successful heuristics for partitioning graphs. This type of methods is a combination of several borrowed algorithms that could be used independently in other contexts. These algorithms correspond to local optimization methods, called refinement methods in the graph partitioning context, and also direct partitioning methods such as the spectral method or region growing method. The multilevel method consists of three main phases: i) coarsening phase, ii)

partitioning phase and iii) uncoarsening/refinement phase. The coarsening phase aims to contract the graph nodes to reduce the overall size of the graph. Then, a partitioning of the coarse graph is done using some algorithms for partitioning, *e.g.*, the spectral method. Finally, the uncoarsening phase maps the found partition to the original graph and perform a refinement on the partition through local optimization swaps on parts. The multilevel partitioning is depicted in Figure 2.2.

In the following, we detail each of the three phases along with algorithms used.

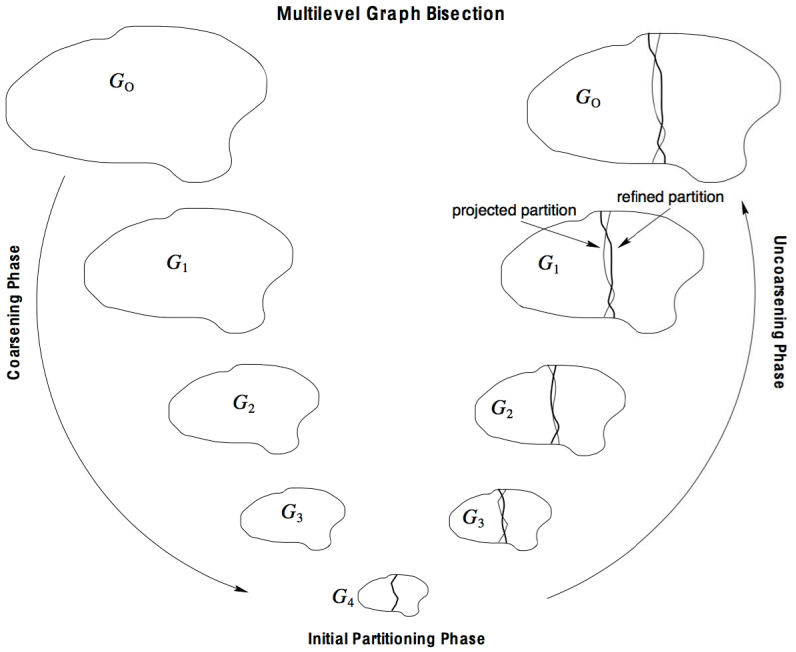


FIGURE 2.2: Multilevel Graph Partitioning [2].

2.2.2.1 Coarsening phase

The coarsening phase aims to successively reduce the size of the graph to be partitioned.

Let G_0 denote the original graph; then during coarsening, a series of smaller graphs $G_1, G_2, \cdots, G_q$ will be created such that $V_0 > V_1 > V_2 > \cdots > V_q$ where G_q is called the coarsest graph and consists of fewer vertices.

Usually, this can be achieved by finding a maximal matching of the graph, the reason is that a maximal matching of the graph has the ability to preserve many properties of the original graph [2, 47].

Definition 2.3. Maximal matching. The matching of a graph, also known as the independent edge set, is a set of edges without common vertices. Moreover, a matching

M of a graph G is maximal if every edge in G has a non-empty intersection with at least one edge in M [48].

There are several ways to compute the maximal matching: Random matching (RM), Heavy edge matching (HEM), Light edge matching (LEM), Heavy clique matching (HCM).

Random matching (RM). Random matching consists in finding a matching using a randomized algorithm as follows: A vertex u is randomly selected. If u is not matched yet, then an unmatched neighbor v is randomly selected. Once v selected, the edge (u, v) is added to the matching, and vertices u and v are marked as matched. In the case of not finding the unmatched neighbor v, the vertex u is then marked unmatched in the final matching.

Heavy edge matching (HEM). HEM is similar to RM, however, it aims at finding a maximal matching with the higher edge-weight, which in turn, will result in a coarse graph with a lower edge-weight. The reason is that a graph with a low edge-weight has a low edge-cut as shown in an analysis given in [49].

HEM is computed using a randomized algorithm similarly to RM described earlier, the vertices are selected randomly. However, a vertex u is matched with vertex v such that the weight of the edge (u, v) is maximum overall incident edges. Though the algorithm does not guarantee that the matching obtained will have maximum weight (overall possible matchings), experiments showed that it works well. The complexity of computing a HEM is $O(m)$ (m is the number of edges).

Light edge matching (LEM). Contrarily to the HEM, the LEM aims to maximize the total edge-weight of the coarser graph. This is achieved by finding a matching with the smallest edge weight, leading to a small reduction in the edge weight of the coarser graph G_{i+1}. It might be surprising to use LEM knowing that it will not reduce the edge-cut during partitioning, however, having a graph with a high average degree is suitable for some algorithms such as KL [50] in order to produce good partitions faster.

In order to compute LEM, there is only a minimal modification to do in the algorithm for computing HEM. Instead of selecting an edge (u, v) that has the largest weight, the edge that has the smallest weight is selected. Hence, the complexity of computing LEM is also O(m).

Heavy clique matching (HCM). In the heavy clique matching, vertices that have high edge density are collapsed. In other words, vertices that form cliques (or almost) will form a multinode and hence, such a clique will not be cut during the partitioning phase, which results in a good partition.

Assuming we have a coarse graph $G_i = (V_i, E_i)$, and we have u and v two vertices in G_i, which means that u and v are multinodes (several vertices are collapsed into them during previous coarsenings) . The edge density between u and v is as follows:

$$\frac{2(C_e(u) + C_e(v) + W_e(u,v))}{(V_w(u) + V_w(v))(V_w(u) + V_w(v) - 1)} \tag{2.1}$$

Where $V_w(u)$ (respectively $V_w(v)$) is the total weight on multinode u (respectively v), which is the sum of weights on vertices that have been collapsed into u (respectively v). And $C_e(u)$ (respectively $C_e(v)$) represents the weight of collapsed edges within multinode u (respectively v), and $W_e(u,v)$ is the weight on the edge (u,v) which corresponds to the sum of the weights on all edges collapsed into (u,v).

The HCM is computed using a randomized algorithm that works as follows: Vertices are randomly selected. The algorithm matches an unmatched vertex u with its unmatched neighbor v, such that the density of the multinode formed by u and v has the largest edge density among all possible multinodes involving v.

It is noteworthy that HCM is similar to the HEM scheme. However, HEM matches vertices regarding only if they are connected with a heavy edge, while HCM matches two vertices if they are connected using a heavy edge and also if each of these two vertices has high contracted edge-weight within.

After the maximal matching is computed, the two vertices of each edge in the matching set are collapsed into one node, thereby reducing the graph size. The process of coarsening is iterated until the coarse graph reaches the desired small size of hundred vertices for example.

2.2.2.2 Partitioning phase

During the partitioning phase, the coarsest graph G_q is partitioned, precisely, the vertex set V_q is partitioned into k parts. And since the coarsest graph G_q consists of few vertices, it is possible to use expensive methods for graph partitioning such as spectral partitioning method to give optimal partitions.

By the same time, various algorithms can be used in this phase, such as spectral method [51], KL algorithm [50], GGP and GGGP algorithms[2].

KL algorithm. The KL algorithm [50] is an iterative algorithm that starts with an initial bipartition of the graph, and through each iteration, it aims at finding a subset of vertices such that if swapped, the overall edge-cut of the partition will be reduced. When these vertices are swapped, they form the partition for the subsequent iteration. The

same process repeats until no vertices to swap could be found. The complexity of each iteration of the KL algorithm in [50] is $O(mlog(m))$, where m is the number of edges in the graph, and it was improved in [52] where the use of special data structures reduced the complexity to $O(m)$. Moreover, the KL algorithm achieves optimal partitions when the initial partition is good and the average degree of the graph is large [53]. If there is any good partition to start with, KL algorithm performs several instances (runs) where each instance starts with a randomly generated partition, and the final partition that achieves the lowest edge-cut is selected. Though it might seem expensive to perform multiple runs of the KL algorithm when the graph in question is large, it is relatively inexpensive since the partitioning is done on the coarse graph which consists of fewer vertices.

Graph Growing Partitioning algorithm (GGP).

GGP algorithm is another alternative to partition the graph. GGP algorithm starts from a vertex and builds a region around it following a breadth-first expansion, until the fixed number of vertices per part have been included [54, 55]. The final result of GGP algorithm depends on the choice of the start vertex, and each start vertex leads to a different partition with a different edge-cut. In order to alleviate this issue, the GGP algorithm is usually used with several different starting vertices, and the best partition (with the smallest edge-cut) is selected. Then, the partition found is refined by passing it as an input to the KL algorithm. Finally, the overall cost if this partitioning is relatively low since it processes the coarsest graph.

Greedy Graph Growing Partitioning algorithm (GGGP). Similarly to the GGP algorithm described in the previous subsection, the GGGP algorithm starts for a vertex and grows a region around, but not in a breadth-first search way. It rather computes an ordering of the vertices to include in the region according to improvements they bring to the partition. In other words, the vertex that will be first included in the region is the one that brings the largest decrease in the edge-cut. The GGGP algorithm is less sensitive to the choice of the start vertex than the GGP algorithm. Moreover, GGGP yields better partitions results compared to GGP algorithm.

2.2.2.3 Uncoarsening and refinement phase

Once the partition on the coarse graph G_q is obtained, it is projected back to G_0 passing through intermediate graphs $G_{q-1}, G_{q-2}, \cdots, G_1$. Moreover, a refinement is performed to enhance the partition quality in the finer graph. The refinement task consists in swapping any two nodes positions whenever it leads to a lower edge cut while preserving the balance of the partition. The Kernighan-Lin (KL) algorithm, described in subsection

2.2.2.2 is usually used for this purpose. In other words, the partition of the graph G_{i+1} is mapped onto the graph G_i, and it is used as the initial partition for the KL algorithm. In fact, the KL algorithm as used in this case will converge to a better partition faster as it starts from a good partition.

2.2.3 Geometric Partitioning

Another class of partitioning technics is called "geometric partitioning". The geometric feature refers to the fact that in some special graphs, spatial coordinates of the nodes can be used to perform a partitioning. This is the case of meshes, which are grids that approximate a geometric domain by dividing it into smaller subdomains. In other words, a mesh could be defined as "the scaffolding upon which a function is decomposed into smaller pieces" [56].

In such partitioning methods, coordinates information about the graph are used to project or to make an embedding of the nodes on a geometric line or plane, and use a geometric technique to make a partitioning, *e.g.*, project the nodes into a line, and use a median of this line to make a bisection.

The simplest geometric method for graph partitioning is recursive coordinate bisection RCB [43], where in each step of the recursion, RCB projects graph nodes onto the coordinate axis and bisects them through the median of their projections. In case of RCB, the bisecting plane is orthogonal to the coordinate axis, which can lead to partitions with large separators in case of meshes with skewed dimensions. The inertial partitioning [57, 58] alleviates this issue. In inertial partitioning, the bisecting plane is orthogonal to a plane L that minimizes the moments of inertia of nodes. That is to say, the plane L is chosen such that the sum of squared distances to all nodes is minimized.

Another interesting algorithm is the random spheres algorithm [59, 60]. It consists on projecting the d dimensional nodes to a random $d+1$ dimensional sphere, and bisecting it by a plane L through its center point. This method is a generalization of the RCB method, and has performance guarantees for k-nearest neighbor graphs and planar graphs.

We give in the following, a highlight of a novel and recently proposed method in the geometric class.

2.2.3.1 Geometric partitioning: Linear Embedding based Graph Partitioning

A recent work in [61], tackles the balanced graph partitioning problem in a distributed setting by using linear embedding technics. The proposed algorithm starts by embedding the nodes onto a line and then processes them in a distributed manner based on the embedding order and using several technics such as minimum cuts and local swaps.

In other words, Aydin *et al.* proposed in [61], a three steps algorithm:

1. The first task is to find a mapping of the graph vertices to a line, where a mapping gives an ordering of the vertices that presumably places neighbors close to each other, which could reduce the minimum-cut partitioning problem to a local optimization problem. The mapping task could be achieved either through *random mapping*, *Hilbert curve mapping* or *affinity-based mapping*. We give a brief description of these mapping technics in the following:

   Random mapping. A naïve method to produce an ordering of vertices is to give random permutation of them. This method does not help in finding good cuts even if it could be very fast.

   Hilbert curve mapping. In the Hilbert curves mapping, geographic/geometric information is needed to construct an ordering using a space-filling curve. This method is known to capture proximity well, that is to say, nodes that are close in space are assumed to be placed nearby on the line. Although there are no theoretical guarantees on the cut generated from Hilbert curves, numerous assumptions on the distributions of edge lengths and node positions enable to bound the cut ratio, showing show that it is significantly less than the cut ratio of random ordering [62–64].

   Affinity-based mapping. The affinity method takes into account the affinity of vertices. Informally, every node starts in a singleton cluster, and then, vertices that are closely connected are grouped into the same cluster, therefore building a tree of connections. The similarity between constructed clusters is computed by a given function of similarities between vertices in the clusters.

   The final ordering is produced by sorting the vertex labels as follows: Let the label for each cluster be the concatenation of vertices IDs strings from the root to the corresponding leaf. Sorting the constructed labels places the vertices belonging to the same cluster in a contiguous piece on the line.

2. The second step consists in improving the ordering obtained through local vertices swapping, such that nodes will be placed next to their most neighbors.

3. The final step is about finding the cut points and refining the partition to improve the cut size. To do so, authors use local post-processing optimization in the "split windows" (a split window is a small interval around the cut points, taking into account permissible imbalance).

2.3 Online Partitioning: Streaming Graph Partitioning

The graph partitioning problem is a well-studied problem with a rich history which witness the importance of this problem and its relevance in solving a wide range of domain-related problems. Nowadays, with the advent of the big data, traditional methods for graph partitioning are not tailored to process big sized graphs and incur an exorbitant cost when dealing with such graphs. In order to alleviate this problem, the streaming graph partitioning problem was introduced in 2012 by Stanton *et. al.*.

In this section we introduce the streaming graph partitioning problem and present the surrounding notions and concepts, then we discuss related work. This section is organized as follows: in Subsection 2.3.1, we first present and describe the streaming model used in the streaming graph partitioning and we give the streaming graph partitioning problem definition. Then, Subsection 2.3.2 reviews and discusses the one-pass streaming heuristics for graph partitioning. And in Subsection 2.3.3, we discuss the the restreaming graph partitioning problem, which is derived method from the streaming partitioning that intends to refine and optimize partitions quality. Finally, we give a brief summary at the end of the section.

2.3.1 Problem Setting & Definition

This section is dedicated to giving definitions surrounding the problem of streaming graph partitioning. First, we define the streaming model, *i.e.*, the setting of the streaming process. Then, we give the definition of the streaming graph partitioning problem.

2.3.1.1 Streaming Model

Streaming graph partitioning was first introduced by Stanton and Kliot [65]. It consists on processing the graph stream in one pass (see Defnition 2.4), where the graph vertices arrive in a certain order (Random, Breadth First Search, Depth First Search $\cdots$ see Definition 2.5), and each vertex is accompanied by its adjacency list.

Definition 2.4. One-pass streaming partitioning. The streaming partitioning is said to be "one-pass" iff the partitioning process performed only one-pass on the graph data stream, *i.e.*, all vertices have been seen no more than once.

Definition 2.5. Stream ordering. The stream ordering represents the ordering in which the vertices of the graph being streamed are explored. We cite three main stream orderings that are usually used in the graph streaming partitioning problem:

- Breadth-First Search BFS: is obtained by randomly selecting a node on each connected component of the graph and starting a breadth-first search from the given node. Component ordering is done at random in case of multiple connected components in the graph.
- Depth-First Search DFS: is similar to BFS but a depth-first search is performed instead of BFS.
- Random: is given by a random permutation of the vertex set.

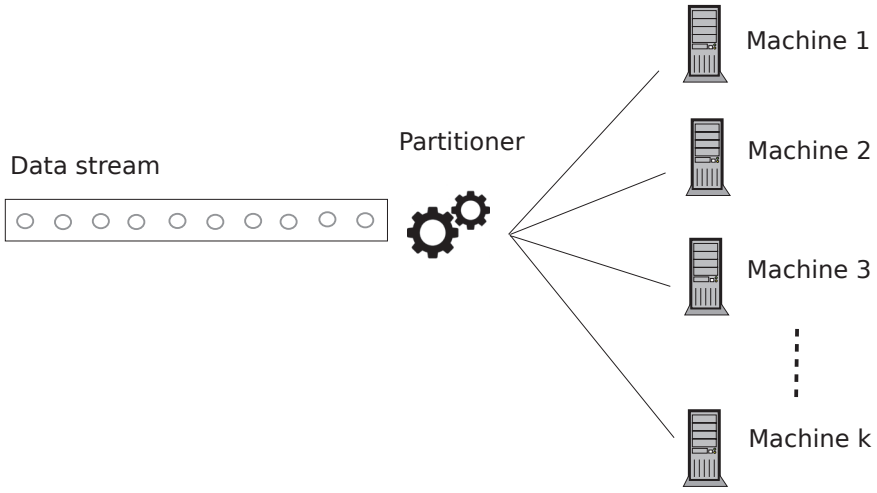


FIGURE 2.3: The streaming model used in the streaming graph partitioning problem.

As depicted in Figure 3.1, the streaming model used in streaming graph partitioning consists of three components: (i) the data stream, (ii) the partitioner and (iii) the target machines. In the following, we provide further details about these three components.

i) Data Stream. The data stream that represents the input of the whole partitioning system, consists of data structures that correspond to the graph vertices along with their adjacency lists. In other words, each data item in the stream is nothing but a vertex accompanied by its adjacency list, where the adjacency list of a vertex v is the list of its neighbors. Particularly, the adjacency list in the item structure includes the references of the neighbors solely, to avoid large size structures to store and increase the processing

speed. In the following, we will refer to data items in the stream as vertices to avoid cluttered sentences.

ii) Partitioner. The vertices in the data stream are serially loaded into the *partitioner*. The partitioner is a program that has the task of assigning each vertex in the stream, to a one selected machine. Precisely, the partitioner selects the one machine where the current vertex will be placed according to the adjacency of the vertex and the available capacity on machines. This machine selection task as performed by the partitioner will be further detailed in section 2.3.2.

iii) Target Machines. As depicted in Figure 3.1, there are k target machines, according to k-way partitioning, where the aim is to partition the graph data being streamed into k parts, where each part is going to be stored in one machine. Suppose that each machine has a storage capacity C, then the total capacity of the k machines should be large enough to handle the whole graph being streamed. In other words, if the size of the graph being streamed is n (the number of vertices), then there is an essential condition for the streaming model to be operational and it is that $n \leq k \times C$.

2.3.1.2 Problem Definition

The streaming partitioning is as follows: when a vertex is loaded on the *partitioner* (along with its adjacency list), a program called the *partitioner* decides on which cluster the vertex will be placed and never relocates it once the assignment is done. The partitioning process is done on the fly without having an access to the whole graph, *i.e.*, the graph does not have to be memory resident. In fact, streaming partitioning uses very limited computational resources in terms of memory and computation time. This property makes the streaming model very suitable when it comes to partitioning billion node graphs, which neither the approximation algorithms nor the heuristics proposed for classical graph partitioning are tailored to process.

2.3.2 One Pass Streaming Partitioning

In this section, we review the core proposed methods in the literature for streaming partitioning in one pass. We highlight in the following three heuristics: Linear Deterministic Greedy heuristic (LDG), FENNEL heuristic and Fractional Greedy heuristic. For each heuristic, we present and explain the objective function used, as well as a discussion on the performances achieved by each heuristic.

2.3.2.1 Linear Deterministic Greedy heuristic

In [65], Stanton and Kliot proposed the first and foremost work in streaming graph partitioning. The authors propose a variety of online partitioning heuristics and conduct a comparative analysis of their performances concluding that the Linear Deterministic Greedy (LDG) remains the most promising heuristic.

As aforementioned in the previous section, the partitioning heuristic is run in the partitioner and has to select the machine where the currently loaded vertex will be located. The LDG heuristic aims at selecting the machine that maximizes the objective function in LDG. In other words, LDG greedily assigns the vertices to machines (clusters) while adding a penalization for big clusters to emphasize balance. The objective function to maximize in LDG is as follows:

$$LDG = \underset{i}{argmax}\ (P_i^t \cap N(v)) * (1 - \frac{|P_i^t|}{C}) \tag{2.2}$$

Where $i \in [k]$ is the index of a given machine, C is the capacity of a machine, v is the current vertex to place, whereas $N(v)$ is its adjacency list. P_i^t is the part of index i at time t, and it refers to the vertices set in machine having the idex i. Given a vertex v, the LDG heuristic works as follows: it selects the machine that has the maximum number of v neighbors, and in the same time, that has available free space to receive additional data.

LDG yields the best results in terms of edges cut in Finite Element Mesh (FEM) datasets, this is due to the structure of FEMs, their edges are highly local which make it possible to obtain very good partitions. However, performances achieved by LDG are far from being optimal. In fact, in a theoretical study on streaming partitioning algorithms [66], Stanton shows that it is impossible for online algorithms to approximate the optimal cut on a single pass over the graph stream within $O(n)$ (where n is the number of vertices of the graph to be partitioned) whatever the streaming order is.

2.3.2.2 FENNEL heuristic

Besides, Tsourakakis *et al.* proposed in [67] an online greedy heuristic for streaming partitioning named FENNEL.

A method proposed by Tsourakakis *et al.* [67], introduces another greedy heuristic approach for graph partitioning FENNEL. This heuristic achieves good performances due to an adjustable objective function.

The objective function of FENNEL is as follows:

$$FENNEL = \underset{i}{argmax}\ (P_i^t \cap N(v)) - \alpha\gamma(|P_i^t|)^{\gamma-1} \tag{2.3}$$

From the objective function, we can see that the first term is similar to LDG (see Equation 3.4), that is FENNEL selects a machine that has the most neighbors of the vertex to be assigned, but has a different second term with two parameters α and γ, that are used for tuning the FENNEL heuristic, and where the best performance values were fixed empirically. FENNEL outperforms LDG by giving partitions of a lower edge cut, however, the balance in LDG is more accurate and respected. This represents the main drawback of FENNEL, as the balance criterion in the graph partitioning problem is extremely important and can cause the computations to slow down as not all the machine will have the same workload.

2.3.2.3 Fractional Greedy heuristic

Another greedy heuristic named Fractional Greedy (FG) for streaming partitioning was proposed in [68]. It aims to enhance the partition quality and yields exactly balanced partitions due to a special penalizing term. The Fractional Greedy heuristic was proposed and developed in the context of this thesis and will be further detailed in Chapter 3.

2.3.3 Restreaming Partitioning

In order to get better partitions, Nishimura *et al.* [69] relaxed the constraint of a single pass over the graph stream and introduced the restreaming graph partitioning as a variant of the original problem which allows for multiple passes over the graph stream. Indeed, the restreaming model enhances the partition quality by reducing the cut and emphasizing the balance.

The restreaming process consists in performing several passes over the data stream. In other words, the first streaming partitioning is done, and a second streaming partitioning is repeated by exploring the graph stream for the second time; and results of the previous partitioning is stored and used in the current partitioning to improve and enhance the final partition quality.

Performances achieved by restreaming methods compete with METIS, a well-known offline method for graph partitioning. However, if restreaming partitioning leads to exact balance - guaranteed for FENNEL - it can be too expensive for partitioning massive

graphs, since it must restream the whole graph several times which incurs a high computational time and memory. Thus, the tradeoff is to be done between computational cost and desired partitions quality.

2.3.3.1 Restreaming the one-pass streaming partitioning heurtistics

The authors took the two streaming heuristics LDG and FENNEL and tweaked them in order to perform a restreaming partitioning. In the following, we give the objective functions for LDG and FENNEL in the restreaming setting:

$$ReLDG = \underset{i}{argmax}\ (P_i^{t-1} \cap N(v)) * (1 - \frac{|P_i^t|}{C}) \tag{2.4}$$

$$ReFennel = \underset{i}{argmax}\ (P_i^{t-1} \cap N(v)) - \alpha\gamma(|P_i^t|)^{\gamma-1} \tag{2.5}$$

In both aforementioned equations for restreaming LDG and FENNEL respectively, only the first term is different from the original objective function in the one-pass setting. In the one-pass setting, this term computes the number of neighbors of the current vertex in part (machine) of index i, but in the restreaming setting, the term computes the number of neighbors in the previously obtained partition. This can be viewed as a learning process, where the previously obtained partitioning is exploited in order to enhance and improve the actual partitioning. The second term of the objective function, also called the penalizing term, is unchangeable and it is relative to the current part (machine) at iteration t.

Although the restreaming partitioning has the strong point of refining the partitions quality, by reducing the edge cut and improving the balance among parts, it actually suffers an important weakness, such that it incurs a high cost as a considerable memory space is needed in order to store information about the partitioning iterations, as well as the computational time being consumed.

Partial Restreaming Partitioning. In order to alleviate the cost incurred by the whole graph restreaming setting, we have proposed and developed in the context of this thesis another variant of the problem named the partial restreaming partitioning [68, 70], where several passes over the graph stream are allowed on a portion of the graph, and the rest of the graph stream is processed in one pass. This work will be further detailed and explained in chapter 3.

Summary. The actual data proliferation is correlated to a sheer increase in size of actual graph datasets, which impels the use of distributed graph processing frameworks

that should consider a good partitioning of the graph dataset, for their performances to be enhanced. In considering this matter, the graph partitioning task turns out to be crucial and important and as a result, recent research in the field is focused on big graph adapted methods. For this reason, the streaming graph partitioning problem was introduced to alleviate the excessive cost being incurred by traditional graph partitioning methods, which are well suited for relatively small graphs.

In this section, we defined the problem of Streaming graph partitioning. We highlighted the streaming setting, and discussed the main advantages of such streaming model. We presented and reviewed recent and leading studies on the Streaming Graph Partitioning problem. We discussed the one-pass streaming heuristics for graph partitioning. Besides, we presented the restreaming partitioning model as a way to improving the partitioning quality given by the one-pass streaming partitioning.

2.4 Chapter Summary

In this chapter, we give an overview of the graph partitioning problem, including definitions of related key notions and reviews on foremost works in the literature as well. We also present different variants of the problem, and present the hardness results and approximation guarantees that were proposed so far. Moreover, we present two main classes of solving methods for the graph partitioning problem: the offline and the online partitioning methods. The offline partitioning is the traditional class of methods, and it is suitable for processing relatively small graphs, since the methods under this class are known to be computationally expensive. The online partitioning was recently proposed, and it exhibits the possibility to handle actual large graphs in a streaming fashion, *i.e.*, the graph to be partitioned, usually of massive size, does not have to be memory resident. This feature makes the online partitioning very efficient to partition large graphs as it does not incur any cost in memory consumption. Moreover, online methods are equally known for being faster than the offline methods for graph partitioning.

Chapter 3

Fractional Greedy Heuristic & Partial Restreaming Partitioning

The graph partitioning problem is a well studied problem with an active present and a rich history. The broadness of this research field witnesses the importance of the graph partitioning problem in a wide range of application domains. In this chapter, we present two of our contributions to the graph partitioning problem, named the *Fractional Greedy* heuristic and the *Partial Restreaming* partitioning model. The Fractional Greedy heuristic was proposed in the context of this thesis to address the shortcoming of partition balance in other state-of-the-art heuristics and also to improve the edge-cut among partitions. On the other hand, the Partial Restreaming partitioning model reduces the computational cost that is incurred in full restreaming partitioning settings while achieving competitive performances.

