UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School

Distributed computing for large-scale graphs

Models, Algorithms, Applications

Alessio Guerrieri

Advisor

Prof. Alberto Montresor

Università degli Studi di Trento

December 2015

Abstract

The last decade has seen an increased attention on large-scale data analysis, caused mainly by the availability of new sources of data and the development of programming model that allowed their analysis. Since many of these sources can be modeled as graphs, many large-scale graph processing frameworks have been developed, from vertex-centric models such as PREGEL to more complex programming models that allow asynchronous computation, can tackle dynamism in the data and permit the usage of different amount of resources.

This thesis presents theoretical and practical results in the area of distributed large-scale graph analysis by giving an overview of the entire pipeline. Data must first be pre-processed to obtain a graph, which is then partitioned into subgraphs of similar size. To analyze this graph the user must choose a system and a programming model that matches her available resources, the type of data and the class of algorithm to execute.

Aside from an overview of all these different steps, this research presents three novel approaches to those steps. The first main contribution is DFEP, a novel distributed partitioning algorithm that divides the edge set into similar sized partition. DFEP can obtain partitions with good quality in only a few iterations. The output of DFEP can then be used by ETSCH, a graph processing framework that uses partitions of edges as the focus of its programming model. ETSCH's programming model is shown to be flexible and can easily reuse sequential classical graph algorithms as part of its workflow. Implementations of ETSCH in HADOOP, SPARK and AKKA allow for a comparison of those systems and the discussion of their advantages and disadvantages. The implementation of ETSCH in AKKA is by far the fastest and is able to process billion-edges graphs faster that competitors such as GPS, BLOGEL and GIRAPH++, while using only a few computing nodes. A final contribution is an application study of graph-centric approaches to word sense induction and disambiguation: from a large set of documents a word graph is constructed and then processed by a graph clustering algorithm, to find documents that refer to the same entities. A novel graph clustering algorithm, named TOVEL, uses a diffusion-based approach inspired by the cycle of water.

Keywords
[graph analysis,big-data,distributed computing]

Contents

I Introduction to Large-scale Graph Analysis **1**

1 Introduction **3**

2 Sources of Graphs **5**
2.1 Large-scale graphs 6
2.2 Sources of graphs 7
2.3 Characteristics of graphs 8

3 Systems for Graphs Analysis **11**
3.1 Motivations for large-scale graph processing systems 11
3.2 Requirements for graph processing systems 13
3.3 Choosing a large-scale graph processing system 14

4 Problems and applications on graphs **15**
4.1 Customer of the application 15
4.2 Timing of the application 16
4.3 Computation on or about graphs 17
4.4 Types of approaches 17

II Foundations of Graph Analysis **19**

5 Building the Graph **21**
5.1 Graphs from clean data 21
5.2 Graphs from dirty data 22
5.3 Graphs from crawls 22
5.4 Indices of graphs 23

6 Partitioning of Large Scale Graphs **25**
6.1 Characteristics of large-scale graphs 25

6.2 Introduction to graph partitioning . . . 26
6.3 Formal definitions . . . 27
6.4 Advantages of edge partitioning . . . 29
6.5 Approaches to graph partitioning . . . 30
6.5.1 Quick heuristic partitioning . . . 31
6.5.2 Specialized partitioning . . . 32
6.5.3 Partitioning via exchanges . . . 32
6.5.4 Multi-level partitioning . . . 33
6.5.5 Streaming partitioning algorithms . . . 33
6.5.6 Distributed partitioning . . . 35

7 DFEP: Distributed Funding-based Edge Partitioning 37
7.1 Distributed Funding-based Edge Partitioning . . . 37
7.1.1 Variants and additions . . . 42
7.2 Results . . . 42
7.2.1 Metrics . . . 43
7.2.2 Datasets . . . 43
7.2.3 Simulations . . . 44
7.2.4 Experiments in EC2 . . . 48
7.3 Future work . . . 48

III Computation on Large-scale Graphs 51

8 Graph Processing Systems 53
8.1 Pregel and Giraph . . . 53
8.2 GraphLab and GraphX . . . 57
8.3 Partition-centric frameworks . . . 59
8.4 Low-memory frameworks . . . 61
8.5 Frameworks for dynamic graphs . . . 62
8.6 Frameworks for asynchronous computation . . . 63
8.7 Graph databases . . . 64
8.8 Other frameworks . . . 65

9 ETSCH: Partition-centric Graph Processing 67
9.1 Introduction . . . 67
9.2 ETSCH . . . 68
9.2.1 Application examples . . . 71

9.2.2 Applicability of Etsch . . . 73
9.2.3 Partitioning schemes . . . 74
9.3 Results . . . 75
9.3.1 Datasets . . . 75
9.3.2 Hadoop . . . 76
9.3.3 Spark . . . 76
9.3.4 Akka . . . 77
9.3.5 Comparison with Blogel and GPS . . . 79
9.4 Conclusions . . . 80

IV Applications of Large-scale Graph Analytics 83

10 Applications of Graph Processing 85
10.1 Strategies . . . 85
10.2 Overview of graph problems . . . 87
10.2.1 Triangle counting . . . 87
10.2.2 Centrality measures . . . 88
10.2.3 Path computation . . . 89
10.2.4 Coloring . . . 90
10.2.5 Subgraph matching . . . 91

11 Graph Clustering for Word Sense Induction 93
11.1 Problem statement . . . 94
11.2 Related work . . . 95
11.2.1 Word sense induction and disambiguation . . . 95
11.2.2 Graph clustering . . . 96
11.3 Graph construction . . . 97
11.4 Tovel: a Distributed Graph Clustering Algorithm . . . 98
11.4.1 Data structures . . . 99
11.4.2 Main cycle . . . 99
11.4.3 Convergence criterion . . . 102
11.4.4 Rationale . . . 102
11.5 Analysis . . . 103
11.5.1 Quality at convergence . . . 103
11.5.2 Sizes at convergence . . . 104
11.5.3 Computing POUR for word sense induction . . . 105
11.6 Extensions for word sense induction and disambiguation . . . 106

11.7 Experimental results 107
11.8 Future work 109

V Concluding Remarks **111**

12 Conclusions **113**

Publications **115**

Bibliography **117**

Acknowledgments **129**

List of Tables

3.1 Timeline of events 13

6.1 Terminology of different definitions of partitioning 29

7.1 Notation 39
7.2 Datasets used in the simulation engine (1-4) and EC2 (5-7) 44

8.1 Overview of frameworks introduced in this chapter. For each framework, we list the programming model, the type of resources used by the framework, if it allows for asynchronous execution and if it can cope with dynamism in the graph 54

9.1 Datasets used with ETSCH/HADOOP and ETSCH/SPARK 76
9.2 Datasets used with ETSCH/AKKA 76
9.3 Comparing different frameworks for ETSCH, using 4 machines. 79
9.4 Comparison of running time of a single PageRank iteration in ETSCH, BLOGEL, GPS (8 `m3.large` machines) 80

11.1 Simple example: 4 documents, *Apple* is the target ambiguous word with two different senses (the fruit and Apple Inc.) 94
11.2 Notations used in the analysis of TOVEL (Section 11.5) 103
11.3 Datasets used in evaluation 107
11.4 Comparison of our approach against online disambiguation services 108

List of Figures

2.1 Structure of the World Wide Web graph, from Graph Structure in the Web [24] 7

6.1 Degree distribution of a) actor collaboration graph b) world wide web graph and c)powergrid data, from [14]. Each of these graphs follows a different power-law degree distribution 26
6.2 Vertex partitioning example: each vertex appears in one partition, cut edges connect partitions 27
6.3 Edge partitioning example: each edge appears in only one partition, while frontier vertices may appear in more than one partition 28
6.4 On the left, a graph with labels on the edges. On the right, its line graph. For each edge in the original graph a new vertex is created. Two vertices are connected if, in the original graph, the corresponding edges had a node in common 30
6.5 Illustrative scheme of multilevel partitioning algorithms, from `http://masters.donntu.org/2006/fvti/shepel/diss/indexe.htm` 34

7.1 Sample run of Step 1 and 2 of DFEP 40
7.2 Behavior of DFEP and DFEPC with varying values of K 45
7.3 Behavior of DFEP and DFEPC with varying diameter ($K = 20$) 46
7.4 Comparison between DFEP, DFEPC, JA-BE-JA and POWERGRAPH's greedy partitioning algorithm ($K = 20$) 47
7.5 Speedup of real implementation of DFEP in Amazon EC2 49

8.1 Vertex's state machine in PREGEL (from [70] 55
8.2 Sample run of max-number computation in PREGEL. Gray vertexes are inactive. (from [70] 55
8.3 View of the scope of Vertex v in GRAPHLAB: it can see and change the state of only its neighbors. (from [68]) 57

9.1 Example of vertex and edge partitioning: on the top figure, vertices are partitioned and a few *cut edges* connect vertices belonging to different partitions. On the bottom figure, edges are partitioned and a few *frontier vertices* appear in more than one partition 69
9.2 Illustrative schema of ETSCH 70
9.3 Running time of single source shortest path algorithm in HADOOP comparing a standard baseline algorithm and ETSCH, with `m3.large` machines on EC2. 77
9.4 Comparison between ETSCH and the standard PREGEL implementation in SPARK/GRAPHX 78
9.5 Scalability of ETSCH/AKKA on the larger datasets, using `m3.large` machines on EC2. For each experiment we measure the average time for a complete iteration of PageRank 80

11.1 Sample graph created from Table 11.1. Blue nodes are ambiguous. 98
11.2 Illustration of the water cycle in TOVEL 102
11.3 Average color per node in a cluster at steady state, for different quality and values of POUR, in a graph with 1000 nodes 105
11.4 Percentage of queried sentence correctly classified, against the number of sampled sentences used in the model construction. 109
11.5 F-score against iteration 109
11.6 Running time against number of machines of SPARK implementation 110

Part I

Introduction to Large-scale Graph Analysis

Chapter 1

Introduction

To a person who has studied extensively the concept of graphs and their possible applications, recognizing graphs in nature becomes trivial. Graphs can represent relations between people, show connections between places, track the spreading of ideas and the interaction between molecules. As people we daily build a network of friendship and relations, follow a network of routes to meet these people and make all these decisions through a network of neurons in our brain. Once you learn to see graphs, you recognize them everywhere.

It may be a surprise then noticing that the concept of graphs is relatively new. Euler used graphs to prove the impossibility of a solution to the Seven Bridges of Koenigsberg problem in 1736 [33], but this concept did not have a name until Sylvester in 1878 used the term "graph" for the first time [109]. In the following decades mathematicians were stimulated by problems such as the four-color problem and the graph enumeration problems, but the field did not start studying real-world graphs in depth before the advent of computer science.

Until only a few decades ago, computer science was pursuing research in more efficient graph algorithms, but the number of real-world graphs that needed to be analyzed was still small. Most research was on problems related to computer networks, such as routing and congestion detection, and problems on geographical graphs, such as path computation.

Internet changed this field completely in three different steps: first by creating a huge example of a graph, the World Wide Web. Later, by creating on top of the web many ways to allow interaction between people, thus making it possible to reconstruct the connections between people at a scale impossible with traditional methods. Finally, connecting huge number of devices via the Internet of Things created another deluge of data and gave access to an incredible variety of graphs of different sizes and nature. In the last 20 years new interesting problems and applications have emerged from new kinds of graphs, all requiring new algorithms to solve it and new systems to permit the execution of these

algorithms.

This thesis is constructed in four parts. The first part gives an high-level overview on the current situation of the field. Chapter 2 introduces where do these graphs come from, when they can be called "large-scale" and what are their defining characteristics. Chapter 3 gives an overview of the history and motivations behind the proliferation of large-scale graph analysis frameworks and what types of problems they try to solve. Chapter 4 introduces the kinds of problems and applications that have grown from these new sources of data and how they influence the choice of the system. The following three parts study in more detail each step of the processing pipeline and describes the novel research completed in that step: Part II introduces the pre-processing issues that arise whenever a graph must be constructed and the partitioning problem. Part III describes the different systems and programming models that have been developed to help the analysis of large-scale graphs. Part IV presents an overview of possible applications of large-scale graph analysis.

Chapter 2

Sources of Graphs

One of the latest trend in computer science is the emergence of the "big data" phenomenon that concerns the retrieval, management and analysis of datasets of extremely large dimensions, coming from wildly different settings. For example, astronomers need to examine the huge amount of observations collected by the new telescopes that are being built both on Earth and in orbit [49]. Biological experiments create large genomic and proteinomic datasets that need to be processed and understood to reach new breakthroughs in the study of drugs [51]. Governments can improve the quality of life of their citizens by analyzing the huge collections of individual events related to traffic, economy, health-care and many other areas of everyday life [48]. The scale of such datasets keeps increasing exponentially, moving from gigabytes to terabytes and now even to petabytes.

Several interesting datasets are structured in a way that makes them easy to be modeled as graphs with additional information labeling vertices and edges. An obvious example is the World Wide Web, but there are many other examples such as social networks, biological systems or even road networks. While graph problems have been studied since before the birth of computer science, the sheer size of these datasets makes even classic graph problems extremely difficult to solve. Computing the shortest path between two nodes needs too many resources to complete in time when the graph is too big to fit into memory.

This chapter introduces the different sources of very large graphs and their characteristics. Since research has shown that efficiency of large-scale graph analysis tools is highly dependent on the input [76], understanding what are the characteristics of the datasets is extremely important to achieve good results.

2.1 Large-scale graphs

Before discussing how to run our analysis and the information that we want to obtain from this data, we need to understand what are large-scale graphs and where do they comes from. While there are many different definition of what is "Big Data", most agree on the *3V* model [64]: a problem is "Big Data" when the volume of data, the velocity of its changes or the variety of its data makes it impossible to use traditional data processing algorithms and systems.

Trying to apply this definition in large-scale graph processing allow us to get some intuition on the particularity of this area of research inside the more general "Big Data" field. The issue of variety becomes less crucial since we are restricting the datasets to those that can be represented as graphs, while the issue of volume becomes more interesting since it could refer to both the size of the graph or the amount of data associated to it.

We can adapt the classical "Big Data" definition to say that there is a need of a large-scale graph processing approach when:

- The size of the graph has become too large: storing the vertices and edges in memory is an issue and the complexity of traditional algorithms makes them impractical for graph of this scale.
- The amount of data on the graph, such as information associated to vertices and edges, has become too large.
- The data is changing too quickly, because of addition and deletion of vertices and edges or continuous updates to the states of both.

From the *3V* model we removed the Variety requirement and split the Volume requirement in two, differentiating between the graph itself and the information on top of it. Computing a centrality metric such as PageRank on a small graph can be easily completed using traditional techniques; when the graph grows larger, however, it becomes costs-ineffective to use just a single machine. For traffic analysis on road networks the volume issue is much different: the graphs will typically be relatively small, but the amount of information associated to them, such as the load of each street at different times of day, can be so huge it makes traditional approaches unusable.

This examples also underline an important consequence of the definition: the size of the graph is not the only factor in deciding if a large-scale graph processing approach is needed. Ultimately, it is the amount of resources needed that define the "large-scale" aspect, not the number of vertices and edges. Even with a graph with millions or billions of nodes the use of large-scale techniques might not be justified if a centralized algorithm is already efficient.

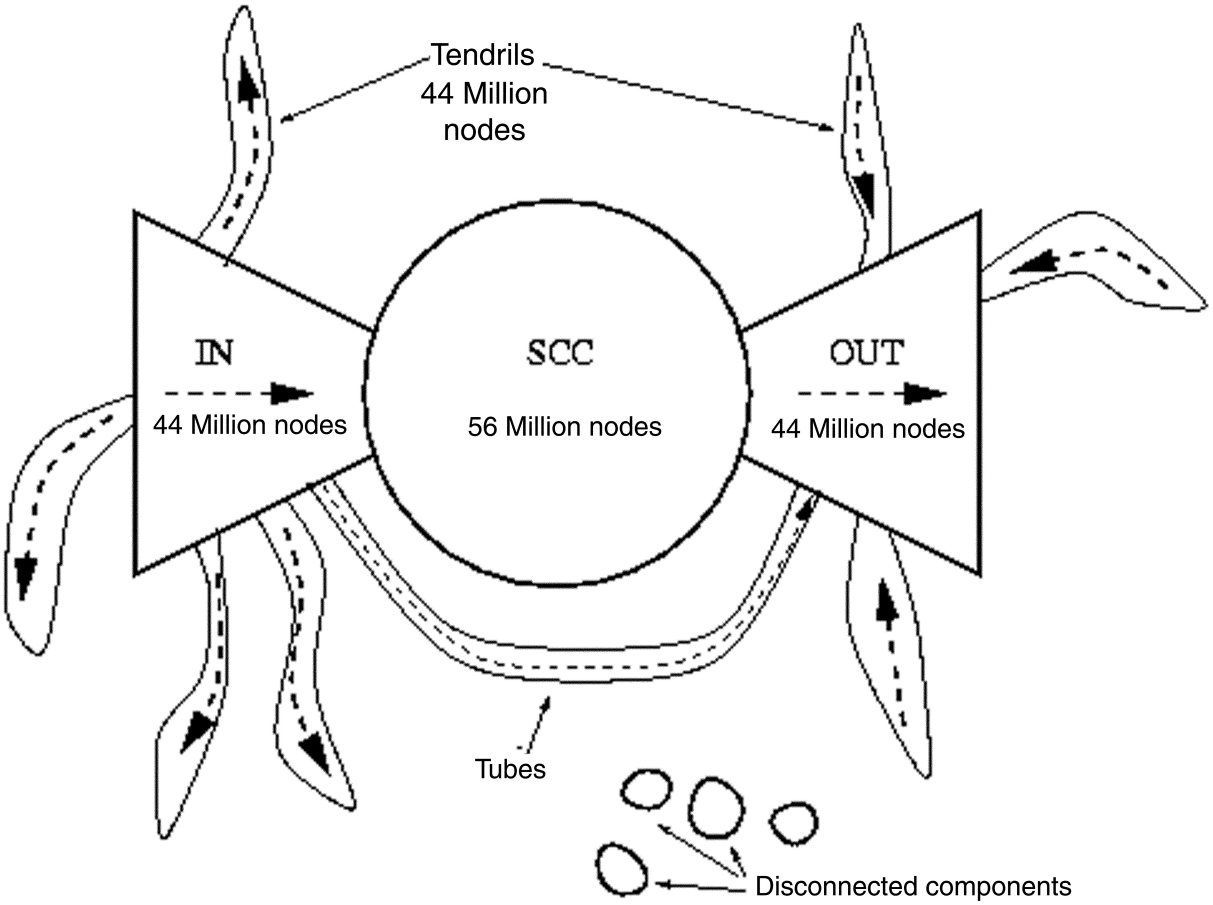


Figure 2.1: Structure of the World Wide Web graph, from Graph Structure in the Web [24]

2.2 Sources of graphs

While the Variety requirement is not as important as in the general "Big Data" setting, large-scale graphs can come from different type of sources that determine some of their characteristics. Most graphs represent activity and connections between entities (be it people or more abstract concepts) and therefore are very dynamic. Only in a few cases the large-scale graphs represents mostly static real-world connection patterns.

Most of the activity-based graphs come from interaction on the World Wide Web. Classic examples are social networks, in which people make connections on the web, or co-purchasing networks, where objects that are purchased by the same accounts are connected together. The World Wide Web itself falls into this category, since it can be seen as a series of events in which pages add links to other pages. These datasets are usually quite clean, since the activity is already stored in a structured format and, once they are retrieved, are easy to process. They can be extremely large in size, since they

are generated by all the users of the Web, but they might come with privacy issues, since they reflect possible private activity by real people. Note that even if in most cases edges and nodes are added to the graph incrementally and there might be information about timestamps of these events, the graphs are often considered as static and the timing information is simply discarded once the new nodes and edges are added to the graph.

A smaller amount of graphs come from activity of people and other entities in the real world, such as connections extracted from news, government documents or sensors deployed in the world. These datasets are not as clean as the ones coming from activity on the web, since they must be digitalized and pre-processed to extract the information needed. While in social networks there is a unique id for each person, in the real world the task of understanding which people does a document refers to becomes a challenging task by itself. Another consequence of this obstacle is that the amount of data that goes through this pipeline to reach the graph analysis computation is much smaller than in the other cases. Problems on graphs coming from these sources can still be considered "Big Data" if the amount of computation is really heavy, but in many cases only the pre-processing phase (such as the text entity recognition) is properly Big Data, while the graph processing could be done using more traditional techniques. Privacy is even more a concern, especially in cases in which the graph regards flow of money or movement tracking.

Finally, there a few graphs that refer to existing structures in the real world. While a social network can be seen as an extension of the real societal network, here we consider more physical graphs such as road networks, biological networks, food networks and similar systems. These are usually very small (in comparison with other networks) and also not easy to obtain and build. What makes these graphs "Big Data" is the amount of information that often is connected to vertices and edges in these systems. For example, road networks are small, but traffic analysis on those road networks require heavy computation and massive quantities of data. In most cases privacy concerns are less heavy in these networks, since their real world counterparts are publicly available.