The chapter is organized as follows: in Section 1, we start by presenting our proposed Fractional Greedy heuristic. In Subsection 1, we introduce and give definitions of key concepts in the *streaming graph partitioning problem*, we also outline the motivation behind the proposal of a new heuristic for streaming graph partitioning such Fractional Greedy. Subsection 2 presents the Fractional Greedy (FG) heuristic, we first describe and explain the objective function of the heuristic, then we describe the FG algorithm. Finally, Subsection 3 presents the experimental evaluation and discusses the obtained results.

In Section 2, we present the *Partial Restreaming Partitioning model*. We first start by defining the restreaming model, and then we define and present the *Partial Restreaming Partitioning model* in its two proposed versions: the simple partial restreaming and the selective partial restreaming models. The same section continues by presenting the

experimental evaluation of the partial restreaming model. Finally, we summarize the chapter in Section 3.

3.1 Fractional Greedy: a proposed streaming partitioning heuristic for large graphs

In this section, we present our proposed method called Fractional Greedy (FG). First, we introduce the streaming model concept and reveal the motivation behind the proposal of such heuristic. Second, we present the actual FG heuristic by presenting the objective function used, and the whole FG algorithm. Last but not least, we present our experimental evaluation of FG over numerous graph datasets and compare it with state-of-the-art heuristics for streaming graph partitioning.

3.1.1 Preliminaries

The Fractional Greedy (FG) heuristic is a streaming graph partitioning method intended to process large graphs in a streaming fashion. Therefore, we begin by briefly explaining the streaming model used in such methods. Then, we give the context related to our proposed FG method and give the motivation behind it.

Notations. We will be using the following notation throughout the chapter. We consider a simple undirected graph $G = (V, E)$, let$|V| = n$ be the number of vertices in G and $|E| = m$ be the number of edges of G. Let the current vertex loaded be v, and $N(v)$ represents his neighbors. k is the number of clusters or parts we wish to divide the graph to. Let $P^t = (P_1^t, \ldots, P_k^t)$ be a partition of the graph. $P_i, \cdots, P_k$ are called clusters such that $P_i \subseteq V$ and $P_i \cap P_j = \emptyset$ for every $i \neq j$.

Let $e(P, V - P)$ be the set of edges with ends belonging to different clusters. We define $\lambda = |e(P, V - P)|/m$ as the fraction of edge cut and it should be minimized during partitioning. We define ρ as the maximum load normalized, it expresses the balance between clusters' size and we have $\rho = \frac{maximumload}{n/k}$, $maximumload$ representing the size of the biggest cluster. Each part P_i is of size C. In our work we set $C = \lceil n/k \rceil$.

3.1.1.1 Streaming Model

Streaming graph partitioning was first introduced by Stanton and Kliot [65]. In this work, the authors presented the streaming model used such that it consists on processing the graph stream in one pass, where the graph vertices arrive in a certain order

(Random, Breadth First Search, Depth First Search), and each vertex is accompanied by its adjacency list. In figure 3.1, we depict the streaming model: it consists on a graph stream, a partitioner, and k machines, provided that the goal is to partition the streamed graph into k parts.

Partitioner. When a vertex is loaded (along with its adjacency list), a program called the partitioner decides on which cluster the vertex will be placed and never relocates it once the assignment is done. The partitioning process is done on the fly without having an access to the whole graph, *i.e.*, the graph does not have to be memory resident. In fact, streaming partitioning uses very limited computational resources in terms of memory and computation time. This property makes the streaming model very suitable when it comes to partitioning billion node graphs.

A natural and evident constraint for such streaming models, is that if the size of the streamed graph is n, then the whole capacity of the k machines, denoted by $k \times C$ (where C is the maximal capacity of one machine), should be greater or equal to n for the partitioning process to be feasible.

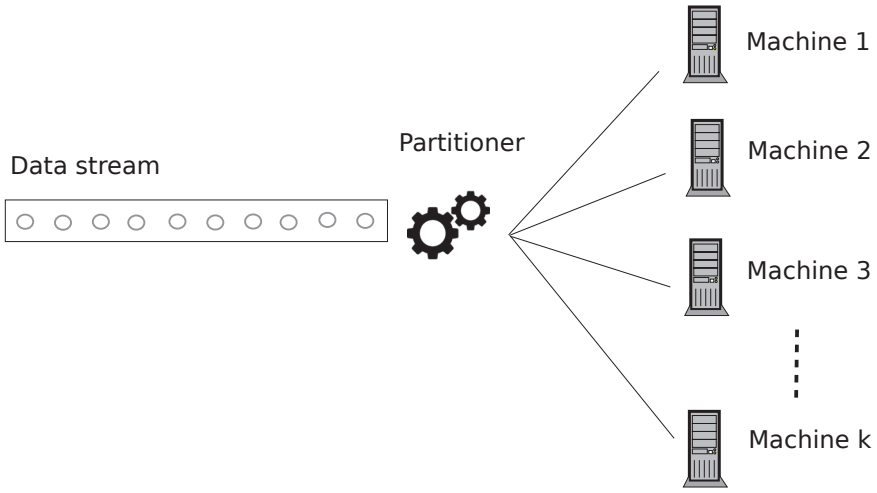


FIGURE 3.1: Streaming model used in the streaming graph partitioning problem.

3.1.1.2 Motivation

The streaming graph partitioning is a fair alternative for partitioning large graphs, the main reason is that it considerably lowers the computational cost when compared to the traditional offline setting for graph partitioning, *i.e.*, the whole graph has to be memory resident in order to perform the partitioning. Since introduced in [65], the streaming partitioning won the interest of many researchers. We cite the Linear Deterministic Greedy LDG heuristic [65], the FENNEL heuristic [67] and the restreaming partitioning

model [69]. The LDG heuristic is known to yield good quality partitions, where a good quality partition has a low edge cut and is exactly balanced. Although LDG achieves exact balance in partitions, it, however, incurs an important edge cut. FENNEL, achieves a lower edge cut compared to LDG, but at the same time, it allows for a deviation in the partition balance, which could be very problematic and present a potential cause for computations to slow down, as not all the machine are assigned the same workload. Taken this into consideration, we proposed the Fractional Greedy heuristic, that achieves lower edge cut and exact balance among partitions. We will detail in the following how FG outperforms the state-of-the-art heuristics due to its special objective function.

3.1.2 Fractional Greedy Heuristic

Used in the context of a streaming model, the FG heuristic partitions the graph by processing each vertex in the stream in a sequential manner. The processing of each vertex consists in finding the target machine, *i.e.* the machine to which the vertex will be assigned. In other words, FG computes the index of the target machine using limited information access, which are the adjacency list of the vertex, and the indexes of vertices of each one of the k parts.

3.1.2.1 Objective function

Our proposed heuristic is designed for the constrained graph partitioning problem, which aims to balance the size of the parts as a priority, then to minimize the crossing edges between parts [71]. Formally, the Fractional Greedy heuristic maximizes the following objective function:

$$f(v, P_i^t) = (P_i^t \cap N(v)) - g(P_i^t) \tag{3.1}$$

In other words, for each vertex loaded, denoted by v, we compute the index ind of the part (or the cluster) as follows:

$$ind = \underset{i}{argmax}\ (P_i^t \cap N(v)) - g(P_i^t) \tag{3.2}$$

The objective function f has two components: the first component computes the intra-parts edges, and the second g computes the penalization cost with regard to the size of the part. The main idea is to assign the loaded vertex v to a part that contains the most of its neighbors and that does not attain the maximal capacity. The cost function g is computed as follows:

$$g(P_i^t) = \frac{1}{1 - \frac{|P_i^t|}{C}} \tag{3.3}$$

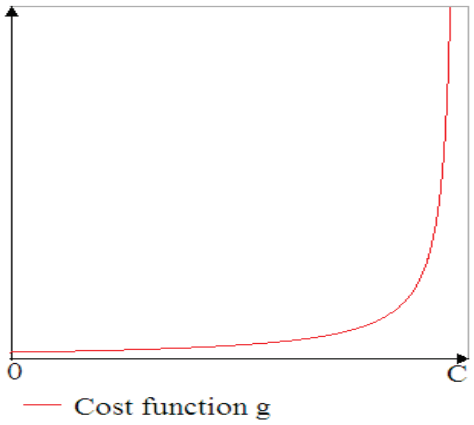


FIGURE 3.2: Cost function g. The x-axis represents the size of a part and the y-axis represents the cost value.

The cost function g penalizes the parts of large sizes. For a given vertex v, the objective function f is computed for each part or machine (among k ones), if the parts/machines that have the most neighbors of v have reached the maximal capacity C, then this part/machine should not be selected. In other words, the corresponding value of the objective function should not be maximal. For that reason, we designed the cost function of FG such that it rapidly penalizes large size parts even though they may contain the most neighbors of the vertex v.

As seen in figure 3.2, the cost function increases gradually as the part's size increases. When the size is high and reaches C, the cost is reaching infinity making sure that parts with size C will never be assigned additional vertices.

3.1.2.2 Fractional Greedy algorithm

We describe the Fractional Greedy algorithm in Algorithm 1. The input parameters are the number of parts k, the graph stream, the maximal capacity of each machine/part denoted by C. Initially, the parts are empty. Then, FG begins the processing of the streaming. It starts by sequentially loading vertices from the stream, such that when a vertex v is loaded along with its adjacency list $N(v)$, the objective function f is computed for every part. Then, we compute the index ind of the machine to which the vertex v will be assigned. The index ind is the index of the part/machine that maximizes the function f (line 6 in Algorithm 1). Then the partition is updated by adding the vertex v to the selected part/machine. The same process is repeated for each vertex in the stream. Ultimately, FG outputs the final partition $P = (P_1, P_2, \cdots, P_k)$.

Algorithm 1 *Fractional Greedy*

Input: k machines represented by k parts P_i^t at time t and $i \in [k]$, the maximal capacity of each machine C, the graph stream represented sequentially by vertex v and its adjacency list $N(v)$.
Output: Partition $P = (P_1, P_2, \cdots, P_k)$.

1. Initialization: $t = 0; P_1 = P_2 = \cdots = P_k = \emptyset$;
2. **for each** v **in the stream**
3. **for each part** P_i^t
4. compute $f(v, P_i^t) = (P_i^t \cap N(v)) - \frac{1}{1-\frac{|P_i^t|}{C}}$;
5. **end for**
6. Select the index ind of the appropriate machine : $ind = \underset{i}{argmax}\ f(v, P_i^t)$;
7. Assign v to P_{ind}: $P_{ind} = P_{ind} \bigcup v$;
8. t++;
9. **end for**
10. **return** P;

3.1.3 Experimental Evaluation

In this section, we present our experimental evaluation of the FG heuristic. First, we present the experimental set up, *i.e.*, the graph datasets used and the evaluation methodology as well. Second, we dicuss the obtained results.

3.1.3.1 Experimental Set up

1. Evaluation datasets:

   Two types of graph datasets were used: web and social. We tested our methods on ten graph datasets listed in Table 3.1, all obtained from the SNAP repository [72]. Vertices with 0 degree and self-loops were removed. All the graphs were made undirected by reciprocating the edges. Graph datasets were chosen in order to be small enough so that we can find offline solutions with METIS and still big enough to capture the behavior of the online heuristics.

2. Methodology:

   We first run FG, LDG and Fennel in one pass stream setting on our graph datasets for $k = 40$ in order to compare the fraction of edge cut represented by λ and the balance represented by ρ. For $Fennel$, the parameter γ was empirically set to 5 in

TABLE 3.1: Graph Datasets used for our tests.

Graph	$\|N\|$	$\|M\|$	**Avgdeg**	**type**
wikivote	7115	100762	14.16	social
enron	36692	183831	5.01	social
Astro ph	18771	198050	10.55	social
slashdot	77360	469180	6.06	social
Web nd	325729	1090108	3.34	web
stanford	281903	1992636	7.06	web
Web google	875713	4322053	4.93	web
Web berkstan	685230	6649470	9.7	web
Live journal	4846609	42851237	8.84	social
orkut	3072441	117185085	38.14	social

order to give as balanced parts as those given by FG and LDG. We compare FG to the other online heuristics LDG and Fennel and to the offline METIS heuristic which is used as a baseline.

3.1.3.2 Experimental results

Obtained results show that Fractional Greedy outperforms LDG and Fennel in most cases with exactly balanced clusters and lower edge cut.

TABLE 3.2: Fraction of edge cut λ and maximum load normalized ρ for 3 streaming heuristics LDG and $FENNEL$ and FG and METIS,(1.001) indicates that the slackness allowed is 001. Results are obtained for 10 graph datasets where $k = 40$.

Graphs	FG		LDG		Fennel		Metis(1.001)	
	λ	ρ	λ	ρ	λ	ρ	λ	ρ
wikivote	0.844	1	0.867	1	0.862	1	0.822	1.001
enron	0.589	1	0.610	1	0.612	1	0.855	1.001
astro ph	0.555	1	0.619	1	0.578	1	0.535	1.001
slashdot	0.758	1	0.787	1	0.777	1	0.711	1.001
webnd	0.249	1	0.261	1	0.270	1	0.036	1.001
stanford	0.349	1	0.392	1	0.347	1.043	0.123	1.001
webgoogle	0.310	1	0.308	1	0.313	1.023	0.009	1.001
web berkstan	0.386	1	0.342	1	0.367	1.023	0.117	1.001
live journal	0.442	1	0.462	1	0.546	1.009	0.309	1.001
orkut	0.627	1	0.639	1	0.696	1.076	0.376	1.001

Performance discussion. Our proposed heuristic Fractional Greedy outperforms LDG and Fennel in terms of balance and also of edge cut. In Table 3.2 we show our results, we see that FG yields partitions of exact balance ($\rho = 1$) and lower edge cut. However, LDG outperforms FG in web-google (0.310 vs 0.308) and web-berkstan (0.368 vs 0.342). Our proposed heuristic is the most adapted for the setting of constrained graph partitioning, it yields exact balanced partitions and also minimizes the edge cut.

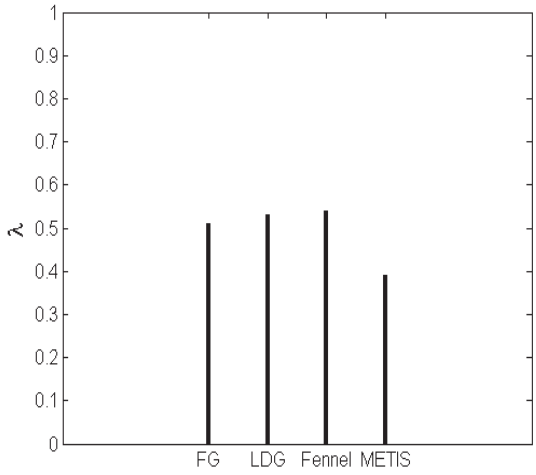


FIGURE 3.3: Average fraction of edge cut λ for $FG, LDG, Fennel$ and $METIS$ over ten graph datasets.

In figure 3.3 we show average results of comparison between $FG, LDG, FENNEL$ and $METIS$ in terms of fraction of edge cut over 10 graph datasets. METIS is the best performing heuristic due to the offline setting, while FG is the second best heuristic outperforming LDG and Fennel.

3.2 Partial Restreaming Partitioning

In this section, we introduce the *partial restreaming partitioning*. First, we give a brief introduction to the *restreaming partitioning model*. Then, we present the partial restreaming model used. Afterward, we present the instance of the partial restreaming partitioning model using Linear Deterministic Greedy LDG [65], Fennel [67] and Fractional Greedy [68] as heuristics for partitioning. Finally, we give another variant of the partial restreaming, called *Selective Partial Restreaming* partitioning, where portions to be restreamed are selected depending on their degree and density.

Notations. The following notations are used in this section, in addition to notations in Section 3.1.1. Let s be the number of the streaming iterations. $P^{t-1} = (P_1^{t-1}, \cdots P_k^{t-1})$ represents the partition obtained from the precedent stream iteration. C represents also the size of the portion to be restreamed (C represents also the maximal capacity of a part/machine). We can choose that several portions of the graph should be restreamed, then β is the number of portions to be restreamed (each portion is of size C).

3.2.1 Restreaming Partitioning: a brief introduction

In order to get better partitions, Nishimura *et al.* [69] relaxed the constraint of a single pass over the graph stream and introduced the restreaming graph partitioning as a variant of the original problem, which allows for multiple passes over the graph stream.

The restreaming process consists in performing several passes over the data stream. In other words, the first streaming partitioning is done, and a second streaming partitioning is repeated by exploring the graph stream for the second time; and results of the previous partitioning is stored and used in the current partitioning to improve and enhance the final partition quality. Indeed, according to experimental results, the restreaming model enhances the partition quality by reducing the cut and emphasizing the balance. The restreaming partitioning is further defined in Chapter 2, Section 2.3.3.

3.2.2 Simple Partial Restreaming Partitioning

In this section, we present the *Simple Partial Restreaming Partitioning*. First, we present the partial restreaming model, then, we present and explain the simple partial restreaming setting.

Partial Restreaming Model. We consider a simple streaming model as described in subsection 3.1.1.1, where vertices arrive in a random order along with their adjacency lists. The heuristic used for partitioning must make a decision about the machine/part to place the current vertex within. In the partial restreaming model, two major phases exist: the restreaming phase and the one pass streaming phase. Namely, a first loaded portion of the graph dataset of size $\beta \times C$ is going to be restreamed and the rest is going to be processed in the simple streaming phase. In other words, in this model, multi-passes of the stream is allowed for only a part of the dataset. Let P^t be the partition obtained at time t, P^{t-1} represents the partition obtained at the precedent iteration of the restream. When we attain the number of restreaming iterations allowed for the portion concerned, we continue streaming the rest of the graph dataset normally, such that we don't use information about the last partitioning to build a new one. In the following, we describe the partitioning heuristics used in our model.

Partial Restreaming Partitioning. As a partitioning strategy, we use the state-of-the-art heuristics [65, 67] in order to compare them with our proposed heuristic FG. In this section, we describe how we adapt these heuristics (LDG, FENNEL, FG) to our partial restreaming model.

1. Linear Deterministic Greedy

   $LinearDeterministicGreedyLDG$ is the best performing heuristic in [65]. It greedily assigns the vertices to clusters while adding a penalization for big clusters to emphasize balance. LDG assigns vertices to clusters that maximize:

$$LDG = \underset{i}{argmax}\ (P_i^t \cap N(v)) * (1 - \frac{|P_i^t|}{C}) \tag{3.4}$$

   In our streaming model, we consider 2 phases: the restreaming phase and the simple streaming phase, which consists in one pass. In the first phase, the LDG function will use information about the last partitioning to decide about the placement of the current vertex such that:

$$PartLDG = \underset{i}{argmax}\ (P_i^{t-1} \cap N(v)) * (1 - \frac{|P_i^t|}{C}) \tag{3.5}$$

   Where $PartLDG$ makes a reference to partial LDG for partially restreaming LDG. In the second phase (the one pass streaming phase) the vertices are assigned following the LDG function as follows:

$$PartLDG = \underset{i}{argmax}\ (P_i^t \cap N(v)) * (1 - \frac{|P_i^t|}{C}) \tag{3.6}$$

2. Fennel

   $Fennel$ yields partitions of good quality compared to LDG, with a lower fraction of edge cut and also emphasizes balance [67]. We adapt Fennel function to the two phases of our streaming model. In the first restreaming phase, $PartFennel$ is as follows:

$$PartFennel = \underset{i}{argmax}\ (P_i^{t-1} \cap N(v)) - \alpha\gamma|P_i^t|^{\gamma-1} \tag{3.7}$$

   Where $PartLDG$ makes a reference to partial $Fennel$ for partially restreaming $Fennel$. While in the second phase, $PartFennel$ is as follows:

$$PartFennel = \underset{i}{argmax}\ (P_i^t \cap N(v)) - \alpha\gamma|P_i^t|^{\gamma-1} \tag{3.8}$$

   The parameter setting of γ will be presented in Section 4.

3. Fractional Greedy [1]

   The adaptation of FG to the partial restreaming model is as follows:

[1] Refer to Section 3.1 for FG description.

In the restreaming phase, PartFG corresponds to:

$$PartFG = \underset{i}{argmax}\ (P_i^{t-1} \cap N(v)) - \frac{1}{1 - \frac{|P_i^t|}{C}} \tag{3.9}$$

And in the second phase, PartFG is defined as:

$$PartFG = \underset{i}{argmax}\ (P_i^{t} \cap N(v)) - \frac{1}{1 - \frac{|P_i^t|}{C}} \tag{3.10}$$

3.2.3 Selective Partial Restreaming Partitioning

Instead of restreaming the first loaded portion of the graph dataset, we try to select portions of size C that would lead to a good quality partitioning. We set degree and density parameters to be the criteria for selecting portions to be restreamed in the first phase of the model. In other words, when a portion is loaded, we check its average degree and its average density, if it is higher than the average degree (respectively average density) of the whole graph, we select this portion for restreaming, otherwise, it will be processed in the second phase of one-pass streaming.

selection criteria. In order to select a portion for restreaming, two criteria are considered:

1. Average degree: The average degree of a portion is the average of vertices degrees inside the portion. Notice that the degree of a vertex is the number of its neighbors no matter they are inside or outside the portion concerned. We take the average degree as a criterion to make sure that the portion which is going to decide for the partitioning of the graph must influence the partitioning decision of a large number of vertices due to its elevated average degree.

2. Average density: The average density of a portion represents the number of edges within the portion. It is important to have edges inside the portion to make better partitioning decision as the objective functions used makes decisions depending on edges, otherwise, the partitioning of the portion will be done at random and it will definitely lead to a lower partitioning quality of the whole graph.

The average degree and density of the whole graph is an information which is not always available, we can substitute this by progressively adding degree and density information as the data is loaded. A portion with high density and high degree vertices should act like a kernel to yield partitions of good quality. In fact, portions with vertices having high degree would attract and influence the partitioning of a large number of other vertices, and the density criterion inside the portion makes sure to take into consideration the

edges to make better decisions for the partitioning. In our experimental evaluation, we show that by selecting portions of high degree average and high-density average we obtain partitions with better quality than those obtained by simply restreaming the first loaded portion.

3.2.4 Experimental Evaluation

In this section, we present the experimental evaluation of the partial restreaming partitioning model. First, we present the experimental set up, *i.e.*, the graph datasets used and the evaluation methodology as well. Second, we discuss the obtained results.

3.2.4.1 Experimental Set up

1. Datasets:

   Graph datasets used for the following experimentations are the same as the ones used in subsection 3.1.3, and are all listed in Table 3.1 .

2. Methodology:

   We vary the restreamed portion size to assess an eventual correlation between the restreamed portion size and the cut. In other words, we examine results of edge cut for different values of β, provided that the portion size is $beta \times C$. We begin by running $PartLDG$ and $PartFennel$ and $PartFG$ (partially restreaming LDG, $Fennel$ and FG respectively) on WebGoogle and LiveJournal for different values of β and see how the fraction of edge cut reacts to the change in β. After that, we run our methods on the ten graph datasets, for $k = 40$, $s = 10$ and $\beta = k/2$. Notice that the ordering of vertices is done at random. $\beta = k/2$ means that we are restreaming half of the graph.
   We compare our results to the whole graph restreaming methods [69] and to METIS, which represents the offline methods and considered as our baseline. Afterwards, we evaluate the partial restreaming methods ($PartLDG, PartFennel$) and the partial selective restreaming methods ($PartSLDG, PartSFennel$) on seven graphs. Last but not least, we show the running time gain for the partial methods (PartLDG, PartFennel and PartFG) over the ten graphs for $k = 40$ and $s = 10$.

The Runtime gain is computed as follows:

$$Gain_{PartLDG} = \frac{ReLDG - PartLDG}{ReLDG - LDG}$$

Where $ReLDG$ refers to the execution time of the version of LDG where the whole graph is restreamed, LDG is the one pass streaming version.

$$Gain_{PartFennel} = \frac{ReFENNEL - PartFENNEL}{ReFENNEL - FENNEL}$$

Same as $Gain_{PartLDG}$, $ReFENNEL$ refers to the execution time of the version of $FENNEL$, where the whole graph is restreamed and $FENNEL$ is the one pass streaming version.

$$Gain_{PartFG} = \frac{ReFG - PartFG}{ReFG - FG}$$

$Gain_{PartFG}$ is the runtime gain for Fractional Greedy, where $ReFG$ refers to restreaming FG and FG is the one pass streaming version. $PartFG$ is the partial restreaming version of FG. We note that the runtime computed includes solely the partitioning runtime.

3.2.4.2 Experimental results

In the following, we present and discuss our results. And before we delve in the results we give a brief summary about it.

Summary of our results

- Results that were obtained show that by augmenting β, *i.e.*, by augmenting the size of the portion to be restreamed, we obtain better quality partitions.
- By restreaming the first loaded half of a graph, we obtain partitions of similar quality than those yielded by restreaming the whole graph.
- The partial selective restreaming method preserves the quality of a partition, unlike the partial methods which depend on the stream order. The reason is that no matter the order, the selective methods always pick the good portions to be restreamed.
- Restreaming only a portion of a graph dataset incurs lower computational cost than restreaming the whole graph, for example, restreaming only the half of the graph takes half of the runtime taken by restreaming the whole graph, while the results are almost the same.

Performance discussion. In Figure 3.4, we see that the bigger the size of the streamed portion β is, the lower the fraction of edge cut. This shows that the information about the precedent stream iteration allows vertices to be more oriented toward the best cluster leading to lower edge cut. In other words, the more information we have about the precedent streaming iteration the better is the edge cut.

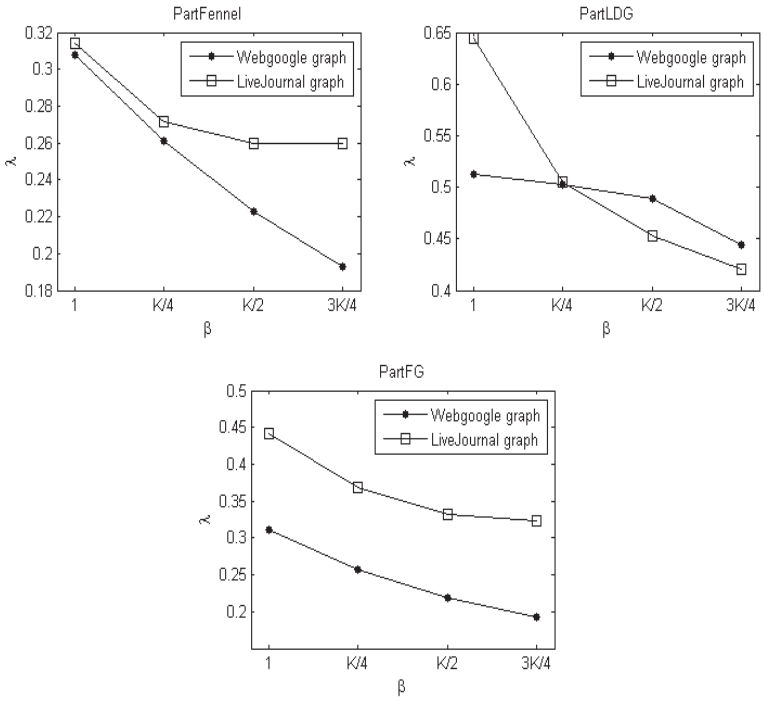


FIGURE 3.4: Variation of λ with the growing size of the portion being restreamed represented by β for Webgoogle graph and LiveJournal. On the left *PartFennel* and on the right *PartLDG* and at the bottom *PartFG*.

TABLE 3.3: Comparison of the fraction of edge cut λ and the normalized maximum load ρ for ReFG and PartFG, ReLDG and PartReLDG, ReFennel and PartFennel. $k = 40$ and $s = 10$ and $\beta = \frac{k}{2}$.

Graph	ReFG		PartFG		ReLDG		PartLDG		ReFENNEL		PartFENNEL	
	λ	ρ	λ	ρ	λ	ρ	λ	ρ	λ	ρ	λ	ρ
wikivote	0.812	1	0.828	1	0.835	1	0.850	1	0.813	1.023	0.826	1.022
enron	0.479	1	0.509	1	0.475	1	0.507	1	0.476	1.098	0.482	1.087
astro ph	0.433	1	0.475	1	0.418	1	0.501	1	0.413	1.019	0.443	1.019
slashdot	0.711	1	0.705	1	0.713	1	0.722	1	0.703	1.041	0.692	1.106
webnd	0.164	1	0.207	1	0.113	1	0.207	1	0.143	1.048	0.193	1.056
stanford	0.200	1	0.267	1	0.204	1	0.319	1	0.193	1.025	0.216	1.109
webgoogle	0.163	1	0.219	1	0.161	1	0.217	1	0.160	1.087	0.222	1.012
web berkstan	0.241	1	0.276	1	0.212	1	0.276	1	0.254	1.037	0.282	1.073
live journal	0.325	1	0.331	1	0.313	1	0.331	1	0.330	1.006	0.319	1.018
orkut	0.398	1	0.503	1	0.395	1	0.503	1	0.410	1.005	0.451	1.017

Table 3.3 shows results in terms of fraction of edge cut and balance. The difference between edge cut of fully restreaming methods and partial restreaming methods is minimal in most cases.

Table 3.4 shows the difference between performances of partial restreaming methods and selective partial methods on seven different datasets. The results show that in most cases, selective partial restreaming methods yield better quality partitions by leading to a lower edge cut. However, in some datasets, portions that conform to the criteria do

TABLE 3.4: Comparison of the fraction of edge cut λ and the normalized maximum load ρ for Partial restreaming methods and Selective partial restreaming methods *PartFG vs PSelectFG*, *PartLDG vs PSelectLDG*, *PartFennel vs PSelectFennel*. Results are obtained for 7 graph datasets where $k = 40$.

Graphs	PSelectFG		PartFG		PSelectLDG		PartLDG		PSelectFENNEL		PartFENNEL	
	λ	ρ	λ	ρ	λ	ρ	λ	ρ	λ	ρ	λ	ρ
wikivote	0.826	1	0.828	1	0.849	1	0.850	1	0.845	1.005	0.826	1.022
enron	0.503	1	0.509	1	0.502	1	0.507	1	0.509	1.003	0.482	1.008
astro ph	0.475	1	0.475	1	0.501	1	0.501	1	0.468	1.008	0.443	1.019
slashdot	0.703	1	0.705	1	0.722	1	0.722	1	0.716	1.011	0.692	1.106
webnd	0.213	1	0.207	1	0.214	1	0.207	1	0.217	1.013	0.193	1.056
stanford	0.271	1	0.267	1	0.339	1	0.319	1	0.258	1.024	0.216	1.109
webgoogle	0.189	1	0.219	1	0.188	1	0.217	1	0.187	1.009	0.222	1.012

TABLE 3.5: Runtime gain computed for *PartLDG* and *PartFennel* and *PartFG* over executions on ten graphs with $k = 40$ and $s = 10$.

Graphs	**PartLDG Gain**	**PartFennel Gain**	**PartFG Gain**
Wikivote	47.2%	58.5%	45.9%
Enron	50%	48.3%	47.6%
Astro ph	44.5%	44%	47.4%
Slashdot	39.6%	47.5%	45.2%
Web nd	51.4%	49.5%	56.2%
Stanford	54.5%	52.4%	56.8%
Web google	57.5%	52.7%	56.1%
Web berkstan	50.6%	49.6%	65%
Live journal	40.5%	45.6%	42.1%
Orkut	52.7%	51.2%	39.9%

not exist, which hinders the selective partial method to give the best results. For this reason, the simple partial restreaming method could be a better alternative for both speeding up computations and imporving partitions quality.

The Runtime gain is represented in Table 3.5. In all our graph datasets, PartFennel has an average gain of 49.93%, PartLDG 48.85% and PartFG 50.26%. It shows that the partial restreaming model reduces the runtime by half.

3.3 Chapter summary

In this chapter, we discuss our proposed methods for the balanced graph partitioning problem. Specifically, we introduce a new online heuristic for streaming partitioning and we show that we improve the partitions quality by partially restreaming several portions of the graph. We showed that our proposed heuristic produces partitions of significantly enhanced quality in terms of balance and edge cut: partitions that were produced are exactly balanced and have a lower edge cut.

We also introduced the partial restreaming graph partitioning, that aims to restream several portions of the graph in order to improve the partition quality. We evaluate our proposed methods on ten different graph datasets and compare it with the state-of-the-art partitioning heuristics (Fennel, LDG and METIS). We showed that our proposed partial restreaming methods produce partitions of similar quality than those produced by fully restreaming methods with the advantage of incurring lower runtime and memory cost.

We also present another variant of the partial restreaming partitioning, called selective partial restreaming partitioning, and showed that selecting relevant portions of the graph using degree and density as selection criteria, does improve the quality of the partitions.

Chapter 4

Streaming METIS Partitioning: an Online version of METIS Heuristic for Big Graph Partitioning

Graph datasets being generated and stored nowadays consist of several terabytes which makes the task of processing and even storing them so challenging. Eventhough there exist efficient tools for graph partitioning, they fall outdated when facing the big data issue. Thus, there is a pressing need to come with and design new tools that would achieve simultaneously effectiveness and efficiency while handling this kind of graph datasets. In this chapter, we present an online alternative, called Streaming METIS Partitioning (SMP), to the offline METIS heuristic for graph partitioning, known to be fast and effective. We show theorectically and experimentally that SMP yields results of similar quality to the original METIS, whereas the time complexity of SMP is lower than for METIS, leading to SMP being faster than METIS.

This chapter is organized as follows: Section 1 introduces and gives general context for the SMP approach. We highlight the METIS algorithm in Section 2, and in Section 3 we introduce the Streaming METIS Partitioning approach. Section 4 shows the experimental evaluation of SMP, and finally, Section 5 gives a chapter summary.

4.1 Introduction & Context

The last two decades have witnessed a sheer increase in the size of graph datasets that emerge in different applications such as social networks, mobile applications to name but a few. For example, the world wide web consists of at least one trillion of hyperlinks [8], The well known social network Facebook amounts to 1.083 billion of daily active users [6] and Twitter totals up to 305 millions of users [7].

This graph datasets plethora has directed many researchers into creating new frameworks for big graph processing: Pregel [11] and its open-source version Giraph [12], Graphlab [9] and Microsoft's Trinity [10] to name a few. The aforementioned platforms perform a partitioning of the graph as a pre-processing step where the graph dataset is distributed across several machines, and then parallel computations are run on each machine.

Even though these graph processing platforms seem to be considerably efficient, they use a hash function to partition the graph, *i.e.*, the graph nodes are assigned to the machines uniformly at random, which leads to a perfectly balanced partitions but incurs an important inter-machine communication overhead that causes the computations to slow down.

The graph partitioning problem addresses this issue, having the main goal of dividing the graph data into several distinct sets of equal size with the constraint of minimizing the number of edges crossing these sets.

In fact, the reason of having these two constraints (balance and minimum edge cut) is that: *(i)* balancing the partition (parts of equal size) ensures that each machine is given the same workload, and *(ii)* the minimized number of crossing edges reduces the network overhead which ultimately makes the graph partitioning an essential pre-processing step in order for the upcoming computations to speed up.

What is the difference between *parts* and *partitions*? It is worth mentioning and clearing up the difference between parts and partitions. A partition is the set of parts which are subsets of the graph vertices; at the end of a partitioning task, we obtain a partition of the graph consisting of several parts.

In its formal form, the graph partitioning problem is known as k-way balanced graph partitioning and is an NP-hard problem [15]. It is a problem that has been well studied and has a rich history, and though many works and heuristics have been proposed in the literature, a wide range is cumbersome and not tailored to process large scale graphs.

Nevertheless, a new variant of graph partitioning which enables fast and efficient big graph partitioning was introduced recently and is known as *streaming graph partitioning (SGP)*.

Having this motivation in mind, we proposed in the context of this thesis, a novel streaming heuristic for graph partitioning, termed Streaming METIS Partitioning (SMP) [73]. and based on an adaptation of the well known offline METIS [2]. The new heuristic extends METIS to the online setting and brings significant enhancements to the streaming graph partitioning.

On the one hand, SMP adapts a high quality partitioning heuristic to an online setting to reduce the computational cost on large scale graphs, while achieving near-optimal quality partitioning.

On the other hand, SMP performs graph partitioning using METIS, known to be one of the most powerful offline partitioning methods as seen in a number of recent experimental studies over a wide range of graph datasets ([65, 67, 68]).

In a nutshell, the main idea of SMP is to apply the METIS partitioning method on small subgraphs in order to reduce computations and fit to the limitations of memory capacity on the partitioner, and also, to extend a powerful offline partitioning method to a steaming setting while ensuring simultaneously a balanced partition along with a minimal edge cut.

The use of METIS algorithm promotes the quality of the partition (*i.e.* number of crossing edges) and leads to minimal crossing edges within the partition. And to ensure balance within the partition, the size of parts is controlled through a multinode weight update strategy that controls the partition growth. Besides, we show through a complexity analysis of the new SMP approach, that its computational time complexity is lower than METIS, and it maintains competitive performances compared to offline METIS. Last but not least, we conducted extensive experiments on various benchmark graph datasets, and the obtained results demonstrate that the SMP method is enjoying considerable advantages compared to state-of-the-art methods [65, 67, 68].

4.2 METIS: Multilevel Graph Partitioning Heuristic

This section is dedicated to introducing the multilevel methods mechanism for graph partitioning. To do so, we focus on the METIS multilevel heuristic [2], which consists of three main phases: A) Coarsening phase, B) Partitioning phase and C) Uncoarsening/Refinement phase.

4.2.1 Coarsening phase

The coarsening phase aims to successively reduce the size of the graph to be partitioned.

Let G_0 denote the original graph; then during coarsening, a series of smaller graphs $G_1, G_2, \cdots, G_q$ will be created such that $V_0 > V_1 > V_2 > \cdots > V_q$ where G_q is called the coarsest graph ad consists of fewer vertices.

Usually, this can be achieved by finding a maximal matching of the graph, the reason is that a maximal matching of the graph has the ability to preserve many properties of the original graph [2, 47].

Definition (Maximal matching). *The matching of a graph, also known as the independent edge set, is a set of edges without common vertices. Moreover, a matching M of a graph G is maximal if every edge in G has a non-empty intersection with at least one edge in M [48].*

After the maximal matching is computed, the two vertices of each edge in the matching set are collapsed into one node, thereby reducing the graph size. The process of coarsening is iterated until the coarse graph reaches the desired small size of hundred vertices for example.

We encourage readers interested in more details to consult [2] as well as references therein.

4.2.2 Partitioning phase

During the partitioning phase, the coarsest graph G_q is partitioned, precisely, the vertex set V_q is partitioned into k parts. And since the coarsest graph G_q consists of few vertices, it is possible to use expensive methods for graph partitioning such as spectral partitioning method to give optimal partitions.

By the same time, various algorithms can be used in this phase, such as spectral method [51], KL algorithm [50], GGP and GGGP algorithms[2].

4.2.3 Uncoarsening and refinement phase

Once the partition on the coarse graph G_q is obtained, it is projected back to G_0 passing through intermediate graphs $G_{q-1}, G_{q-2}, \cdots, G_1$. Moreover, a refinement is performed to enhance the partition quality in the finer graph. The refinement task consists in swapping two nodes positions whenever it leads to a lower edge cut while preserving the

balance of the partition. The Kernighan-Lin algorithm is usually used for this purpose [2].

4.3 Streaming METIS Partitioning Approach

This section presents our proposed Streaming METIS Partitioning Heuristic. First, we define key notations used along the section and give an overview of the heuristic. Then, we detail the different phases of SMP, and we finally give a complexity analysis of the approach.

4.3.1 Notations

Let G be the graph to be partitioned such that G is simple and $G = (V, E)$ where V is the vertex set with size $n = |V|$ and E the edge set with size $m = |E|$. Assuming that u and v are vertices in G, $w(v)$ denotes the weight on vertex v, and $w(u, v)$ the weight on edge (u, v). Let k be the desired number of parts.

We denote by $P^t = (S_1^t, \cdots, S_k^t)$ the partition of the graph at time/iteration t where $S_1^t, \cdots, S_k^t$ are called parts, and are represented by a set of multinodes $M_1^t, \cdots, M_k^t$ such that at the final iteration we have:
i) $S_i \subseteq V$, ii) $S_i \cap S_j = \emptyset$ for every $i \neq j$ and iii) $\bigcup_{i=1}^{k} S_i = V$. We define the fraction of edge cut to be minimized during the partitioning process as $\lambda = \frac{|E_c|}{m}$, where $E_c = \{e(S_i, V \backslash S_i), i \in \{1, \cdots, k\}\}$ represents the set of edges with ends belonging to different parts. The normalized maximum load is defined by ρ which represents the balance of the partition: $\rho = \frac{maximumload}{\frac{n}{k}}$; where $maximumload$ is the size of the biggest part.

Finally, we denote by p the number of vertices loaded from the graph data stream at each iteration.

4.3.2 Overview of the method

The proposed method SMP is a natural extension of the well-known METIS to the online setting. It aims to partition the full graph into k parts in an online fashion, *i.e.*, the graph is not memory resident but arrives in a stream where each vertex is accompanied by its adjacency list.

SMP builds an accurate partition of the streamed graph using the METIS heuristic over small subgraphs loaded from the graph data stream. In other words, at each iteration

of the method, SMP loads a graph portion of p vertices from the data stream, where p remains constant during the overall execution of the algorithm and we have $p << n$.

For each newly loaded subgraph, SMP first builds a weighted graph connecting new loaded nodes to previously obtained partitions (multinodes M_i). This intermediate graph is called the *atomic graph*, and is weighted on both edges and vertices. Then, METIS is used to partition the *atomic graph*, where multinodes update their weights. Until the end of the data stream, the same process is performed: p vertices are loaded, the *atomic graph* is constructed, then METIS is used to partition it and multinodes weights are updated. In the following, we elaborate these steps in more details.

4.3.3 Streaming METIS Partitioning Approach

The SMP method works in two main stages. The first is specific to the graph partition a $t = 0$, and the second concerns subsequent operations where $t \geq 1$.

At $t = 0$, p vertices are loaded from the graph data stream, that we call simple nodes. METIS is then used to partition the subgraph and produce k parts $S_1, \cdots, S_k$, corresponding respectively to multinodes $M_1, \cdots, M_k$. Then, SMP assigns a weight to each multinode M_i that represents the number of vertices in S_i. Finally, the cut λ_0 for the first iteration $t = 0$ is recorded.

At $t \geq 1$, SMP processes two node groups (*i.e.* the previously obtained k multinodes and the p new loaded simple nodes). To address this case, SMP considers jointly the two groups as an *atomic graph*, which is a weighted graph, with weights on both vertices and edges. Weight on edges connecting a simple node v to multinode M_i is measured as the number of direct neighbors of v in S_i. Besides, edges connecting two simple nodes are weighted 1. On another side, edges between two multinodes are intentionally omitted (see subsections *1) Atomic graph construction* and *2) Partitioning*). Then, METIS is used to partition the new weighted graph of $p + k$ vertices. Finally, the multinodes weights are updated according to the partition obtained.

4.3.3.1 *Atomic graph construction*

As aforementioned, the *atomic graph* is constructed for $t \geq 1$, which is a weighted graph that is going to be partitioned using METIS. The *atomic graph* consists of the multinodes obtained from previous iteration and a p new nodes loaded from the graph data stream. The new p nodes are simple nodes with weight equal to 1 since that the original streamed graph G is simple. On the other hand, the k multinodes have weights

referring to the size of their corresponding parts. Multinodes update their weights after each partitioning, meaning that their weights increase along the iterations.

Edges connecting the p simple nodes are weighted by 1 as they constitute a subgraph of G. Besides, edges connecting simple nodes to multinodes are weighted according to the number of neighbours in multinode, *e.g.*, for a new loaded vertex v with 5 neighbours in part S_i which is represented by multinode M_i, the edges connecting v to M_i is rated $w(v, M_i) = 5$. Figure 4.1 illustrates the process of creating the *atomic graph*.

Inter-multinode edges are omitted because they represent the cut obtained on the previous partitioning λ_{t-1}. Moreover, considering those edges in the *atomic graph* would skew the partitioning process: as stated in section 4.2, the coarsening phase of METIS uses the heavy edge matching strategy which starts on a random vertex u and matches it with a neighbor v such that (u, v) has the largest weight among the adjacency list. Inter-multinode edges have high weight compared to other edges in the *atomic graph*, if they are not omitted, there is a high probability that two multinodes will be selected and added to the matching and hence the partition balance will be compromised. Therefore, inter-multinode edges are omitted in the *atomic graph*.

4.3.3.2 Partitioning

Once the *atomic graph* constructed, it is partitioned using METIS where the coarsening, partitioning and uncoarsening phases of METIS are performed (as explained in section 4.2).

However, at every iteration, the partitioning should satisfy a *partitioning condition*: each part should contain one multinode plus several simple nodes, in other words, simple nodes are assigned to the k existing parts which are represented by multinodes. Moreover, placing two multinodes in the same part is equivalent to merging two parts, which is not allowed in order to prevent high imbalance among the partition and also to avoid replacing nodes. In fact, two multinodes can end up in the same part due to either the coarsening or partitioning phase.

To address this shortcoming, two constraints will be added to the algorithm: (i) the *atomic graph* should not consider inter-multinodes edges and (ii) the coarsening phase of METIS is skipped or executed once.

Constraints.

This subsection will explain how these two constraints ensure the partitioning condition (each obtained part consists of one multinode and several simple nodes) at each iteration, at the coarsening and the partitioning phases.

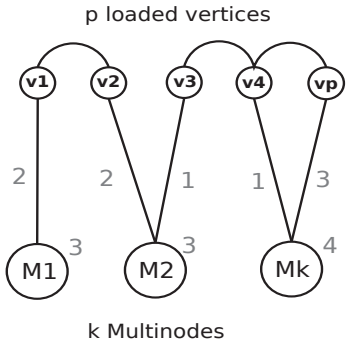


FIGURE 4.1: Constructing the *atomic graph* at iteration t.

At the coarsening phase, a maximal matching is found using heavy edge matching strategy which starts from a random vertex u and matches its neighbor v if the edge (u, v) has the largest weight on the adjacency list of u [2]. Every edge in the matching is collapsed, and the two correspondent vertices are combined to form one node.

Thus, to avoid collapsing edges connecting two multinodes, we explicitly ignore them while constructing the *atomic graph*. Nevertheless, the problem still persists as several coarsening iterations are performed. For instance, in figure 4.2, at the first iteration of coarsening, starting from node d, the edge (d, M_2) is selected and added to the matching since it has a large weight, then nodes d and M_2 are combined to form d-M_2 node, but at the second coarsening iteration, the edge (d-M_2, M_3) will be selected and added to the matching set because of its large weight, and then the two multinodes M_2 and M_3 will end up placed in the same part. Hence, we either allow only one iteration of coarsening or completely skip the coarsening phase since the *atomic graph* processed is of a small size ($p + k << n$).

The partitioning phase in METIS can lead to a similar issue (more than one multinode in one part) without incurring partition imbalance. In fact, the partitioning algorithm used (Graph Growing Partitioning GGP [2]) starts from a random vertex and grows a region around it until the sum of vertices weights reaches the maximum allowed weight, then it selects randomly another vertex from the remaining ones and repeats the same process until k parts are obtained. This partitioning algorithm could place two or more multinodes in the same part if the sum of their weights is lower than the maximum weight per part. Hence no partition imbalance would occur but still, the nodes will be replaced which is not allowed in our setting.

Proposition 4.1. *Given a positive integer t that denotes the iteration of the algorithm ($t = 0$ refers to the initial iteration), at $t > 1$, no two (or more) multinodes will be assigned to the same part using the SMP method.*

Proof. Let k be the number of parts and P_i^t be the part of index i at iteration t (initial iteration is 0), p is the block size of loaded vertices at each iteration, k the number of parts. Let $w(P_i^t)$ be the weight of P_i^t, then $w(P_i^t) = \frac{p*t}{k}$. The weight of two parts P_i^t and P_j^t is then $w(P_i^t) + w(P_j^t) = \frac{2p*t}{k}$ since the partition is balanced.

The GGP algorithm used by METIS, yields an exactly balanced partition, which means that in our case, at time t, each part should be of weight $\frac{\sum_{i=0}^{k-1} w(P_i^t)}{k}$.

Two multinodes representing P_i^t and P_j^t won't be assigned to the same part at $t+1$ after running the *GGP* algorithm if the sum of their weights exceeds the maximum allowed weight per part, in other words:

$$(i, j \in \{0, .., k-1\}, i \neq j) \quad w(P_i^t) + w(P_j^t) > \frac{\sum_{i=0}^{k-1} w(P_i^{t+1})}{k}$$

$$\Leftrightarrow \frac{2p*t}{k} > \frac{p*(t+1)}{k} \Leftrightarrow t > 1$$

Therefore, at $t > 1$ no two (or more) multinodes will be placed in the same part, which is tolerable because the parts' weight at $t = 1$ (equal to $\frac{p}{k}$) is a minimum compared to parts' weight in further iterations, and as a result, the partition balance would not be skewed and nodes replacing operations would not be costly.

For the uncoarsening phase, several vertices may switch parts if that is going to improve the cut and preserve the balance, or just improve the balance without improving the cut. Thus, it is not possible for two (or more) multinodes to end up in the same part because it directly affects the partition balance. □

- **Multinode weight update**

  After METIS partitioning is done at iteration t, the weights on multinodes are updated. As aforesaid, each part obtained at time t consists of one multinode M_i and α simple nodes where α is a positive integer. The weight on multinode M_i is incremented by α.

- **Edge cut**

  The final edge cut is represented by λ, and we have $\lambda = \sum_{t=0}^{t_f} \lambda_t$ where λ_t is the cut obtained at iteration t, and t_f is the final iteration by which the graph data stream ends, and we have $t_f = \lceil \frac{n}{p} \rceil$.

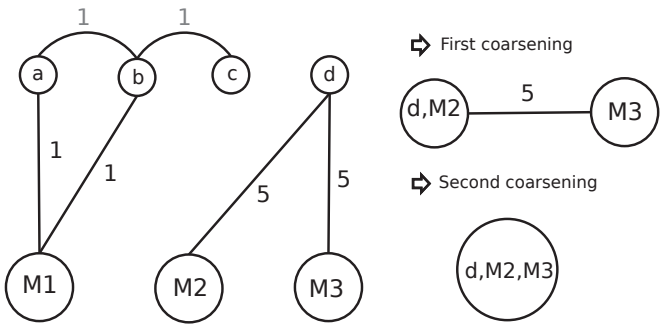


FIGURE 4.2: Coarsening phase: an example with a focus on nodes d, M_2 and M_3.

4.3.3.3 Complexity Analysis

The aim of extending METIS to an online/streaming setting was to enable high-quality METIS partitioning for large scale graphs in a streaming setting, *i.e.*, with a cheaper computational cost.

- **Time complexity**

  Originally, the complexity of METIS is $O(n + m + klog(k))$ [74]. In our setting, the algorithm has the following *atomic complexity* (according to the *atomic graph*): $O(n_i + m_i + klog(k))$ where n_i and m_i are the the number of vertices and edges in the *atomic graph*, k is the number of parts. Please note that n_i is fixed to a constant value such that $n_i << n$ and $m_i << m$, and we have $n_i = p + k$. However, the algorithm performs METIS partitioning several times, as long as it takes to process all the graph. Specifically, it takes $\lceil \frac{n}{n_i} \rceil$ times until all the graph is processed, hence the complexity of the SMP algorithm would be: $O(n + \frac{n}{n_i}(m_i + klog(k)))$. The highest will be the complexity when the processed graph is complete, *i.e.*, $m = \frac{n(n-1)}{2}$ and $m_i = \frac{n_i(n_i-1)}{2}$ for the *atomic graph*. Let $f(n)$ (*resp* $f_i(n)$) represents the complexity of METIS (*resp SMP*): $f(n) = n + \frac{n(n-1)}{2} + klog(k)$ and $f_i(n) = n + \frac{n}{n_i}(\frac{n_i(n_i-1)}{2} + klog(k))$. We have $\lim_{n \to +\infty} \frac{f_i(n)}{f(n)} = 0$, thus, $O(f_i(n)) < O(f(n))$, so we can state that SMP has lower time complexity than METIS.

- **Space complexity**

  The space complexity of both METIS and SMP are the same since SMP uses METIS to partition the graph. However, it is apparent that SMP requires smaller memory space than METIS, the reason is that SMP processes the graph in small portions during several iterations, whereas METIS processes the whole graph at once.

4.4 Experimental Evaluation

We conducted an extensive experimental evaluation to assess the performance of our proposed SMP method on various real graph datasets and compared it to alternative methods. In this section, we first give a description of the evaluation settings such as the datasets used and the evaluation methodology. Also, we present the performance metrics used and finally, we present and discuss the obtained results.

4.4.1 Settings

Datasets.

We used 11 real graphs for our experiments, all obtained from the SNAP repository [72]. The graph datasets were chosen to be small enough so that we can find offline solutions with METIS and still big enough to capture the behavior of the online heuristics, whereas the largest graph considered in this work is ComFriendster with more than 65 millions of nodes. Table 4.1 summarizes basic statistics of the graph datasets: the number of vertices ($|V|$), the number of edges ($|E|$) and also the graph type where two types of graph datasets were used, web and social. All graphs were made undirected by reciprocating the edges, and vertices with 0 degree and self-loops were removed.

Methodology.

Through all experiments, k is set to 40.

We first run our SMP method on all graph datasets listed in Table 4.1, and compare it to its offline counterpart METIS to assess whether the same partition quality is conserved.

Secondly, we compare SMP to the state-of-the-art streaming partitioning heuristics: LDG, FENNEL and FG. The comparison is also extended to restreaming methods: ReLDG, ReFENNEL and ReFG where the number of restreaming iterations is set to 10.

In order to verify the stability of our method, we run SMP on all graph datasets with three different streaming orders:

- Breath First Search BFS: is obtained by randomly selecting a node on each connected component of the graph and starting a breath first search from the given node. Component ordering is done at random in case of multiple connected components in the graph.

- Depth First Search DFS: is similar to BFS but a depth-first search is performed instead of BFS.
- Random: is given by a random permutation of the vertex set.

Random ordering is a standard ordering in streaming literature and a general assumption when theoretically analyzing streaming algorithms. On the other hand, BFS and DFS orderings have the main advantage of allowing to see edges immediately in the stream, unlike adversarial orders.

Finally, we check the impact of the *atomic graph* size on SMP performances: we run SMP on two graph datasets while varying p, the number of nodes loaded from the graph data stream at each iteration.

All algorithms have been implemented in C++, and all experiments were performed on a single machine, with Intel Xeon(R) CPU at 1.9GHz, and 16GB of main memory, except for Comfriendster graph dataset (file size of 63GB) where a machine of 100GB of main memory has been used to find the offline solution (METIS). Reported runtime includes only the partitioning execution, excluding the required time to load the graph into memory.

TABLE 4.1: Graph Datasets used for our tests.