Almost all applications that are explored in this thesis come from the first two categories, while analysis of physical graphs is used just to show the impact of their characteristics on their analysis.

2.3 Characteristics of graphs

The most important property that helps us to classify sources of large-scale graphs is distinguishing between *static* and *dynamic* sources. In a static scenario, the source will provide a dataset and the system will execute the chosen algorithm on it. In a dynamic

scenario, the source will continuously send data across the pipeline and both systems and algorithms will have to adapt to it. While it is possible to process dynamic graphs with static tools, it becomes infeasible when either the flow of data is too fast or the problem is time critical and delays in updating the result can be damaging.

Note that temporal graph analysis, in which an algorithm uses or studies the changes in shape of the graph through time, can be both static and dynamic. Often changes to such graphs are not that quick to force the use of dynamic techniques and the analysis can be run again from scratch whenever the application needs an updated result, but that is not always the case.

Another distinction is between homogeneous or heterogeneous graphs. Co-purchasing networks, graphs that contain information about which products are purchased together, can be stored in both ways. In an homogeneous network each node will represent a product and there will be an edge between two products if they have been purchased by the same user. In an heterogeneous network nodes can be either products or users, while edges will connects users and their purchases. This graph is heterogeneous because the nodes are not all of the same class of entity and will thus have completely different types of data associated to them.

Finally, there are differences on how the data is collected and delivered to the system. In many cases the data is collected by a single entity that will then send it to the system for the analysis. This is the classical scenario where data gathering and data processing are two completely separated steps in the pipeline. There are cases in which this approach is infeasible: when the data is collected by separated entities, be them different machines, sensors or even different coordinating companies, the collection of the data in a single place can be too costly or time-consuming. These *distributed* large-scale graph datasets must be processed in place by the system and thus need better coordination between the data source and the system.

Chapter 3

Systems for Graphs Analysis

The data analysis field has reacted in different ways to the growth in size of available datasets. As the new architectures have shown a reduced interests in faster and faster CPUs and have and focused on efficient multi-core architectures, the parallelism of the analysis has also become a crucial characteristic.

While parallel (multi-CPU, multi-core) systems have been used to deal with this deluge of data, there are many cases in which distributed approaches are the only viable road. The disadvantages of distribution cannot be ignored, though: distributed algorithms are inherently more difficult to develop and implement, and they bring a larger communication overhead. Nevertheless, the advantages outweigh the disadvantages. A distributed system is able to cope with potentially unlimited datasets, is more robust to hardware failures, is often cheaper and, with the emergence of distributed frameworks for data analysis, is also much easier to use than it was a decade ago. These new distributed frameworks abstract away most of the challenges of building a distributed system and offer the analysts straightforward programming models for their data analysis programs.

This chapter illustrates the underlying reasons for this proliferation of large-scale graph processing frameworks both in academic and industrial settings, and introduces the different types of framework and their different approaches.

3.1 Motivations for large-scale graph processing systems

The immense growth in number and complexity of large-scale graph analysis frameworks can be explained not only by the availability of data but also by the challenges posed by the analysis of these graphs.

While writing a graph analysis program can be seen as a simple task, writing an efficient, scalable and reusable system is far from trivial. Even the basic task of storing the graph in memory can create unforeseen challenges. Users must be able to quickly

obtain outgoing edges of a single node, follow those edges and recover or update data associated with those nodes or edges. Classical approaches, such as the commonly used adjacency lists, can be extremely inefficient when the size of the graph is so large that every byte of memory is important.

These challenges become even more harsh when you need to cope with multiple threads, multiple processes or even multiple machines that are trying to access the graph. Writing a system that can solve all those problems and is flexible enough to not be limited to a single algorithm is not a task that users will want to tackle, if also given access to a readily prepared framework.

Another very important factor for the proliferation of graph processing framework has been the fact that the development of these framework did not necessarily have to start from scratch. In many cases they were created as a tools inside already existing ecosystems, along side other "Big Data" tools. HAMA [102] and GIRAPH [8] started as part of the HADOOP [16] ecosystem, GRAPHX [121] was born by adding graph-specific code to SPARK [124] and GELLY as an extension of FLINK [5]. On the developers side, this opportunity made it possible to reuse already available tools and start from an already existing user base. The users were more likely to test these frameworks if they had previous experience with the ecosystem and were allowed to combine standard machine learning techniques with graph analysis algorithms and obtain data pipelines impossible in stand-alone graph frameworks. One of the main selling points of SPARK has been the fact that users could get data, clean it, pre-process it, create a graph, run a graph algorithm and finally obtain the output all in one program, without having to leave the ecosystem.

The only extremely successful standalone graph processing framework, GRAPHLAB [68], also stopped being standalone just when it had reached maturity. To widen the scope of the project, the developers kept the extremely efficient core of graph analytics and reused many techniques to build a generic machine learning oriented framework around it. This choice could signal that in the future, the place of distributed graph processing framework will be inside larger ecosystems and the competition will be between the ecosystems that host the frameworks, not on the framework themselves.

Interestingly, this pattern has not been necessarily true for non-distributed graph databases, where standalone frameworks such as NEO4J [78], ORIENTDB [83] or SPARKSEE [105] continue to thrive. The most likely reason for these exception is the lesser need of an ecosystem in a non-distributed scenario, where it is easier to manually combine different systems.

Table 3.1: Timeline of events

Year	**hadoop derivatives**	**akka derivatives**	**Other frameworks**
2004			MAPREDUCE [31] introduced
2005	HADOOP [16] introduced		
2007			NEO4J [78] introduced
2009	Amazon offers EMR	AKKA [3] introduced	GRAPHLAB [68] introduced
2010			PREGEL [70] presented
2011	GIRAPH [8] introduced	SPARK [124] introduced	
2012			TITAN [111], GRAPHCHI [63]
2013		GRAPHX [121] introduced	GRAPHLAB expands its scope

3.2 Requirements for graph processing systems

Before starting to differentiate these systems, we can start by looking at what is requested by the users. In this section we consider clients both the cluster administrators (that need to run the system on their cluster) and the programmers that use the system.

The core concept that all these systems have to take in mind is the idea that in the "Big Data" scenario efficiency is less important than manageability. In clusters with thousands of nodes and petabytes of data, efficiency is still important but secondary to the assurance that the computation will continue and data will not be lost even in presence of hardware failures in the system. Since failures are virtually guaranteed to appear, being resilient to failures is actually the most important feature of a large-scale distributed system.

A second, important property of a successful system is the ease of use of both its interface to the programmer and its interface to the administrator. The programming model should allow different levels of abstraction, to give satisfaction to programmers with different skill levels. On the administrator side, these systems should be as predictable as possible, to allow for accurate estimation of costs and time spent on different analysis.

These systems should be *scalable* in both directions. They should be able to cope with vertical scaling, an increase in the size of the data, and increase its efficiency in case of horizontal scaling, an increase in the number of nodes. Both are highly desirable properties, since we cannot assume that size of the data will remain the same. Vertical scaling guarantees that the system will not break when a larger amount of data needs to processed, while horizontal scaling gives us a way to accelerate computation in exchange of computing power.

3.3 Choosing a large-scale graph processing system

Considering the wide range of sources of graphs and applications, it is not a surprise to find that graph processing frameworks can be immensely different from each other. To find a pattern, it is necessary to study three different factors: resources, data and algorithms.

The type of resources available to the user is the biggest motivator in deciding which framework to use. From less powerful to more powerful, frameworks can target commodity level machines, shared-memory clusters and distributed systems. While it might seem counter-intuitive to create a "Big Data" framework and run it on a single cheap machine, forward-looking users might decide to adopt the framework at very early stages of development to make sure that they will be able to scale to larger quantity of data by simply upgrading the hardware and not changing the software. Having an efficient, small-scale implementation can also increase the visibility of the framework. As an example, the SparkSee graph databases not only claims to be extremely efficient on shared-memory clusters, but also the first graph database for Android, in the hope of gaining visibility from mobile developers. While there are similarities between frameworks for commodity-level machines and shared-memory clusters, frameworks for distributed systems tend to be extremely different in structure and technology because of the need of distributed file systems and message passing architectures. While distributed systems can be run on single machines, in that setting they can be useful only to develop and test the programs that will be eventually run on a large clusters.

Another important factor in choosing the right system is the type of data that the user needs to analyze. Frameworks can focus on RDF data or generic graph data. While RDF is usually represented by subject-predicate-object triples, generic graphs can contain additional data inside vertices or edges and therefore is more difficult to fit into a specialized system for RDF analysis.

Finally, a third factor is the type of algorithm that the user need to run on the framework. Each of these frameworks offer different programming models, each of them with strong and weak points. Some models can allow algorithms to be expressed using a query/transaction model or a graph-centric programming model, some might allow static (executed only once) or continuous (needs to adapt to new data in real time) algorithms. Understanding the type of analysis that will need to be done can inform greatly the choice of the system.

Chapter 4

Problems and applications on graphs

It is impossible to categorize all possible applications of large-scale graph analysis, but there are some patterns that can be evinced from how these systems are used in the industry. The best indicators for understanding which type of application are (i) who is the computation for (who will use its output); (ii) how quickly is an answer needed; (iii) whether it is directly a graph problem or a problems that needs the use of graphs; finally, (iv) the type of solution that this problem will need. As is shown in this chapter, all these properties influence each other and can be important factors in the choice of system to tackle the problem.

4.1 Customer of the application

The first important factor to consider is who wants the problem to be solved. In some cases, the application is just used internally by the owner of the data, without showing its results directly to the public. In other cases, the information gathered is then presented to the users and changes the way they interact with the system. A special case appears when the owner of the data is an information provider, a company which product is the information itself.

When the owner of the data needs to understand something about that data, there is seldom a question of timeliness. Unless the application uses large-scale graph processing to create an alert system, the owner will only need sporadic updates to keep track of patterns of change in the graph. Twitter might want to track the spread of influence between its users to develop better way to recommend connection to other users, but daily updates are not needed. Computing the community structure of its users (and the rate of activities of those community) can be helpful to encourage activity on the system, but it can be executed only when there is a specific need.

If the output is shown to the user, then there is a bigger need for it to be updated.

If a user needs to know similar items to an already bought item, they cannot receive outdated information. Periodical updates to the knowledge base require the system to be run often and to keep track of new activities by the users. Another requirement appears in this scenario: this information must be either be precomputed and readily available, or computed very quickly. More than a few seconds of delay could already be not acceptable in the modern web.

These requirements becomes even more stringent if the owner of the data is an information provider. The users of these services pay for access to the output of the information provider's analysis and require updated, thorough and quick answers to their query. Predictability is key: if an update to the data is promised for a certain day, the information provider cannot delay it without incurring in significant costs to its image. Choosing algorithm with precise running time in stable, fault-tolerant systems becomes a priority.

4.2 Timing of the application

How quickly must an answer be given to the customer is an extremely important factor that has repercussions on all subsequent choices. There is a sliding scale that goes from real-time computation to time insensitive, batch processing.

The most demanding scenario is one in which answers are expected almost on real-time. For this scenario to appear the data must appear very quickly, either directly through interaction on the web or through sensors in the real world. Systems must typically find interesting anomalies and react quickly by alerting experts or specialized systems. This type of applications is infrequent in large-scale graph processing, especially compared with its frequency in the more general Big Data scenario. Even the example of adapting shortest route computation in case of congestion (either in road or computer networks) seldom requires an amount of computation sufficient to justify large-scale graph processing systems.

In a more relaxed scenario, where efficiency is not crucial but still very important, low-level systems are at advantage over general large-scale processing frameworks. There is a trade-off between abstraction and efficiency, therefore systems that offer PREGEL's programming model will seldom be as efficient as systems that just offer fast message passing primitives. Since every problem becomes time-sensitive if the amount of data becomes too big, using lower level primitives is a valid approach whenever more general programming models are too slow to complete the computation quickly enough.

These concerns disappear whenever there is no urgency to complete the computation. If the specific application needs to be computed at specific, relaxed intervals then using generic graph computing platforms allow for a smaller effort by part of the programmers

and higher stability. There are still some concerns about efficiency, since executing in less time would mean smaller costs, but they are less pressing than in other scenarios.

4.3 Computation on or about graphs

Another important factor that influences the choice of system to be used is how this application uses graphs in its computation. It can either study a graph to obtain some insight, or it can build and use a graph as a tool to highlights connection in unstructured data.

As example, computing the centrality of accounts in a social network is a problem that studies properties of a specific, already existing graph. Finding paths in such a network needs only the graph itself and does not need any steps before or after the actual graph computation.

This is not true whenever graphs are used as tools inside a larger system. Chapter 11 presents an approach that creates a network of words from unstructured documents, and clusters the resulting graph to get insight about the properties of those documents. Even simpler graphs, such as co-purchasing networks, are built from information that needs pre-processing to be seen as graphs.

The biggest consequence of this factor is the choice of a stand-alone, specialized graph processing system against a more general system that can be connected with other big data frameworks inside an ecosystem. If the application needs pre- or post-processing before and after the graph computing phase, the user might well choose a slower graph computing framework that at least lives inside a large ecosystem of big data frameworks.

4.4 Types of approaches

Even if the problem that needs to be solved is novel, it is usually the case that the developers know which type of approach they will need. Knowing if it is a problem that can be solved from local informations, from knowledge about paths or from the similarity between nodes or edges gives an indication of what type of systems the user should use.

Problems that can be solved with local information, such as queries about the degrees of nodes or finding nodes with specific characteristics can typically be solved by large-scale graph databases. RDF queries are quite powerful and modern graph databases can process these queries at a speed that would not be reachable by more general graph processing frameworks.

A different class of problems contains those that involve paths in the graph. These problems include not only distance computation, but also all centrality measures that

define the position of a node by looking at the paths in which the node is involved. These problems typically needs many more iterations to converge, but often the running time depends on the diameter of the graph. On small-world graphs, such as most social interaction networks, the diameter is small enough to limit the loss of efficiency caused by having to synchronize all the machines at the start of each iteration. In cases where diameter is large there are two possible solutions: either use more specialized systems (such as frameworks for analysis of geographical graphs) or frameworks with asynchronous programming models.

Finally, a broad class of problems regard similarities between nodes, between groups of nodes or between subgraphs. This class contains problems such as finding similar nodes (clustering), finding recurring similar graphs (frequent subgraph mining) and studies about communities of nodes. These problems have similar requirements to the path-based problems, but it is usually harder to find solution that can be easily implemented on top of commonly used vertex-centric programming models.

Part II

Foundations of Graph Analysis

Chapter 5

Building the Graph

The first step across the pipeline that connects the origin of the data to the results given to the users is the extraction and pre-processing of the graph. This process can be trivial in cases where the graph is already natively stored by the owner of the original data, but it can become a challenge by itself when the source data is unstructured, in a remote location, or is a composition of different sources.

In this Chapter we give an overview of the challenges in obtaining the data, constructing the graph and preparing for the analysis of the graph.

5.1 Graphs from clean data

In the least challenging scenario the graph is constructed from structured data owned by the analyst itself. For example, Facebook or Google might want to analyze the social network of its users or study their activity.

If the data is well-structured and the connections between the entities are already explicit in the data, then the problem is trivial. Social networks such as Facebook or Twitter have a complete view of their users and the connection between them, without any ambiguity. Networks that can be indirectly inferred from structured data, such as co-purchasing networks, can also be constructed without complications.

Challenges may arise when the analyst wants to also use unstructured data to add information to the graph. Connections in Google Plus can be grouped in circles, but the meaning of each circle (usually described by a single word) is different for each user. This problem gives a foretaste of the difficulties of building a graph in a less clean scenario.

From a practical side, since in most cases the owner of the data will use the same infrastructure for the storage of its data and its analysis (a locally managed cluster or rented machines from the cloud), the costs of the pre-processing may be minimal and the entire pre-processing activity only involves the conversion of data from the storage system

to a format readable by the graph processing system.

5.2 Graphs from dirty data

While the previous setting is frequent, there are many complications that may arise when the graph must be constructed from less clean sources or if the owner of the data is not the same as the analyst.

These challenges appear most often when the analyst is using data coming from different sources; in such cases, hoping that companies will use a common ontology to define the connections inside their data is futile. Consider, for example, the case in which two different datasets containing corporate positions of workers in companies must be integrated. The terminology used by the two sources to define the type of position will often be different and creating a mapping between the two terminologies to reconciliate the two datasets will be a thankless job that will be probably be assigned to a human worker.

This setting can be even more challenging when the graph is in an unstructured form, such as when a network of interactions must be extracted from text data. If we want to create an edge between two entities when they co-appear in a piece of news, the first problem to solve is understanding which sequence of words are entities. Then those entities must be mapped to the ontology used by the system, making sure to differentiate between entities that have different meaning but same text (such as people with the same name). Extracting the type of connection from the text is only the last step to obtain a well-structured and meaningful graph [77] [35].

Each of these steps can and will introduce errors in the graph. The analysts need to carefully consider the impact of these errors on their applications and if they might invalidate their results. Not all errors are equals: services that collect news regarding different companies are more worried about mistakes in the data regarding its largest users, while errors in recognizing small, local companies can be less crucial.

5.3 Graphs from crawls

Graphs can be difficult to build for an orthogonal reason: the data may be well-structured, but not directly available. If the data is distributed between different entities and is not given directly to the analyst, building the graph becomes an issue.

In a few important cases, the analysts have an interface with the data that allows them to collect all information regarding a specific entity, but not global information (such as the list of all entities). It is quite easy to extract all outlinks of a webpage, but reconstructing the World Wide Web graph is a very difficult task, not only because of its

size, but also because of the difficulty in obtaining a list of all web pages. These graphs must be reconstructed via crawling techniques that start from known entities and try to discover the rest of the network by following all links.

There is much research about the different techniques that can be used to crawl efficiently such a network [13]. A really big challenge in this area is the risk of creating bias in the resulting graph. Studies have shown that traditional, BFS-based crawling algorithms tend to discover more easily high-degree nodes over nodes with smaller degree. The graph that is obtained from such a crawl can have completely different characteristics than the original graph [43].

This kind of data can also become outdated very quickly. By the time the crawl is completed, the oldest vertices that have been recovered may be too old to be used. Instead of having a snapshot of the graph, the crawled graph will be a continuously changing mix of updated and outdated vertices and edges. Online graph algorithms that do not need to start from scratch are especially useful to allow the fast computation of updated results.

5.4 Indices of graphs

A useful step that should be taken before the start of the computation is managing the identifiers of the vertices in the graph.

Often the identifiers of the entities in the original data are in the form of strings, such as URLs representing web pages or social security numbers representing tax payers. Since most systems require integer identifiers, the pre-processing step should also compute those identifiers: this is trivial on small graphs, but can be a challenge on large distributed ones.

In some applications, this step is not strictly necessary, since vertices might be already defined via unique, non-consecutive integer identifiers, but their direct use in the system would cause many inefficiencies. If the computing system uses a standard edge list implementation to store the graph, it will also need a Map that connects each id to the position of the edges of its node. If the identifiers were consecutive integer between 0 and N-1, a simple array would have solved the problem, but in the case of non-consecutive identifiers a much more costly hash map is needed.

Some systems already implements this pre-processing step internally, but many are more efficient if the graph already has integer ids. While transforming string id to integer is trivial, mapping a huge sequence of strings to a set of consecutive integers is computationally heavy since it requires at least one sorting of the edge set. If the system is able to internally use such mapping the costs of this pre-processing is justified by the increase in efficiency in the computation phase, otherwise using non-consecutive identifiers might still be the easiest route.

Chapter 6

Partitioning of Large Scale Graphs

This chapter introduces the problem of partitioning graphs to allow distributed and parallel computing systems to execute independently on each partition. While this task might not be as useful for some parallel systems, it is a crucially important step for all distributed frameworks.

6.1 Characteristics of large-scale graphs

This section describes the characteristics of the graphs constructed during the pre-processing phase. These characteristics have been under deep scrutiny since it was discovered that real-world graphs are extremely different from completely random graphs. The seminal paper of Watts and Strogatz [120] looked at two metrics that differentiate real-world graphs: the average path length between random nodes and the clustering coefficient. Real-world graphs with the "small-world" property have a very small average path length and a very high clustering coefficient.

The fact that the average path length is small in real network was not a surprise since a very famous experiment executed by Milgram in 1967 [73] showed that it was possible to connect random persons using only five connections on average. The classical random graph model by Erdos and Renji [32] also generates graphs that have this property, but what was unknown was the very high clustering coefficient of these graphs.

The clustering coefficient of a graph measures how often, if two nodes have a common neighbor, they will have a edge connecting them. It has a huge impact on the dynamics of the graph: graphs with high clustering coefficient will have a much larger number of triangles than random graphs and therefore will be more connected. Watts and Strogatz showed that, according to all models of spreading of diseases, graphs with this property are much more vulnerable.

Another important property of most real-world networks is the presence of a power-law

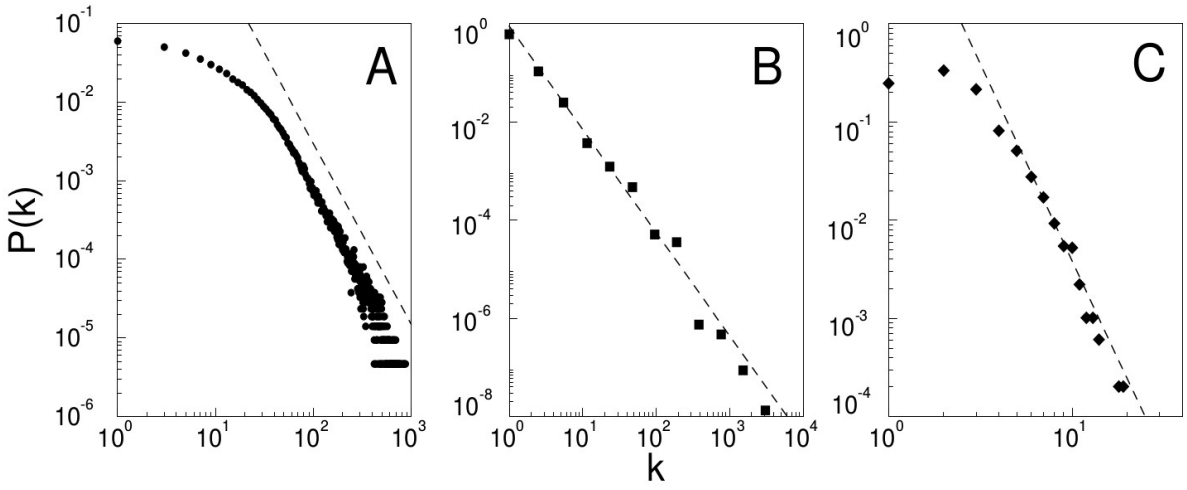


Figure 6.1: Degree distribution of a) actor collaboration graph b) world wide web graph and c)powergrid data, from [14]. Each of these graphs follows a different power-law degree distribution

degree distribution. Barabasi and Albert [14] [4] showed that for many of these networks, described as "scale-free networks", there is a huge quantity of nodes with small degree and very few nodes with huge degree. Their model to generate graphs with these properties follows a rich-gets-richer approach that underscore the importance of these high degree nodes (hubs) in the dynamics of the network.

In his review, Newman [79] showed that these three properties (short average path length, higher clustering coefficient and power-law degree distribution) change completely the dynamics of these networks. While there are natural networks that do not have one or more of these properties, knowledge about them is necessary to choose the correct strategies to process these graphs. In the experimental results of this thesis, we underline the impact of a change in diameter or of the degree distribution on the computation of different measures.

6.2 Introduction to graph partitioning

The most common approach to cope with these huge graphs using multiple processes or machines is to divide them into pieces, called *partitions*. When such partitions are assigned to a set of independent computing nodes (being them actual machines or virtual executors like processes and threads, or even mappers and reducers in the MAPREDUCE model), their size matters: the largest of them must fit in the memory of a single computing entity and imbalances in the processes's work load will cause the entire system to slow down.

The partitions should also be well-connected, to minimize the amount of communication needed by the processes to coordinate their execution. These concerns differentiate this problem from the Clustering problem, where the graph must be divided in well-connected clusters, without any requirement about their sizes.

The partitioning problem is not well-defined without a common definition of what is a partition. Figure 6.2 shows the classic definition, *Vertex Partitioning* in this thesis, in which each partition is defined by the subgraph induced by a subset of the vertex set of the original graph. The edges that have its nodes in different partitions are called *cut edges* and are the communication channels that the processes will use to coordinate. Using a different definition, partitions can be defined as graphs induced by subsets of the edge set where each edge is inside exactly one partition and there are "frontier vertices" that are present in more than one partition, as shown in Figure 6.3. When we use this definition we talk about *Edge partitioning*.

The differences between these two approaches and their respective advantages and disadvantages are covered in the rest of this chapter.

6.3 Formal definitions

Given a graph $G = (V, E)$ and a parameter K, a *vertex partitioning* of G subdivides all vertices into a collection $V_1, \ldots, V_K$ of non-overlapping edge partitions:

$$V = \cup_{i=1}^{K} V_i \qquad \forall i, j : i \neq j \Rightarrow V_i \cap V_j = \emptyset$$

The i-th partition is associated with a edge set E_i, composed of the end points of its edges:

$$E_i = \{(u, v) : u \in V_i \vee v \in V_i\}$$

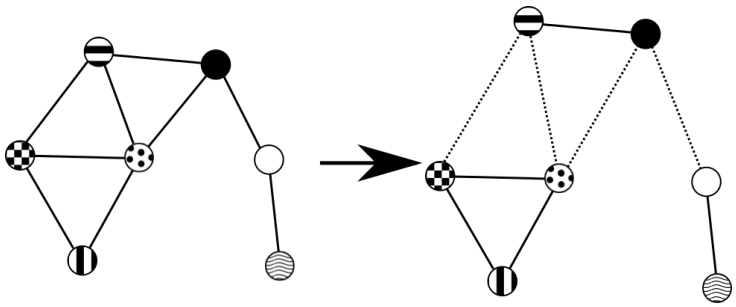


Figure 6.2: Vertex partitioning example: each vertex appears in one partition, cut edges connect partitions

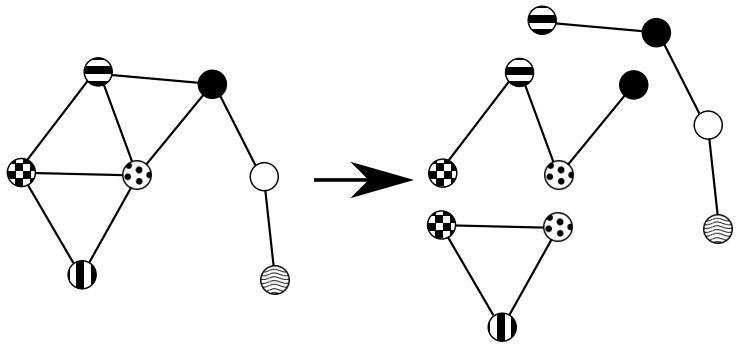


Figure 6.3: Edge partitioning example: each edge appears in only one partition, while frontier vertices may appear in more than one partition

The edges of each partition, together with the associated vertices, form the subgraph $G_i = (V_i, E_i)$ of G. The cut edges of this partitioning are all those edges that appear in more than one partition.

$$Cut(G_1, \ldots, G_K) = \{e \in E : \exists i \neq j : e \in (E_i \cap E_j)\}$$

Similarly, an *edge partitioning* of G subdivides all edges into a collection $E_1, \ldots, E_K$ of non-overlapping edge partitions:

$$E = \cup_{i=1}^{K} E_i \qquad \forall i, j : i \neq j \Rightarrow E_i \cap E_j = \emptyset$$

The vertex set of the ith partition, V_i, is composed of the end points of its edges:

$$V_i = \{u : \exists v : (u, v) \in E_i \vee (v, u) \in E_i\}$$

We denote with $F_i \subseteq V_i$ the set of vertices that are frontier in the i-th partition.

$$F_i = \{u \in V_i : \exists j \neq i : u \in V_j\}$$

The *size* of a partition is proportional to the amount of edges and vertices $|E_i| + |V_i|$ belonging to it. Vertices may be replicated among several partitions, in which case they are called *frontier vertices*.

The vertex-partitioning problem asks, given a graph G and an integer K, to find a partitioning of G into K vertex partitions such that:

- $\forall i, size(G_i) < (1 + \epsilon) \times \dfrac{size(G)}{K}$
- $|Cut(G_1, \ldots, G_K)|$ is minimized.

	Partition defined by	**Partitions connected through**
Vertex partitioning	Vertices	Cut edges
Edge partitioning	Edges	Frontier vertices
Hypergraph partitioning	Vertices	Cut hyperedges

Table 6.1: Terminology of different definitions of partitioning

The definition for the edge-partitioning problem is very similar, but tries to minimize the number of frontier vertices. In both versions the partitioning problem is not only NP-complete, but even difficult to approximate [6].

A third definition of the partitioning problem regards partitioning of hypergraphs [112]. This subproblem has different applications from the other kinds of partitioning, such as load balancing, circuit design or parallel databases, but often uses similar ideas in its solutions. As in vertex partitioning, the vertex set is partitioned into similar-sized partitions and the hyperedges can connect two or more partitions.

In Table 6.1 we show the terminology that is used through this thesis for each type of partitioning.

6.4 Advantages of edge partitioning

While traditionally vertex partitioning has been prominent, in the last decade there has been a push toward edge partitioning by some of the big players in large-scale graph analysis, as shown in Chapter 8. The reasoning behind these choices is the following: dividing the vertex set in equal-sized partitions can still lead to an unbalanced subdivision: having the same amount of vertices does not imply having the same size, given the unknown distribution of their degrees and the potential high assortativity of some graphs. Given that each edge $(u, v) \in E_i$ contributes with at most two vertices, $|V_i| = O(|E_i|)$ and the amount of memory needed to store a partition is strictly proportional to the number of its edges, this fact can be exploited to fairly distribute the load among machines.

This problem is much more common on scale-free graphs, because of the power-law degree distribution of its nodes. When only a few nodes can contain a large percentage of the total edges, their position in the partitioned graph can become a big issue. Using the edge partitioning formulation allows the algorithm to cut these hubs into different partitions, thus leading to more balanced partitions.

In theory, a vertex-partitioning algorithm can be used to compute the edge partitioning of G, by using the line graph of G. The line graph of $G = (V, E)$ is constructed by creating a vertex for each edge in the original graph and creating connections between edges that

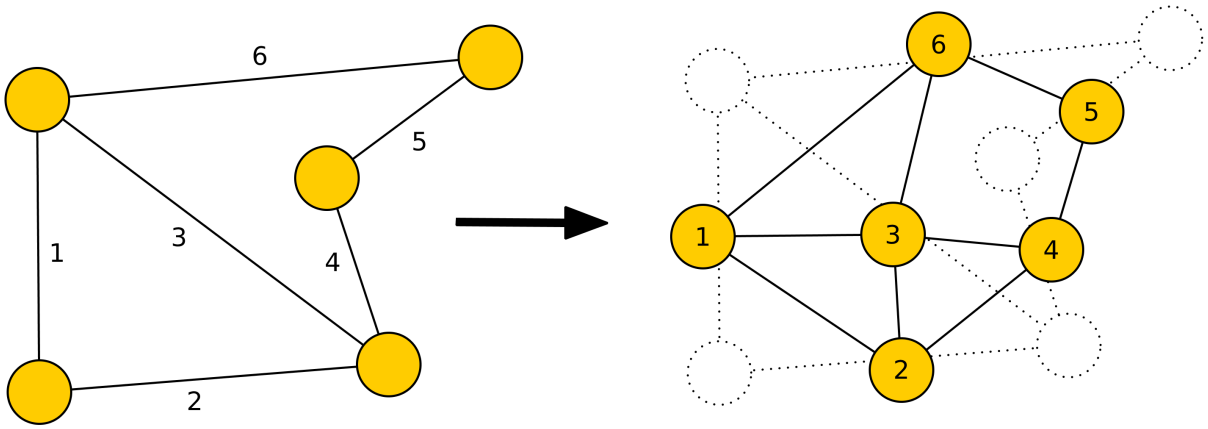


Figure 6.4: On the left, a graph with labels on the edges. On the right, its line graph. For each edge in the original graph a new vertex is created. Two vertices are connected if, in the original graph, the corresponding edges had a node in common

had a vertex in common (see Figure 6.4 for an example). Given a graph $G = (V, E)$, the line graph of G is defined as $L_G = (E, E')$ with

$$E' = \{(e_0, e_1) : e_0 = (u, v) \in E, e_1 = (w, y) \in E, e_0 \neq e_1, e_0 \cap e_1 \neq \emptyset\}$$

It is easy to see that the edge partitioning of G corresponds to a vertex partitioning of L_G, but this approach is infeasible for real-world graphs because of the size of their line graphs. The number of edges of the line graph grows proportional to the sum of the squares of the degrees of the nodes in the original graph.

$$|E'| = \sum_{v \in V} deg_v \times (deg_v - 1)$$

Since real-world graphs follow a power-law degree distribution, their size grows order of magnitude bigger when the line graph is constructed. A more efficient approach would be to use the line graph as an hypergraph, but the same issues of performance may appear.

6.5 Approaches to graph partitioning

The type of dataset and the characteristics of the chosen system greatly inform the choice of a partitioning algorithm. As example, if the chosen system runs on a vertex-partitioned graph, the choice is obviously restricted to algorithms that compute vertex partitionings. Knowing in advance the input needed by the system will already restrict the range of possible partitioning algorithm.

What still needs to be decided is the amount of computation to invest in the partitioning phase. There is a huge variance in efficiency between graph partitioning algorithms:

there are extremely quick algorithms that can process huge graphs in seconds but do not compute high-quality partitions, and slow, methodical, algorithms that get close to the optimal solution but will take some time to get there.

Since the system will be able to work more efficiently if the graph has been well-partitioned, there is a tradeoff between fast partitioning and fast execution. The right choice can only be made if the users already have an idea of how computationally heavy is the analysis they want to execute. If the analysis is a quick query, such as computing degree centrality, then the quality of the partitioning is not crucial and a quick approximation algorithm, such as those in the next section, will suffice. If the analysis is a more complex computation, such as computing the Betweenness Centrality, then investing more time in a slower, more precise partitioning algorithm can lead to much better results in the computing phase. Choosing a graph analysis system that has predictable running time can help inform this choice.

6.5.1 Quick heuristic partitioning

There are many different partitioning strategies that can be applied very quickly, possibly in parallel, to get to a rough partitioning of the given graph. As representative examples, we describe the partitioning strategies implemented by SPARK [104] in their graph processing framework GRAPHX. Since this framework uses edge partitioning, all of the proposed algorithm try to solve the edge-partitioning problem. Nevertheless, it is easy to build their corresponding vertex-partitioning algorithms by simply reusing the same ideas in the context of vertex partitions.

The most obvious partitioning strategy regards random placement of edges. Each edge will end in a random partition, thus creating an huge number of frontier vertices. Usually, instead of using completely random placement, the implementation will use a randomly chosen hash function to make sure that two edges that have the same source and destination vertices will end in the same partition.

An equally fast, but more precise heuristic uses just the source vertex of the edge to choose the resulting partition. The id of the source vertex will be passed to an hash function and, therefore, all of its outgoing edges will be put in the same partition. Most nodes that have both outgoing and ingoing edges will become frontier vertices, but they will be replicated in fewer partitions than in the completely random scenario.

The last partitioning strategy offered by GRAPHX is named EdgePartition2D: the sparse matrix representation of the graph, which is the $N \times N$ matrix that contains for each pair of nodes if they have an edge between them, is partitioned into smaller squares. If the algorithm receive a request for 9 partitions, it will divide the matrix into 3×3 smaller submatrix and assign all edges to partitions according to their position in the

matrix. This approach has the advantage that hubs are effectively split into different partitions, but all vertex are split in at most $(O(\sqrt{K}))$ partitions. The fact that this strategy cannot be applied if K is not a perfect square effectively limits its range of applications.

6.5.2 Specialized partitioning

A special consideration must be made for partitioning algorithms that use information outside of the graph structure. If the graph to be partitioned is part of the World Wide Web and the algorithms has access to the URL of the pages, then it is much easier to obtain good partitioning by clustering the pages by their domain. Since most edges are between pages of the same domain, such an algorithm will be cheap and obtain very high-quality partitions.

Similar approaches can be applied to road networks, where the coordinate of the vertices can inform the choice of the partition. A common technique computes the 2D rectangle that surrounds the data points, partitions the rectangle and then map each vertex according to its coordinates.

As a general approach, additional information about vertices and edges should be used before trying general graph partitioning methods, since it is likely that such method will obtain a partitioning so close to the optimal solution, that the motivation behind more complex algorithms disappears.

6.5.3 Partitioning via exchanges

Because of the complexity of this problem, most research focused on heuristics algorithms with no guaranteed approximation rate. Kernighan and Lin developed the most well-known heuristic algorithm for binary graph partitioning in 1970 [61]. At initialization time each vertex in the network is randomly assigned to one of two partitions and the algorithm tries to optimize the vertex cut by exchanging vertices between the two partitions. The process is repeated until is not possible to find exchanges that improve the solution. In case there is a need for a larger number of partitions it is possible to generalize this approach by changing the initialization procedure and adapting the scoring function that is used to decide which vertices the partitions should exchange. This technique has been later extended to run efficiently on multiprocessors by parallelizing the computation of the scoring function used to choose which vertices should be exchanged [42].

Similar approaches have been developed for hypergraph partitioning. Fiduccia et al. [36] start with random partitioning and organize moves of nodes between partitions in *passes*. For each pass, the algorithm computes the gain of all possible exchanges of positions between nodes, chooses and executes the best one and lock the nodes. A pass

is finished when all nodes are locked and the best solution seen during that pass is the starting solution for the next pass.

Both these algorithms depends from the choice of the starting solution and therefore might easily incur in local minima. Techniques to alleviate this issue are more complex heuristics to choose the starting solution or running multiple instances of the algorithm and choosing the best result.

6.5.4 Multi-level partitioning

An highly successful strategy for graph partitioning has been the idea of multi-level partitioning. The original graph is first coarsened into a sequence of smaller and smaller graphs by collapsing edges and nodes. The smallest graph is then partitioned using slower, precise algorithms. The results of this partitioning are then sequentially applied to the larger graphs and refined, until the results have reached the original graph.

The most successful algorithm that uses this strategy is METIS [57]. The authors introduced improvements at each step of the strategy: edges to be collapsed are chosen using the *heavy edge matching* strategy, which guarantees a smaller number of levels before reaching a graph small enough to apply partitioning. The partitioning phase is implemented via a simple breadth-first search from a random node, with bias toward nodes that create less cut-edges. Finally, the refinement phase is implemented with a boundary algorithm that greatly improves over the previously used Kernighan-Lin algorithm.

METIS has been quickly expanded to work for hypergraph partitioning [58] and to K-way partitioning [59]. An effort to create a parallelizable version of the program has lead to ParMETIS [60], a version built for multicore machines. The quality of the partitions obtained with this approach does not seem to be of the same quality than the centralized version.

While the quality of METIS is considered very high, there have been improvements on the process by which the graph is coarsened and the different heuristics used at different points of the multilevel approach. Lotfifar and Johnson [67] show an improvement in partitioning hypergraphs by filtering "unimportant edges" using Rough Sets, a data structure useful to extract less important items from a large data set [86].

6.5.5 Streaming partitioning algorithms

The presence of additional constraints has driven the research field towards more specialized algorithms. For example, in the streaming scenario it is infeasible to use the classical partitioning algorithms, since the data is continuously arriving. Many greedy algorithms, starting from very simple heuristics [106], assign each incoming vertex to a partition in a

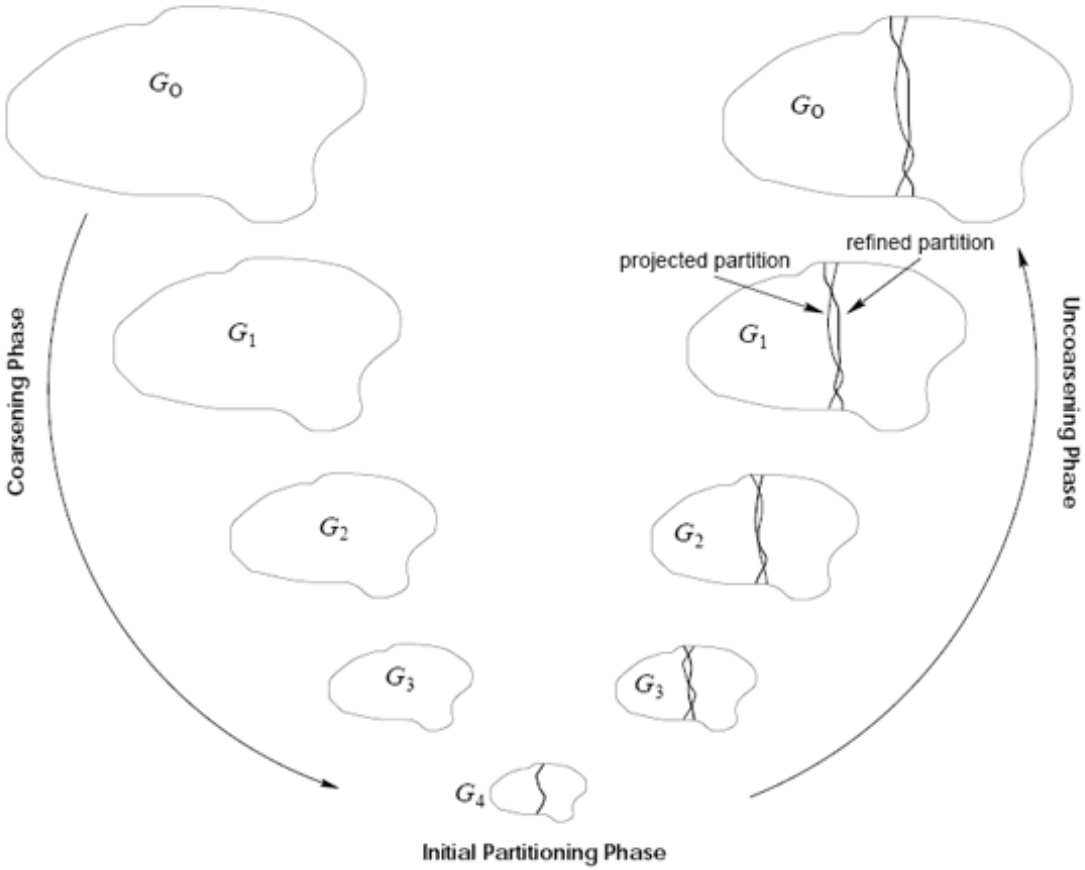


Figure 6.5: Illustrative scheme of multilevel partitioning algorithms, from `http://masters.donntu.org/2006/fvti/shepel/diss/indexe.htm`

streaming scenario.