Graph	$\|V\|$	$\|E\|$	**type**
Wikivote	7115	100762	social
Enron	36692	183831	social
Astro ph	18771	198050	social
Slashdot	77360	469180	social
Web nd	325729	1090108	web
Stanford	281903	1992636	web
Web google	875713	4322053	web
Web berkstan	685230	6649470	web
Live journal	4846609	42851237	social
Orkut	3072441	117185085	social
ComFriendster	65608367	1806067137	social

4.4.2 Performance Metrics

Three metrics are used to show and compare performances of our proposed method SMP and other competitors:

- λ refers to the fraction of edge cut and should be minimized for optimal results.

- ρ represents the partition balance (see section 4.3.1); when it is equal to 1, the partition is said exactly balanced, *i.e.*, the closest ρ to 1, the more balanced is the partition.
- T_{exec} reports the partitioning execution time without considering other program tasks, *e.g.*, loading the graph into memory.

4.4.3 Experimental Results

SMP vs. METIS. Table 4.2 reports the fraction of edge cut obtained by SMP and METIS, the online vs. the offline version.

As expected, METIS outperforms SMP, due to the offline setting (in METIS) where the whole graph is memory resident which allows it to give optimal results.

However, results of both methods are comparable and the average difference of the obtained cut between SMP and METIS over all datasets is *26%*.

Moreover, SMP outperforms METIS in Wikivote, *0.761* vs *0.770*. The remaining results in the table show the closeness between the two methods, *e.g.*, for Slashdot, SMP cuts only *7%* more edges than METIS. However, the worst performances of SMP are obtained in Web google and Stanford, where SMP cuts more than *50%* of edges compared to METIS.

TABLE 4.2: Fraction of edge cut λ given by SMP vs METIS. The balance ρ is 1.001 for both methods.

Graph	SMP	METIS
Wikivote	0.761	0.770
Enron	0.612	0.412
Astro ph	0.557	0.372
Slashdot	0.761	0.694
Web nd	0.200	0.034
Stanford	0.650	0.114
Web google	0.585	0.014
Web berkstan	0.256	0.108
Live journal	0.443	0.300
Orkut	0.672	0.347
ComFriendster	0.460	0.283

SMP vs. Online methods. Results in table 4.3 show that our proposed SMP method outperforms all other online competitors (LDG, FENNEL and FG) for almost all graph datasets.

Nevertheless, FG outperforms SMP in Enron, Astro ph and Slashdot with a slight difference, *i.e.*, for Enron resp.(Astro ph, Slashdot), FG cuts *2.3%* resp.(*0.2%*, *0.3%*) less edges than SMP. For Stanford and Web google, SMP gives the worst results, cutting up to *20%* of additional edges.

In table 4.4, we report the results of SMP vs restreaming methods ReLDG, ReFENNEL and ReFG. The table shows comparable performances of SMP and restreaming methods where the average difference between SMP and restreaming methods is *17%*.

TABLE 4.3: Fraction of edge cut λ and normalized maximum load ρ given by SMP vs. online competitors: LDG, FENNEL and FG.

2***Graphs**	SMP		LDG		FENNEL		FG	
	λ	ρ	λ	ρ	λ	ρ	λ	ρ
Wikivote	0.761	1,001	0.867	1	0.862	1.000	0,844	1
Enron	0.612	1,001	0.610	1	0.612	1.000	0,589	1
Astro ph	0.557	1,001	0.619	1	0.578	1.000	0,555	1
Slashdot	0.761	1.001	0.787	1	0.777	1.000	0,758	1
Web nd	0.200	1.001	0.261	1	0.270	1.000	0,249	1
Stanford	0.650	1.001	0.392	1	0.347	1.043	0,349	1
Web google	0.585	1.001	0.308	1	0.313	1.023	0,310	1
Web berkstan	0.256	1.001	0.342	1	0.367	1.023	0,386	1
Live journal	0.443	1.001	0.462	1	0.546	1.009	0,442	1
Orkut	0.672	1.001	0.639	1	0.696	1.076	0,627	1
ComFriendster	0.460	1.001	0.619	1	0.535	1.095	0,582	1

Stream orders. In order to investigate the stream order impact on SMP performances, we run SMP on eight datasets each with three different orders: BFS, DFS and Random. Reported results in table 4.5 reveal the considerable influence of the stream order on SMP. For example, for Web google, on the random order, it cuts *58.5%* of edges whereas only *7.7%* of edges are cut on DFS order.

According to Table 4.5, SMP gives optimal results when it is run under DFS order: it actually outperforms all other online methods (FG, FENNEL, FG) and is the closest to METIS, with an average difference of *4.3%* (instead of *26%* on a random order). On the other hand, BFS order doesn't improve the results *w.r.t.* the random order, it makes the results even worst for some datasets, *e.g.*, Web nd and Web berkstan.

SMP and online competitors were all run under a DFS order to assess their performances. Reported results in table 4.6

show that SMP outperforms all competitors (LDG, FENNEL and FG) on all datasets when they are run under DFS order. For some datasets, the difference is so important:

TABLE 4.4: Fraction of edge cut λ and normalized maximum load ρ given by SMP vs. restreaming methods: ReLDG, ReFENNEL and ReFG.

2***Graphs**	SMP		ReLDG		ReFENNEL		ReFG	
	λ	ρ	λ	ρ	λ	ρ	λ	ρ
Wikivote	0.761	1.001	0.835	1	0.813	1.023	0.812	1
Enron	0.612	1.001	0.475	1	0.476	1.098	0.479	1
Astro ph	0.557	1.001	0.418	1	0.413	1.019	0.433	1
Slashdot	0.761	1.001	0.713	1	0.703	1.041	0.711	1
Web nd	0.200	1.001	0.113	1	0.143	1.048	0.164	1
Stanford	0.650	1.001	0.204	1	0.193	1.025	0.200	1
Web google	0.585	1.001	0.161	1	0.160	1.087	0.163	1
Web berkstan	0.256	1.001	0.212	1	0.254	1.037	0.241	1
Live journal	0.443	1.001	0.313	1	0.330	1.006	0.325	1
Orkut	0.672	1.001	0.395	1	0.41	1.005	0.398	1
ComFriendster	0.460	1.001	0.425	1	0.431	1.001	0.428	1

for Stanford, SMP cuts *11.8%* of edges whereas LDG cuts *48.2%*, FENNEL *35.1%* and FG *37.6%*. On average, competitors cut *16%* more edges than SMP on DFS order.

DFS order enables SMP to have great performances mainly because it sends connected portions of the graph to SMP and in fact, this allows SMP to conduct a better partitioning process hence giving low edge cut partitions.

On the contrary, when there are no edges in the stream, *i.e.*, the *atomic graph* is made of a set of isolated vertices, where the best SMP can do is to balance the partition, and in later iterations when edges will finally arrive, the *atomic graph* will be highly connected such that a high edge cut partition becomes unavoidable.

TABLE 4.5: Fraction of edge cut λ given by SMP in the BFS, DFS and Random orders.

Graph	**BFS**	**DFS**	**Random**
Wikivote	0.955	0.752	0.761
Enron	0.687	0.463	0.612
Astro ph	0.588	0.496	0.557
Slashdot	0.897	0.719	0.761
Web nd	0.678	0.078	0.200
Stanford	0.426	0.118	0.650
Web google	0.481	0.077	0.585
Web berkstan	0.312	0.093	0.256

Impact of parameter *p*. We run SMP on Enron and Slashdot graph datasets while varying p (the size of the graph portion loaded from the data stream at each iteration) to see the impact on the obtained cut. In figure 4.3, one can clearly see that the cut

TABLE 4.6: Fraction of edge cut λ given by SMP vs. online competitors: LDG, FENNEL and FG under the DFS streaming order.

Graph	$\mathbf{SMP}_{DFS}$	$\mathbf{LDG}_{DFS}$	$\mathbf{FENNEL}_{DFS}$	$\mathbf{FG}_{DFS}$
Wikivote	0.752	0.863	0.841	0.822
Enron	0.463	0.726	0.555	0.583
Astro ph	0.496	0.813	0.540	0.537
Slashdot	0.719	0.848	0.737	0.761
Web nd	0.078	0.204	0.148	0.141
Stanford	0.118	0.482	0.351	0.376
Web google	0.077	0.545	0.174	0.195
Web berkstan	0.093	0.391	0.280	0.286

decreases when p is bigger for both Enron and Slashdot. In other words, when we process big portions of the graph at each iteration, the cut is more likely to be minimal,

with the best cut being obtained necessarily when $p = n$ (METIS).

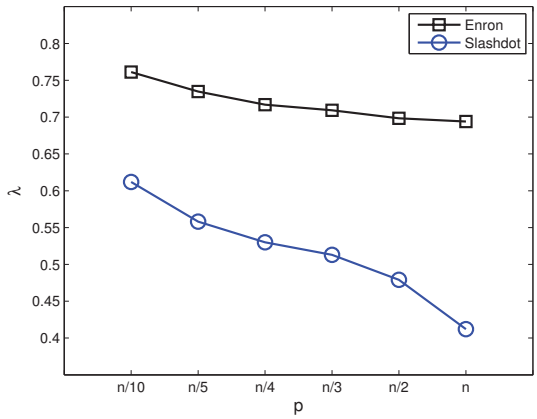


FIGURE 4.3: SMP method: Variation of λ for different values of p for Enron and Slashdot graph datasets.

Runtime. In Table 4.7 we show the execution time of SMP and METIS over all datasets except Comfriendster which was run on a different machine. Results show that for small graphs, *i.e.*, Wikivote, Enron, Astro ph, Web nd, Stanford, Web google and Web berkstan, METIS is faster though it has higher complexity compared to SMP. However, when n (number of vertices) grows and approaches values in the order of millions, SMP is faster than METIS. Hence, SMP is faster for big instances of n, *i.e.*, large graphs. Those results confirm our complexity analysis in section 4.3.

TABLE 4.7: Execution runtime T_{exec} of SMP and METIS given in seconds.

Graph	**SMP**	**METIS**
Wikivote	0.472	0.260
Enron	0.684	0.315
Astro ph	0.496	0.275
Slashdot	1.367	1.766
Web nd	1.244	0.660
Stanford	3.900	1.194
Web google	9.387	3.647
Web berkstan	7.522	2.590
Live journal	59.630	165.100
Orkut	97.220	354.709

4.5 Chapter Summary

In this chapter, we present our proposed streaming graph partitioning heuristic based on the offline de facto METIS and called SMP: Streaming METIS Partitioning [73]. We investigated the expansion of the offline multilevel heuristic METIS to an online setting (where the graph dataset is streamed) in order to obtain approximate partition quality and also to alleviate the challenging computational cost incurred when processing large scale graphs in the offline setting.

Our complexity analysis showed that SMP has a lower time complexity than METIS and conducted experiments confirm it. Moreover, SMP gives high-quality partitions that are competitive to METIS, especially when it is run within a Depth-First Search (DFS) streaming order. In fact, the streaming order of the graph has a considerable impact on SMP performances and DFS order allows it to give optimal results.

On the other hand, experiments showed that SMP outperforms all online competitors by giving minimal edge cut.

Part II

Aggregated Graph Search

This part of the thesis is about the Aggregated Search in Graphs (ASG). The novelty brought by this work is that the ASG enables simultaneous querying of multiple graphs, a deeper granularity graph search and building an answer or a match from multiple graph fragments.

Organization of the part. This part consists of three chapters: in the first chapter we give a general definition of the aggregated search and gives an overview of related research disciplines and all research work related to the keyword aggregation encountered in the literature. The second chapter focuses on the search methods in graphs. It gives definitions of related concepts, as well as a taxonomy of graph search methods. It gives a defined context of the aggregated search in graphs among other graph search proposed in the literature. The third chapter presents the core contribution work. We propose and present a new framework for graph search based on aggregated search.

Chapter 5

Aggregated search: General definition and overview

Aggregated search was first introduced as a new search paradigm in information retrieval (IR) domain. The need for such a search framework is to bring more focus and relevance to user query answering. In other words, when a user submits its query in classical web search engines, the latter give back a ranked list of links which is to the user to go through in order to find an answer. For some queries, the answer is definite and could be widely found in one link (e.g. eiffel tower construction year). However, for other types of queries, finding an answer in one document would be striking (e.g. Mariah Carey) because of the ambiguity carried by the query due to the user confusedness about its information need. For such queries, it would be interesting to give the user a short biography about the artist, a photo, several videos and related news. The building of the answer makes the search engine very effective as the user won't have to scroll all the list of links and try to summarize his or her information need. The aggregated search enables this feature of answer building from multiple sources of information to enhance and improve the effectiveness.

In addition to the aforementioned aggregated search concept, the keyword aggregation is encountered in literature with a meaning of summarization [75–77].

This chapter will cover the aggregated search paradigm in the realm of Information Retrieval (IR) First, we will give a definition of the aggregated search, then we give an overview about related (IR) disciplines and tools.

5.1 Aggregated search: Motivation and definition

Aggregated search is a novel search paradigm that arose in the IR field. It presents numerous advantages and opens a wide range of functionalities to improve the search engines performances in a way that better results (results are more precise and more relevant) would be given in response to a user query. This section starts by giving an introduction about IR, what is IR, and we describe the main trends in this field of research as well. Then, we give the main motivation behind the use of aggregated search in IR and for the aggregated search paradigm in general. Then, we give a formal definition of the aggregated search as a novel and promising paradigm search in IR.

A brief introduction to Information Retrieval (IR). Information Retrieval could be any action taken to extract information from given resources. For instance, it could be just the simple act of checking the weather of the day in a smart phone. But, Information Retrieval as an academic and research field could be defined as:

- "Information retrieval is often regarded as being synonymous with document retrieval, and nowadays, with text retrieval, **implying that the task of an IR system is to retrieve documents or texts with an information content that is relevant to a user's information need.**" (Sparck Jones and Willett 1997)
- "Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)." (Manning *et al.* 2008)

These two definitions agree that IR is the task of finding some *information material* that is relevant to an *information need* expressed by the user, from a large collection of information resources.

Definition 5.1. Information material. The information material is the output of an IR system following an information need expressed by the user. The information material is defined by its format, which could be a textual document, a video, an image $\cdots$; and is also defined by its granularity: a whole document or a paragraph, a video sequence, an image portion, etc.

Definition 5.2. Information need and User query. The information need of a user is the abstract form of information he intends to obtain, and the query is a representation or an expression of this need. The query is usually on a textual form for web search engines for example, or it could in any structured form or special language, *e.g.*, the

query could be in SPARQL language in order to retrieve information from an RDF graph.

One of the main challenges in IR is the maintaining of the large collections of resources, especially with the advent of big data and the repercussion that it implies in the storing and the processing task. Also, building efficient indexing such that the retrieval task will result in high efficiency and effectiveness is another main challenge in IR.

5.1.1 Motivation

Classical web search engines select a list of relevant documents or links and return it to the user, who should go through it in order to find an answer to his query. Nowadays, this task of scrolling the list of documents could be very cumbersome as the most majority of users are using mobile devices and on the other hand, the plethora of information available on internet in this era of big data, makes the list really long which makes these classical engines out dated with regard to the satisfaction of users information need.

Furthermore, the matching between the query and the document, or the document ranking task in classical IR engines is done on a document-level, *i.e.*, the whole document is said to be relevant or not; and this is achieved by using several models for documents scoring like the vector space model [78], the probabilistic model [79] and language models [80]. For instance, in the vector space model [78], the query and documents are represented by vectors and the angle between these vectors represents their similarity. Specifically, the cosine of the angle is used instead of computing the angle itself, and it is given by the dot product of the two vectors.

The document-level ranking, as discussed above, might not suit the information need of the user, as it might be just a section of a document, a mix of some multimedia and text content or an overlap of several documents. Thus, there is a need of more in-depth search methods that **(i)** go in a deeper granularity level than the document level and that *(ii)* enable building an answer by aggregating several documents parts. The aggregated search paradigm gathers these two main features (*i* & *ii*), which will be elucidated in the following section.

5.1.2 Aggregated search definition

Aggregated search was defined for the first time in [81] as *"the task of searching and assembling information from a variety of sources, placing it into a single interface".*

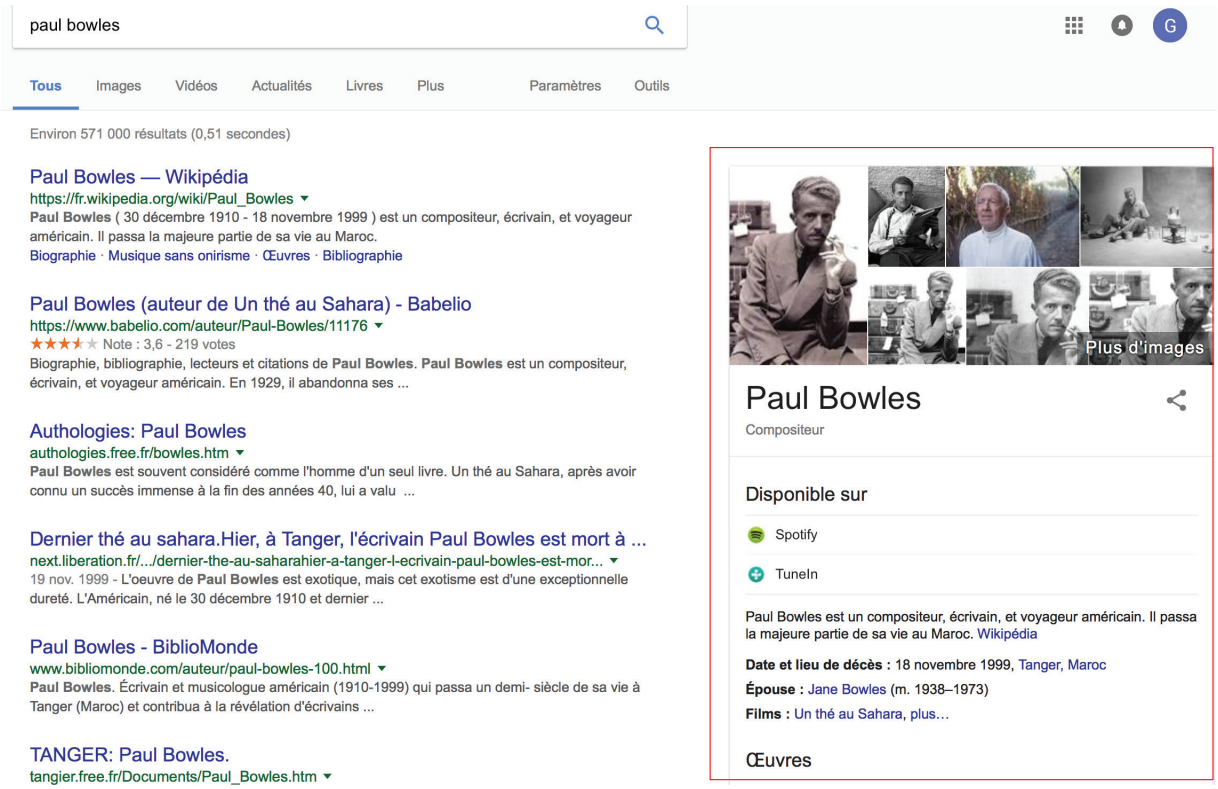


FIGURE 5.1: Example of an Aggregated search result in Google web search engine.

In other words, aggregated search is a process that retrieves fragments of information from different sources and build a consistent answer to be returned to the user, where the information fragments could be of any granularity (document, paragraph, image, video sequence,...) or format(text, video, image,...). Figure 5.1 gives an example of aggregated search result in Google search, where the red rectangle represents the aggregated search result, which consists of images, name and role of the entity "Paul Bowles" in the example, as well as a brief extract from his biography along with a short list of his most famous artwork. Moreover, the aggregated search is also seen as a threefold process [82]: ***(1)*** Query dispatching, ***(2)*** Fragment retrieval and ***(3)*** Result aggregation, which will be detailed in the following.

1. **Query dispatching:** This step refers to the preliminary processing of the query before proceeding to the query matching. This includes three main operations: source selection, query interpretation and query reformulation. Each of these operations is detailed in the following:

    - Source selection: is about the selection of the sources that are prone to answer the query given a set of sources. One usual way to select the appropriate source is the identification of special keywords in the user query. It is true that special keywords are useful to predict the kind of answers the user is

expecting [82], for example, if the query is like "what is the weather like in Bali in ten days ?", then relevant words would be *weather* and *Bali*, and they will be used as special keywords for selecting relevant sources fast.

- Query interpretation: is the task of analyzing the query to discover the intent of the question which is useful in order to determine the type of the question (e.g. what, where, when, ...) and also to identify semantic relations between the query and potential answers. For instance, from the query "what is happening now in Paris ?", the word *Paris* will be matched semantically with the Paris entity (in RDF graphs), and all related triples will be selected. Also, the words *what* and *now* will also give indications about the intent of the query: *what* would expect that the user is expecting an answer that is different from a time or a location type answer, and *now* gives an indication about actuality and news.
- Query reformulation: When the user submits its query to a general web search engine, and after selecting the source, it is necessary to add some features to the query to adapt it to the source format that will be queried. To illustrate this, consider a search engine that interrogates an RDF graph, then the user query that is usually typed in a textual format should be translated to a SPARQL format by adding special keywords.

2. **Fragment retrieval:** As depicted in figure 5.2, *Fragment retrieval* is the task that follows the *Query dispatching* where the selected source(s) matches the query to potential information fragments. The task of fragment retrieval depends on the type of the fragment to be returned, *i.e.*, there is a distinction between *document retrieval* when the whole document is expected to be retrieved and *passage/focused retrieval* which corresponds to a section of a document. Also, fragment retrieval depends on the multimedia type of the fragment (text, image, ...) as there is a difference between *textual retrieval* and *multimedia retrieval*. Upon completion, this step is supposed to output information fragments that will form the final answer to be returned to the user.

3. **Result aggregation:** Once the fragments are given, the task is to put them together in a coherent way to build the final result that will be returned to the user: this task is called *result aggregation*. There are numerous ways of aggregating content, here we cite five among them and which will be further detailed in the following: grouping, merging, sorting, splitting and extracting.

   - Grouping: Like the "groupBy" operation in SQL (Structured Query Language) [83] which tends to make group of results that share some features, the grouping task in *result aggregation* aims at grouping together fragments

that have several features in common, *e.g.*, same multimedia format, same location, etc.

- Merging: Slightly similar to *grouping*, the merging consist in merging all the fragments into one aggregate output, *e.g.*, one document; unlike the grouping, where the output is a set of fragments groups.
- Sorting: the sorting action takes the list of information fragments and sorts it according to some feature, *e.g.*, relevance score, time, etc. The output of the sorting action is also a list of fragments.
- Splitting: given a list of information fragments, the goal is to split the fragments into an even more smaller ones. This action is the opposite of the *merging* action, and its output is a list of even more smaller fragments. The splitting task is used when the fragments involved presents a relatively elevated size, and is usually accompanied by another action, like the sorting or grouping.
- Extracting: This step is often used in conjunction with another actions and it consists in identifying the main semantic information in the fragments. The output of this action could be a named entity (Paris, Michael Jackson...), images, videos or pronouns, verbs in a textual fragment.

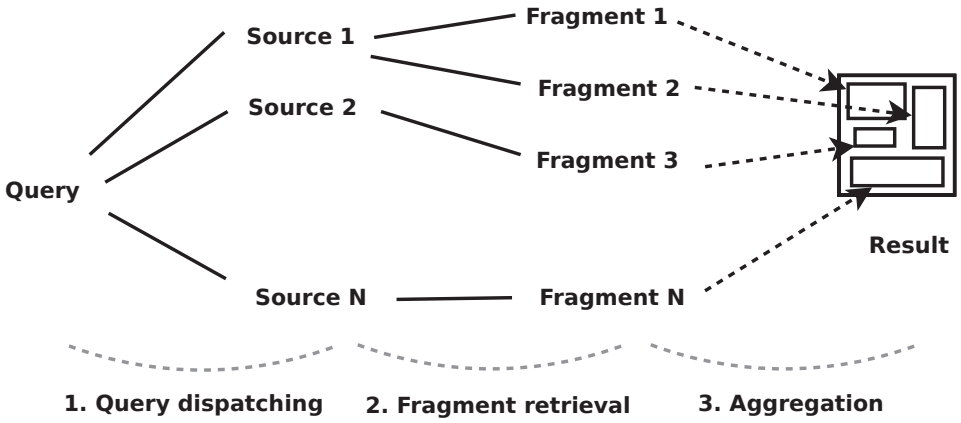


FIGURE 5.2: Aggregated search framework.

5.2 Aggregated search: related research and IR disciplines

Aggregated search finds its roots in numerous disciplines in IR such as federated search, vertical search, to name a few. It is straightforward that aggregated search is not a fully novel research area but rather a conjunction of many existing ones. In the following, we explain how federated search, query answering and natural language generation areas are related to the aggregated search.

5.2.1 Federated search

Federated search, also known as *distributed information retrieval*, is an IR paradigm with multiple distributed data sources and search algorithms [84–86]. As for the aggregated search framework, the *query dispatching* step of federated search has to select appropriate sources that are most likely to answer the query. Besides, to make this source selection task fast and lightweight, every source has its own local representation or summary which is used to swiftly identify potential sources. Finally, the obtained results from each source are assembled into a ranked list.

It is worth mentioning that *Metasearch* is considered as the equivalent of Federated search in Web search context [87], and uses multiple web search engines as an intermediary to query different sources on the web. Then, the obtained results from each engine are assembled into one single interface.

Federated and aggregated search are similar in a way that they both have to query distributed multi-sources and have to assemble the final result. Federated search tools do not bring an improvement over mono-source IR tools, however, they are preferred when the available sources are too heterogeneous to be assembled into one central source.

5.2.2 Question answering

Instead of giving a ranked list of documents as in traditional IR engines, Question Answering (QA) paradigm aims to provide a set of answers [88] that are made up through information extraction and assembling.

Similar to the *query dispatching* step in the aggregated search framework, the queries in QA are in the form of questions with different taxonomies/types, *e.g.*, "what", "when", "yes/no" etc. Also, QA uses technics for identifying named entities and facts present in the question to help in understanding the query type which is useful for determining potential sources. The *fragment retrieval* step in QA consists in having a set of matching documents and the main task is to extract text sections and build the answers which is usually a critical and error prone process. Finally, for the *result aggregation* step, QA returns a list of answers supported with text passages extracted from the matching documents. Interested readers are referred to [89] for an interesting study of the similarities between aggregated search and QA.

5.2.3 Natural language generation

The similarity between aggregated search and Natural Language Generation (NLG) is that the expected output is a document built by using information fragments rather than a list of ranked documents, as in the setting of classical IR engines. Specifically, NLG is a research area that focuses on content aggregation using predefined ways to organize textual information in a consistent linguistic way [90]. Consequently, tasks of query dispatching and fragments retrieving are absent in NLG as the information fragments are given beforehand from search engines or databases [91]. Once the information fragments given, NLG uses different ways of information organization called *discourse strategies*. Those strategies could be either predefined/static, e.g., organizing the information using a cause-effet or chronological form; or learned/dynamic through the use of learning models on example documents that serve as a training base. However, the learning strategy is concept dependent and should be specific to one known domain [92], and that clearly explain why the NLG is successful solely in specific domain applications. Further information about NLG could be found in an interesting survey in [90] and references therein.

5.2.4 Composite retrieval

Composite retrieval is similar to the aggregated search in a way that it avoids returning a list of ranked documents but rather a coherent built result, represented by a set of item bundles. Composite retrieval was first introduced in [93], and is defined as the problem of returning k bundles of complementary items, where each bundle adheres to a budget constraint, and the k bundles have to be diverse.

Let us depict the composite retrieval process in the light of the following application scenario: consider a user that intends to plan a trip; he has to check for flights, hotels, important places to visit in the corresponding city, restaurants, the weather, etc. The task of composite retrieval will be to return k bundles (k is a user fixed parameter) of *complementary* items, *e.g.*, within one bundle we need to have items such as a flight to Paris, a hotel, a restaurant, etc; whereas each bundle should be valid provided a *budget constraint* that could be for example a time interval of the trip or literally a budget (money). Moreover, the whole answer should present diversity where each bundle should be about a different destination city in the context of our example.