A more complex example is Fennel [113], an algorithm that computes partitions of only slightly less quality than most centralized algorithms using a fraction of resources. For each incoming vertex, the algorithm computes an objective function that measures the quality of the results for each possible choice. Fennel's uses a framework to define the objective function, which is shown to contain most of the already known heuristics and can be used to control the complexity of the algorithm.

Powergraph's greedy vertex cut algorithm [44] also uses a similar approach, processing and assigning each edge before moving to the next. It keeps in memory the current sizes of each partition and, for each vertex, the set of partitions that contain at least one edge of that vertex. If both endpoints of the current edge are already inside one common partition, the edge will be added to that partition. If they have no partitions in common, a partition will be chosen by the node having the most edges still to be assigned. If only one node is

already in a partition, the edge will be assigned to that partition. Otherwise, if both nodes are free, the edge will be assigned to the smallest partition. The advantage over Fennel's approach is that there is no objective function to optimize, therefore the algorithm can be run efficiently even on larger datasets. This heuristic can be run independently on N subsets of the edge set to parallelize the workload, at the cost of lower quality partitions.

GraSP [15] shows a real distributed implementation of these greedy strategies. To balance the limited view of each process, the greedy strategy is applied in n_s different passes. At each pass, the requirements for balance become more strict, to allow for higher quality but less balanced partitions at the beginning and then improve the solution toward a more balanced partitioning. The processes only communicate between passes.

HDRF [87] instead concentrates on the greedy rules that are applied on the incoming edges or vertices, with the aim of using the power-law degree distribution of real graphs. If the algorithm needs to create a frontier vertices, it will try to choose the highest-degree nodes.

Another interesting contribution of HDRF is how it can adapt to non-randomly ordered streams. A particularity of most greedy streaming partitioning algorithms is that, while they perform extremely well in case of randomly ordered streams, they can obtain extremely bad results if the vertices arrive in a specific order, such as one created by a breadth first visit of the graph. Since in many applications the graph is constructed using such processes, the user need to apply a costly random sorting before running the algorithm. HDRF has the advantage of being able to run on partially ordered by correctly setting the parameters of its greedy strategies.

6.5.6 Distributed partitioning

Aside from centralized algorithms that have been expanded to work on parallel and distributed settings, there has been also research on native algorithms for partitioning in distributed and peer-to-peer settings. A few algorithms on distributed graph clustering have also been developed, but they cannot be used for graph partitioning since they do not obtain balanced partitions. Two examples are DIDIC and CDC: DIDIC [41] uses a diffusion process to move information across the graph and make sure that clusters are properly recognized, while CDC [95] simulates a flow of movement across the graph to compute the community around an originator vertex.

JA-BE-JA [94] was the first completely decentralized partitioning algorithm based on local and global exchanges. As in the Kernighan and Lin algorithm, each vertex in the graph is initially mapped to a random partition. At each iteration, it will independently try to exchange its partition with either one of its neighbor or with one of the random vertices obtained via a peer selection algorithm. If the exchange decreases the edge cut

size, it will be approved. Since the algorithm has a large risk of falling into a local minima, the authors added a layer based on simulated annealing. At the beginning of the computation, the nodes will be happy to exchange their partitions, even if if this slightly decrease the quality, while toward the end they will be more careful and move partitions only if there is a gain. This peer-to-peer approach is extremely scalable and can therefore be used easily in distributed scenarios.

The authors have also extended their work to apply the same approach to edge partitioning [93], by changing the peer sampling strategy to allow for exploration of edges instead of vertices.

In Chapter 7 we present an alternative to JA-BE-JA, a distributed edge-partitioning algorithm closer to the concept of diffusion and that needs less iterations to converge to a solution.

Chapter 7

DFEP: Distributed Funding-based Edge Partitioning

As presented in Chapter 6, there are many options for graph partitioning that cover the entire spectrum of requirements. There are rough and fast partitioning algorithms for situations in which fast partitioning is more important and algorithms that do more complex computation to get closer to the optimal answer.

The area that has not seen the same level of scrutiny is natively distributed partitioning that can scale to large sizes without losing quality. Here, JA-BE-JA is the main competitor, but its simulated annealing approach require many iterations to complete. If the users set JA-BE-JA's parameters according to the guidelines described by the authors, JA-BE-JA will need several hundred iterations to reach its answer. This large number of iterations may be costly in synchronized settings, where there is a synchronization barrier at the end of each iteration.

In this chapter we present DFEP [45], a novel distributed graph partitioning algorithm that uses the concept of diffusion and needs only a small coordinator to keep the sizes of each partition as similar as possible. DFEP is scalable, computes dense, connected partitions and can be implemented on top of many vertex-centric programming models.

7.1 Distributed Funding-based Edge Partitioning

The properties that a "good" edge partitioning must possess are the following:

- **Balance**: partition sizes should be as close as possible to the average size $|E|/K$, where K is the number of partitions, to have a similar computational load in each partition. Our main goal is to minimize the size of the largest partition.
- **Communication efficiency**: given that the amount of communication that crosses

the border of a partition depends on the number of its frontier vertices, the total sum $\sum_{i=1}^{K} |F_i|$ must be reduced as much as possible.

There are other, less crucial properties that can cause advantages when specific computation must be done on top of the partitioned graph.

- **Connectedness**: the subgraphs induced by the partitions should be as connected as possible. This is not a strict requirement and we also illustrate a variant of DFEP that relax this condition.
- **Path compression**: a path between two vertices in G is composed by a sequence of edges. If some information must be passed across this path, it will need to cross partitions every time two consecutive edges belong to different partitions. The smallest the number of partitions to be traversed, the faster the execution will be.

Balance is the main goal; it would be simple to just split the edges in K sets of size $\approx |E|/K$, but this could have severe implications on communication efficiency and connectedness. The approach proposed here is thus heuristic in nature and provides an approximate solution to the requirements above.

Since the purpose is to compute an edge partitioning as a pre-processing step to help the analysis of very large graphs, we need the edge-partitioning algorithm to be distributed as well. As with most distributed algorithms, we are mostly interested in minimizing the amount of communication steps needed to complete the partitioning.

Ideally a simple solution could work as follows: to compute K partitions, K edges are chosen at random and each partition grows around those edges. Then, all partitions take control of the edges that are *neighbors* (i.e., they share one vertex) of those already in control and are not taken by other partitions. All partitions will incrementally get larger and larger until all edges have been taken. Unfortunately, this simple approach does not work well in practice, since the starting positions may greatly influence the size of the partitions. A partition that starts from the center of the graph will have more space to expand than a partition that starts from the border and/or very close to another partition.

To overcome this limitation, we introduce DFEP (Distributed Funding-based Edge Partitioning), an algorithm based on concept of "buying" the edges through an amount of *funding* assigned to partitions. Initially, each partition is assigned the same amount of funding and an initial, randomly-selected vertex. The algorithm is then organized in a sequence of *rounds*. During each round, the partitions try to acquire the edges that are neighbors to those already taken, while a coordinator monitors the sizes of each partition and sends additional units of funding to the smaller ones, to help them overcome their slow start.

Table 7.1: Notation

$d(v)$	degree of vertex v
$E(v)$	edges incident on vertex v
$V(e)$	vertices incident on edge e
$M_i[v]$	units of partition i in vertex v
$M_i[e]$	units of partition i in edge e
E_i	edges bought by partition i
$owner[e]$	the partition that owns edge e

Algorithm 1: DFEP Init

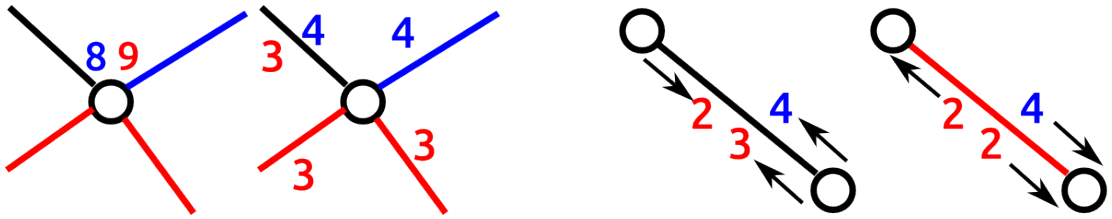
Executed by the coordinator

```
foreach edge e ∈ E do
    owner[e] = ⊥
for i = 1 to K do
    v ← random(V)
    M_i[v] = |E|/K
```

Algorithm 2: DFEP Step 1

Executed at each vertex v

```
for i = 1 to K do
    if M_i[v] > 0 then
        eligible = ∅
        foreach e ∈ E(v) do
            if owner[e] = ⊥ or owner[e] = i then
                eligible = eligible ∪ {e}
        foreach e ∈ eligible do
            M_i[e] = M_i[e] + (M_i[v]/|eligible|)
        M_i[v] = 0
```



(a) Step 1 (b) Step 2

Figure 7.1: Sample run of Step 1 and 2 of DFEP

Algorithm 3: DFEP Step 2
Executed at each edge e

$best = argmax_p(M_p(e))$
if $owner[e] = \perp$ **and** $M_{best}(e) \geq 1$ **then**
  $owner[e] = best$
  $M_{best}[e] = M_{best}[e] - 1$
for $i = 1$ **to** K **do**
  if $owner[e] = i$ **then**
    foreach $v \in N(e)$ **do**
      $M_i[v] = M_i[v] + M_i[e]/2$
  else
    $S =$ vertices that funded partition i in e
    foreach $v \in S$ **do**
      $M_i[v] = M_i[v] + M_i[e]/|S|$
  $M_i[e] = 0$

Table 7.1 contains the notation used in the pseudocode of the algorithm. For each vertex and edge we keep track of the amount of units that each partition has committed to that vertex or edge. Algorithm 1 presents the code executed at the initialization step: each partition chooses a vertex at random and assigns all the initial units to it. The edges are initialized as unassigned. Each round of the algorithm is then divided in three steps. In the first step (Algorithm 2), each vertex propagates the units of funding to the outgoing edges. For each partition, the vertex can move its funding only on edges that are free or owned by that partition, dividing the available units of funding equally among all these eligible edges. During the second step (Algorithm 3), each free edge is bought by the partition which has the most units committed in that edge and the units of funding of the losing partitions are sent back in equal parts to the vertices that contributed to that funding. The winning partition loses a unit of funding to pay for

Algorithm 4: DFEP Step 3
Executed by the coordinator

$AVG = \sum_{i \in [1...K]}(|E_i|)/K$
for $i = 1$ **to** K **do**
  $funding = min(10, AVG/E_i)$
  foreach $v \in V$ **do**
    if $M_i(v) > 0$ **then**
      $M_i(v) = M_i(v) + funding$

the edge and the remaining funding is divided in two equal parts and sent to the vertices composing the edge. In the third step (Algorithm 4), each partition receives an amount of funding inversely proportional to the number of edges it has already bought. This funding is distributed between all the vertices in which the partition has already committed a positive amount of funding.

Figure 7.1a illustrates Step 1 of the algorithm. The vertex has 8 units on the blue partition, 9 units on the red one, two edges are owned by the red partition, one by the blue, and the black one is still unassigned. When Step 1 is concluded, the 9 red units have been committed to the two red edges and the black one, while the 8 blue units have been committed to the blue edge and the black one. The blue partition will be allowed to buy the black edge. Figure 7.1b illustrates Step 2 executed on a single edge. The edge receives 5 red units and 4 blue units, and thus is assigned to the red partition. All the blue units are returned to the sender while the remaining $5 - 1$ red units are divided equally between the two vertices.

DFEP creates partitions that are connected subgraphs of the original graph, since currency cannot traverse an edge that has not been bought by that partition. It can be implemented in a distributed framework: both Step 1 and Step 2 are completely decentralized; Step 3, while centralized, needs an amount of computation that is only linear in the number of partitions.

In our implementation the amount of initial funding is equal to what would be needed to buy an amount of edges equal to the optimal sized partition. A smaller quantity would not decrease the precision of the algorithm, but it would slow it down during the first rounds. The cap on the units of funding to be given to a small partition during each round (10 units in our implementation) avoids the over-funding of a small partition during the first rounds.

In a distributed setting the algorithm will follow the Bulk Synchronous Processing model: each machine receives a subset of the graph, executes Step 1 on each of its vertices

independently, sends money to the correct edges (that may be on other machines), wait for the other machines to finish Step 1, and executes Step 2. Step 3 must be executed by a coordinator, but the amount of computation is minimal since the current sizes of the partitions can be computed via aggregated counting by the machines. Once the coordinator has computed the amount of funding for each partition, it can send this information to the machines that will apply it independently before Step 1 of the successive iteration. If the coordinator finds that all edges have been assigned, it will terminate the algorithm.

7.1.1 Variants and additions

If the diameter is very large, there is the possibility that a poor starting vertex is chosen at the beginning of the round. A partition may be cut off from the rest of the graph, thus creating unbalanced partitions. A possible solution for this problem involves adding an additional dynamic, at the cost of losing the connectedness property.

A partition is called *poor* at round i if its size is less than $\frac{\mu}{p}$, with μ being the average size of partitions at round i and p being an additional parameter; otherwise, it is called *rich*. A poor partition can commit units on already bought edges that are owned by rich partitions and try to buy them. This addition to the algorithm allows small partitions to catch up to the bigger ones even if they have no free neighboring edges and results in more balanced partitions in graphs with larger diameter.

Another interesting application of DFEP is that by changing the amount of funding given to partitions during the initialization and funding phase, it is possible to get purposely unbalanced partitions. The main application scenario is for use in dis-homogeneous networks where the system can use machines with different resources: a machine with twice the resources than another should receive a partition of twice the size than the others. DFEP can adapt easily to this scenario and return partitions of the desired size.

7.2 Results

This section starts by introducing the different metrics that have been measured during the experiments and the datasets that have been used. The evaluation is split in two parts: first we evaluate in detail the behavior of DFEP through a simulation engine; then, using the Amazon EC2 cluster, we evaluate its scalability.

7.2.1 Metrics

We evaluated our algorithms with both simulations (experiments repeated 100 times) and actual implementations (experiments repeated 20 times). The metrics considered to evaluate DFEP in our simulation engine are the following:

- **Rounds**: the number of rounds executed by DFEP to complete the partitioning. This is a good measure of the amount of synchronization needed and can be a good indicator of the eventual running time in a real world scenario.
- **Balance**: Each partition should be as close as possible to the same size. To obtain a measure of the balance between the partitions we first normalize the sizes, so that a partition of size 1 represents a partition with exactly $|E|/K$ edges. We then measure the standard deviation of the normalized sizes, computed as in the following formula, where E is the number of vertices, K is the number of partitions and $|E_i|$ is the size of the i-th partition:

$$NSTDEV = \sqrt{\frac{\sum_{i=1}^{K}\left(\frac{|E_i|}{E/K} - 1\right)^2}{K}}$$

- **Communication costs**: Each partition will have to send a message for each of its frontier vertices, to share their state with the other partitions. We thus use the frontier nodes to estimate the communication costs: $M = \sum_{i=1}^{K} F_i$.

7.2.2 Datasets

Since the simulation engine is not able to cope with larger datasets, we used different datasets for the experiments in the simulation engine and the real world experiments. For both types of datasets we list the size of the graphs, the diameter D, the clustering coefficient CC and the clustering coefficient RCC of a random graph with the same size.

The first four datasets in Table 7.2 have been used in the simulation engine. ASTROPH is a collaboration network in the astrophysics field, while EMAIL-ENRON is an email communication network from Enron. Both datasets are small-world, as shown by the small diameter. The USROADS dataset is a road networks of the US, and thus is a good example of a large diameter network. Finally, WORDNET is a synonym network, with small diameter and very high clustering coefficient.

The three larger graphs are used in our implementation of DFEP on the Amazon EC2 cloud. DBLP is the co-authorship network from the DBLP archive, YOUTUBE is the friendship graph between the users of the service while AMAZON is a co-purchasing network of the products sold by the website.

#	Name	$\|V\|$	$\|E\|$	D	CC	RCC
1	ASTROPH	17903	196972	14	1.34×10^{-1}	1.23×10^{-3}
2	EMAIL-ENRON	33696	180811	13	3.01×10^{-2}	3.19×10^{-4}
3	USROADS	126146	161950	617	1.45×10^{-2}	2.03×10^{-5}
4	WORDNET	75606	231622	14	7.12×10^{-2}	8.10×10^{-5}
5	DBLP	317080	1049866	21	1.28×10^{-1}	2.09×10^{-5}
6	YOUTUBE	1134890	2987624	20	2.08×10^{-3}	4.64×10^{-6}
7	AMAZON	400727	2349869	18	5.99×10^{-2}	2.93×10^{-5}

Table 7.2: Datasets used in the simulation engine (1-4) and EC2 (5-7)

All the networks have been taken from the SNAP graph library [66] and cleaned for our use, by making directed edges undirected and removing disconnected components.

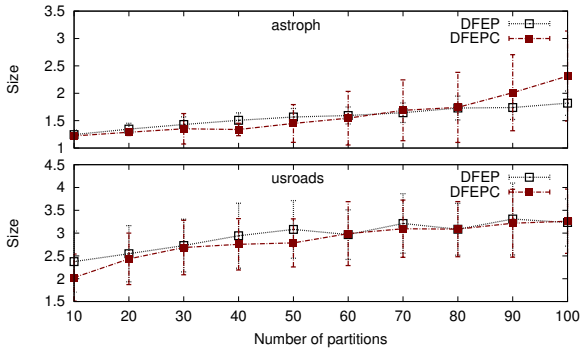
7.2.3 Simulations

Figure 7.2 shows the performance of the two versions of DFEP against the parameter K, in the ASTROPH and USROADS datasets. As expected, the larger the number of partitions, the larger is the variance between the sizes of those partitions and the amount of messages that will have to be sent across the network. The rounds needed to converge to a solution go down with the number of partitions, since it will take less time for the partitions to cover the entire graph.

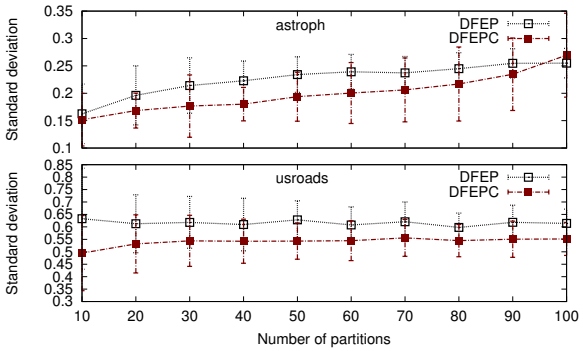
The diameter of a graph is a strong indicator of how our proposed approach will behave. To test DFEP on graphs with similar characteristics but different diameter, we followed a specific protocol: starting from the USROADS dataset (a graph with a very large diameter) we remapped random edges, thus decreasing the diameter. The remapping has been performed in such a way as to keep the number of triangles as close as possible to the original graph, to avoid introducing bias in the experiment by radically changing the clustering coefficient.

Figure 7.3 shows that changing the diameter leads to completely different behaviors. The size of the largest partitions and the standard deviation of partitions size rise steeply with the growth of the diameter, since in a graph with higher diameter the starting vertices chosen by our algorithm affect more deeply the quality of the partitioning. As expected, the number of rounds needed by DFEP to compute the partitioning also rise linearly with the diameter. Since the partitions will be more interconnected, the amount of messages sent across the network will decrease steeply with a larger diameter. Our variant of DFEP is able to cope well also in case of graphs with large diameter.