Authors in [93] show that the composite retrieval problem is NP-hard and propose two distinct heuristic approaches to solve it: first, the Produce and Choose (PAC) approach produces many valid bundles and selects k among them accordingly to the problem of

Maximum edge subgraph. Second, the Cluster and Pick approach finds a k-clustering of bundles, then select one valid bundle from each cluster. It was shown through experiments in [93], that the CAP approach is more appropriate when diversity among bundles is highly required, otherwise, the PAC approach would be more appropriate.

5.3 Graph aggregation: Miscellaneous approaches around "aggregation" in graphs

The keyword "graph aggregation" have a different meaning in several works encountered in the literature, where the aim is usually to perform graph summarization or compression by merging nodes that share common features [75–77]. In [75], "graph aggregation" is considered as a process of grouping together nodes that share or have the same feature value, and the links between these nodes groups exist only if all the nodes in the two groups are related.

In [76], authors propose and formalize rules, that they called graph aggregation rules, to best summarize preferences and choices within a group of users. The main idea is, from N input oriented graphs that all have the same vertex set and that represent preferences or choices for N users, to build one (oriented) graph that represents all N graphs.

Similarly, numerous works for ontologies aggregating and merging [77] find inspiration in social choice theory [94] in order to aggregate opinions, preferences and judgments to optimally summarize and determine the global choice of a group.

5.4 Relational Aggregated Search

In this section, we highlight a special direction in aggregated search field called *Relational Aggregated Search (RAS)*. In a way, RAS is related to graph search due to the focus given to relations between entities in RAS which is similar to the graph structure where entities or vertices are related to each other via edges. Since this thesis is originally situated in the graph theory research field, we decided to highlight and focus on Relational Aggregated Search as an intersection between Aggregated Search and Graph Search. We present key notions used in Relational Aggregated search along with a description of the according framework.

5.4.1 Relational Aggregated search framework

Relational Aggregated Search (RAS) considers relations between information fragments, and uses them to perform an even more coherent result aggregation. RAS has several benefits such as it gives more relevance and focus by answering some queries directly (for example, the query *ex-president of US?* a direct answer of *Barack Obama* could be given in a direct way). Moreover, relations between information fragments can help in giving more structured results, by giving a list of attributes related to the entity in the query (for the entity "Paul Bowles", the attributes like name, date and place of birth/death, novels... can be returned).

Similarly to the aggregated search framework depicted in Figure 5.2, the Relational Aggregated Search fits into the same framework, with the same three main steps (*Query Dispatching, Information Retrieval* and *Result Aggregation*), however, RAS uses special approaches in these steps as it focuses on the relation between entities of information. For instance, the second step in RAS is called Relation retrieval instead of information retrieval.

5.4.1.1 Query Dispatching

As mentioned in subsection 5.1.2, the first step in the aggregated search framework is the query dispatching. The main goal of this first step is to prepare the query before proceeding to the matching with sources. It consists in selecting the relevant sources, interpreting and reformulating the query.

Query types. In [95], authors identify and propose a simple taxonomy of query types that are most likely to benefit from RAS. In the following, we describe the *relational query types* that benefit the most from RAS.

1. Attribute query: This type of query is the most succinct, as it asks for a given value of an attribute, for example "the birth place of Paul Bowles", the "address of university Claude Bernard", etc.

2. Instance query: The instance query is about one known instance of a class, and all related attributes will be returned as an answer. As an example for instance query, we cite: "Paul Bowles", which is an instance of the class Artist, "MacBook Pro", an instance of Laptop class, etc.

3. Class query: This query type has a wider scope than the types above mentioned. A class query could be as "amphibious animals", "Pop singers", etc. To this kind

of query, a more voluminous answer would be returned (the list of all instances in the class, along with their attributes).

The query type is an important information such that, depending on it, an according solution for result aggregation will be triggered (see 5.4.1.3).

5.4.1.2 Relation Retrieval

In RAS, relations between information fragments are the core information to be retrieved. Hence, the main goal in this step is to retrieve relations between information fragments. To do so, it is possible to use special sources that have already data items that are structured as entities along with their inter-relations, these are known as knowledge bases (ontologies). Another possible way to retrieve relation between information fragments is to borrow techniques of the *Information Extraction* research field. In a nutshell, *Information Extraction* field is about studying techniques and rules that extract classes, instances and attributes and their relations as well whenever applied to simple documents or any unstructured information material [96]. We will not focus on *Information Extraction*, however, we direct interested readers to [95, 96] and references therein for additional information about this research field. In the following, review three types of relations, as a taxonomy that was proposed in [95].

5.4.1.3 Result Aggregation

When the search results or the information fragments are retrieved, they are aggregated and combined to form one compact answer. In RAS, the aggregation of the search result depends on the query type as follows:

1. Result aggregation of the *Attribute query*:

   The *attribute query* type requires the correct value of the attribute being requested. However, there may be many candidates to answer such a query, while not being able to decide which one is correct and most relevant. In general, attribute queries are answered by returning the attribute value right away or to by returning a list of the candidates along with textual elements with each candidate.

2. Result aggregation of the *Instance query*:

   Generally, the instance queries are answered by giving values of all the attributes of the instance. The attributes of the instance could be extracted from several pages and summarized. There are also specific methods that are related to people search [97], product search [**?**], bibliographic search [**?**], etc.

3. Result aggregation of the *Class query*:

   This type of queries is usually answered by a list of instances of the class in the query. The list of instances could be presented in two-dimensional tables, where each row represents an instance of the class, and each column the attributes [98].

5.5 Chapter summary

In this chapter, we tackled a general definition of aggregated search. We define the aggregated search in its original context, the Information Retrieval, as a process of information retrieval that provides the user with a more exploratory experience by giving results in a combined manner rather than a ranked list of documents. The main motivation between aggregated search is to avoid the user the burden of checking all the documents in the ranked list.

Then, we cite some important and related work to the aggregated search in the IR context, such as the federated search where the aim is to query multiple sources and gather the according results in one final list. We also noticed the similarity between the aggregated search and other research disciplines such as question answering, natural language generation.

We also explained how aggregated search is similar to the composite retrieval problem, as the latter aims to provide the user with a number of diverse item bundles that satisfy several constraints.

We further define the keyword "graph aggregation" in a more general fashion by citing other works where the aim of aggregation is different from the context of aggregated search in IR. In these works, the meaning of graph aggregation is to perform some graph compression or a graph summary by merging nodes that have features in common. Also, graph aggregation may refer to the aggregation of choices and opinions of a group of users accordingly to some rules borrowed from the social choice theory.

The following chart in figure 5.3 gives a brief summary and classification of related works cited in this chapter. As depicted in figure 5.3, *Aata Aggregation* can be divided into two categories: the (semi)structured data aggregation and the unstructured data aggregation. The unstructured data aggregation concern the aggregation in the IR context and the composite retrieval problem, mainly because the contents being aggregated are unstructured (text, images, videos). The second category is the semi-structured data aggregation which is about graph aggregation (graph summarization, choices aggregation) or about performing aggregated search in graphs. The latter problem, *i.e.*, aggregated

search in graphs, is the main problem considered in this second part of the thesis, and will be presented along with definitions and related work in the next chapter.

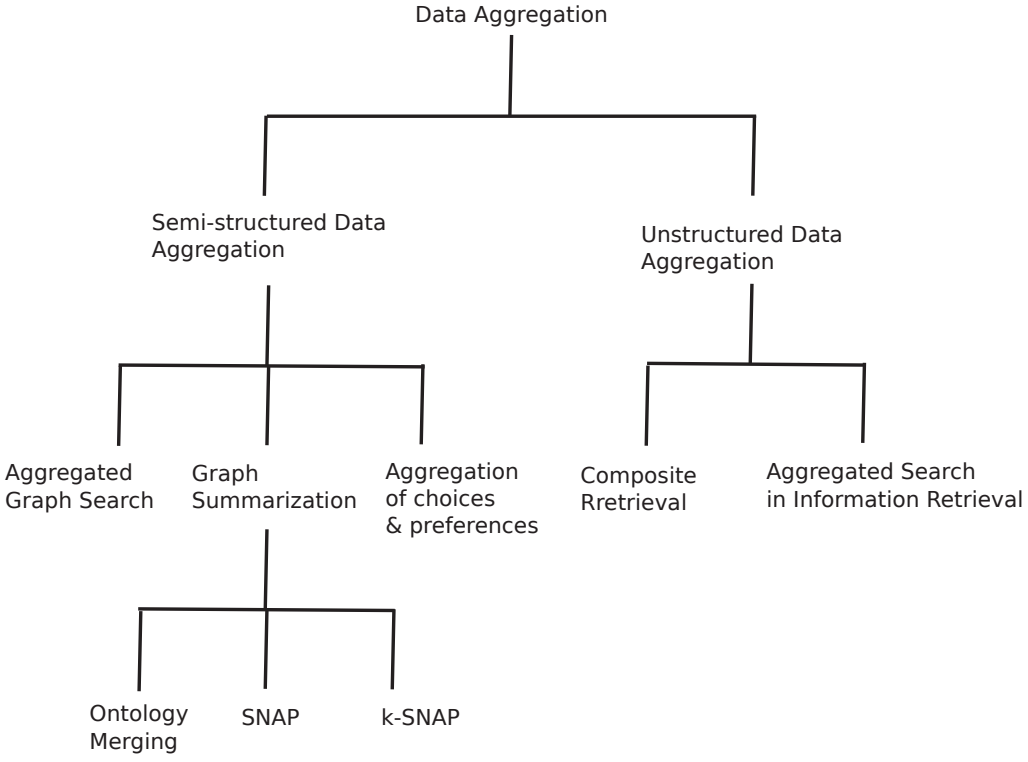


FIGURE 5.3: A taxonomy of Data Aggregation.

Chapter 6

Graph search methods

Graphs represent an efficient and a broadly used representation formalism in information processing. The advent of *big data* in the last two decades results on a plethora of information that is stored and exploited as graph datasets. For this reason, the graph querying or the graph search task becomes extremely challenging as traditional methods incur an exorbitant computational cost. As a result, most of research works in the field are more concerned about designing and developing new techniques that will achieve the best tradeoff between results precision and the incurred computational cost (time and memory). Furthermore, the graph search task is very important as the performances of most applications significantly relies on the efficiency of the querying technique being used. In this chapter, we will discuss the graph search task and give an overview of graph querying methods in the literature. We first give a general definition of the *graph search* task. We also give definition about related concepts to the graph search such as *the graph matching problem*. Furthermore, we highlight and give a taxonomy of the most important works and methods related to the graph search task.

Chapter organization. The chapter is organized as follows: Section 1 introduces and defines the graph *search problem* and gives a classification of used approaches, also, it further defines the problem by giving definitions of key concepts related to the graph search problem. The following sections are dedicated each to define and review related work to each class of *graph search* approaches.

6.1 Preliminaries

This section is dedicated to introduce and define key concepts related to *graph search* that will be used throughout the chapter. First, we define the problem of graph search

in its general form and setting, then we give a classification of methods and approaches used to search graph databases. We introduce and explain each class as well as the underlying concepts.

6.1.1 Graph search

In its general definition, graph search is the task of searching a given information, usually in the form of a *query graph*, in a large graph, often called the *target graph*. In other words, let q be a query graph, and G be a target graph. The graph search task consists in finding all occurrences of q in G. In a wide range of graph querying frameworks, the target graph is a set of graphs, and the aim is then to find all graphs within that have the query as a subgraph. In similar settings, a classical way of querying graph databases is to perform a sequential scan through all the graphs in the database and check subgraph isomorphism between each graph and the query which is very costly. Hence, most of graph querying approaches follow the framework of filtering and verification, which prunes false candidates in the graph database using an efficient graph indexing technique in the filtering step, and then perform graph isomorphism to check each remaining candidate. In the following, we discuss *graph indexing* as a preprocessing step toward efficient graph querying.

6.1.1.1 Graph Indexing

Graph indexing is a major step for a large class of graph querying methods. The technique plays an essential role in query processing and is generally used as a first step in filtering and verification frameworks. The main aim of the technique is to withdraw, in the *filtering* step, false candidate graphs in order to reduce the graph data base, and thus, to enable an efficient *verification* step, where (sub)graph isomorphism testing is performed within a reduced time. The concept of *graph indexing* is to organize the graphs in patterns that represent a number of graphs. Specifically, and given a graph g, a graph index i_g of g is a graph or a structure that represents fairly the initial graph g but with few vertices and edges. The interest of using a graph index in the *filtering step* is to accelerate the false candidate graphs detection, *i.e.*, instead of spending time in comparing the whole query with each graph candidate, one can use an index of the query and compare it to candidate graphs (see Section 6.2).

There is a variety of works in the literature with regard to the graph indexing [99–102], and the main challenge in this field is to find effective graph indexes having a lightweight structure that would be fast to process.

Moreover, different indexing techniques have been proposed for graph queries. In [103] Shasha *et al.* proposed a path-based technique termed *GraphGrep* where the idea is to enumerates paths up to a threshold length from each graph. In *GraphGrep*, the index table is constructed so that each row stands for a path and each column expresses a graph. Each entry in the index table is the number of occurrences of the path in the graph.

In [99], Yan *et al.* proposed a different indexing alternative in graph querying which aims to use the frequent patterns as index features. The idea behind is to reduce the index space as well as improving the filtering rate. Experimental results in [99] showed that the technique has smaller space index size than the *GraphGrep's* space size.

Nevertheless, the index construction requires an exhaustive enumeration of paths or fragments with high space and time overhead. Furthermore, since paths or fragments carry little information about a graph, this may lead to a loss of information. Furthermore, they are not adequate for graphs where attributes on edges or vertices are continuous values, since that the index features requires an exact match with the query.

Berretti *et al.* [104] proposed a metric based indexing on Attributed Relational Graphs (ARGs) for content-based image retrieval. The main idea is to cluster the graphs hierarchically according to their mutual distances to be indexed by M-trees [105]. The concept is to route the query along the reference graphs of clusters in a top-down manner.

Besides, [101] proposed the Closure-tree indexing technique. The main goal is to organize the graphs hierarchically where each node resumes its descendants by a graph closure. The underlying purpose of using the Closure-tree is to take advantage of their ability to support both subgraph queries and similarity queries. The Subgraph queries aim to find graphs containing a specific subgraph, whereas the similarity queries aims to find similar graphs to a query graph.

6.1.1.2 Graph Matching

Graph matching is the process of finding a mapping between the nodes and the edges of two graphs that satisfy some (more or less stringent) constraints, ensuring that similar substructures in one graph correspond to similar substructures in the other.

The graph matching is a crucial process in many applications where the structured information is represented by graphs. It aims to perform a comparison between two objects or between an object and a model to which the object could be related.

In the graph literature, a graph matching between $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where V_i and E_i are respectively the node set and the edge set, is defined as a mapping $f : V_1 \rightarrow V_2$ such that $\forall (u, v) \in E_1$ we have $(f(u), f(v)) \in E_2$.

Graph matching methods are classified into two broad categories. The first category concerns *exact matching* methods, which require a strict and exact correspondence among the two graphs to be associated, or at least within their subparts. The second category represents *inexact matching or approximate* methods. In this category of methods, a matching can occur even if the two compared graphs are structurally different to some extent.

- **Exact graph matching**

  Exact graph matching methods are characterized by the ability of their mapping to be edge preserving for each mapping for each node of the two graphs. In other words, if two nodes in the first graph are linked by an edge, their correspondents in the second graph are also linked by an edge. In the most stringent form of exact matching, graph isomorphism, this property must be conserved in both directions, and the mapping is necessary bijective. That said, a one-to-one correspondence must be found between each node in the first graph and each node in the second graph to have $|V_1| = |V_2|$, *i.e.*, the cardinality of both graphs is even.

  A weaker form of graph matching the subgraph isomorphism, which requires that an isomorphism holds between one of the two graphs and a node-induced subgraph of the other graph.

  The subgraph isomorphism can also be used in a slightly weaker sense as in [106], where the condition that the mapping function should be edge-preserving in the both directions is dropped. This resulting matching, also known as monomorphism [107], requires that each node has a distinct corresponding node in the second graph and each edge of the first graph has a corresponding edge in the second graph. However, nothing hinders the second graph to have both extra nodes and extra edges.

  Finally, another interesting matching variant maps a subgraph of the first graph to an isomorphic subgraph of the second graph. Usually, the goal of the matching algorithm is to find the largest subgraph for which such a mapping exists. This problem is commonly known in the literature as finding the maximum common subgraph (MCS) of the two graphs [108].

  It is important to highlight that, though very costly to implement, the graph isomorphism has not yet been demonstrated if it belongs or not to NP [108]. For

particular graph structure, polynomial isomorphism algorithms have been developed (trees [109], planar graphs [110], bounded valence graphs [111]). However, no polynomial algorithms are known for the general case.

Therefore, exact graph matching has exponential time complexity in the worst case. Nevertheless, in many applications, the computation time can be still acceptable, since that the structure of the graphs encountered in practice are usually different from the worst cases for the algorithms. Furthermore, the node and edge attributes can be used to reduce the search time.

- **Inexact or approximate Graph Matching**

  The stringent conditions imposed in the exact matching are in some circumstances too strict for the association of two graphs. In many application contexts, the data graphs are subject to deformations due to variation in the data patterns, noise in the data acquisition process, to missing or incomplete information. These deformations create some differences between the actual query graph and its ideal answer graph (*i.e.*, $|V_1| < |V_2|$).

  Thus, the matching should be able to accommodate such differences by relaxing, to some extent, the constraints defining the matching scheme. Even when such deformation is not expected, such robust matching can be useful. Furthermore, exact graph matching algorithms (except for special classes of graphs) require exponential time in the worst case. Moreover, it may be more practical to use algorithms that do not guarantee to give the best solution, but that, at least, reach a good approximate solution in reasonable time. The two different needs (which can also be simultaneous), have led to the rise of inexact graph matching algorithms. Usually, in this category of algorithms, the association between two nodes that do not satisfy the edge-preservation specification of the matching is not excluded. Instead, it penalized the cost of the matching, where the matching is defined as a cost function that takes into account other differences, such as the difference between the nodes and edges attributes. In this case, the algorithm must find a mapping that minimizes the matching cost.

  Among the approximate graph matching algorithms, the optimal inexact matching algorithm tries to find the solution achieving the global minimum of the matching cost. In other words, if any exact solution exists, it will be determined by the algorithm. These algorithms can be seen as a generalization of exact matching algorithms.

  Alternatively, the approximate or suboptimal matching algorithms, on the other hand, only ensure to find a local minimum of the matching cost. Usually, the obtained minimum is not very far from the global one, but there are no guarantees

to reach the exact solution even if it exists. The suboptimality of the solution is amply compensated by a shorter, usually polynomial, matching time.

Usually, inexact graph matching algorithms formulate the matching cost based on the notion of the matching errors, such as missing nodes or edges. These approximate matching algorithms are known as error-correcting or error-tolerant.

Similarly, the matching cost can be defined as a set of graph edit operations (node insertion, node deletion, etc.) between the two graphs. Since that each operation is assigned a cost, the cheapest sequence of operations required to transform one of the two graphs into the other, is estimated. The cost of this series of transactions is called the graph edit cost.

Graph Simulation. An alternative to alleviate the prohibitive cost of graph isomorphism is the graph simulation. It can be solved in quadratic time and is less restrictive than graph isomorphism so it enables approximate matching which becomes widely needed in actual applications, *e.g.*, plagiarism detection.

We say that a graph G matches a pattern Q, via subgraph simulation, if there exists a binary relation $M = V_q$, such that 1) for each $(u, v) \in M$, u and v have the same label, and 2) for each node u in Q, there exists v in G such that a) $(u, v) \in M$ and b) for each edge (u,u') in Q there exists an edge in G such that $(u', v') \in M$.

To reduce the complexity and capture the need of novel applications, graph simulation has been adopted for pattern matching. It is less restrictive than subgraph isomorphism, and can be determined in quadratic time. Graph simulation and its extensions play a critical role for the analysis of social positions/roles in social networks [112, 113]. Several extensions of graph simulation exist and have a cubic time complexity. In [114], authors propose a new variant of the graph simulation problem called strong simulation in order to rectify the loss of structural similarity in simple simulation. Strong simulation adds up some constraint in order to improve the quality of the matching while preserving the cubic time complexity of graph simulation extensions.

Graph Homomorphism. A still weaker type of matching is the homomorphism, which releases the condition that first graph nodes have to be mapped to distinct nodes of the second graph; hence, the correspondence can be many-to-one. Given two node labeled graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the problem of graph homomorphism is to find a mapping from V_1 to v_2 such that each node in V_1 is mapped to a node in V_2 with the same label and each edge in E_1 is mapped to an edge in E_2. Graph homomorphism has been revisited for graph matching. The reason is identical label matching is an overkill, and edge to edge mappings only

allow strikingly similar graphs to be matched. These conventional notions are often too restrictive for graph matching in emerging applications. In a nutshell, graph matching is to decide whether a graph G matches another graph G_p, although not necessarily identical.

6.2 Filtering and verification methods

As a consequence to the prohibitive cost of sequential scan of the database, a large class of graph querying methods follow the framework of filtering and verification [99, 103, 115, 116], where the key idea is to reduce the search space, i.e., the graph database, by withdrawing all negative candidates quickly based on a graph indexing technique. As a result, the graph database is reduced, and then the verification step performs subgraph isomorphism checkings with the remaining graphs in the database after the filtering step. Those methods, in general, rely on the indexing technique used, as an inefficient indexing may not filter enough the database resulting in a costly verification step as the number of graphs in the database won't decrease considerably.

In the following, we cite some of the well-known filtering and verification techniques.

GraphGrep [103] is a basic filtering and verification method that uses a path-based index structure, where all existing paths in the graph database are enumerated accordingly to a threshold length and indexed. Then the path-based index is used to select candidate graphs. Although this method showed good performances, it actually suffers a considerable structural information loss which usually leads to skewed results.

To alleviate this issue, Yan *et al.* proposed gIndex [99], having a graph-feature based index which is considered as a more complicated structure index aiming to improve the index pruning capabilities. Although gIndex outperforms GraphGrep, it actually inccurs a higher computational cost due to the complicated structure of the index.

Alternatively, Zhang *et al.* proposed TreePi [115] which uses a tree-feature based index. Tree graphs are known to be easy to use compared to graphs, and incur a lower computational cost while capturing more structural information about the original graphs in the database. Zaho *et al.* proposed Tree+Δ [116], which extends TreePi by adding some frequent discriminative graphs feature to the tree based index in order to enhance the index pruning.

While the above-mentioned methods consider the relationship between the index quality and the runtime cost to be a tradeoff, Gouda *et al.* proposed an efficient feature-based indexing approach that improves the pruning capabilities of their index by compressing

multiple features into one feature [117]. Their feature construction technique allows for effective indexing performances along with efficient performances regarding the runtime cost.

Interested readers are directed to [100, 118, 119] and references therein for a survey on graph indexing methods and other filtering and verification graph querying methods.

6.3 Graph querying methods based on graph matching

Graph querying is a crucial task as most of the actual datasets are being stored and exploited as graphs. Many graph querying techniques exist in the literature and the proposed framework is closely related to aggregated search and approximate (sub)graph matching, the latter being considered as an elementary operation in our matching framework. So in this section, we provide a brief description of the graph querying methods that are based on (sub)graph matching and the aggregated search in graphs with a highlight on three graph querying methods: SAGA [120], NEMA [121] and BLINKS [122].

6.3.1 (Sub)graph matching

Subgraph matching is a well-studied problem with a rich literature. Two main categories fall under the subgraph matching problem, the *exact* and the *inexact subgraph matching*. Exact subgraph matching finds exact answers to the query via graph isomorphism [119, 123]. Approaches in this group are criticized for their intractability [124], their cost prohibitive characteristic and for being restrictive *w.r.t.* to the number of obtained answers to the query.

On the other hand, inexact graph matching allows slight differences between the query and the matches which is highly suitable for actual needs as graph datasets are usually noisy, and relevant answers could be found using approximate matching. Under the category of inexact (sub)graph matching, we find subgraph matching based on graph simulation [112, 114], which can be determined in quadratic time and is defined as a *relation* between the query nodes and the target nodes. However, it suffers a considerable loss of structural similarity which makes it untailored for many applications such as in bioinformatics.

Another variant of inexact graph matching is the graph homomorphism [125]. The idea is to find mappings to the query such that node labels difference falls under a threshold whereas the query edges are mapped to paths of a given maximum length.

6.3.2 Graph matching based on similarity metrics

There is another class of (sub)graph matching based on graph similarity metrics [126], where one aims to measure and quantify the similarity between two graphs which is an important task in so many applications such as image processing for objects identification. Graph similarity metric could be given or computed in different ways [127], *e.g.*, by using the error correcting graph matching also known as graph edit distance, where edit distance is defined as the number of operations to transform a graph into another one, and these operations include node/edge insertion, deletion and relabelling. Graph similarity metric could be also given by using maximum/minimum common subgraph (MCS) where the graph similarity is proportionally given by the size of the maximum common subgraph, *i.e.*, the bigger the MCS, the more similar are two graphs.

6.3.2.1 SAGA: a subgraph matching tool for biological graphs

Tian *et al.* proposed a tool for approximate graph matching called SAGA [120]. The approach allows approximate matching by using a distance measure for similarity computing, and also by breaking the query into subgraphs and finding small hits that are assembled to form a match.

SAGA relies on an index-based algorithm to accelerate the exploration of graphs in the database. The algorithm proceeds in three steps: the query is first divided into several fragments, and the index is probed. Then, the subgraphs that have been found using the index are assembled to form larger matches to the query. Finally, the matches obtained are purged to provide the final results.

1. Distance measure for subgraph matching

   SAGA uses a distance value to measure graph similarity, such that similar graphs have smaller distance. Consider two graphs $G_1 = (V_1, E_1, o_1)$ and $G_2 = (V_2, E_2, o_2)$, where V_i is the node set, E_i the edge set, and o_i is a labeling function which associates labels to nodes. Let λ be the matching between two graphs G_1 and G_2, where λ is a bijection function which maps nodes from G_1 to G_2, $\lambda : V_1' \leftrightarrow V_2'$ where $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$. The subgraph distance, denoted by SGD, with respect to λ, is as follows:

$$SGD_\lambda(G_1, G_2) = w_e \times StructDist_\lambda + w_n \times NodeMismatches_\lambda + w_g \times NodeGaps_\lambda \tag{6.1}$$

In Equation 6.1, w_e, w_n and w_g represent the weights for each component StructDist, NodeMismatches and NodeGaps, and they are mainly used to tune the emphasis on the different components of the similarity distance measure. We detail in the following each component of the distance measure given in Equation 6.1.

The *"StructDist"* component. The StructDist component estimates the structural differences for the matching node pairs in the given two graphs: G_1 and G_2.

$$StructDist_\lambda = \sum_{u,v \in V_1', u<v} |d_{G_1}(u,v) - d_{G_2}(\lambda(u), \lambda(v))| \tag{6.2}$$

The distance between nodes u and v of G_1 is given as the length of the shortest path between them. The StructDist component examines the distance between each pair of matched nodes (u, v) in one graph to the distance between the corresponding nodes in the other graph $(\lambda(u), \lambda(v))$, and add up the differences.

The *"NodeMismatches"* component.

$$NodeMismatches_\lambda = \sum_{u \in V_1'} mismatch(o_1(u), o_2((u))) \tag{6.3}$$

In Equation 6.3, the *mismatch()* function gives a penalty for nodes with different labels. Hence, the *NodeMismatches* component, with regard to the matching λ, accumulates all the penalties given by matching nodes with different labels.

The *"NodeGaps"* component. This component computes the penalties associated to the gap nodes in the query graph, *i.e.*, nodes that were not matched to any node in the target graph G_2. The $gap_{G_1}(u)$ function in equation 6.4 associates a penalty with each gap node in the query graph G_1, and the NodeGaps component sums up all the penalties of all gap nodes.

$$NodeGaps_\lambda = \sum_{u \in V1 - V_1'} gap_{G_1}(u) \tag{6.4}$$

Each one of the aforementioned components is minimal when the two graphs G_1 and G_2 are similar and hence, the subgraph distance SGD is minimal when the two graphs are similar.

2. The index-based matching algorithm

   SAGA uses an index-based algorithm for exploring the target graph database fast, as the sequential subgraph checking of the query with each graph in the database is extremely expensive. The algorithm works as follows: an index is computed for small substructures of graphs in the database. Then, fragments of the query

are matched to fragments in the database using the index. At the final step, the fragments obtained are assembled to form larger matches.

The index structures. The index used is referred to as the FragmentIndex. The FragmentIndex is built as a set of x nodes from the database graphs, where x is a user fixed parameter, and is usually a small number ranging from 2 to 4. The x node set is formed by randomly selecting nodes from graphs in the database, and the distance between a pair of nodes is used to decide whether the node will be added to the index or not. A parameter d_{max} is used as when the distance between two nodes u and v is less than d_{max}, nodes u and v are added to the fragmentIndex and connected by a pseudo-edge. And finally, the fragmentIdex is validated when it is connected by the pseudo edges. The reason of using pseudo edges in the index is to allow gap nodes to be part of the node selection.

Items in the FragmentIndex have the following structure: [nodeSeq, groupSeq, distSeq, sumDist, gid], where nodeSeq is a series of node IDs for the fragment nodes, groupSeq is the list of group labels linked to nodes, distSeq is the sequence of distances between fragment nodes, sumDist is the sum of these pairwise distances, and gid denotes a unique graph ID. Another index called DistanceIndex is additionally used to efficiently estimate the subgraph distance between a query graph and a database graph. The aim of using this index is to look up the distance between any pair of nodes in a graph.

The matching algorithm. Once the index computed, the matching algorithm works as follows: the query is first split into small subgraphs. Then, the hits from the index probes are combined to provide larger candidate matches. Finally, each candidate is checked to give the final results. In the following, we give further details about the algorithm.

- Finding small hits: In this first step, the query is divided into small fragments and the index (FragmentIndex) is investigated to find fragments in the database that matches or are similar to the query fragments. The query is divided into fragments by enumerating all *k-node sets* as done for the database graphs to build the *FragmentIndex*. Then, for each query fragment, the groupSeq, nodeSeq, sumDist, and distSeq values are calculated. Next, each of these query fragments is examined *w.r.t.* the FragmentIndex through a multi-level filtering strategy: First, the groupSeq and sumDist values are used to filter out the non-matching fragments. Then, the candidate list is further filtered using the distSeq values. In the first filtering, database fragments are selected only if they have the same groupSeq as the query fragment and their sumDist values are within the following bounds: $f_q.sumDist \pm \frac{k(k-1)}{2} \times \frac{MaxPairDist}{w_e}$, where f_q represents the query fragments

and k the fragment size. The MaxPairDist is a user-fixed parameter that aims to restrict the weighted pairwise distance difference between the query and the database fragments as follows: $w_e \times |d_{G_1}(u,v) - d_{G_2}(\lambda(u),\lambda(v))| \leq MaxPairDist$. At this point, a list of candidate database fragments for every query fragment is generated. This list can be even more filtered by using the distSeq information (that contains the pairwise distances) to verify that all pairwise distances satisfy the MaxPairDist constraint.

- Assembling small fragments: The set of candidate fragments produced in the previous step are grouped into bigger matches as follows: First, the fragments are classified by the database graph IDs. Next, a fragment-compatible graph is built for each matching graph, where each node in a fragment-compatible graph represents a couple of matching database and query fragments. Two nodes in the fragment-compatible graph are connected by an edge iff two query fragments share zero nodes, or more nodes and the corresponding database fragments in the fragment-compatible graph also share the same nodes. An edge connecting two nodes tells us that the two fragments concerned can be joined to form a larger fragment since they have no conflicts in the union. Therefore, a clique in the fragment-compatible graph represents a set of fragments that can be joined. Once the fragment-compatible graph is formed, the fragments assembling problem reduces to the maximal clique detection problem, and the set of fragments in each maximal clique is considered as a candidate match.
- Examining candidates: In this step, each candidate match is examined in order to produce a set of final matches. A parameter P_g is defined by the user, and it denotes the percentage of gap nodes in the match. Once P_g fixed, the matches having a percentage of gap nodes greater than P_g are eliminated. For the remaining candidate matches, the DistanceIndex is probed, and the subgraph matching distance SGD defined above is computed, where the matching to be chosen is the one that minimizes the SGD distance.

Although SAGA method is based on an efficient indexing to speed up the processing, it actually reduces to solving the maximum clique problem which represents a costly operation.

6.3.2.2 NEMA

A tool called NeMa (Network Match), proposed by Khan et al. in [121], is a subgraph matching technique based on neighborhood information to query real-life network

datasets.

NeMa consists in finding the (top-k) matches of the query graph with minimum costs w.r.t the target graph. The cost function used is a metric that estimates the similarity between the query graph and the target graph and hence, should be minimized. NeMa is a heuristic that is based on an inference model in order to solve the underlying problem. Int he following, we describe the cost metric used in NeMa, along with the heuristic used and the inference model.

1. Cost metric

   Before delving into the definition of the cost metric used in NeMa, let us start by giving notations that will be used throughout the section.

   Used notations. A **target graph** usually represents a network dataset, and is defined as a labeled undirected graph G = (V, E, L) where V denotes the node set, E the edge set and L a labeling function that assigns labels to nodes. **A query graph** is an undirected, labeled graph denoted by $Q = (V_Q, E_Q, L_Q)$ where V_Q is the node set, E_Q the edge set, and L_Q is a label function that associates each query node with a corresponding label $L_Q(v)$. Given a query graph Q and a target graph G, the set $M(v)$ of query node v represents the target nodes **candidates set** which is the set of target nodes that are likely to match with query node v. Formally, a target node u is considered as a candidate for node v ($u \in M(v)$) only if the difference between their respective labels fall under a given threshold. A **subgraph matching function** ϕ is defined as a many-to-one function: $V_Q \rightarrow V$, such that $\forall v \in V_Q, \phi(v) \in M(v)$.

   Neighborhood vectorization. Given a node v in a graph G, the neighborhood of u is represented by a neighborhood vector $R_G(u) = \{< u', P_G(u, u') >\}$, where u' is a neighbor of u within h-hops, and $P_G(u, u')$ denotes the proximity of u' from u in G. $P_G(u, u')$ is defined as follows:

$$P_G(u, u') = \begin{cases} \alpha^{d(u,u')} & \text{if } d(u, u') \leq h \\ 0 & \text{otherwise.} \end{cases} \tag{6.5}$$

   Where α is a propagation factor anywhere between 0 and 1, and $h > 0$ is the hop number of the neighborhood for vectorization. $d(u, u')$ represents the distance between node u and u'. The neighborhood vector is to encode the distance information about node u and its h-hops neighbors.

   Neighborhood cost. The neighborhood cost is defined based on neighborhood vectors, and it corresponds to the cost of matching the neighborhoods of a query node and a target node. Let the neighbor nodes of v within h-hops be denoted by

N(v), and let consider a matching function ϕ. The neighborhood matching cost between v and $u = \phi(v)$, denoted by $N_\phi(v, u)$ is defined as:

$$N_\phi(v,u) = \frac{\sum_{v' \in N(v)} dig(P_Q(v,v'), P_G(u, \phi(v')))}{\sum_{v' \in N(v)} P_Q(v,v')} \tag{6.6}$$

Where the function *dig* is as follows:

$$dig(x,y) = \begin{cases} x - y & \text{if } x > y \\ 0 & \text{otherwise.} \end{cases} \tag{6.7}$$

A function that computes the label difference denoted δ_l is considered, and it is supposed to range between 0 and 1. Then, the matching cost for each node given a matching function ϕ is defined as the aggregation of the label difference function and the neighborhood matching cost function. The matching cost for individual nodes v and $u = \phi(v)$ is defined as follows:

$$F_\phi(v, \phi(v)) = \Lambda.\delta_l(L_Q(v), L(u)) + (1 - \Lambda).N_\phi(v,u) \tag{6.8}$$

Where Λ is a parameter in [0,1].

The subgraph matching cost. The subgraph matching cost function sums up all individual matching cost for all query nodes and is defined as follows:

$$C(\phi) = \sum_{v \in V_Q} F_\phi(v, \phi(v)) \tag{6.9}$$

The minimum cost subgraph matching problem. The underlying problem that NeMa aims to resolve is called the minimum cost subgraph matching MCSM. The MCSM consists in finding a matching ϕ that incurs the lowest subgraph matching cost denoted by $C(\phi)$. The MCSM problem was proved to be APX-hard.In other words, the MCSM problem aims to find a matching Φ given in Equation 6.10.

$$\Phi = \underset{\phi}{argmin}\ C(\phi) \tag{6.10}$$

2. Heuristic algorithm

NeMa is based on an inference model, referred to as *NemaInfer*, to compute the minimum cost matchings by identification with the *Max-sum* inference problem. In a nutshell, the *Max-sum* inference problem aims to find values of variables $x_1, x_2, \cdots, x_M$ resulting in maximizing $p(X) = \Pi_i f_i(X_i)$ or $logp(X) =$

$\sum_i log f_i(X_i)$, where $p(X)$ is a joint probability distribution of the set of variables $X = \{x_1, x_2, \cdots, x_M\}$. Provided that the goal of the minimum cost subgraph matching problem is to minimize the overall subgraph matching cost $C(\phi)$, one can see that there is a similarity with the max-sum inference problem.

- NemaInfer algorithm: Consider a query graph Q and a target graph G, first, NemaInfer calculates the set of candidates for each query node using the node label similarity function. Then, an inference cost denoted by $U_0(v, u)$ is initialized with the minimum possible value of a node matching cost $F_\phi(v, u)$, considering all possible matching functions ϕ, where $\phi(v) = u$. Next, the algorithm computes the inference cost for each query node v and its candidate matching nodes, and picks the optimal match of v as the candidate u having the minimum inference cost. It is important to note that NemaInfer records the optimal matches for each query node. The same process iterates until it reaches a fixpoint, such that the optimal matches for a fixed number of query nodes stay the same in two successive iterations.
- Inference cost and Optimal match:

  In each iteration, the algorithm NemaInfer enhances the quality of the matching, based on the joint notion of the inference cost and the optimal match which will be discussed later. The inference cost, computed at every iteration i of NemaInfer and for each $v \in V_Q$ and $u \in M(v)$ ($M(v)$ is the candidate nodes set for node v), is denoted by $U_i(v, u)$ and is defined as follows:

$$U_0(v, u) = \min_{\{\phi:\phi(v)=u\}} F_\phi(v, u) \tag{6.11}$$

$$U_i(v, u) = \min_{\{\phi:\phi(v)=u\}} [F_\phi(v, u) + \sum_{v' \in N(v)} U_{i-1}(v', u')] \tag{6.12}$$

  In Equation 6.12, we suppose that $i > 0$. Then, the inference cost is defined as the minimum sum of the node matching cost $F_\phi(v, u)$ for each node, and inference costs of the previous iteration $U_{i1}(v, \phi(v))$ for all neighbors v of v, considering all possible matching functions ϕ, where $\phi(v) = u$.

  Optimal match. For every iteration, there is a match for each query node that is called the optimal match. We define the optimal match of a node v at the i^{th} iteration, denoted by $O_i(v)$, as follows:

$$O_i(v) = \underset{u \in M(v)}{argmin}\ U_i(v, u); \qquad i \geq 0 \tag{6.13}$$

NEMA proves to be an efficient subgraph matching tool based on a graph matching cost metric that considers node label similarity, however, it doesn't address the problem

of multi-target graphs. Moreover, subroutines in NeMa can be very expensive such as the neighborhood vectorization process, the neighborhood matching cost, and also the inference query processing algorithm used *NemaInfer*.

On another hand, the notion of h-hops neighbors considered in NeMa, unless $h = 1$, is irrelevant and adds nothing but complexity and computational expense, especially for h values that are greater than 1, although it permits a certain form of generalization in the whole method.

It is also worth mentioning that there are parallel works related to the approximate graph matching in bioinformatics. This category of methods includes PathBlast [128], NetAlign [129] and IsoRank [130] to name but a few. These methods are found to be efficient in tasks related to the application domain such as protein interaction networks alignment [128, 129]. Besides, a different type of approximate structural matching works has been proposed in [131, 132]. These works are based on concept propagation [133] and spreading activation [134] instead of classical matching schemes (graph isomorphism and similarity metric). Nevertheless, these works consider a strict label node matching, which is still too restrictive.

Finally, approximate and inexact (sub)graph matching encompasses decades of research work. Further details about the problem could be found in [135] as well as references therein.

6.4 Querying semi-structured data (RDF graphs)

In the realm of querying RDF graphs, SPARQL is a widely used query language. It requires a complete knowledge of the schema of the graph database, *i.e.*, the structure, node labels and types of entities in the graph. Moreover, writing queries in SPARQL proves to be a complicated task that requires users to be familiar with the language. In order to alleviate this, Zou *et al.* studied the problem of answering SPARQL queries via subgraph matching and proposed gStore [136], which allows approximate node label matching but adheres to strict structural matching leading the method to be restrictive.

6.5 Aggregated search in graph databases.

Aggregated search is a familiar search paradigm in Information Retrieval in the context of documents [82]. However, few works tackled the problem of aggregated search in graph databases. Elghazel *et al.* [137, 138] studied the problem of aggregated search in

labeled graphs. Their method looks for exact matches which is restrictive on the one hand, and on the other hand very expensive due to the maximum common subgraph search and the maximum clique detection tasks.

6.6 A taxonomy on graph search methods and discussion

To give a synoptic view of the works around the graph search task and associated problems, we propose a taxonomy of graph search methods. We derive the graph search methods into four classes: the filtering and verification frameworks, the (semi)structured data querying, keyword graph search and the most important class: the graph matching methods. The graph matching is derived into two sub-categories: the exact matching, also known as the graph isomorphism problem; and the inexact graph matching which is the most suitable for actual graph search application as the datasets are huge in size. Figure 6.1 depicts the proposed taxonomy organized in a hierarchy.

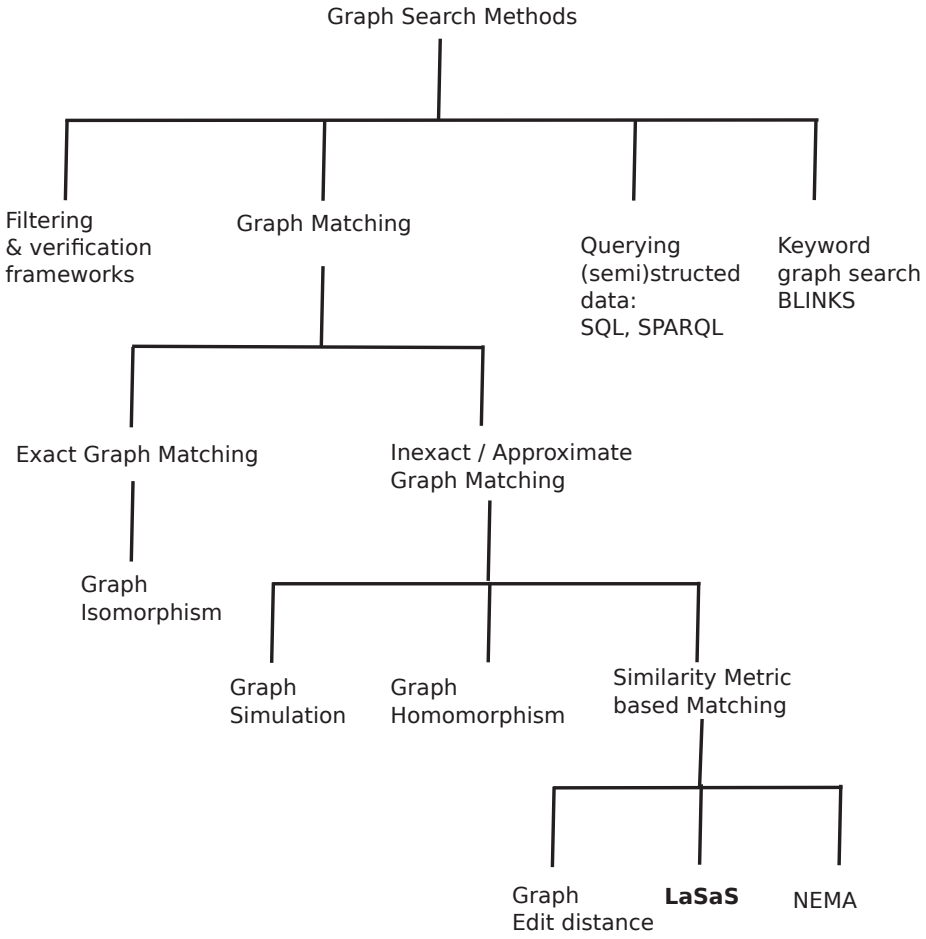


FIGURE 6.1: A taxonomy of Graph Search methods.

Most of the aforementioned works (described in previous sections) altogether lack several critical points: (*i*) They present a restrictive tool for query answering either by imposing a strict node label similarity or by a strict structural similarity, which does not allow approximate relevant answers to be discovered, (*ii*) the graph matching is usually done by checking the maximum common subgraph which is a costly computational task [139]. (*iii*) Methods querying semi-structured data requires a complete knowledge of the graph schema and are often complicated. (*iv*) Few works considered a query answering by several graphs in concomitant, and if so, they seek exact matches which is still restrictive. To address these shortcomings, we propose a new graph matching framework that aims to answer a query by allowing approximate label and structural similarity through a lightweight similarity metric which alleviates the task of maximum common subgraph search. Moreover, our proposed framework enables efficient RDF graph querying without having any knowledge about the schema of the graph. Last but not least, our proposed framework is based on the aggregated search to enable a joint query answering by several heterogeneous graphs in order to benefit from the information wealth within these graphs.

6.7 Chapter summary

In this chapter, we tackled the problem of graph search, also known as graph querying. We first define the process of graph search as well as the underlying concepts, such as the graph matching and the graph isomorphism, graph simulation and graph homomorphism. We also propose a taxonomy of graph search methods as depicted in figure 6.1.

As the main focus of this part of the thesis is on aggregated search in graph databases, our proposed method in Chapter 7, named LaSaS (Label and Structure similarity using aggregated Search) goes under the category of similarity metrics based approximate graph matching methods.

Chapter 7

LaSaS: A new Approximate Graph Matching framework based on Aggregated Search

Querying graph datasets proves to be an important task as most of the real-life datasets are represented by networks of labeled entities. Moreover, the noisy nature of those graph datasets makes the approximate graph matching tools highly required in order to alleviate restrictive query answering. In this chapter, we introduce and propose a new framework for graph querying based on aggregated search called *Label and Structure Similarity Aggregated Search* (LaSaS). First, we give the main motivation behind the use of the aggregated search paradigm in graph databases. We explain how the aggregated search in graphs benefits the querying process compared to classical and traditional graph querying setting. Then we present our proposed aggregated search graph matching framework in three main sections: we introduce and define theoretical terms and concepts used in the framework. Then, we present the query processing algorithm and explain all the underlying routines and steps from the query acquisition to the result output. We ultimately give an intensive experimental evaluation of our proposed method by exploring its performance under a variety of settings and by comparing it to related state-of-the-art works. Results from our experimental evaluation approved the effectiveness and the stability of the proposed method over different parameter settings. Besides, results also indicate that LaSaS leads to significant improvement over state-of-the-art related approaches by finding more precise matches in a shorter amount of time. Finally, in the last section, we give a chapter summary and outline several perspectives and future work directions.

7.1 Graph search using Aggregated Search

Recent years have witnessed a sheer increase in data generated from numerous applications. This data proliferation favors the use of graphs as a storage support to fully exploit the knowledge within the dataset. In fact, graphs are a popular data model that enables efficient data processing benefiting from all the graph theory findings and technics. In considering this matter, graph querying or graph search has attracted the interest of many researchers as it represents an essential task to exploring the knowledge in these datasets.

In order to query these graph databases, generally, a *graph matching* task is performed using either *graph isomorphism* to find exact answers to the query, or other technics that allows *approximate graph matching*. In the case of graph isomorphism, seeking exact answers to the query can be very expensive as well as restrictive since actual datasets are usually noisy. Moreover, it requires the user to have a complete knowledge of the data structure which is not always the case. To bypass these restrictions, approximate graph matching is widely used in many real life applications such as web anomaly detection [140], search result classification [141] and spam detection [142] to name a few.

However, few works on graph querying have addressed the graph matching problem by building an answer to the query based on the aggregation of heterogeneous graphs. The idea is to find a matching to a given query by combining together several subgraphs that when aggregated, they form an approximate match for the query. This search paradigm is known as aggregated search in graphs [137, 138], *i.e.*, given a query graph q and a set B of graphs, aggregated search aims to find matches to q by combining or aggregating subgraphs in B. The aggregated search is different from the classical graph matching problem where all occurrences of the query are to be found in one target graph G. In the following, we first give a definition of the aggregated search in graph databases, then we give the key motivation behind the aggregated search paradigm in graphs.

7.1.1 Aggregated search in graph databases: Definition

Aggregated search is a recent search paradigm that was first introduced in the field of Information Retrieval [81]. In a nutshell, aggregated search aims at building a result from several and complementary information shards given a query, where the information shards represent an elementary piece of information found in a source or document and could be a paragraph, an image, a video section, etc. Aggregated search is mainly distinguished by its ability to search for any granularity level of information, and for merging all these information shards to build a coherent final result. In the realm of

graph datasets, the aggregated search still refers to the process that retrieves deeper granularity levels of information and merges it to build a result. However, there are some particularities as the data being used is in a format and context that is different from IR. Aggregated Graph Search (AGS) is defined as follows:

Given a query graph q, and a set of fragment graphs B, where each fragment f_i has eventually a common subgraph with q. The aim of the aggregated graph search problem is to build a graph called the aggregate a that matches with q through exact matching, *i.e.*, graph isomorphism, or inexact matching, where the graph a is the union of several fragments f_i. In other words, the aggregated graph search is to build an aggregate using complementary fragments which are all relevant to the query, such that the final aggregate matches with the query exactly or approximately. The aggregated graph search process will be further detailed and explained in the following sections, nevertheless, we give an illustration of this process in figure 7.1.

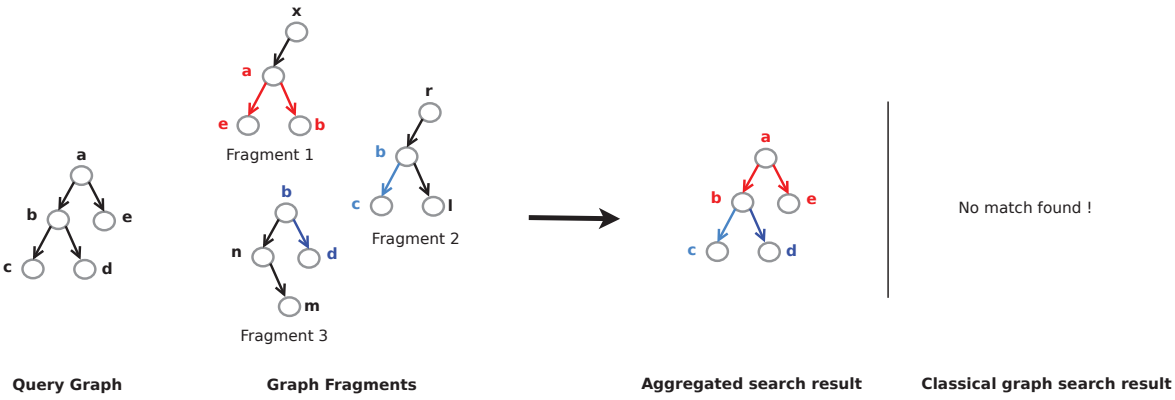


FIGURE 7.1: Aggregated Graph Search vs. Classical Graph Search.

Figure 7.1 depicts the difference between aggregated graph search and classical graph search, *e.g.*, graph isomorphism. One can clearly see that with aggregated graph search, we can still find matches to a given query, whereas with traditional graph search methods, no results could be found due to the fact that traditional graph search are query wise, *i.e.*, each graph or fragment is said to contain the query or not, while for aggregated graph search, each fragment is explored to find substructures of the query.

7.1.2 Aggregated search in graph databases: Motivation

Graph search methods include a broad range of methods: exact algorithms, heuristics etc. Since the graph model represents a very efficient data storage model, it becomes widely used in almost all application domains (social networks, the world wide web, user generated data, institution data etc.). Thus, methods for querying those graphs in

order to explore knowledge within and to perform processes over these data is necessary. Graph search methods or graph querying methods include a broad set of multiple approaches varying from exact algorithms to heuristics. A commonly known graph querying concept is the graph matching problem, where a query graph is to be matched with several graphs from the target graph database. The graph matching problem generally supposes a searching for the whole query through performing subgraph matching, graph isomorphism which are high-cost operations (NP complete [143]) in graphs. This compels the research work to be directed toward heuristic solutions in order to compute a graph matching, and several problems were defined in the light of a heuristic approach: inexact/approximate graph matching where a query graph matches some graphs under a fixed threshold and does not have to be fully isomorphic. The relevance and usefulness of each version (either exact or approximate) of the graph matching problem are dependant on the application domain and the context on which it is used. For example, in biology, exact matching schemes are usually preferred due to the high precision required by the domain; on the contrary, approximate matching schemes are favored in querying big graph datasets as in social networks analysis (graph pattern mining $\cdots$) [121]. On the other hand, few works have tackled the problem of aggregated search in the graph context [137, 138]. The problem of aggregated search in the context of graphs can be seen as a graph matching task in the way that it takes a query graph and try to find corresponding matches, and similarly to the aggregated search in the information retrieval context, the aggregated graph search aims at finding a matching for a query graph by building a match from several subgraphs called fragments. While the usual graph matching setting seeks an answer to a query as a whole, graph aggregated search consider query subgraphs mining and then building an answer that matches the query. The motivation behind aggregated graph search is two fold: first, it is a great tool for supplementing graph query processing frameworks by finding more answers through the investigation of building matches from graph fragments that match each with a query subgraph. Secondly, aggregated graph search is very useful for discovering scattered patterns in distributed or distinct graph datasets. For example, in plagiarism detection, aggregated graph search could easily detect plagiarism cases where several parts (in text format) were copied from different sources and combined into a refined and original looking document.

7.2 Preliminaries

This section is dedicated to introducing notions used in our proposed method. First, we give the formal definitions of the data structures used in the rest of the chapter: the query, the target graphs and the answer set. Then we introduce a new graph similarity

metric and the objective function used in our proposed matching scheme. Finally, the section ends by giving the formulation of the aggregated graph search problem.

7.2.1 Query, Target graph, Answer

Query. The query q is an undirected labeled graph $q = (V_q, E_q, \mu_q)$ where V_q represents the vertex set, and E_q the edge set. While μ_q is a function $\mu_q : V_q \rightarrow L_{V_q}$ associating the labels to vertices, with L_{V_q} the vertex label set.