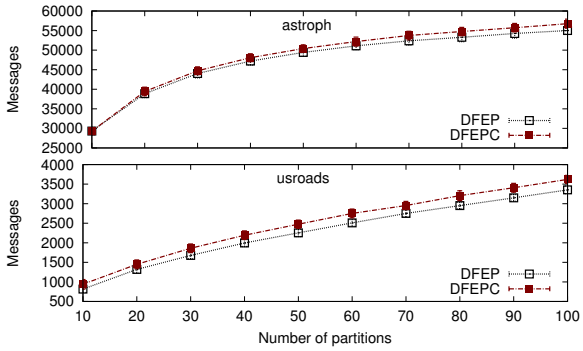
Finally, we compare the two version of DFEP against JA-BE-JA [94] and POWER-



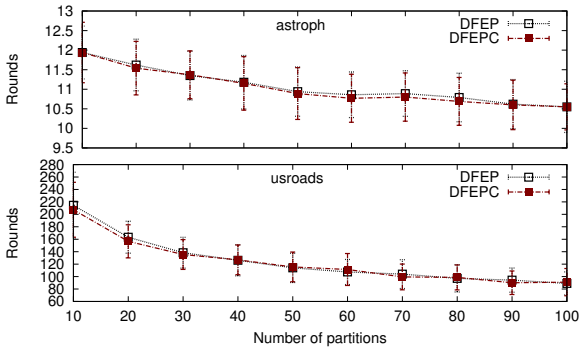
(a) Size of the largest partition



(b) Std. Dev. of partition sizes



(c) Communication cost



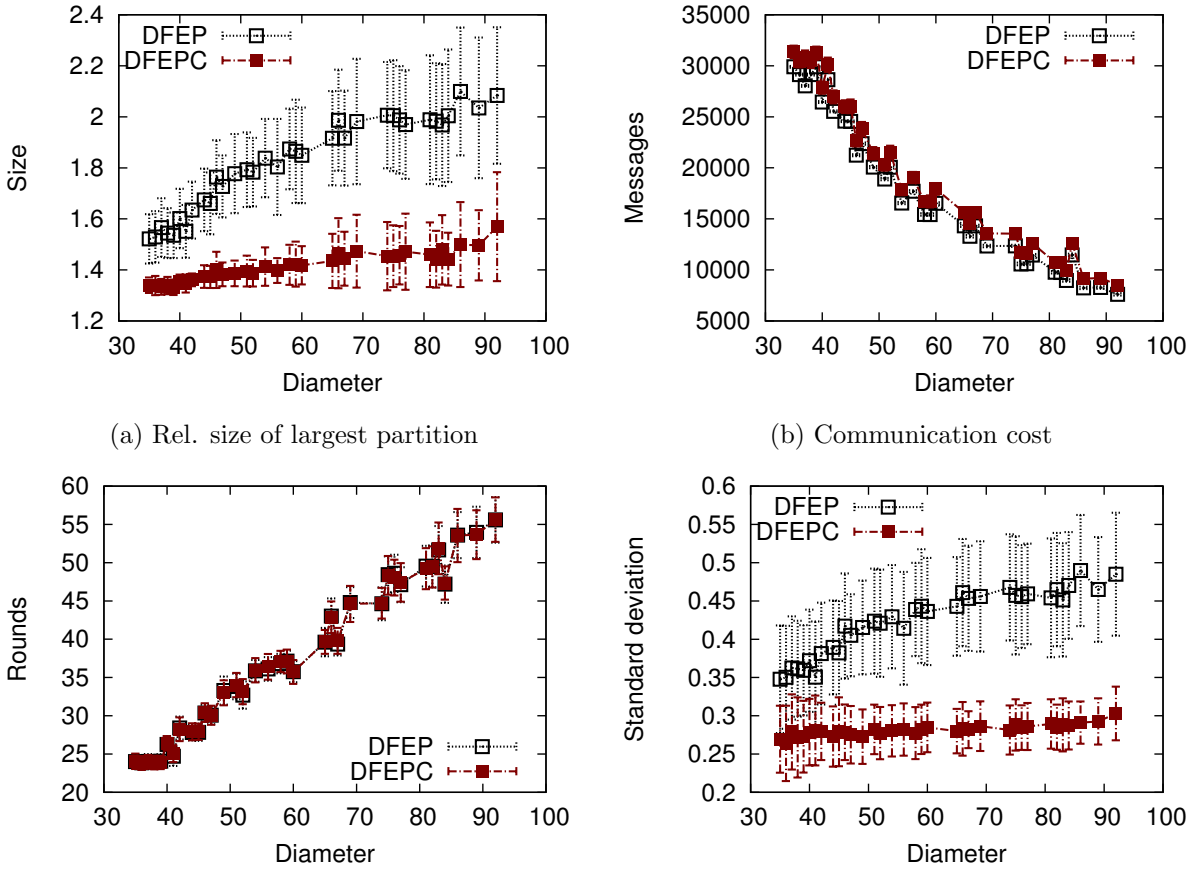
(d) Rounds to converge

Figure 7.2: Behavior of DFEP and DFEPC with varying values of K

GRAPH's greedy partitioning algorithm [44]. Since JA-BE-JA is a vertex-partitioning algorithm, its output has been converted into an edge-partitioning.

Two approaches have been considered: running the algorithm directly on the line graph of the input graph, creating a vertex for each edge in the original graph, or assigning each edge to a partition by following the vertex-partitioning and assigning each cut edge randomly to one of the two neighboring partitions. Since the line graph can be orders of magnitude bigger than the original graph we followed the second approach.

POWERGRAPH is an edge-centric graph processing framework that offers a greedy edge partitioning algorithm (see Section 8.2). It processes the graph one edge at a time, assigning it to the best partition according to which partitions already contain the nodes of the current edge. The sequential version of the algorithm needs at each step complete knowledge of the choices of the previous iteration. The authors also illustrate an "oblivious" version in which each process behaves independently on a subset of the edges. The



(a) Rel. size of largest partition

(b) Communication cost

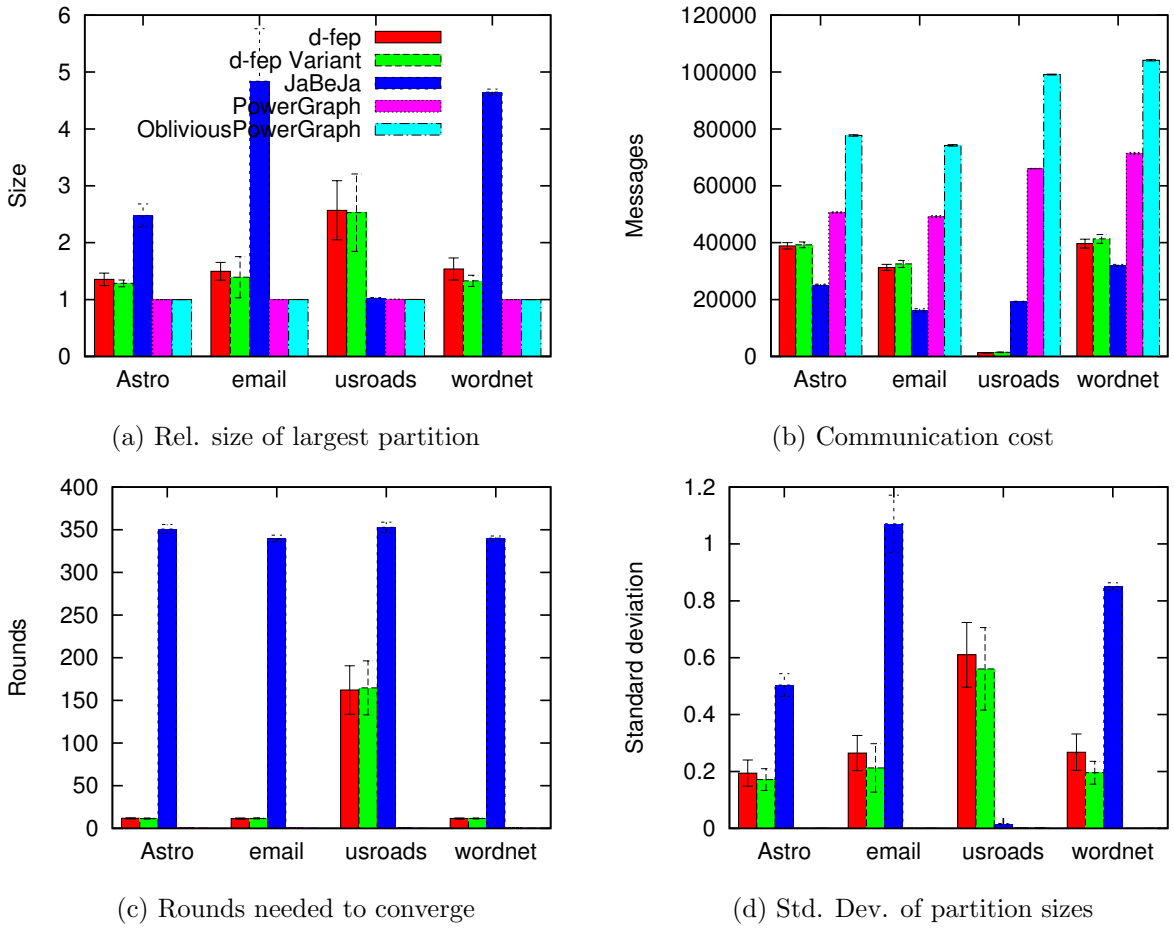
(c) Rounds needed to converge

(d) Std. Dev. of partition sizes

Figure 7.3: Behavior of DFEP and DFEPC with varying diameter ($K = 20$)

quality of the partitioning thus depends on the number of independent processes used. In our comparison, we used both the centralized version (labeled "PowerGraph") and the oblivious version (labeled "Oblivious PowerGraph"). In the oblivious version, we tested the algorithm by simulating two distinct processes. Both versions create remarkably balanced partitions and are extremely fast, since they work in a single pass over the graph. On the downside, their partitions are less connected than DFEP and thus incur in more communication costs.

Figure 7.4 shows the experimental results over 100 samples, on the four different datasets. A pattern can be discerned: the algorithms have wildly different behaviors in the small-world datasets than in the road network. In the small world datasets our approaches results in more balanced partitions, while needing less rounds to converge than JA-BE-JA. In the USROADS dataset JA-BE-JA creates more balanced partitions, but with a communication costs that is roughly ten times higher. This result shows the importance of



(a) Rel. size of largest partition

(b) Communication cost

(c) Rounds needed to converge

(d) Std. Dev. of partition sizes

Figure 7.4: Comparison between DFEP, DFEPC, JA-BE-JA and POWERGRAPH's greedy partitioning algorithm ($K = 20$)

creating partitions that are as much connected as possible. Powegraph's approach instead gets balanced, but not very connected partitions in all cases. With the oblivious version of the algorithm the quality degrades, since the approach will obtain a partitioning of worse quality the higher the number of the processes that participate in the computation.

Since JA-BE-JA uses simulated annealing to improve the candidate solution, the number of round needed is mostly independent from the structure of the graph. As shown in Figure 7.3 the number of rounds DFEP needs depend mostly from the graph diameter. Both versions of POWERGRAPH's algorithm work in a single pass over the edge set, and therefore is a better choice if the amount of computation needed after the partitioning step is not large enough to warrant a more precise partitioning.

7.2.4 Experiments in EC2

DFEP has been implemented in both HADOOP in the MAPREDUCE model and in SPARK/GRAPHX, and has been tested over the Amazon EC2 cloud. All experiments have been repeated 20 times on *m1.medium* machines.

HADOOP has arguably the largest user base between all big-data analysis frameworks, and its implementation, while not very efficient, is very stable without any unpredictable behavior. As the experimental results show, SPARK is much more efficient but is still unstable and we encountered strange behaviors when the memory available is small. This is expected since SPARK is still a young framework and makes a larger use of the memory than HADOOP. To show that our results are generic, the rest of the current section presents the results of our experiments on both frameworks.

It was not possible to implement DFEP in HADOOP using a single MAPREDUCE round for each iteration while keeping exactly the same structure as in the pseudocode. Each instance of the Map function is executed on a single vertex, which will output messages to its neighbor and a copy of itself. Each instance of the Reduce function will receive a vertex and all funding sent by the neighbors on common edges. The part of the algorithm that should be executed on each edge is instead executed by both its neighboring vertices, with special care to make sure that both executions will get the same results to avoid inconsistencies in the graph. This choice, which sounds counterintuitive, allows us to use a single MAPREDUCE round for each iteration of the algorithm, thus decreasing the communication and sorting costs inherent in the MAPREDUCE model.

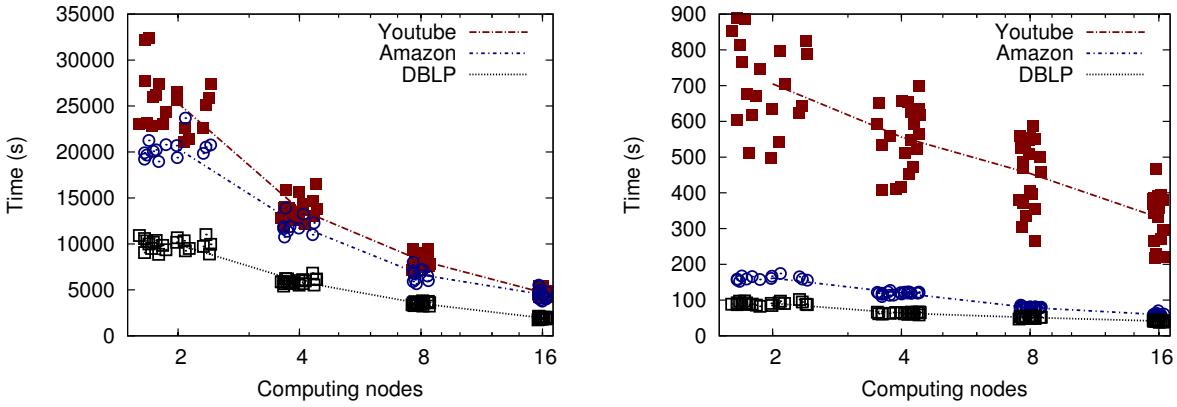
Figure 7.5a presents the scalability results, when run with the datasets in Table 7.2, with $K = 20$. The algorithm scales with the number of computing nodes, with a speedup larger than 5 with 16 nodes instead of 2.

Our SPARK/GRAPHX implementation of DFEP is still unstable, and thus, while faster, it is not able to reach the scalability of the HADOOP implementation. Figure 7.5b shows a speedup of just 2 with 16 nodes instead of 2 nodes, with a very large variance.

7.3 Future work

As future work, we are working on an efficient SPARK implementation of DFEP, to allow us to partition larger graphs and analyze the scalability of our approach. The current approach is not efficient enough to scale effectively over a certain number of nodes, but there are many opportunities to optimize it. An implementation in a lower level framework such as AKKA would probably allow for the partitioning of much larger graphs and to a bigger number of nodes.

Another line of study regards how to adapt the algorithm in presence of dynamism



(a) HADOOP (b) SPARK

Figure 7.5: Speedup of real implementation of DFEP in Amazon EC2

such as addition and deletion of edges. If the graph is already partitioned and a new batch of vertices and edges need to be added, then there are more efficient ways to process this batch without starting from scratch the computation. From our experiments, the most efficient strategy looks at the nodes of the original graph that also appear in new batch and uses their distribution of currency to initialize their copy. Running DFEP only on the new batch of nodes using this initialization techniques will cheaply obtain high quality partitions.

Part III

Computation on Large-scale Graphs

Chapter 8

Graph Processing Systems

A large number of independent research groups have focused their attention on the creation of large-scale graph processing systems. While only a few of these system have grown to maturity and are used by commercial entities, many more have not been widely used and are mostly interesting for a few improvements and ideas. Still, while the systems themselves might be unsuccessful, they still show some interesting ideas that could have applications in the larger, mature systems.

All of these frameworks have a common goal: giving an easy to use programming interface for graph analysis that can be then executed using the resources given. They will use large-scale distributed systems, parallel systems or even single commodity-level machines. They can offer query based computation, vertex, edge or partition-centric programming models. They can allow for synchronous or asynchronous algorithms and handle changes to the graph. Many are just incremental improvements over existing systems, while some are so completely different from anything else.

In Table 8.1 you can see an overview of the frameworks that are covered by this chapter. We will first introduce the big players, the frameworks that have the largest share of users and had the largest influence on the field. All the other systems are then grouped in the different categories, with emphasis on their specific improvements and ideas over the standard approaches.

8.1 Pregel and Giraph

After Google introduced MAPREDUCE, a generic processing framework that could also be used for graph computation, it started developing new systems to make graph analysis programs both more efficient and easier to develop. PREGEL [70] was the main breakthrough: it had a very simple programming model that, while restricting the developer, allowed the system to hide all the technical difficulties of running complex computations

	Prog. Model	**Resources**	**Asyn**	**Dyn**
PREGEL	Vertex-centric	Distributed system	No	No
GIRAPH	Vertex-centric	Distributed system	No	No
GRAPHLAB	Vertex-centric	Parallel systems	Yes	No
POWERGRAPH	Edge-centric	Distributed system	No	No
GRAPHX	Edge-centric	Distributed system	No	Yes
NEO4J	Query	Parallel systems		Yes
TITAN	Query	Distributed system		Yes
GIRAPH++	Graph-centric	Distributed system	No	No
GPS	Graph-centric	Distributed system	Yes	No
BLOGEL	Graph-centric	Distributed system	No	No
GRAPHCHI	Vertex-centric	Single machine	No	Yes
XSTREAM	Edge-centric	Single machine	No	No
GRIDGRAPH	Vertex or Edge-centric	Single machine	No	No
GRAPHQ	Graph-centric	Single machine	No	No
KINEOGRAPH	Vertex-centric	Distributed system	Yes	Yes
CHRONOS	Vertex-centric	Parallel systems	No	No
EAGR	Vertex-centric	Parallel systems	No	Yes
GRACE	Vertex-centric	Parallel systems	Yes	No
EXPREGEL	Vertex-centric	Distributed system	Yes	No
GIRAPH UNCHAINED	Vertex-centric	Distributed system	Yes	No
PEGASUS	Matrix operations	Distributed system	No	No
GREFT	Graph-centric	Distributed system	No	No
HAGP	Vertex-centric	Distributed system	No	No

Table 8.1: Overview of frameworks introduced in this chapter. For each framework, we list the programming model, the type of resources used by the framework, if it allows for asynchronous execution and if it can cope with dynamism in the graph

on large graphs. This highly influential paper was crucial in opening the path toward programmers-friendly graph frameworks and its main ideas have been reimplemented in almost all modern graph processing systems.

PREGEL's programming model is very simple. Each vertex is associated to a state machine (see Figure 8.1) that controls its activity. Each vertex can decide to halt its computation, but can be woken up at every point of the execution by an incoming message. At each *superstep* of the computation a user-defined vertex program is executed for each active vertex: this user-defined function will take the vertex and its incoming messages as

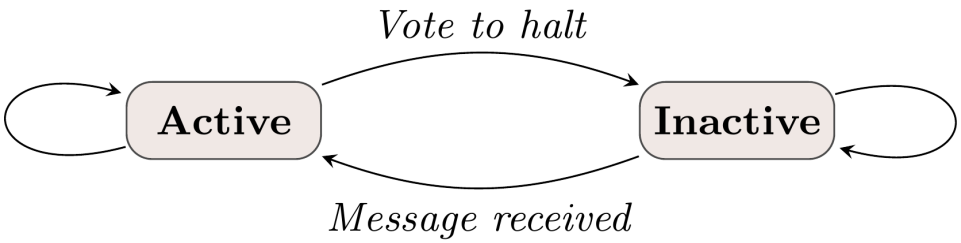


Figure 8.1: Vertex's state machine in PREGEL (from [70]

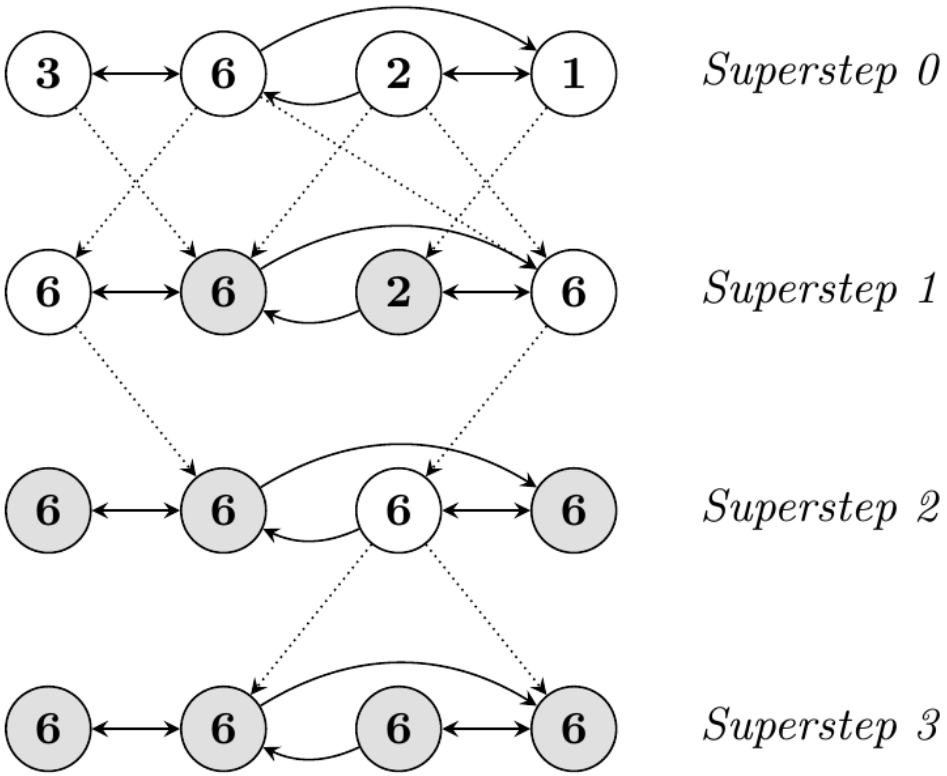


Figure 8.2: Sample run of max-number computation in PREGEL. Gray vertexes are inactive. (from [70]

input, change the vertex's value and eventually send messages to other vertices through the outgoing edges. Between each superstep of the computation a synchronization barrier makes sure that all vertices have executed their instance of the user-defined function, as defined by the Bulk Synchronous Parallel model [115]. Since all instances of the vertex program can be executed independently, different machines can execute the computation on different partitions of the graph and communicate before the synchronization barrier to allow the passage of messages between machines. In the next superstep, the user-defined program will be executed only on the active nodes, those that received messages or chose

to remain active in the previous superstep. If there are no active nodes, the computation has finished.