Target graphs. Target graphs are represented by the set $B = \{f_1, \dots f_p\}$ of p graphs which are referred to as fragments f_i, $i \in [p]$. Fragments $f_i = (V_i, E_i, \mu_i)$ are undirected labeled graphs *s.t.* V_i (resp. E_i) is the vertex set (resp. edge set) of f_i. μ_i is a function $V_i \rightarrow L_{V_i}$ associating labels to vertices, with L_{V_i} is the vertex label set of f_i.

Answer set. The expected answer set A is defined as follows: $A = \{a_1, \cdots, a_k\}$ where a_i are called aggregates and k is the number of aggregates. We have $a_i = \bigcup_{j=1}^{p_i} f_j$ *s.t.* p_i is the number of fragments in a_i and we have $p_i \leq p$, *i.e.*, a_i is constituted by as few fragments as possible.

Though our method focuses on undirected labeled graphs, it is straightforward to extend it to process other kinds of graphs.

7.2.2 Objective function

Graph similarity metric. We first introduce a graph similarity metric that computes the similarity between two graphs based on two main features: *(i)* the node label similarity and *(ii)* the structure similarity. The similarity function is denoted by $s(f_i, q)$ where $s : B \times \{q\} \rightarrow [0, 1]$ is a function that quantifies the similarity that f_i shares with q in terms of label and structure similarity *s.t.* the closest s to 1, the more similar are f_i and q. In the following, we present the components of the similarity metric s: *label similarity* and *structure similarity* components.

7.2.2.1 Label similarity component

Label similarity of fragment f_i and query q is denoted by $\Delta_L(f_i, q)$ and is computed as follows:

$$\Delta_L(f_i, q) = \frac{1}{n_i}. \sum_{v \in V_{f_i}} J(\mu(v), \mu(Q(v))) \tag{7.1}$$

where n_i is the number of vertices in f_i, and J is the jaccard similarity coefficient such that $J(l_1, l_2) = \frac{W_{l_1} \cap W_{l_2}}{W_{l_1} \cup W_{l_2}}$, with W_{l_1} (resp. W_{l_2}) is the words set in label l_1 (resp. l_2). Q is an application $Q : V_i \rightarrow V_q$ that associates vertices from the fragment to their counterpart in the query and we have: $Q(v) = u$ iff $J(\mu_i(v), \mu_q(u)) > \tau$ where τ is a user fixed threshold.

7.2.2.2 Structure similarity component

On the other hand, $\Delta_D(f_i, q)$ denotes the structural similarity between fragment f_i and query q, and is defined as follows:

$$\Delta_D(f_i, q) = \frac{1}{n_i}. \sum_{u,v \in V_{f_i}, u<v} \delta_t(d(u, v), d(Q(u), Q(v))) \tag{7.2}$$

with

$$\delta_t(x, y) \begin{cases} 1 & \text{if } |x - y| < t \\ 0 & \text{otherwise.} \end{cases}$$

$d(u, v)$ denotes the distance between vertices u and v, while Δ_D computes the structure similarity in terms of distances in f_i and q. Δ_D increases each time the distance between two nodes in the target graph is equal or near to the distance between their corresponding nodes in the query graph. Precisely, Δ_D increases when the distance difference is under a threshold t called the *structure disparity threshold*. In practice, threshold t should be small in order to avoid disparate matches and to preserve the structural similarity. Besides, t should neither be equal to zero otherwise it would be very restrictive as only strikingly similar matches will be favored.

Taking these both components into account, we define the similarity function s as follows:

$$s(f_i, q) = \lambda.\Delta_L(f_i, q) + (1 - \lambda).\Delta_D(f_i, q) \tag{7.3}$$

where λ is a tunable parameter in $[0, 1]$.

Objective function. Let η denote the matching that associates the aggregates a_i in the answer set A to the query q (matching query nodes to target nodes). Based on the similarity function s, we define the objective function of the matching η as follows:

$$\Psi(\eta) = \frac{1}{k}. \sum_{i=1}^{k} s(a_i, q) \tag{7.4}$$

7.2.3 Problem Formulation

Given a set of fragments B, a query q. The aggregated graph search problem consists in finding the matching η that associates the answer set A of k aggregates to q *s.t.* $\Psi(\eta) = 1$.

Proposition 7.1. Aggregated graph search problem is NP-Hard.

Proof. Let us consider the simple case of $k = 1$. That is to say, the expected answer set consists of one aggregate a_1 *s.t.* $a_1 = \bigcup_{j=1}^{p_1} f_j$, where $\Psi(\eta) = 1$, *i.e.*, $s(a_1, q) = 1$, with η is the matching that associates aggregate a_1 to q. By definition, the similarity function $s(a_1, q) = 1$ means that we are looking for minimum elements from B (see section 7.2.1) *s.t.* their union covers q: label similarity component equals one means that all nodes are matched and their labels are exactly similar, on the other hand, structure similarity component should be one, meaning that the query and the aggregate have similar structure. This reduces to resolving the NP-Hard set cover problem where the universe is the query q that we aim to cover with minimum elements from B. This ends the proof. □

7.3 Query Processing Algorithm

In this section, we present our query processing algorithm named LaSaS, which is a heuristic solution intended to solve the problem of aggregated graph search problem. We expose in details the proposed algorithm and its different steps.

LaSaS algorithm works in three distinct steps: first, the *Selection step* is the most important step that aims to select the most relevant fragments from B based on the similarity metric s (see section 7.2.2). Second, the *Aggregation step* combines the relevant fragments found by the *Selection* step to form the aggregate a. Third, the *Refinement step* enhances the quality of the aggregate by pruning irrelevant nodes. Furthermore, in the case of unmapped edges, this step finds mapping paths of a length under a fixed threshold. In the following, we present each step in details. Note that this process is necessary for obtaining one aggregate, whereas in the general case of k aggregates, this process will repeat k times. LaSaS is described in algorithm 2.

7.3.1 Selection step

The first step consists in selecting the most relevant fragments, as many as necessary to cover the whole query q. The selection is based on successive iterations of two main

Algorithm 2 *Label and Structure Similarity Aggregated Search*

Input: Fragments set B, Query graph q, number of expected aggregates k.
Output: Answer set A.

1. $i = 1$;
2. **while** $i < k + 1$
3. Select potential fragment from B that are similar to q;
4. Add the selected fragments to set S;
5. Aggregate all fragments in S to form an aggregate a_i;
6. Refine a_i;
7. Prune irrelevant nodes;
8. If there are unmapped edges in a_i:
9. Map paths to missing edges;
10. $A = A \cup \{a_i\}$;
11. $i = i + 1$;
12. $S = \emptyset$;
13. **end while**
14. **return** A;

substeps: *(i)* similarity checking and *(ii)* query updating until a stopping condition is verified. In the similarity checking, a rank is associated with each fragment $f_i \in B$ which is given by the similarity metric $s(f_i, q)$ (see section 7.2.2), then the fragment f_{max} having the maximal similarity is selected and we have $f_{max} = \underset{f_i}{argmax}\ s(f_i, q)$ where $f_i \in B$. Then, the query is updated according to the best-ranked fragment f_{max} such that the query part that has been covered by f_{max} is withdrawn according to several conditions detailed in the following. When the query is updated, the selection process repeats: similarity checking and query update substeps are performed until at least one of the stopping conditions is verified: *(i)* $|V_q| < \epsilon$, meaning that almost all query nodes have been matched and *(ii)* the fragment set B is empty, given that a fragment is removed from B once chosen during the selection step.

Query Update. The foremost role of query update step is to ensure complementarity among selected fragments. The query is updated by pruning elements that are covered by the selected fragment to enable subsequent selections to choose fragments covering complementary parts of the query. The updating can be done in two distinct ways: using either Maximum Common Subgraph (MCS), or Weight Update (WU).

Query update using MCS. As aforementioned, the query is updated when the best fragment f_{max} is found. First, the MCS between q and f_{max} is computed, then the MCS is withdrawn from q while keeping the boundary nodes that belong to the MCS as follows:

let MCS(q, f_{max}) denote the maximum common subgraph between q and f_{max}, and let q' denote the updated query q. The vertex set of q' is $V_{q'} = V_1(q') \cup V_2(q')$, where $V_1(q') = \{v | v \in V_q \,\&\, v \notin MCS(q, f_{max}\}$ and $V_2(q') = \{u | (u, v) \in E_q \,\&\, v \notin MCS(q, f_{max}\}$. On the other hand, the edge set of q' is given by $E_{q'} = \{(u, v) \in V_{q'}^2 \mid (u, v) \in E_q\}$.

Query Update using Weight Update (QUWU). Computing the maximum common subgraph is known to be NP-complete by reduction from the maximum clique problem. Hence, it is cost prohibitive to use it in a repetitive operation as the query update. Alternatively, we perform a weight update on the query vertices with a polynomial time complexity. The QUWU process builds on the assumption that all query vertices are initially weighted by 1, where a node weighted by 1 stands for an uncovered query node, and if weighted by 0, the node is said covered. As described in Algorithm 3, QUWU works in 2 steps:

1. Weights on covered vertices are set to 0.5 (line 1 to 6). At this point of QUWU algorithm, *i.e.*, prior to the weight binarization step, only vertices belonging to the MCS between q and f_{max} will be weighted by 0.5. The equivalence of MCS and this first step of QUWU is further depicted in Figure 7.2. It is straightforward that this step of QUWU has, in the worst case, a polynomial time complexity (w.r.t. the number of vertices in the query and the fragment) provided that function Q has a linear time complexity (w.r.t. the number of query vertices $O(n_q)$). Whereas, the MCS problem is known to be NP-complete and related algorithms have either an exponential or factorial time complexity [144].

2. The query q goes through a weight binarization process (described in algorithm 4) where: the nodes weighted by 0.5 and having adjacent vertices all weighted either by 0.5 or 0 will have their weights set to 0, and the nodes weighted by 0.5 and have at least one adjacent vertex weighted by 1 will have their weights set to 1.

The stopping condition of the selection step would be that all vertices of q are weighted by 0, *i.e.*, $W \leq \epsilon$, where $W = \sum_{v \in V_q} w(v)$.

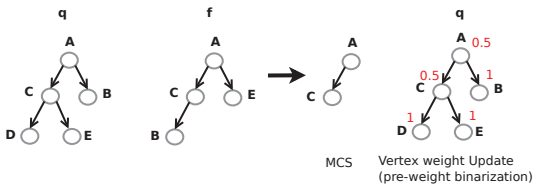


FIGURE 7.2: Vertex weight update vs. Maximum Common Subgraph.

Since QUWU does not reduce the query size as it does not withdraw vertices but update their weights, it is necessary to tweak the similarity function s in order to consider only vertices that are weighted by 1, *i.e.*, not yet covered.

Similarity function using QUWU. The above-mentioned similarity function s is slightly modified in order to take into account the weight condition on the query when comparing it to a fragment. In other words, when computing the similarity between fragment f and query q, the function s should consider only nodes and edges that are weighted by 1 and omit the remaining ones. Consequently, selected fragments would be complementary. When using weight update, the two components of the similarity function, Δ_L and Δ_D are computed as follows:

$$\Delta_L(f_i, q) = \frac{1}{n_i}. \sum_{\substack{v \in V_{f_i} \\ w(Q(v)) \neq 0}} J\left(\mu(v), \mu(Q(v))\right) \tag{7.5}$$

And Δ_D becomes:

$$\Delta_D(f_i, q) = \frac{1}{n_i}. \sum_{\substack{u,v \in V_{f_i}, u<v \\ w(Q(v)) \neq 0}} \delta_t(d(u,v), d(Q(u), Q(v))) \tag{7.6}$$

Algorithm 3 *Query Update using Weight Update (QUWU)*

Input: Query graph q, Fragment graph f.
Output: Query graph q with updated weights.

1. **foreach** $(u, v) \in E_f$:
2. $u' = Q(u), v' = Q(v)$;
3. **if** $(u', v') \in E_q$, then :
4. $w(u') = w(v') = 0.5$;
5. **end foreach**
6. weight_binarization(q);
7. **return** q;

7.3.2 Aggregation step

The fragments obtained upon successful completion of the selection step are stored in the solution set denoted by S. The aggregation step constructs an aggregate a_i from the solution set S as follows: $V_{a_i} = \bigcup_{f_i \in S} V_i$ and $E_{a_i} = \bigcup_{f_i \in S} E_i$.

Algorithm 4 *Weight_binarization*

Input: Query graph q.
Output: Query graph q with binary weights.

1. **foreach** $v \in V_q$:
2. **if** $w(v) = 0.5$ **and** all neighbors of v are weighted either by 0 or 0.5, then:
3. $w(v) = 0$;
4. **end foreach**
5. **foreach** $v \in V_q$:
6. **if** $w(v) = 0.5$ **and** there are neighbors of v weighted by 1, then:
7. $w(v) = 1$;
8. **end foreach**
9. **return** q;

7.3.3 Refinement step

The last step consists in improving the quality of the aggregate a_i. This is achieved by *(i)* connecting the aggregate whenever it is disconnected, *i.e.*, when there are unmapped edges, our method maps them to paths of a given length; and *(ii)* by pruning irrelevant nodes and edges from a_i.

Connecting the aggregate. Cases when A is disconnected may occur when selected fragments cover disjoint areas of query q leaving some query edges unmapped. Our algorithm maps these edges to paths by performing a path search in B in order to interconnect the Connected Components (CCs) in a_i. To this end, we search for a path between any two nodes belonging to different CCs of a_i in the graph G, where G is a weighted graph such that all nodes and edges are weighted by 1, and we have $G = \bigcup_{f_i \in B} f_i$. The path search stops either when a_i becomes connected or when no path could be found. For the path finding task, we used the Dijkstra algorithm and added a constraint relatively to the maximal path length *s.t.* paths having a length that is greater to a fixed threshold are ignored. That is to say, we consider only paths that have a relatively small length in order to preserve a semantic relevance (a long path is considered as semantically irrelevant). As the Dijkstra algorithm searches incrementally for paths, when a path length exceeds the threshold, the search for this path is abandoned, and as a result, the algorithm takes lower time to execute.

Pruning. The aggregate a_i is further refined by withdrawing irrelevant nodes. To do so, we iteratively prune irrelevant leaves from a_i until no irrelevant leaf is left, where a leaf is a vertex of degree 1. As depicted in figure 7.3, not all irrelevant nodes are pruned from A by the end of the pruning process, however, by limiting the pruning to irrelevant

leaves, the connectivity of the aggregate is preserved as the removal of an irrelevant node with a degree>1 may disconnect the aggregate.

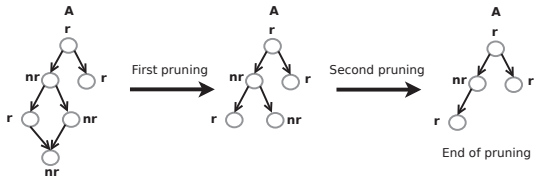


FIGURE 7.3: Pruning process of the aggregate.

7.4 Experimental evaluation

In this section, we evaluate the performances of the proposed approach by analyzing its stability w.r.t. its inner parameters. Then we confront its performances to recently proposed and related works. We first describe the experimental set up along with the graph dataset and the evaluation methodology. Then we present the evaluation metrics. Finally, we present and discuss the obtained results.

7.4.1 Experimental set up

Graph Datasets. In order to evaluate our method, we used three real-life datasets: **(i)** DBpedia Knowledge Base [145] that consists of RDF triples extracted from Wikipedia. We considered a total of 666043 triples from entities of Senator, anime, award, beauty queen, castle, chess player, film festival, Hollywood cartoons, novel and Olympics data. **(ii)** IMDB Network [146] is the internet movie database that consists of entities of movies, tv series, actors, directors, producers... and their relationships as well. **(iii)** YAGO Entity Relationship Graph [147] (Yet Another Great Ontology) is a knowledge base with information extracted from wikipedia, WordNet and GeoNames with a total of 120 million triples.

Query generation. Two parameters are necessary to generate a query q: the number of nodes denoted by n and the query diameter d which represents the largest distance between any two nodes. A query is then generated by extracting a subgraph from the dataset and introducing some label and structural noise to it. The label noise is generated by randomly modifying some words in the original label, while the structural noise is provided by randomly removing or adding some edges.

Fragments generation. In order to form the fragments set B, we partition each of the three datasets in such a way that no part is isomorphic to the query q with a size

randomly ranging from 5 to 10 nodes, and the obtained parts constitute the fragments. To avoid crossing edges between fragments, nodes belonging to the crossing edges are duplicated in the fragments involved.

Evaluation metrics. The F1-measure is used as the main evaluation metric to show the effectiveness of our method. It combines the recall (R) and precision (P) where the recall shows the ratio of the correctly found node matches overall correct matches, and the precision is the ratio of correctly found matches overall found matches. F1-measure is computed as follows:

$$F1 = \frac{2}{(1/R + 1/P)}$$

In addition to the F1-measure, the runtime is also reported.

Methodology. Our experiments are based on the following guidelines: we create 4 query sets and generate 100 queries under each set. First, we limit LaSaS to select the top-1 aggregate by fixing the parameter k to 1 to further emphasize the impact of other parameters over the performances. First, we assess the influence of the label noise on our method by fixing the structural noise and varying the label noise for all the 4 query sets. Then, to assess the influence of the structural noise, we fix the label noise and vary the structural noise and run our method on all the 4 query sets. On the other hand, we vary the ratio of the matching nodes in the query, *i.e.*, the query nodes that are surely present in the fragment set B, and see the repercussion on the performances of our method. Furthermore, we run our method while varying the parameter λ of the similarity metric (see equation 7.3), to see the influence of the emphasis on either the label similarity or the structural similarity. Moreover, we run LaSaS and compare it to state-of-the-art tools: SAGA [120], NEMA [121] and BLINKS [122]. BLINKS is a ranked keyword graph search method that relies on an extensive indexing technique to accelerate the search process, and focuses solely on node labels similarity. We decided to compare our proposed framework LaSaS to BLINKS to show that our framework is different from keyword graph search and also to show that when labels on nodes and structure similarities are jointly taken into consideration, it can lead to significant improvement in the results quality.

Last but not least, we evaluate the performances of our method on different values of parameter k, *i.e.*, the number of expected aggregates for all query sets and datasets.

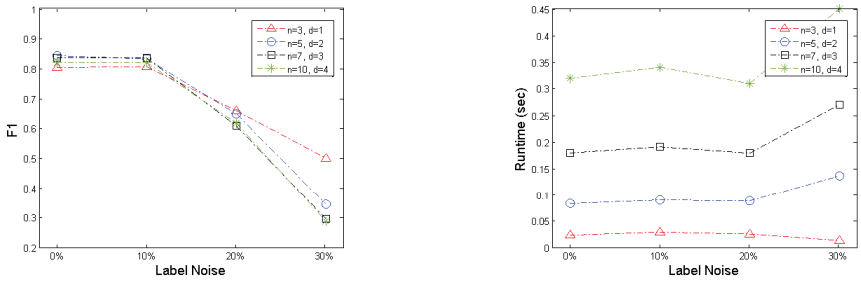
All algorithms have been implemented in C++, and all experiments were performed on a single machine, with Intel Xeon(R) CPU at 1.9GHz, and 16GB of main memory.

7.4.2 Experimental results

Label noise influence. We vary the label noise and report the F1-measure in figure 7.4-(a). Having the structural noise fixed to 10%, F1-measure does not reach 1 when label noise equals 0%. Results show that LaSaS maintains the same performances for 0% to 10% of label noise, meaning that it efficiently discovers the correct matches despite the noise. However, when the label noise goes up to 30%, F1-measure drops considerably, which shows that LaSaS is label noise sensitive.

In figure 7.4-(b), we show the runtime for all query sets while varying the label noise. Results show that for each query set, the runtime is even for different values of label noise, however, a slight increase of the runtime is noticed when the label noise reaches 30%, which is normal as the method spends additional time to look for other candidate matches. Altogether, increasing the label noise does not incur a considerable computational cost. Although not reported here due to space limitation, results on YAGO and IMDB graphs present similar trends.

FIGURE 7.4: F1-measure and Runtime (in seconds) reported on DBpedia graph, upon four different query sets with 100 query within each set while varying the label noise, having the structural noise set to 10%.



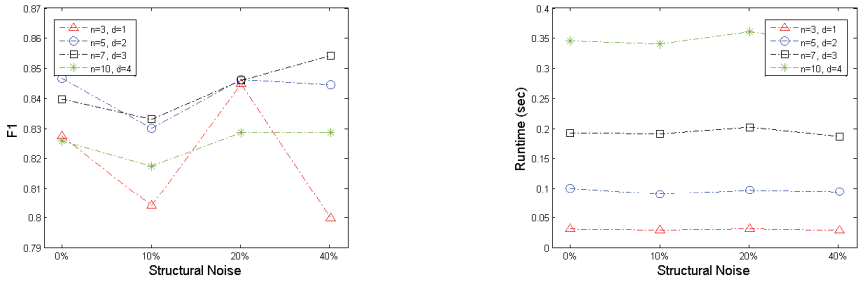
(a) *(b)*

Structural noise influence. Figure 7.5-(a) reports F1-measure for different values of structural noise while label noise was set to 10%. When no structural noise is added, the F1-measure does not attain 1 due to the label noise being set to 10%. Moreover, the F1-measure does not have an abrupt variation for all query sets (with the lower bound being 0.8 and the upper bound being 0.86), which translates the capacity of LaSaS to find the correct matches despite the added noise on the structure.

Figure 7.5-(b) reports the runtime while we vary the structural noise, and results show that the runtime is almost steady for all query sets, that is to mention that the computational cost incurred by the added structural noise is negligible.

Ratio of query matching nodes. We refer to the ratio of matching nodes that are already in the fragments set B by Φ, and we investigate its influence on the performances of LaSaS. Figure 7.6-(a) reports the F1-measure, and shows that performances

FIGURE 7.5: F1-measure and Runtime (in seconds) reported on DBpedia upon four different query sets with 100 query within each set while varying the structural noise, having the label noise set to 10%.

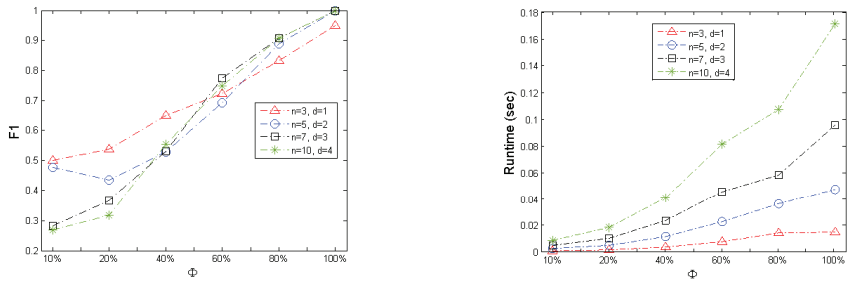


(a) *(b)*

are optimal when the ratio of the matching nodes is equal to 100%, and it decreases gradually when the ratio of the matching nodes decreases. This shows the effectiveness of our method, as it finds the best results when they are available.

The reported runtime as shown in figure 7.6-(b) increases with the ratio Φ, which means that the method takes more time to process as there are additional relevant answers in the B set.

FIGURE 7.6: F1-measure and Runtime (in seconds) reported upon four different query sets with 100 query within each set while varying the matching nodes ratio in the B set.



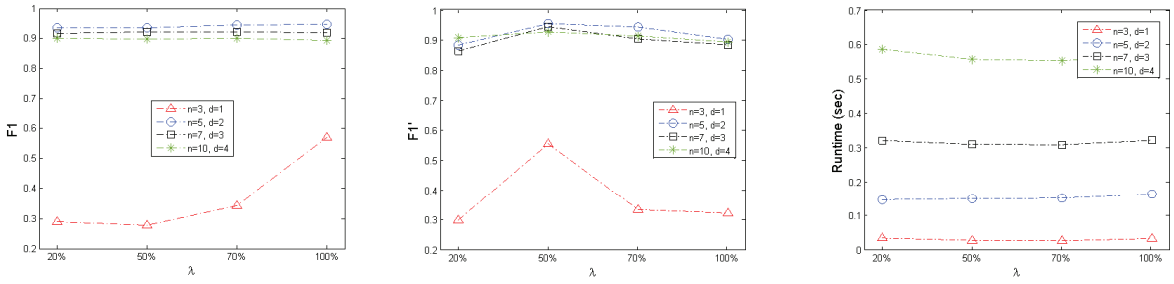
(a) *(b)*

Influence of parameter λ. We investigate the influence of parameter λ of the similarity metric (see equation 7.3).

For all query sets, the F1-measure reported in figure 7.7-(a) increases when λ increases (We recall that increasing λ means that the emphasis is given to the label difference Δ_L when computing the similarity metric). This is straightforward as the emphasis on the label similarity will favor nodes that have a similar label, hence increasing the F1-measure. In figure 7.7-(b), we show **F1'** which reports the F1-measure but in terms of *graph* matches instead of node matches. We see that we obtain the best performances when λ equals 0.5, *i.e.*, when the label similarity and structure similarity are equally considered or given the same importance.

Figure 7.7-(c) reports the runtime when varying the parameter λ. The runtime is even for all query sets as it does not incur an additional computational cost. It is also important to notice that we obtain similar results over IMDB and YAGO datasets.

FIGURE 7.7: F1-measure in terms of node matches, graphs matches and runtime reported upon four different query sets with 100 query within each set while varying the parameter λ.

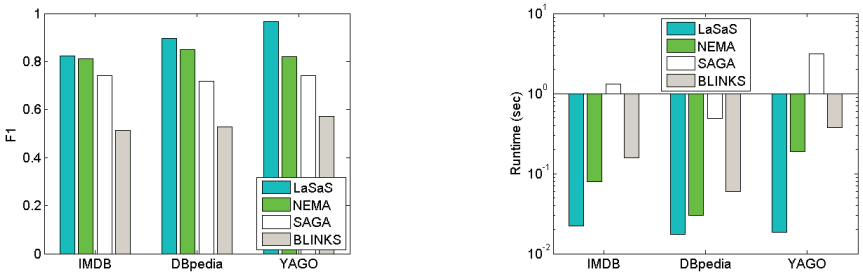


(a) *(b)* *(c)*

LaSaS *vs.* SAGA, NEMA and BLINKS. In figure 7.8, we show the comparison results of our method LaSaS to three state-of-the-art tools: *(i)* SAGA [120] and *(ii)* NEMA [121], two subgraph matching tools for approximate graph matching, and *(iii)* BLINKS [122], a graph querying tool based on keyword search.

Results in figure 7.8 show that our proposed method LaSaS outperforms all the three competitors in effectiveness by achieving greater F1-measure (figure 7.8-(a)), and also in efficiency by finding results faster (figure 7.8-(b)).

FIGURE 7.8: F1-measure in terms of node matches and runtime of LaSaS, NEMA, SAGA and BLINKS over all datasets. Results are reported for query set 3 (n=7,d=3), where $k = 1$ and label and structural noise were set to 10%.



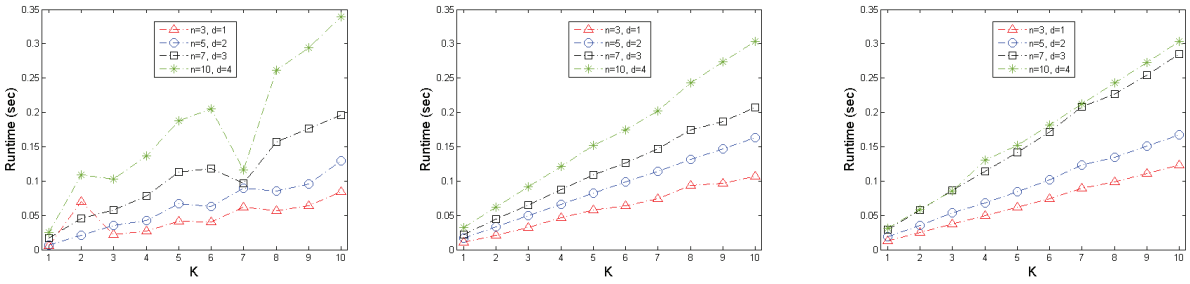
(a) *(b)*

Number of aggregates k. In figure 7.9, we report the runtime on all four query sets over each graph dataset while varying the number of expected aggregates k.

Generally, the runtime increases with the increasing k, as the algorithm takes more time to search for additional aggregates. However, as the number of fragments in the B set decreases whenever k increases, the selection step in building the aggregate gets faster, this is shown by the fact that in the majority of cases, the time for finding k aggregates

is lower than finding one aggregate k times. We skipped reporting the F1-measure while varying k, however, we note that the F1-measure decreases with the increasing k as the B set of fragments runs out of potential fragments for building good aggregates.

FIGURE 7.9: Runtime reported upon four different query sets for all three datasets while varying the parameter k (the number of expected aggregates).



(a) *DBpedia* *(b)* *IMDB* *(c)* *YAGO*

7.5 Chapter summary

In this chapter, we discussed a novel framework for approximate graph matching based on aggregated search called Label and Structure Similarity Aggregated Search (LaSaS). The proposed approach enables an effective graph querying without any knowledge of the schema of the data graph. The framework joins ideas from aggregated search and graph matching to effectively find approximate matches for a query from a set of heterogeneous graphs. LaSaS is based on three key ideas: *(i)* aggregated search strategy in order to enrich the set of answers, *(ii)* a lightweight graph similarity metric that takes into account both the nodes label and graph structure similarity to enable finding approximate matches and *(iii)* a graph weight update that replaces the maximum common subgraph search task and reduces the complexity cost. Our method enables approximate matching by allowing slight label difference and by mapping edges to paths with a thresholded length. This feature makes our method unrestrictive and hence enables it to find more answer results compared to existing strict matching schemes like graph isomorphism. Empirical evaluation over the three real-life used datasets, illustrates the effectiveness of our method over different parameter settings and corporates the stability of LaSaS. Moreover, the experimental results indicate that the proposed method outperforms the state-of-the-art related approaches by finding more precise matches. Future works will be conducted to build an index on the query and fragments to further accelerate the process of the selection step. We also plan to extend the empirical study to compare our method to additional matching tools.

Chapter 8

Conclusion and Perspectives

In this thesis, we addressed two important problems arising in the graph theory field. First, we tackled the problem of *graph partitioning*, specifically for massive graphs and in a streaming setting, which is a lightweight memory-consuming setting. Second, we addressed the *aggregated graph search* problem, which consists in simultaneous querying of several target graphs and provides an answer to the query through a coherent combination of the target graphs.

Our main contributions are: *(i)* a new heuristic named Fractional Greedy for streaming graph partitioning, achieving competitive results with state-of-the-art heuristics. *(iii)* A new restreaming model called the Partial Restreaming Partitioning model, which is a hybrid streaming model that combines a one-pass and multiple-passes streaming partitioning strategies, with the goal of reducing the computation costs while maintaining similar performances as in the full restreaming setting. *(iv)* We proposed a novel heuristic named Streaming Metis Partitioning (SMP). The proposed heuristic combines two powerful features for optimal graph partitioning: the accurateness of the multilevel method Metis, and the lightness of the streaming setting. *(v)* Last but not least, we proposed a new graph querying framework named LaSaS (Label and Structure similarity using aggregated Search), that is based on the concept of aggregated search.

We would like to point out that the objective function in streaming partitioning heuristic really influences the overall performances of the heuristic. For this reason, our proposed Fractional Greedy heuristic outperforms other state-of-the-art streaming heuristics, where its special penalizing term (in the objective function) plays an essential role in reducing the edge-cut while preserving exact balance.

An important finding in the thesis is regarding the partial restreaming partitioning. Instead of restreaming th whole graph, one can restream just a part of it and still

can achieve almost similar results, with the important advantage of reducing time and memory consumption considerably. Moreover, the streaming partitioning results depend on the first nodes being processed, as they will influence the partitioning of the remaining nodes by attracting their neighbors. This is shown in the selective partial restreaming model, where the results considerably improve when restreamed portions are selected based on sensed criteria.

It is noteworthy that, in the context of graph partitioning, the combination of a powerful offline heuristic and a lightweight and fast online setting into one hybrid approach that we call SMP, has achieved a fair tradeoff between precision and computation cost. In fact, studies in this thesis showed that SMP is faster than its offline counterpart *Metis*, theoretically and empirically. Moreover, the results quality between the offline Metis and SMP are closely similar.

Furthermore, the theoretical and empirical studies on the LaSaS framework, stresses the fact that the application of aggregated search concept in graph databases brings many benefits to classical graph querying task. In fact, the query dispatching on several graphs and the answer building from numerous fragments allows additional relevant answers to be found.

An interesting follow-up to our work would be to extend our theoretical and experimental results on the streaming graph partitioning problem, especially on investigating performance guarantees under a given streaming order for the SMP method. Another interesting direction for future work would be the study of the convergence of the partial retreaming model, as well as the investigation on other selection criteria for the selective partial restreaming partitioning. For the aggregated graph search framework LaSaS, it would be interesting to extend the empirical study by using additional graph datasets, and by comparing with more graph querying tools.

Bibliography

[1] A. Kaveh and B. Alinejad. Hypergraph products for structural mechanics. *Advances in Engineering Software*, 80:72 – 81, 2015. ISSN 0965-9978. doi: http://dx.doi.org/10.1016/j.advengsoft.2014.09.017.

[2] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), December .

[3] J. J. Sylvester. On an application of the new atomic theory to the graphical representation of the invariants and covariants of binary quantics, with three appendices. *American Journal of Mathematics*, 1(1):64–104, 1878. ISSN 00029327, 10806377. URL http://www.jstor.org/stable/2369436.

[4] https://en.wikipedia.org/wiki/graph_(discrete_mathematics). .

[5] M. A. Eshera and K. S. Fu. An image understanding system using attributed symbolic representation and inexact graph-matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(5):604–618, Sept 1986. ISSN 0162-8828. doi: 10.1109/TPAMI.1986.4767835.

[6] https://www.statista.com/statistics/346167/facebook-global-dau/. .

[7] https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/.
.

[8] http://www.worldwidewebsize.com. .

[9] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2010.

[10] http://research.microsoft.com/trinity, jan 2012. .

[11] 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 ACM SIGMOD International Conference on Management of Data 2010*.

[12] http://giraph.apache.org.

[13] L. Hyafil and R. Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. *Technical Report 33, IRIA – Laboratoire de Recherche en Informatique et Automatique*, 1973. URL https://people.csail.mit.edu/rivest/pubs/HR73.pdf.

[14] M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237 – 267, 1976. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(76)90059-1. URL http://www.sciencedirect.com/science/article/pii/0304397576900591.

[15] K. Andreev and H. Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, 2004. ISBN 1-58113-840-7.

[16] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.

[17] Guy Even. Fast approximate graph partitioning algorithms. *SIAM J. Comput.*, 28(6):2187–2214, August 1999. ISSN 0097-5397. doi: 10.1137/S0097539796308217. URL http://dx.doi.org/10.1137/S0097539796308217.

[18] U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM J. Comput.*, 2002.

[19] Robert Krauthgamer, Joseph (Seffi) Naor, and Roy Schwartz. Partitioning graphs into balanced components. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2009.

[20] Umit Catalyurek and Cevdet Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 28–28, New York, NY, USA, 2001. ACM. ISBN 1-58113-293-X. doi: 10.1145/582034.582062. URL http://doi.acm.org/10.1145/582034.582062.

[21] David A Papa and Igor L Markov. Hypergraph partitioning and clustering., 2007.

[22] Aleksandar Trifunović and William J Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *Journal of Parallel and Distributed Computing*, 68 (5):563–581, 2008.

[23] Bas Fagginger Auer and Rob H Bisseling. Abusing a hypergraph partitioner for unweighted graph partitioning. *Graph Partitioning and Graph Clustering*, 588: 19–35, 2012.

[24] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3): 75–174, 2010.

[25] Mark EJ Newman. Community detection and graph partitioning. *EPL (Europhysics Letters)*, 103(2):28003, 2013.

[26] Hao Li, Gary W Rosenwald, Juhwan Jung, and Chen-Ching Liu. Strategic power infrastructure defense. *Proceedings of the IEEE*, 93(5):918–933, 2005.

[27] Juan Li and Chen-Ching Liu. Power system reconfiguration based on multilevel graph partitioning. In *PowerTech, 2009 IEEE Bucharest*, pages 1–5. IEEE, 2009.

[28] Björn H Junker and Falk Schreiber. *Analysis of biological networks*, volume 2. John Wiley & Sons, 2011.

[29] Ulrich Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. *Geoinformation und Mobilität-von der Forschung zur praktischen Anwendung*, 22:219–230, 2004.

[30] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed time-dependent contraction hierarchies. In *SEA*, pages 83–93. Springer, 2010.

[31] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In *Proceedings of the 10th International Conference on Experimental Algorithms*, SEA'11, pages 376–387, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-20661-0. URL http://dl.acm.org/citation.cfm?id=2008623.2008657.

[32] Dennis Luxen and Dennis Schieferdecker. Candidate sets for alternative routes in road networks. *J. Exp. Algorithmics*, 19:2.7:1–2.7:28, January 2015. ISSN 1084-6654. doi: 10.1145/2674395. URL http://doi.acm.org/10.1145/2674395.

[33] Daniel Delling and Renato F Werneck. Faster customization of road networks. In *International Symposium on Experimental Algorithms*, pages 30–42. Springer, 2013.

[34] Bo Peng, Lei Zhang, and David Zhang. A survey of graph theoretical approaches to image segmentation. *Pattern Recogn.*, 46(3):1020–1038, March 2013. ISSN 0031-3203. doi: 10.1016/j.patcog.2012.09.015. URL http://dx.doi.org/10.1016/j.patcog.2012.09.015.

[35] K Santle Camilus and VK Govindan. A review on graph based segmentation. *International Journal of Image, Graphics and Signal Processing*, 4(5):1, 2012.

[36] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure*. Springer Science & Business Media, 2011.

[37] R. Andersen and Y. Peres. Finding sparse cuts locally using evolving sets. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, 2009.

[38] W.E. Donath and A.J. Hoffman. Algorithms for partitioning graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972.

[39] W.E. Donath and A.J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425, 1973.

[40] Miroslav Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4): 619–633, 1975. URL http://eudml.org/doc/12900.

[41] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996. ISBN 0-8018-5414-8.

[42] Alex Pothen, Horst D Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM journal on matrix analysis and applications*, 11(3):430–452, 1990.

[43] Horst D Simon. Partitioning of unstructured problems for parallel processing. *Computing systems in engineering*, 2(2-3):135–148, 1991.

[44] Lars Hagen and Andrew B Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE transactions on computer-aided design of integrated circuits and systems*, 11(9):1074–1085, 1992.

[45] Stephen T Barnard and Horst D Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency and computation: Practice and Experience*, 6(2):101–117, 1994.

[46] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM. ISBN 0-89791-816-9. doi: 10.1145/224170.224228. URL http://doi.acm.org/10.1145/224170.224228.

[47] Richard J. Lipton and Robert E Tarjan. A separator theorem for planar graphs. Technical report, Stanford, CA, USA, 1977.

[48] https://en.wikipedia.org/wiki/matching_(graph_theory). .

[49] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM. ISBN 0-89791-816-9. doi: 10.1145/224170.224229. URL http://doi.acm.org/10.1145/224170.224229.

[50] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 1970.

[51] L. Hagen and A. B. Kahng. New spectral methods for ratio cut partitioning and clustering. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 2006.

[52] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press. ISBN 0-89791-020-6. URL http://dl.acm.org/citation.cfm?id=800263.809204.

[53] Thang Nguyen Bui and Curt Jones. A heuristic for reducing fill-in in sparse matrix factorization. Technical report, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (United States), 1993.

[54] Todd Goehring and Yousef Saad. Heuristic algorithms for automatic graph partitioning. Technical report, Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1994.

[55] P. Ciarlet and F. Lamour. On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint. *Numerical Algorithms*, 12(1):193–214, Mar 1996. ISSN 1572-9265. doi: 10.1007/BF02141748. URL http://dx.doi.org/10.1007/BF02141748.

[56] Bruce Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clother? (extended abstract). In *Proceedings of the 5th International Symposium on Solving Irregularly Structured Problems in Parallel*, IRREGULAR '98, pages 218–225, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64809-7. URL http://dl.acm.org/citation.cfm?id=646012.677019.

[57] Roy D Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency and Computation: Practice and Experience*, 3(5):457–481, 1991.

[58] Charbel Farhat and Michel Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal for Numerical Methods in Engineering*, 36(5):745–764, 1993.

[59] Gary L Miller, S-H Teng, and Stephen A Vavasis. A unified geometric approach to graph separators. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pages 538–547. IEEE, 1991.

[60] John R Gilbert, Gary L Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[61] Kevin Aydin, Mohammadhossein Bateni, and Vahab Mirrokni. Distributed balanced partitioning via linear embedding. In *WSDM 2016: Ninth ACM International Conference on Web Search and Data Mining*, 2016.

[62] C. Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. *Trans. Img. Proc.*, 5(5):794–797, May 1996. ISSN 1057-7149. doi: 10.1109/83.499920. URL http://dx.doi.org/10.1109/83.499920.

[63] Rolf Niedermeier, Klaus Reinhardt, and Peter Sanders. Towards optimal locality in mesh-indexings. *Discrete Applied Mathematics*, 117(1):211 – 237, 2002. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/S0166-218X(00)00326-7. URL http://www.sciencedirect.com/science/article/pii/S0166218X00003267.

[64] Peter Fishburn, Prasad Tetali, and Peter Winkler. Optimal linear arrangement of a rectangular grid. *Discrete Mathematics*, 213(1):123 – 139, 2000. ISSN 0012-365X. doi: http://dx.doi.org/10.1016/S0012-365X(99)00173-9. URL http://www.sciencedirect.com/science/article/pii/S0012365X99001739.

[65] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *18th ACM SIGKDD -KDD '12*.

[66] Isabelle Stanton. Streaming balanced graph partitioning algorithms for random graphs. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*.

[67] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, 2014.

[68] Ghizlane Echbarthi and Hamamache Kheddouci. Fractional greedy and partial restreaming partitioning: New methods for massive graph partitioning. In *2014 IEEE International Conference on Big Data, 2014*, .

[69] J. Nishimura and J. Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013.

[70] Ghizlane Echbarthi and Hamamache Kheddouci. Partial restreaming approach for massive graph partitioning. In *Tenth International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2014*, .

[71] Charles-Edmond Bichot and Patrick Siarry. Graph Partitioning, September 2011. URL http://liris.cnrs.fr/publis/?id=5317. ISTE-Wiley, 368 pages, 13 chapters, ISBN 978-1-84821-233-6.

[72] http://snap.stanford.edu/data/. .

[73] G. Echbarthi and H. Kheddouci. Streaming metis partitioning. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 17–24, Aug 2016. doi: 10.1109/ASONAM.2016.7752208.

[74] http://glaros.dtc.umn.edu/gkhome/node/419.

[75] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 567–580. ACM, 2008.

[76] Ulle Endriss and Umberto Grandi. Graph aggregation. *Artificial Intelligence*, 245:86 – 114, 2017. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2017.01.001. URL http://www.sciencedirect.com/science/article/pii/S0004370217300024.

[77] Daniele Porello and Ulle Endriss. Ontology merging as social choice. In *International Workshop on Computational Logic in Multi-Agent Systems*, pages 157–170. Springer, 2011.

[78] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975. ISSN 0001-0782. doi: 10.1145/361219.361220. URL http://doi.acm.org/10.1145/361219.361220.

[79] S. E. Robertson and S. Walker. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '94, pages 232–241, New York, NY, USA, 1994. Springer-Verlag New York, Inc. ISBN 0-387-19889-X. URL http://dl.acm.org/citation.cfm?id=188490.188561.

[80] Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, pages 275–281, New York, NY, USA, 1998. ACM. ISBN 1-58113-015-5. doi: 10.1145/290941.291008. URL http://doi.acm.org/10.1145/290941.291008.

[81] Vanessa Murdock and Mounia Lalmas. Workshop on aggregated search. *SIGIR Forum*, 42(2):80–83, November 2008. ISSN 0163-5840. doi: 10.1145/1480506.1480520. URL http://doi.acm.org/10.1145/1480506.1480520.

[82] Arlind Kopliku, Karen Pinel-Sauvagnat, and Mohand Boughanem. Aggregated search: A new information retrieval paradigm. *ACM Comput. Surv.*, 46(3): 41:1–41:31, January 2014. ISSN 0360-0300. doi: 10.1145/2523817. URL http://doi.acm.org/10.1145/2523817.

[83] https://fr.wikipedia.org/wiki/structured_query_language.

[84] Aditya Pal and Jaya Kawale. Leveraging query associations in federated search. In *Proceedings of the SIGIR 2008 Workshop on Aggregated Search*, volume 3, 2008.

[85] Thi Truong Avrahami, Lawrence Yau, Luo Si, and Jamie Callan. The fedlemur project: Federated search in the real world. *J. Am. Soc. Inf. Sci. Technol.*, 57(3): 347–358, February 2006. ISSN 1532-2882. doi: 10.1002/asi.v57:3. URL http://dx.doi.org/10.1002/asi.v57:3.

[86] Jamie Callan. Distributed information retrieval. *Advances in information retrieval*, pages 127–150, 2002.

[87] Mounia Lalmas. Chapter: Aggregated search. *Advanced topics on information retrieval*, 2011.

[88] Hoa Trang Dang, Diane Kelly, and Jimmy J Lin. Overview of the trec 2007 question answering track. In *Trec*, volume 7, page 63, 2007.

[89] Véronique Moriceau and Xavier Tannier. Fidji: using syntax for validating answers in multiple documents. *Information Retrieval*, 13(5):507–533, 2010. ISSN 1573-7659. doi: 10.1007/s10791-010-9131-y. URL http://dx.doi.org/10.1007/s10791-010-9131-y.

[90] Cécile Paris, Stephen Wan, and Paul Thomas. Focused and aggregated search: A perspective from natural language generation. *Inf. Retr.*, 13(5):434–459, October 2010. ISSN 1386-4564. doi: 10.1007/s10791-009-9121-0. URL http://dx.doi.org/10.1007/s10791-009-9121-0.

[91] Cécile Paris, Stephen Wan, Ross Wilkinson, and Mingfang Wu. Generating personal travel guides-and who wants them? In *International Conference on User Modeling*, pages 251–253. Springer, 2001.

[92] Christina Sauper and Regina Barzilay. Automatically generating wikipedia articles: A structure-aware approach. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, ACL '09, pages 208–216, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics. ISBN 978-1-932432-45-9. URL http://dl.acm.org/citation.cfm?id=1687878.1687909.

[93] Sihem Amer-Yahia, Francesco Bonchi, Carlos Castillo, Esteban Feuerstein, Isabel Mendez-Diaz, and Paula Zabala. Composite retrieval of diverse and complementary bundles. *IEEE Transactions on Knowledge and Data Engineering*, 26(11):2662–2675, 2014.

[94] Amartya Sen. Social choice theory. *Handbook of mathematical economics*, 3: 1073–1181, 1986.

[95] Arlind Kopliku, Karen Pinel-Sauvagnat, and Mohand Boughanem. Aggregated search: a new information retrieval paradigm. *ACM Computing Surveys*, vol. 46 (n 3). pp. 1-31. ISSN 0360-0300, 2014.

[96] Fabian M. Suchanek, Mauro Sozio, and Gerhard Weikum. Sofie: A self-organizing framework for information extraction. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 631–640, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-487-4. doi: 10.1145/1526709.1526794. URL http://doi.acm.org/10.1145/1526709.1526794.

[97] Craig Macdonald. The voting model for people search. *SIGIR Forum*, 43(1): 73–73, June 2009. ISSN 0163-5840. doi: 10.1145/1670598.1670616. URL http://doi.acm.org/10.1145/1670598.1670616.

[98] Arlind Kopliku, Karen Pinel-Sauvagnat, and Mohand Boughanem. Attribute retrieval from relational web tables. In *String Processing and Information Retrieval, 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings*, pages 117–128, 2011. doi: 10.1007/978-3-642-24583-1_12. URL https://doi.org/10.1007/978-3-642-24583-1_12.

[99] Xifeng Yan, Philip S. Yu, and Jiawei Han. Graph indexing: A frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 335–346,

New York, NY, USA, 2004. ACM. ISBN 1-58113-859-8. doi: 10.1145/1007568.1007607. URL http://doi.acm.org/10.1145/1007568.1007607.

[100] Sherif Sakr and Ghazi Al-Naymat. Graph indexing and querying: a review. *International Journal of Web Information Systems*, 6(2):101–120, 2010.

[101] Huahai He and Ambuj K. Singh. Closure-tree: An index structure for graph queries. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 38–, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2570-9. doi: 10.1109/ICDE.2006.37. URL http://dx.doi.org/10.1109/ICDE.2006.37.

[102] David W Williams, Jun Huan, and Wei Wang. Graph database indexing using structured graph decomposition. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 976–985. IEEE, 2007.

[103] Rosalba Giugno and Dennis E. Shasha. Graphgrep: A fast and universal method for querying graphs. In *ICPR (2)*, pages 112–115. IEEE Computer Society, 2002. ISBN 0-7695-1695-5. URL http://dblp.uni-trier.de/db/conf/icpr/icpr2002-2.html#GiugnoS02.

[104] Stefano Berretti, Alberto Del Bimbo, and Enrico Vicario. Efficient matching and indexing of graph models in content-based retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10):1089–1105, 2001.

[105] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc. ISBN 1-55860-470-7. URL http://dl.acm.org/citation.cfm?id=645923.671005.

[106] Andrew KC Wong, Manlai You, and SC Chan. An algorithm for graph optimal monomorphism. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(3): 628–638, 1990.

[107] David E Ghahraman, Andrew KC Wong, and Tung Au. Graph optimal monomorphism algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 10(4):181–188, 1980.

[108] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International journal of pattern recognition and artificial intelligence*, 18(03):265–298, 2004.

[109] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[110] John E Hopcroft and Jin-Kue Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 172–184. ACM, 1974.

[111] Eugene M Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of computer and system sciences*, 25(1):42–65, 1982.

[112] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. Graph pattern matching: From intractable to polynomial time. *VLDB*, 2010.

[113] Joel Brynielsson, Johanna Högberg, Lisa Kaati, Christian Mårtenson, and Pontus Svenson. Detecting social positions using simulation. In *Advances in Social Networks Analysis and Mining (ASONAM), 2010 International Conference on*, pages 48–55. IEEE, 2010.

[114] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. Capturing topology in graph pattern matching. *Proc. VLDB Endow.*, 2011.

[115] Shijie Zhang, Meng Hu, and Jiong Yang. Treepi: A novel graph indexing method. In *in Proc. of ICDE*, 2007.

[116] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. Graph indexing: Tree + delta = graph. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 938–949. VLDB Endowment, 2007. ISBN 978-1-59593-649-3. URL http://dl.acm.org/citation.cfm?id=1325851.1325957.

[117] Karam Gouda and Mosab Hassaan. Compressed feature-based filtering and verification approach for subgraph search. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 287–298, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1597-5. doi: 10.1145/2452376.2452411. URL http://doi.acm.org/10.1145/2452376.2452411.

[118] James Cheng, Yiping Ke, Wilfred Ng, and An Lu. Fg-index: Towards verification-free query processing on graph databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 857–872, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247574. URL http://doi.acm.org/10.1145/1247480.1247574.

[119] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *VLDB*, 2008.

[120] Yuanyuan Tian, Richard C Mceachin, Carlos Santos, Jignesh M Patel, et al. Saga: a subgraph matching tool for biological graphs. *Bioinformatics*, 2007.

[121] Arijit Khan, Yinghui Wu, Charu C. Aggarwal, and Xifeng Yan. Nema: fast graph search with label similarity. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, 2013.

[122] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: Ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07.

[123] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26, 2004.

[124] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM.*

[125] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. Graph homomorphism revisited for graph matching. *Proc. VLDB Endow.*, 3.

[126] Horst Bunke and Xiaoyi Jiang. *Graph Matching and Similarity*, pages 281–304. Springer US, Boston, MA, 2000. ISBN 978-1-4615-4401-2. doi: 10.1007/978-1-4615-4401-2_10. URL http://dx.doi.org/10.1007/978-1-4615-4401-2_10.

[127] Laura A. Zager and George C. Verghese. Graph similarity scoring and matching. *Applied Mathematics Letters*, 21(1):86 – 94, 2008. ISSN 0893-9659. doi: http://doi.org/10.1016/j.aml.2007.01.006. URL http://www.sciencedirect.com/science/article/pii/S0893965907001012.

[128] Brian P. Kelley, Bingbing Yuan, Fran Lewitter, Roded Sharan, Brent R. Stockwell, and Trey Ideker. Pathblast: a tool for alignment of protein interaction networks. *Nucleic Acids Res*, 2004.

[129] Zhi Liang, Meng Xu, Maikun Teng, and Liwen Niu. Netalign: A web-based tool for comparison of protein interaction networks. *Bioinformatics*, 2006.

[130] Rohit Singh, Jinbo Xu, and Bonnie Berger. Pairwise global alignment of protein interaction networks by matching neighborhood topology. In *Proc. of RECOMB'11*.

[131] Venkata Snehith Cherukuri and Kasim Selçuk Candan. Propagation-vectors for trees (pvt): Concise yet effective summaries for hierarchical data and trees. In *Proc. of the LSDS-IR'08 ACM Workshop*.

[132] Arijit Khan, Nan Li, Xifeng Yan, Ziyu Guan, Supriyo Chakraborty, and Shu Tao. Neighborhood based fast graph search in large networks. In *Proc. of the 2011 ACM SIGMOD*.

[133] Jong Wook Kim and K. Sel#231;uk Candan. Cp/cv: Concept similarity mining without frequency information from domain describing taxonomies. In *Proc. of CIKM'15*.

[134] John R Anderson. A spreading activation theory of memory. *Journal of verbal learning and verbal behavior*, 1983.

[135] Brian Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS*, 2006.

[136] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. gstore: Answering sparql queries via subgraph matching. *Proc. VLDB Endow.*, 2011.

[137] Haytham Elghazel and Mohand-Said Hacid. Aggregated search in graph databases: preliminary results. In *International Workshop on Graph-Based Representations in Pattern Recognition*, 2011.

[138] Thanh-Huy Le, Haytham Elghazel, and Mohand-Saíd Hacid. A relational-based approach for aggregated search in graph databases. In *International Conference on Database Systems for Advanced Applications*, 2012.

[139] John W Raymond and Peter Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of computer-aided molecular design*, 2002.

[140] Panagiotis Papadimitriou, Ali Dasdan, and Hector Garcia-Molina. Web graph similarity for anomaly detection (poster). WWW '08.

[141] Adam Schenker, Mark Last, Horst Bunke, and Abraham Kandel. Classification of web documents using graph matching. *International Journal of Pattern Recognition and Artificial Intelligence*, 2004.

[142] Manu Aery and Sharma Chakravarthy. emailsift: Email classification based on structure and content. ICDM '05.

[143] Michael R Garey, David S. Johnson, and Larry Stockmeyer. Some simplified np-complete graph problems. *Theoretical computer science*, 1(3):237–267, 1976.

[144] Donatello Conte, Pasquale Foggia, and Mario Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *J. Graph Algorithms Appl.*, 2007.

[145] http://dbpedia.org.

[146] http://www.imdb.com/interfacesplain.

[147] https://datahub.io/dataset/yago.