Figure 8.2 shows an illustrative example of the execution of a simple program. Each vertex is associated with a number and the algorithm tries to find the maximum of these numbers. The user-defined function takes the maximum of the incoming messages, updates the vertex state (if there is message that has larger value than the current state of the vertex) and disseminates its information through the outgoing edges. As shown in the example, the value 6 is quickly sent around the network until all vertexes have received it and all vote to halt the computation. In the first superstep, the neighbors of the highest-value vertex will receive its value; in the second superstep, those with distance 2 will receive its value, and so on until all vertices have been reached. The number of supersteps needed to complete the execution is therefore equal to the radius of the highest-value node, which is bounded by the diameter.

While this simple programming model is extremely effective, the original paper already presents some improvements that can be used to increase the efficiency of the system. Similarly to MAPREDUCE, it is possible to implement a user-defined function that combines outgoing messages into a single one, to decrease the number of messages that need to be sent between different machines. There is also the possibility of using aggregators to compute global information that is then available to all nodes in the next superstep.

On the execution side, PREGEL is implemented in C++ and runs a single master process and several slave processes. Each slave reads a piece of the graph from input and sends the corresponding data to the right machine, according to the partitioning strategy adopted. The master will then take care of coordinating the execution by waiting for all machines to complete their current superstep before allowing them to start the following one. The master also controls the fault tolerance mechanism: all machines checkpoints all their data to persistent storage at precise intervals. Once a machine stops responding to the master, it will choose another machine to continue the execution starting from the most recent checkpoint.

PREGEL's implementation is not available to companies outside of Google, but an open-source implementation of the module was quickly introduced: GIRAPH [8] offers a very similar API to PREGEL's. While there are no noticeable differences in the programming model, the execution plan is quite different: GIRAPH uses as much as possible of the already developed Apache systems: HDFS is used to store the input data, HADOOP starts the cluster and Zookeeper offers the synchronization service between machines and the computation of aggregate data. Since GIRAPH lives inside the HADOOP framework, it can be used inside a more complex pipeline and has seen much exposure thanks to the large user base of HADOOP itself.

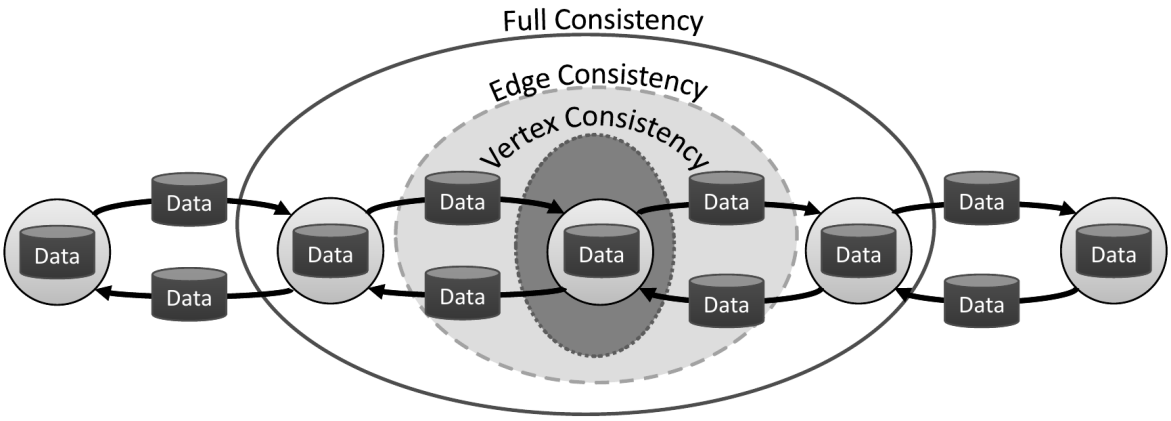


Figure 8.3: View of the scope of Vertex v in GRAPHLAB: it can see and change the state of only its neighbors. (from [68])

8.2 GraphLab and GraphX

The other two biggest players in the general graph processing space are GRAPHLAB [68] and GRAPHX [121]. While GIRAPH's evolution comes directly from the introduction of PREGEL, these two frameworks have very different starting point but, after many iterations, now occupy a similar space: graph processing systems with an immediate interface for developers, bindings in numerous languages and the opportunity to execute the entire pipeline, from pre-processing to computation, without the need of external tools.

GRAPHLAB was developed independently from PREGEL, but had the same motivations: being a middle ground between low-level implementations and systems like MAPREDUCE, which were too abstract to be used efficiently for graph computation. The main difference between the two ideas was that while PREGEL's authors were targeting Google's large distributed system, GRAPHLAB addresses shared memory parallel systems: there is more focus on efficient locking and parallel access of memory than on the issue of efficient message passing and synchronization.

GRAPHLAB's programming model is also quite simple: users must define an *update* function that, given as input a node and its entire neighborhood, can change all data associated to that node, to its edges or to its neighbors. Figure 8.3 shows the scope of a vertex: the *update* function called on that vertex will be able to read and write all data in its scope.

One issue with this approach is that, to allow for parallel execution of the update function, GRAPHLAB must have some policy to avoid race conditions. The system allows the user to define the safety of the execution by choosing between a *Fully Consistent*, a *Vertex Consistent* or an *Edge Consistent* model. Note that there is no synchronization

barrier in this system: nodes can and will be seen in any order, according to a scheduling model that can be chosen by the user. This asynchronicity allows GRAPHLAB to be extremely efficient by avoiding waiting times, but makes it difficult to keep consistent in distributed systems.

One issue that both GRAPHLAB's and PREGEL's programming model have in common is that in scale-free graphs the scope of a vertex can be huge. A hub may have a huge amount of edges and loading all this data into memory may be too costly and cause a decline in performance. To overcome these difficulties the developers of GRAPHLAB worked on a new graph processing frameworks, named POWERGRAPH [44]. Its programming model, called Gather-Apply-Scatter (or GAS) integrates the idea of combiners directly inside the programming model, to avoid the definition of user define function that need the entire neighborhood of a node. The *gather* function is applied separately on each edge and possibly computes a message to destination vertices. All messages to a single vertex are aggregated into a single message via the repeated application of a *sum* function that, from two messages, computes a single one. The message is then *applied* to the vertex, an operation that may change its state. Finally, a *scatter* function is given an edge and its nodes as input and can update the edge's state and, possibly, create new messages.

This programming model has been hugely influential because it moved the computation from being completely vertex-centric to a more efficient edge-centric model. Each function has a fixed amount of data to work on and the presence of hubs are not a problem, since the *apply* function is called on a single message and the process of combining all incoming messages into one can be easily parallelized. POWERGRAPH still keeps the same different level of asynchronicity then GRAPHLAB, allowing for both synchronous execution, completely asynchronous execution and a more relaxed *asynchronous serializable* execution that allows more control on the sequence of execution of the user-defined functions.

The developers of GRAPHLAB and POWERGRAPH decided to expand the scope of the project and founded GraphLab Create, a more generic framework that not only does graph computation using GRAPHLAB and POWERGRAPH core implementation, but also allow generic data processing. GRAPHLAB can now be seen as a complete system that can process the data, build the graph and execute some computation without the need of external tools.

GRAPHX [121] has a different history. It started from SPARK [124], a generic distributed programming framework implemented using AKKA [3] as an extension of the MAPREDUCE model. SPARK introduces RDD, resilient distributed datasets, that can be split in partitions and kept in memory by the machines of the cluster that is running the system. These RDD can be then passed to of predefined meta-functions such as map, reduce, filter

Listing 8.1: Word count in SPARK, using Scala.

```
val textFile = spark.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                     .map(word => (word, 1))
                     .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

or join, that will process them and return a new RDD. As shown in Listing 8.1, SPARK's programming model is much closer to imperative programming than MAPREDUCE.

GRAPHX uses much of SPARK's structure: graphs are defined as a pair of two specialized RDD, one containing the Vertex data and one containing the Edge data. New operations are then defined on these RDD, to allow to map vertices's values via user-defined functions, join them with the edge table or external RDDs, or also run iterative computation.

The strong points of GRAPHX's programming model are two. On the execution side, it is a natively edge-based programming model, thus allowing easy processing of scale-free graphs. On the programming side, it has a very complete API with many options for the developer. GRAPHX offers the option of running programs defined following PREGEL's programming model, but also permits the arbitrary combinations of the meta-functions.

Since GRAPHX is internal to SPARK it is possible to run algorithms that use not only the graph, but also other data structures defined in SPARK. For example, SPARK can be used in a streaming mode in which updates are collected in batches processed by user-defined algorithms. If the user-defined algorithm is actually a GRAPHX program, this setting could also allow for the execution of graph algorithms on dynamic graphs.

Both GRAPHLAB and GRAPHX have as a major selling point the fact that they can cover the entirety of the graph processing pipeline, but their different origins still have consequences: while GRAPHLAB is more efficient, it is much more restrictive when used as a generic data processing framework, while GRAPHX still maintains the flexibility inherited from SPARK, at a cost of a slower execution.

8.3 Partition-centric frameworks

While vertex-centric and edge-centric programming models have moved away from traditional algorithms with complete view of the graph, there has also been research toward finding a middle ground. Since graphs are already partitioned in subgraphs to be given to the different machines that participate to the computation, it is possible to give each of these partitions to a user-defined function that has complete view of the partition. This

approach allows the developer to define more complex functions and define algorithms that can converge much quicker. In some applications the advantages are clear: a problem such as distance computation, that usually needs a number of supersteps equal to the diameter of the graph, needs a much smaller number of supersteps in a partition-centric frameworks, since all paths inside a partition can be explored by the single user-defined function. Nevertheless, since the user-defined functions now work on subgraphs, this approach but can also easily make the programming model very difficult to use.

GIRAPH++ [110] is one example of this class of graph processing frameworks, and has been built as an extension of GIRAPH. It allows users three different options to execute their algorithms: users can write vertex programs through an interface similar to PREGEL, can run algorithms that receive the entire partition as input, or use an hybrid setting. In this last setting the users define vertex-centric programs as in PREGEL, but messages between vertices that are in the same partitions can be processed asynchronously, while messages across partitions are sent via the network. GIRAPH++ also offers an interface that allows changing the graph, which can be useful for problems such as graph coarsening and graph summarization.

GPS [100] also starts from PREGEL's programming model, but adds three important optimizations. It permits the interleaving of vertex-centric computation with more global computation, by allowing the master to execute its own user-defined function on a list of global objects. It uses the pattern of messages exchanged between vertices to reassign them to a better partition, thus refining the initial partitioning according to the execution of the algorithm. Finally, GPS includes an optimization called LALP (large adjacency list partitioning), in which adjacency lists of high-degree vertices are not stored in a single worker, but rather are partitioned across workers. This optimization can improve performance, but only for algorithms with these two properties: vertices use their adjacency lists (outgoing neighbors) only to send messages and not for computation and, if a vertex sends a message, it sends the same message to all of its outgoing neighbors. Many applications, including e.g. PageRank, follow this structure and can be seamlessly implemented in this model.

Finally, the most efficient of these partition-centric frameworks is BLOGEL [122]. BLOGEL executes vertex-centric algorithms, block-centric algorithms and even hybrid algorithms, in which all vertices execute before the entire block. These modes of execution allow BLOGEL to offer a very fast implementation of PageRank that first operates in block mode, initializing the PageRank of the vertices using only local information and the connections between blocks, and then execute the classical vertex-centric PageRank algorithm, which will converge faster than normal thanks to the better starting values. BLOGEL also offers readily available implementations of specialized partitioning algorithms,

such as URL partitioning algorithms or 2D spatial partitioning, which can lead to much faster execution of the framework when additional data about vertices is available.

Chapter 9 covers ETSCH, our own partition-centric graph processing framework that uses edge partitions to offer a more natural programming interface to the developers. We compare the efficiency of our approach with those of the other three frameworks in this section.

8.4 Low-memory frameworks

While most systems in the previous sections cover the use of parallel or distributed systems, there is a relatively new niche that has seen much research in the last few years: framework that allow the processing of large-scale graphs on single commodity-level machines. These frameworks have a very small amount of memory to use and therefore need to dump efficiently the states of vertices and edges to disk and to be able to read quickly the correct parts of the graph. These systems have many applications: if the graph is large, but not huge, using these system can be cheap enough to be available even to small companies or startup, while forcing the developer to adopt algorithms that can be eventually ported in a more large-scale graph analysis.

GRAPHCHI [63] is an extension of GRAPHLAB, based on the same programming model. It first process the graph and divides it in a set of shards (small partitions), that are stored on disk. During each iteration the shards are loaded into memory one by one, processed using a vertex-centric programming model similar to GRAPHLAB, but where updates can only be applied on the central vertex and then written to disk. If the number of vertices is small, then the states of the vertices are stored into main memory, otherwise they will be stored to disk. Since in this case the system will have only a single shard in memory, it is possible to use GRAPHCHI on very large graphs even with a single commodity machine. The process of creation of the shards can be costly, but then it is possible to update them in case of addition of edges or vertex, thus allowing analysis of dynamic graphs.

XSTREAM [97] has the same target of GRAPHCHI but uses a slightly different programming model. It uses a variation of the Gather-Apply-Scatter model of POWERGRAPH in which the Apply function is removed. For each iteration, the Scatter function is executed on each edge and the updates generated are then applied independently in the Gather phase. To have efficient execution of the system the graph is divided in partitions and only the states of vertices that appear in that partition are stored in memory. Since the edges are read in a streaming faction, no storage is needed for the edge data. XSTREAM does not need pre-processing and therefore is able to start the computation much quickly than GRAPHCHI.

GRIDGRAPH [126] uses a similar approach, but changing the random partitioning used by XSTREAM. This new approach is based on the construction of a grid over the adjacency matrix: the vertex space is divided into K chunks, thus subdividing the adjacency matrix into $K * K$ blocks. When the system will have to process edges in block (i, j), it will only need to touch vertices in chunk i and j. GRIDGRAPH can therefore load efficiently in memory the vertex states of the right chunks and two blocks that have no chunk in common can execute in parallel. Users can define functions that are executed on each edge or each vertices, and combine them in any order using an imperative programming model. A second layer of partitioning over the $K * K$ matrix allows for more optimization of disk accesses.

GRAPHQ [119] overcomes the limitations of a single machines by using the assumption that for most vertex specific queries there is only need to study a small neighborhood of that vertex. Its solution is to use a multi-level approach to partitioning and a check-refine loop to find the solution. The user will decide which should be the starting partition and the system will check if it is able to answer the query just by looking at that partition. If the system fails, it will refine the partition by adding edges from other partitions. If the partition has become so big that it cannot fit in memory, the system will stop the computation with a partial result. The multi-level partitioning helps the system in deciding from which partitions it should get edges and vertices to refine the current partition. While this approach can be extremely useful for local queries, it is not applicable for algorithms that need global information.

8.5 Frameworks for dynamic graphs

The issue of velocity can be crucial even in graph-centric computation, since these systems need to process streams of updates of vertices and edges. While more generic systems also allow for dynamism, looking at specialized systems for dynamic graphs can give insight on the techniques that could later be adopted by the larger frameworks.

KINEOGRAPH [27] receives continuous streams of updates to the graph, in the form of addition/deletion of vertices and edges. Those updates are applied in epochs, to allow users to run both incremental algorithms and static algorithms. The technical issues comes from keeping consistent each snapshot of the graph, using a mechanism named Epoch Commit that waits before committing an update to an epoch until that epoch has been defined. KINEOGRAPH offers both a push model, in which each node can send updates to its neighbors, and a pull model, in which each node can update its value by looking the states of its neighbors.

EAGR [75] has a smaller scope than KINEOGRAPH: its target is to allow fast com-

putation of continuous queries over the neighborhood of nodes in dynamic graphs. The applications for this problem are many: it can be used for anomaly detection, personalized trends or even local search. Its programming model involves the creation of an overlay graphs that contains the definition of the dataflow needed to compute the query. An important optimization allows queries of close nodes to share computation, thus speeding up the process.

CHRONOS [47] gives a different view of dynamism in graphs. While it does not process streams of changes in a graph, it is optimized for *temporal graphs*: datasets that contain information about evolution of a graph across a period of time. CHRONOS is able to use information about previous snapshots of the graph to execute computations on subsequent snapshots at a much higher speed, thus outperforming standard systems that are only able to look at each snapshot independently. Their proposed scheduling mechanism groups close snapshots together and applies updates to vertexes across all snapshots.

8.6 Frameworks for asynchronous computation

Most frameworks adopts a variation of the BSP model, which expects a synchronization at the end of each superstep, to ease the construction of correct algorithms. Asynchronous algorithms are usually more difficult to study, since their behavior can depend on the order of execution of the single vertex programs, but they can be more efficient because of the less need for synchronization. To find a middle ground between two options many frameworks have studied ways to allow options for asynchronous execution of the user-defined functions without having to complicate the programming model.

GRACE [118] follows a relaxed BSP paradigm, by allowing users to programmatically decide when a vertex is executed. The user can adjust the scheduling priority of each vertex whenever it has received a message and can make sure that vertices that receive many messages are executed more often than less important vertices. By relaxing this simple property GRACE is shown to reach a convergence rate close to asynchronous systems, without losing the advantages of the BSP model.

EXPREGEL [99] expands PREGEL by giving priority to messages between nodes of the same partition over messages across partitions. Each partition executes the compute function on its nodes and all updates that have been generated inside the partition are immediately processed. Once all vertices have consumed all the internal messages, the external messages are sent across the network to the other partitions. This approach's advantages are twofold: similarly to what happens in partition-centric frameworks, the rate of convergence of EXPREGEL is much faster than PREGEL, but since it keeps the same programming interface it is easy to port PREGEL's applications to EXPREGEL.

Listing 8.2: Example of query in CYPHER: find the cast of all movies which names start with the letter T and return the first 10 in alphabetical order

```
MATCH (actor:Person)-[:ACTED_IN]->(movie:Movie)
WHERE movie.title =~ "T.*"
RETURN movie.title as title, collect(actor.name) as cast
ORDER BY title ASC LIMIT 10;
```

GIRAPH UNCHAINED [46] adds barrierless asynchronous execution to GIRAPH's programming model. GIRAPH++ uses both *local barriers* and the traditional *global barriers*. Local barriers divides local superstep and are executed by single workers. Workers can execute a different number of local superstep, without restrictions on waiting for other workers. Similarly to EXPREGEL, workers can decide to ask for a global barrier if it has finished processing internal messages. These improvements to GIRAPH make this framework over 5 times faster than the synchronous version.

8.7 Graph databases

A wildly different category of graph processing systems are large-scale graph databases. These systems grows from a sense of inadequacy of traditional SQL systems for graph data, with a focus on scalability and distribution. Graph databases can typically scale to a large number of machines, but are less mature both in their programming language and implementation. While optimization of SQL queries has a long history, optimization in graph databases is still at its early stages: many graph databases run their own specialized query language and that makes it difficult to create common optimization between systems. There is also a question of security, since many graph databases are not yet mature and still keep in memory their data in plain text, without any of the protections offered by traditional, mature databases [84].

The most widely used and mature of these graph databases is NEO4J [78], with clients such as Walmart, Ebay and Telenor. Neo4j offers its own querying language, named CYPHER, which allows the definition of queries that span not only vertices or edges, but also paths of any length. Listing 8.2 illustrates a simple query on a database that contains relations between actors and movies.

Internally, NEO4J is implemented on top of the JVM and uses many optimizations to minimize the storage space. It offers drivers in many languages and integration with systems such as SPARK and Elastic Search. There are still some scalability concerns, since it has been mostly used in parallel settings. The distributed version just keeps copies of

Listing 8.3: Query in TITAN: find the name of all vertices that have been created by a vertex that has "marko" as its name

```
g.V().has("name","marko").out("created").values("name")
```

the same database and reconciliate the data whenever there have been changes.

TITAN is a younger, natively distributed graph database. It lives inside the HADOOP ecosystem and is therefore well-integrated with both SPARK and HADOOP. Its programming model, Gremlin, allows for more imperative syntax (as seen in Listing 8.3) and is converted to a query in a later stage. When compared with NEO4J, TITAN is more scalable and flexible, since it allow for different backends such as HBase and Cassandra, but is still slower on average.

An overview of graph databases [52] shows that the rate of reads to write should inform the choice of the database. NEO4J is still the fastest on the market if there is a larger number of read operations, but if there are many writes and there are many changes in the graph then other backends, such as Cassandra or HBase, can be even faster.

8.8 Other frameworks

We end this chapter with a few unrelated frameworks that introduce some very specific, interesting optimizations. While none of these frameworks is as prominent as the ones presented in the rest of this chapter, they all introduce some ideas that might be harnessed to optimize more general graph processing framework.

PEGASUS [55] defines a much different programming model, named *GIM-V*: Generalized Iterative Matrix-Vector multiplication. The intuition behind GIM-V is to take the operations used in matrix-vector multiplication and create a programming model that only allows those operations. The most interesting contribution of this approach is the unexpected flexibility of the programming model. By combining these simple three operations the authors can implement different algorithms such as PageRank, Random Walk, connected components, and diameter estimation. The author's implementation on top of HADOOP can scale to graphs with billion of edges.

GREFT [90] is the first to introduce the issue of byzantine fault tolerance in large-scale graph processing frameworks. The authors propose an approach to keep the data consistent in case of a single machine with accidental arbitrary faults. The author implemented GREFT as an extension of GPS and run in over several large-scale graphs. Experimental results shows that GREFT only uses twice the resources than vanilla GPS.

Chapter 9

ETSCH: Partition-centric Graph Processing

As presented in Chapter 8, there are many different frameworks for large-scale graph analysis, covering from small machines to large distributed systems. This chapter introduces our novel approach, named ETSCH. Our proposal falls in the area of partition-centric frameworks, similar to GIRAPH++ and BLOGEL, but with a small wrinkle: partitions are edge-centric instead of vertex-centric. We investigate the consequences of this choice on the programming model and use it to examine the performance of systems such as HADOOP, SPARK and AKKA when used as building blocks for the construction of ETSCH. We show that our AKKA implementation can scale to billion edges graphs using few resources and offers to the user a simple, intuitive programming model.

9.1 Introduction

In recent years, there has been strong focus on frameworks that are specialized on graph computation. PREGEL [70], the most influential of these frameworks, has been designed following a vertex-centric philosophy: each vertex is considered as a single processing unit that receives information from neighbor vertices, computes a very simple function and sends up-to-date information back; all these steps are performed in periodic rounds.

While this is a very simple and elegant paradigm, it does not take into consideration two facts:

- many interesting graph properties could be more easily computed by considering larger groups of vertices and edges, beyond the adjacency list of a single node;
- when partitioned along their vertices, real-world graphs tend to generate unbalanced partitions, particularly in natural graphs with power-law degree distributions.

In this chapter we make the case for a different approach based on the following ideas:

- graphs are partitioned into a collection of subgraphs, and each of them is assigned to a different worker. Each worker computes a function over the *entire subgraph*. Often such functions are simple implementations of classical graph algorithms, e.g. graph traversals. The results obtained in each separate subgraph may be later reconciled into a global view by using the vertices and edges shared between different workers as *communication channels*.

- the partitioning process happens along the edges and not the vertices, meaning that each partition is actually a collection of edges; the vertices associated to those edges may be replicated between distinct partitions.

Both these ideas are not entirely new. In most parallel systems, the vertices are divided into non-overlapping subsets. Edges between vertices that have been assigned to distinct partitions act as communication channels between the partitions themselves. In a distributed setting this approach has been followed by a few frameworks, such as BLOGEL [122] and GIRAPH++ [110]. Furthermore, to solve the unbalance problem, systems like GPS [100] are able to partition the outgoing edges of high-degree vertices in different workers.

ETSCH is the first distributed graph processing framework that combines the ideas expressed above: computation is associated to partitions rather than vertices, and partitions are edge-disjoint rather the vertex-disjoint. Each machine will be responsible for a collection of edges, while communicating with other machines through vertices that appear in multiple partitions. A simple programming model is defined to support such abstraction. Experiments performed in the Amazon EC2 cluster show that ETSCH is highly scalable and outperforms or is on par to the most important distributed graph processing frameworks.

The rest of this chapter is structured as follows: Section 9.2 presents the architecture of ETSCH, together with a few sample algorithms implemented on top of it. Section 9.2.3 discusses the different options for partitioning the graph the best way to allow speedy computation by ETSCH. The experimental results are presented in Section 9.3, while ideas for future work and extension are included in Section 9.4.

9.2 ETSCH

Figure 9.1 shows the differences between partitioning the vertex set and partitioning the edge set (*vertex partitioning* and *edge partitioning*).

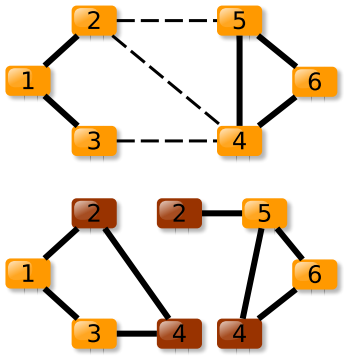


Figure 9.1: Example of vertex and edge partitioning: on the top figure, vertices are partitioned and a few *cut edges* connect vertices belonging to different partitions. On the bottom figure, edges are partitioned and a few *frontier vertices* appear in more than one partition

When a graph is subdivided using a vertex-partitioning algorithm, each subgraph has a number of external edges that connect vertices across partitions. These are called *cut edges* and are not really part of the subgraph, since the partition does not have knowledge of the other endpoint of the edge. The approach in this case is to consider vertices as computational entities that "send" messages to their neighbors, potentially across partitions using cut edges.

This is not the case with edge partitioning. Both vertices and edges of a local graph can be associated with local state. Edges are part of exactly one subgraph, therefore their states belong exactly to one partition. The same happens with vertices that are not replicated. The nodes that are replicated in different partitions are called *frontier edges*; their state need to be periodically reconciled.

Figure 9.2 shows the organization of ETSCH. First of all, the graph is decomposed into K partitions by an edge-partitioning algorithm. Each partition is assigned to a different *worker*, which executes the following steps:

1. The *initialization phase* is run once, by taking the subgraph representing the partition as input and initializing the local state of vertices and edges.

2. Once completed the initialization, each subgraph state is fed to the *local computation phase*, that runs an independent instance of a sequential algorithm that updates the local state of the subgraph.

3. The *aggregation phase* logically follows the local computation: for each frontier vertex, the framework collects the distinct states of all replicas and computes a new state, that is then copied into the replicas.

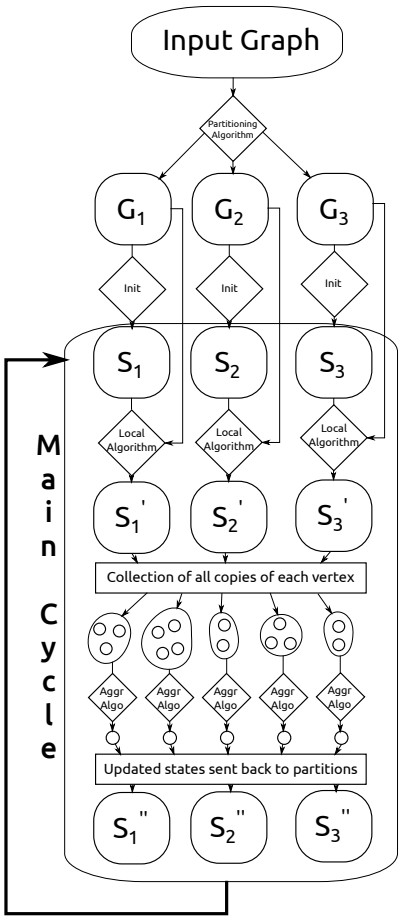


Figure 9.2: Illustrative schema of ETSCH

Step (2) and (3) are executed iteratively, until the desired goal is reached and the distributed algorithm has completed its goal.

In order to use ETSCH, three functions corresponding to the three phases must be implemented. Functions init() and localComputation() take a subgraph as input and perform their computation on it. aggregation() takes an array of replica state (whose type is defined by user) and should return a single state that reconcile those contained in the array. The framework takes care of calling init() and localComputation() in each of the worker, and provides them with a subgraph to be computed; it then collects the replicated states from the replicas, calls aggregation() on them and then copy the aggregated state back to the replicas.

Init Algorithm	$G \to G'$	applied only once
Local Algorithm	$G \to G'$	for each partition at start iteration
Aggregation Algorithm	$State\ list \to State$	for each frontier node at end iteration

Algorithm 5: Distance computation

```
function init(SubGraph G_i)
    foreach v ∈ G_i.V do
        if v = source then
            v.dist = 0;
        else
            v.dist = ∞;

function localComputation(SubGraph G_i)
    changed = true;
    while changed do
        changed = false;
        foreach e ∈ G_i.E do
            s = e.sourceVertex;
            d = e.destVertex;
            if s.dist + 1 < d.dist then
                d.dist = s.dist + 1;
                changed = true;
            else
                if d.dist + 1 < s.dist then
                    s.dist = d.dist + 1;
                    changed = true;

function Distance aggregation(Distance[] D)
    return min(D);
```

9.2.1 Application examples

Algorithms 5 and 6 provide a couple of application examples; the former shows how to compute the distances of vertices from a source vertex, while the latter shows how to identify the connected components of a graph using ETSCH.

For the problem of distance computation (Algorithm 5), each vertex is associated with a state containing just the distance variable *dist*. Initially, all vertices are initialized to $+\infty$, apart from the *source* vertex which is initialized to 0. In the local computation phase, the vertices distances are updated by executing the Bellman-Ford algorithm until no changes are made. In the aggregation phase, replicated states of vertices are represented as a vector of distances, from which the minimum distance is taken.

Algorithm 6: Connected components computation

```
function init(SubGraph G_i)
    foreach v ∈ G_i.V do
        v.id = random();

function localComputation(SubGraph G_i)
    PQ = new PriorityQueue⟨NODE⟩();
    foreach v ∈ g_i.V do
        PQ.add(v);
    while not PQ.() do
        q = PQ.pop();
        foreach v ∈ q.neighbors do
            if v.id > q.id then
                v.id = q.id;
                PQ.update(v);

function aggregation(ID[] D)
    return min(D);
```

This approach decreases substantially the number of iterations needed by the framework to complete its execution. If the shortest path between a node and the *source* passes through K partitions, that node will have the correct distance from *source* after only K iterations. Since the length of a path is an upper bound to the number of partitions traversed by that path and a vertex-centric algorithm can only move by one edge during each iterations, our approach can reduce the amount of iterations needed to converge. Comparing the number of iterations needed by the vertex-centric approach against our partition-centric approach, we measured a 30% decrease on small-world graphs and over 95% decrease on road network with large diameter.

The algorithm for computing the connected components uses a variation of Dijkstra's Algorithm (Algorithm 6). Each vertex is associated with a connected component identifier id, which is generated randomly for each vertex. The local computation phase epidemically spread the smallest component identifier by passing it through the local edges, until all vertices have been reached. In the aggregation phase the smallest identifier is selected from all the replicas and returned as their connected component identifier. Eventually, each connected component will be identified by a single value, which is the smallest identifier randomly generated in each connected component.

Algorithm 7: Gather-Apply-Scatter

```
function localComputation(SubGraph G_i)
    foreach (u, v) ∈ g_i.E do
        D_{u,v} = scatter(D_u, D_{(u,v)}, D_v)
    foreach u ∈ G_i.V do
        foreach v ∈ u.neighbors do
            u.Accum = sum(u.Accum, gather(D_u, D_{(u,v)}, D_v));

function aggregation(Accum[] a, D curstate)
    Accum cur = nil;
    foreach i ∈ a do
        cur = sum(cur, i);
    return apply(cur, curstate);
```

Both are basic problems that can be used as building blocks for other, more complex computations. For example, the problem of distance computation is needed to compute properties like betweenness centrality [22]. It is also possible to implement Luby's maximal independent set algorithm [69] in ETSCH, by spreading the random values in the local phase and choosing if a vertex must be added to the set in the aggregation phase.

9.2.2 Applicability of Etsch

As a general guideline, problems that need several iterations to complete in a vertex-centric framework are the ones that can gain the most from a partition-centric framework such as ETSCH. In other problems, such as computing the number of triangles in a graph or solving PageRank, most algorithms need only local computation and therefore a vertex-centric interface can allow simpler algorithms. For these reasons we implemented the *gather–apply–scatter* model on top of ETSCH, as shown in Algorithm 7. The *scatter* phase is executed in the local phase and the messages sent inside the partitions can already be *gathered* and collected. The ETSCH aggregation phase will collect the messages sent to that node from different partitions and *apply* it to the state of the node.

This additional module will allow users to run a program written in GAS or following ETSCH programming model, while following the same partitioning. Some of the experiments presented in Section 9.3 has been obtained by running the standard PageRank algorithm on top of this *gather–apply–scatter* module (see [44] for the pseudocode).

9.2.3 Partitioning schemes

The quality of the partitioning chosen for the execution of ETSCH can have a huge impact on the efficiency of the system and deserves some careful consideration. Since ETSCH uses edge partitioning, we will use the same definition introduced in Chapter 6. The most important metric to consider to evaluate the quality of such partition is the number of frontier nodes, the vertices that appear in more than one partition and thus cause the creation of replicas and the need of the aggregation phase. A smaller amount of frontier nodes affects the execution of ETSCH in different ways:

- Workers need less memory to store all the replicas that refers to the copies of the frontier vertices
- ETSCH sends less messages to reconciliate the replicas of each node
- The aggregation phase of ETSCH is shorter, since there are less nodes on which it has to execute.

While having less frontier nodes is always better, there is a trade-off between the time spent to improve the partitioning and the time needed by ETSCH to complete its execution. It is clearly not a good idea to run a complex, slow partitioning algorithm if the algorithm to be run is not very time-consuming, or if the system will spend more time in partitioning the graph than in analyzing it.

There are many very fast hash-based random partitioning algorithms that can be used when the quality of the partitioning is not very important, but they tend to create extremely disconnected partitions and a huge number of frontier nodes. ETSCH can still be executed on these partitions but the advantages of a partition-based framework might disappear.

A better compromise is POWERGRAPH's random vertex cut [44], a greedy algorithm that computes a good edge partitioning by following few simple rules for each edge. If there is a need of a more connected partition, DFEP [45] is a diffusion-based algorithm that, while slower, will allow ETSCH to run more efficiently. JA-BE-JA [94] needs even more iterations to converge, but computes extremely balanced partitions with good connectedness.

In many cases, the graph comes with additional information about its vertices and edges. A web graph can contain the URL of the pages, a road network can contain information about the coordinates or the type of connections, collaboration networks can contain data about the people involved. This information can be exploited to obtain high quality partitions very quickly. In our experiments in Section 9.3 that involve web data, we used a variant of the URL partitioning algorithm offered by BLOGEL [122] that groups together all out-edges of each web domain and then assign greedily each group to the

less loaded partition. We found that the fraction of frontier nodes in web graphs can be very few (around 2% in our experiments) and therefore the amount of bandwidth needed is very small. We estimate less than 7Mb of data was sent around the network for each iteration of PageRank of ETSCH on the ARABIC graph, which contains over 20 million nodes.

9.3 Results

ETSCH has been implemented on top of three different frameworks for distributed computation, HADOOP, SPARK and AKKA. This section compares the three implementations of ETSCH against native vertex-centric algorithm in the respective frameworks. A final comparison shows that our AKKA implementation of ETSCH is orders of magnitude faster than the other implementation and obtains generally better results with respect to competitor frameworks such as BLOGEL and GPS.

All these experiments have been executed using Amazon AWS. The machines used in the experiments are `m3.large`, equipped with High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors (2 virtual cores) and 7.5 GB of RAM.

9.3.1 Datasets

In our experiments, we used different datasets to test the performance of our framework on different problems. The five graphs that are used in our implementation of ETSCH in SPARK and HADOOP are presented in Table 9.1. DBLP is the co-authorship network from the DBLP archive, YOUTUBE is the friendship graph between the users of the service while AMAZON is a co-purchasing network of the products sold by the website. Finally, ROADNET-PA is a large-diameter graph representing a road network and LIVEJOURNAL is a large social network graph with a small diameter. All these networks have been taken from the SNAP graph library [66] and cleaned for our use, making directed edges undirected and removing disconnected components. These datasets do not contain any additional information outside of the structure of the graph and have been partitioned using DFEP [45].

The larger datasets used in AKKA, presented in Table 9.2, have been downloaded from the Laboratory of Web Algorithmics of the University of Milan[1]. Such datasets are compressed via LLP [19] and WebGraph [20] and contain additional information about the domain of the web page. We partitioned them using a variant of BLOGEL's URL partitioner based on edges rather than vertices.

[1] `http://law.di.unimi.it`

Name	$\|V\|$	$\|E\|$	$\varnothing$	d_{max}
DBLP	317,080	1,049,866	21	343
YOUTUBE	1,134,890	2,987,624	20	28754
AMAZON	400,727	2,349,869	18	9905
ROADNET-PA	1,087,562	1,541,514	784	9
LIVEJOURNAL	3,997,962	34,681,189	16	14815

Table 9.1: Datasets used with ETSCH/HADOOP and ETSCH/SPARK

Name	$\|V\|$	$\|E\|$	$\varnothing$	deg_{max}
INDOCHINA	7,414,866	194,109,311	28.12	6985
UK-2002	18,520,486	298,113,762	21.59	2450
ARABIC	22,744,080	639,999,458	22.39	9905
UK-2005	39,459,925	936,364,282	23.19	5213

Table 9.2: Datasets used with ETSCH/AKKA

9.3.2 Hadoop

ETSCH has first been implemented as a MAPREDUCE job on top of Apache HADOOP. HADOOP has arguably the largest user base between all big-data analysis frameworks, and its implementation, while not very efficient, is very stable without any unpredictable behavior.

To test the practical advantages of ETSCH we first prepared a HADOOP implementation of the framework in which the user can define the three functions as defined in Section 9.2. We compared this approach against running a baseline vertex-based implementation of the shortest path algorithm on the unpartitioned graph, in which every vertex sends messages in the Map phase and receive them in the Reduce phase. Figure 9.3 shows that our approach is much more efficient when the number of processing nodes is small, since the partitions are larger and paths are more easily compressed. When the number of partitions grows, the baseline approach gets closer to ETSCH, but the latter is still more efficient.

9.3.3 Spark

Apache SPARK/GRAPHX is both faster and more flexible than HADOOP. Since GRAPHX uses internally edge partitions it was possible to implement ETSCH on top of it with minimal additions to the GRAPHX codebase.

As the experimental results show, SPARK is much more efficient but is still unstable

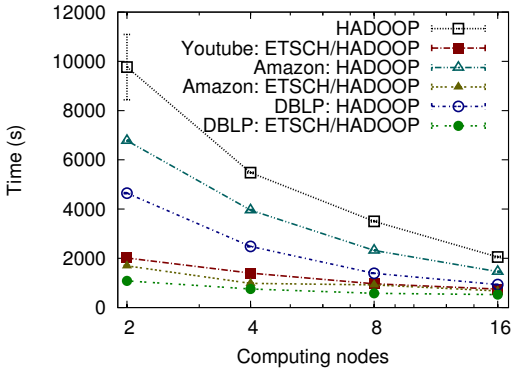


Figure 9.3: Running time of single source shortest path algorithm in HADOOP comparing a standard baseline algorithm and ETSCH, with `m3.large` machines on EC2.

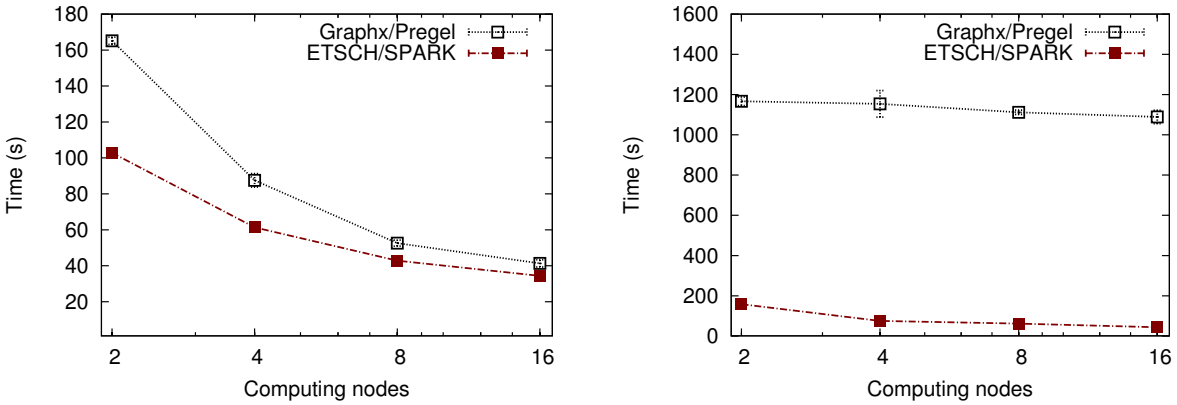
and we encountered strange behaviors when the memory available is smaller than optimal. This is expected since SPARK makes a larger use of the memory than HADOOP.

As before, we compared the single-source shortest path algorithm implemented on ETSCH with the standard PREGEL approach, following the example presented in the GRAPHX programming guide. Given the large speedup obtained by moving from HADOOP to SPARK, we were able to use the last two datasets from Table 9.1 as well.

We show the experimental results in Figure 9.4. In the LIVEJOURNAL graph we can see that our approach is faster, but the greater the number of partitions the smaller is the speedup caused by processing each partition as a subgraph. The ROADNET-PA graph shows that, as expected, when the diameter is huge ETSCH is extremely efficient. While the PREGEL version needs hundreds of iteration to complete, ETSCH finishes in only few iterations thus decreasing the synchronization overhead.

9.3.4 Akka

AKKA [3] is a Java framework for parallel and distributed Actor-based programming. AKKA offers the possibility to create generic *actors*, stored using very few memory, that can react asynchronously to incoming, user-defined messages. While the absence of graph-specific services meant more work to implement ETSCH, it allowed us to avoid using work-arounds to overcome the other frameworks' limitations. Our current implementation does not yet use AKKA's services for obtaining fault-tolerance and load-balancing and assumes failure-free executions. While we plan to extend our framework to deal with failures in a future work, we underline that even the largest datasets tested in this section are analyzed in less than one minute, and thus it is perfectly feasible to re-start the entire job in case of failures.



(a) LIVEJOURNAL (b) ROADNET-PA

Figure 9.4: Comparison between ETSCH and the standard PREGEL implementation in SPARK/GRAPHX

Our ETSCH implementation creates one single actor for each worker and one actor as a master. Currently the master actor functions only as a check-in point for the actors to discover the system and does not do any computation. The worker actor store their partitions and executes the algorithm defined on top of ETSCH programming model.

Since these experiments are executed on web graphs, in the rest of the section we focus on the PageRank problem and we compare ETSCH, BLOGEL and GPS against it. We believe that the advantages of a partition-centric approach in problems such as single-source shortest path have been already shown by the previous sections and by results in [122] and [110]. For ETSCH, we implemented the standard PageRank algorithm on top of the Gather-Apply-Scatter module presented in Section 9.2.

Figure 9.5 shows the scalability of our approach, running PageRank on the 4 large datasets in Table 9.2 on different numbers of `m3.large` machines. ETSCH can analyze even the largest, 1–billion edges dataset with only 4 machines, but the addition of more machines allow the system to use the memory more efficiently and thus significantly speed up the computation. Using 8 machines on the UK2005 causes a 2.37 speedup with respect to the same experiment with 4 machines. Investigating this result indicates that when the number of edges is too big, the system is slowed down by Java's garbage collector. Moving away from Java's collections and using ad-hoc, array-based data structure might be a solution that we will explore in future works.

Even with these memory issues, our AKKA implementation is order of magnitude faster than implementations on other frameworks, as seen in Figure 9.3

Dataset	ETSCH/ HADOOP	ETSCH/ SPARK	ETSCH/ AKKA
DBLP	755s	7.8s	1.75s
YOUTUBE	1400s	11.1s	2.45s
AMAZON	984s	12.2s	2.33s
ROADNET-PA	NA	75.1s	1.43s
LIVEJOURNAL	NA	61.3s	8.78s

Table 9.3: Comparing different frameworks for ETSCH, using 4 machines.

9.3.5 Comparison with Blogel and GPS

Comparison against other frameworks is shown in Table 9.4. Given that in most datasets we were not able to run BLOGEL and GPS on only 4 machines, we decided to run the following experiments using 8 `m3.large` machines in EC2. We executed BLOGEL's URL partitioner on the datasets and then ran both BLOGEL and GPS on the same partitioned graph. Since GPS implements only a round-robin scheme, we exploited the BLOGEL URL partitioning scheme and assigned node identifiers so that GPS's round-robin distribution reflects the URL partitioning. For each of them we measure the running time of the vertex-centric PageRank part of computation, discarding the setup phase, and we divided it by the number of iterations needed to converge to measure the average running time per iteration.

Aside from UK-2005, where some memory issues with the Java's garbage collector make ETSCH slightly slower than BLOGEL, our approach is significantly faster. An interesting behavior can be noticed in the processing of INDOCHINA. Despite the smaller size of that dataset with respect to UK-2002, both BLOGEL and GPS take more time to complete a PageRank iteration. We investigated the problem and we found that the reason is that the partitions created by BLOGEL are balanced with respect to the number of vertices, but not with respect to the number of edges. This is a limitation of vertex-partitioning schemes: since we are dealing with graphs with a power-law degree distribution, the number of edges inside the partitions can be wildly different even if the number of the nodes stays the same. In the case of INDOCHINA, BLOGEL created a partition that has 4X more edges with respect the other partitions, thus slowing the system significantly.

GIRAPH++'s implementation of PageRank has been executed on UK-2002 and UK-2005 by the authors in their paper. Their reported results (respectively 4 and 13 seconds) are worse than what both BLOGEL and ETSCH/AKKA obtain; furthermore, note that they used more resources (10 quad-core machines with 32G of ram against 8 `m3.large` machines).

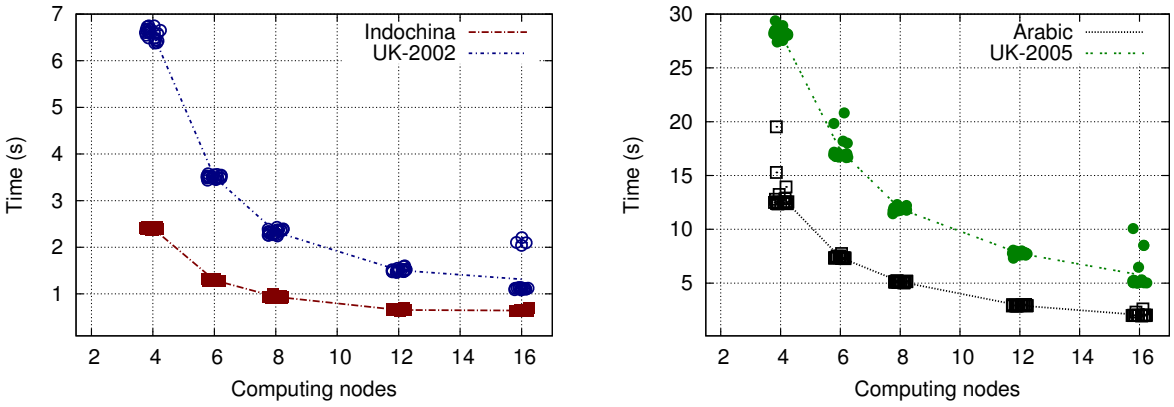


Figure 9.5: Scalability of ETSCH/AKKA on the larger datasets, using `m3.large` machines on EC2. For each experiment we measure the average time for a complete iteration of PageRank

Dataset	ETSCH/AKKA	BLOGEL	GPS
INDOCHINA	$0.94s \pm 0.01s$	$4.51s \pm 0.01s$	$10.37s \pm 0.17s$
UK-2002	$2.33s \pm 0.05s$	$3.44s \pm 0.02s$	$8.11s \pm 0.18s$
ARABIC	$5.12s \pm 0.05s$	$8.49s \pm 0.04s$	$15.42s \pm 0.09s$
UK-2005	$11.90s \pm 0.18s$	$10.91s \pm 0.08s$	$23.51s \pm 0.05s$

Table 9.4: Comparison of running time of a single PageRank iteration in ETSCH, BLOGEL, GPS (8 `m3.large` machines)

9.4 Conclusions

Our experimental results, obtained through real implementations and actual deployment on an Amazon EC2 cluster, show that ETSCH scales well and can process even very large graphs efficiently, but is still far for being a mature system.

As future work, we plan to thoroughly study the ETSCH framework, both from a theoretical and a practical point of view. We plan to investigate how flexible the model is, to understand which type of graph problems are solvable and which ones need a completely different framework. For some problems, the classical solutions could be easily translated into ETSCH, while for others novel algorithms could be needed. On the technical side we plan to add fault tolerance and to the system, by exploiting AKKA's features, and make it usable by general users. We also want to add the partitioning algorithm to the system, to allow users to download, partition and analyze a large graph with a single command.

An even more exciting project would be to extend ETSCH programming model to avoid the need for synchronization. In this extended frameworks both replicas and partitions would be able to execute in random order, thus hopefully allowing less waiting time and

more efficient analysis of the graph. Section 8.6 shows that adding asynchronism to a synchronous model even with simple optimizations can lead to a more efficient, scalable system.

Part IV

Applications of Large-scale Graph Analytics

Chapter 10

Applications of Graph Processing

Because of the extremely different meanings that can be given to different graphs, there are at least as many application scenarios as sources of data. Some of them are just applications of well-known problems in graph theory: for example, computing shortest paths in a graph can be useful in almost any type of graph, even if the meaning assigned to the paths can be different. Some are more application specific: studying congestion in a network is an important problem in computer and road networks, but is not very useful on a social one.

The development of graph processing systems gave researchers new opportunities for developing and running large-scale graph algorithms using the new programming models. Many old, parallel or peer-to-peer algorithms have been adapted to the different programming models [101] but some of them do not lend themselves to such a process and need to be reinvented.

In this chapter, we look at a representative set of interesting graph problems. For each of them we introduce the standard, parallel approaches to solve those problems and the work done to adapt such algorithms to modern programming models. Chapter 11 will focus on the clustering problem, introducing a new distributed clustering algorithm for large-scale graphs and showing a direct application in the field of word sense induction and disambiguation.

10.1 Strategies

We first start this overview by introducing three different strategies commonly used in distributed graph algorithms. While different problems ask for different algorithms, many challenges can be tackled using one or more of the following strategies: using *diffusion* algorithms to disseminate the information across the graph; using *divide-and-conquer* algorithms to independently analyze subgraphs and compute common results; using *com-*

pression algorithms to allow analysis on the coarsened the graph or compress the amount of data associated to vertices or edges.

Diffusion In this class of algorithms, each node in the graph has a starting state, communicates with its neighbors and disseminates the information across the network. This approach is commonly used when nodes must compute a value, like distance from a seed node or a centrality index, and has been the inspiration for the PREGEL programming model. Usually algorithms that use this technique have a running time that heavily depends on the diameter D, the length of the shortest path of maximum length, since only after D steps we have a guarantee that the information contained in a single node at the start of the computation has had enough time to propagate to all other nodes. While many interesting graphs have the "small-world" property [120] and the diameter might be in the order of $\log(N)$, this property might still entail a large number of iterations to complete.

Divide-and-conquer If the graph is divided into K subgraphs, such as what happens in distributed graph analysis frameworks, it is possible to use a "divide-and-conquer" approach by computing intermediate solutions for each of the subgraphs and then compose them to get the final solution. This approach can also decrease the average path-size, improve the performance of diffusion-based algorithms and is the inspiration of the partition-centric programming models such as those introduced in Section 8.3 and studied in Chapter 9.

Compression Compact representation of data can decrease both the memory costs and the time needed to complete the computation. This approach can be applied either on the graph itself, by shrinking the graph to a more manageable size and working on a smaller version of it, or by using different kind of "sketches" to decrease the amount of memory needed by each node to keep track of its state. These sketches are very common in the field of streaming algorithms, where the amount of memory available for the algorithm is very scarce. In both cases, this approach allows for approximate answers and can be combined with the other two strategies. This class of approaches has also been widely used in partitioning algorithms (see Section 6.5.4) and can be applied to more general computation.

10.2 Overview of graph problems

The three strategies introduced above can be combined and independently applied to a large set of graph problems. In this section, we introduce a few of them, with some possible solutions based on one or more of the three strategies.

10.2.1 Triangle counting

A triangle in a graph is defined as a triple of vertices such that there is an edge between any pair of them. While finding these triangles has no immediate application, counting the number of triangles in graph is a basic operation that can be used to compute many important statistics, such as the clustering coefficient. We describe this problem to introduce practical applications of the three strategies introduced in Section 10.1.

One simple, yet inefficient method to compute the number of triangles is to take the adjacency matrix $\mathbb{M}$ and compute $\mathbb{M}^3$. The values on the diagonal will represents the number of paths of length 3 from a node to itself, and dividing them by three will give us the number of unique triangles. This process must be done without explicitly writing the matrix in memory, or the space needed would be too large.

Since this problem has only local dependencies, there is no need to diffuse information across the entire graph: diffusion-based algorithms do not need many iterations to converge. If each node had a view of the adjacency list of all its neighbors, it could easily compute the number of triangles it takes part in. This is still not a feasible solution, since most real-world graphs have a power-law degree distribution and therefore a single node might have to work on a huge quantity of data. A possible solution to this problem [29] is to convert the undirected graph into a directed one, with edges going from the node with the lowest degree to the node with the highest degree. Using this strategy the out-degree is kept low even in presence of a power law and each node will send a reasonable amount of messages to its neighbors.

The natural divide-and-conquer strategy of counting the triangles in each subgraph and then starting aggregate queries to count the triangles that are on the frontier between partitions looks promising but is not very efficient, since the number of triangles on the frontier can be extremely large. A more elaborate approach [108] can bring better results: all vertices are split into k subsets $(V_1, V_2, ..., V_k)$. Each process in the distributed system will work on the subgraph induced by $V_i \cup V_j \cup V_k$, for different values of i, j, k and count the number of triangles inside that subgraph. Since all triangles are into at least one of these triplets of sets, each process will be able to work completely independently and the system will compute the correct number of triangles.

We complete this overview with an example of a compression-based algorithm [114].

The idea is very simple: each edge is kept with probability p and deleted with probability $1 - p$. Each triangle in the resulting graph is equivalent to $\frac{1}{p^3}$ triangles in the original graph. The paper shows that the mean resulting value of the algorithm is correct and provides upper bounds on the variance, while showing that in practice this approach gets very precise results.

10.2.2 Centrality measures

The most famous centrality measure of modern graph analysis is PageRank [23], which simulates infinite random walks over the web graph with a probability of a jump to a random node. The standard implementation is a vertex-centric algorithm based on diffusion, which converges in a fixed number of rounds.

Interestingly, even this natively diffusion-based problem can be solved more efficiently by using a combination of divide-and-conquer and compression strategies. BlockRank [53] uses the block-centric structure of the web to compute a good starting value for each node and decrease the number of iterations needed to converge. The input graph is divided in blocks according to the domain of the vertices. For each block, each process will run the local PageRank of those nodes (thus using the divide-and-conquer approach). Additionally, BlockRank construct a block graph in which each domain is represented by a single node. Running the PageRank algorithm on this compressed graph will obtain an estimation of the importance of each block in the web. BlockRank will then compute a starting PageRank value for each node using both its local PageRank and the PageRank of its block.

A more challenging but extremely useful centrality measure is the *betweenness centrality* [98]. The betweenness centrality of a single node n is defined as the fraction of shortest paths between all pairs of nodes passing through n. Intuitively, a node with high betweenness centrality is highly involved in the shortest paths of the graph. This measure is often used by sociologists to find influential vertices and by network analysts to compute better routing and recover connectivity in case of node failure.

The most efficient centralized algorithm [22] follows this procedure: it repeatedly runs a BFS from each single node and then collects information on the paths moving from the bottom of the BFS tree to the original node. This procedure must be repeated for each possible source node with total complexity equal to $O(V(V+E))$. Both steps of this algorithm can be easily converted in a diffuse pattern and implemented in a vertex-centric programming model. Still, each node will need to take track of the $O(N)$ parallel BFS's which is infeasible on large networks. Running the BFS's in batches can alleviate the memory concerns, but increase greatly the number of iterations needed to converge.

Research has focused on two possible solutions for the demanding amount of resources

needed to compute this centrality measure: heuristics that use common pattern in real world networks can speed up computation by a large factor by collapsing similar nodes and paths [91], or algorithm can relax the requirements for an exact answer and allow approximated results. For example, it is possible to reach relatively precise results using just a fraction of the resources needed by the exact algorithm by using compressed data structure that allow each node to store the approximate states of many BFS at the same time, or running the BFS's just from few carefully chosen nodes [9].

10.2.3 Path computation

The distance from a node can be computed quite naturally using a diffusion-based algorithm in a programming model such as PREGEL, but this standard approach has some drawbacks. The main issue of that algorithm is that it needs needs $O(D)$ iterations in the BSP model to compute the distance between any two nodes. Since pre-computing all the distances between all pairs of nodes is also infeasible on a large graph, there is a need for faster algorithms whenever a quick response is needed.

A divide-and-conquer approach can help in decreasing that large number of iterations. The intuition is that if a graph is "perfectly" partitioned, then the path between two nodes should only traverse each subgraph once, thus reducing the number of communication rounds needed to complete. In Chapter 9 we show a few implementations of this approach and their gain in efficiency.

A possible compression-based approach is presented in [30]. This approach uses the triangle inequality (for every vertices v,w and y, $D(v, w) \leq D(v, y) + D(y, w)$) and takes the sum of distances to a common node as an upper bound on the distance between two nodes. A set S of k nodes is chosen at random and we compute for every node in the graph its distance to the closest node in S. This property can be computed with a single breadth-first search or a single instance of Dijkstra's algorithm if the graph is weighted. If two nodes v and w have found the same node $s \in S$ as the closest, $D(v, s) + D(w, s)$ is an upper bound on $D(v, w)$. Choosing k correctly is a crucial problem, since a low k means that s can be very far from v and w, while a large k will decrease the probability of getting v and w to have the same choice as the closest node in S. The solution is to have $O(\log(N))$ instances of this algorithm with k increasing exponentially. After the precomputing part, consisting in $O(\log(N))$ instances of a breadth-first search, each node will keep a table of $O(\log(B))$ pairs nodes/distance as a compact representation of the entire graph. The estimation of the distance between two nodes consist in simply comparing the two sketches to find distances to common nodes and can be easily done in a single iteration.

Computing the diameter of a graph, the length of the longest of all shortest paths,

can give insight into the structure of the graph. The obvious algorithm computes the distances between all pairs of nodes, but this approach is infeasible on very large graphs where even a $O(N^2)$ running time can be too large. Even if it could be possible to compute the distance between two nodes in $O(1)$ using one of the methods shown in the previous section, checking all pairs of nodes would be too slow to complete in time.

One possible approach [54] uses the Flajolet-Martin counting sketches [37] to compute an approximation of the diameter. The FM-sketch is a compact way to approximate the number of distinct values in a sequence, with the useful property of having the union of two sketches to simply be the bitwise-or of the two sketches. Be $S_{v,k}$ the sketch corresponding to the set of nodes reachable from v in k steps and $N(v)$ the set of neighbors of v, we can compute $S_{v,k} = \bigcup_{w \in N(v)} S_{w,k-1}$. At each step each node will send its FM-sketch to its neighbors and compute the bitwise-or of the sketches received to get its own sketch for the next iteration. When all estimates of the number of reachable nodes have reached a certain threshold, we have an estimate of the diameter.

In the presence of additional knowledge about the graph, it is possible to use that information to get more efficient and precise algorithms. Aridhi et al. [7] show an interesting application of the divide-and-conquer technique for computing the shortest path over geographical graphs: the vertices are partitioned according to their position relative to the straight line connecting the source and destination vertices. Each subgraph is then processed using a standard single-source shortest path algorithms and the union of the different shortest paths are then combined via an iterative improvement technique.

Holzer et al. [50] introduce an algorithm that allow the distributed computation of diameter and all-pairs shortest path by carefully managing the starting time of each visit of the graph. The algorithm first constructs a BFS tree of the graph and starts traversing this tree using a special *pebble*. Whenever the pebble reaches a new node, it will wait for an iteration before starting a concurrent BFS from that node. The authors prove that by using this approach all the different BFS's will not touch each other, thus avoiding congestion and extreme load on the different nodes.

10.2.4 Coloring

Graph coloring is one of the oldest problem in graph theory and has been object of great scrutiny. In particular, the four color conjecture, stating that any map can be colored using only four colors in such a way that no neighbors have the same, has been studied since the 1800s by mathematicians all over the world. In general, finding the minimum number of colors necessary to color correctly a graph is one of the original NP-complete problems studied by Karp [56]. While this problem has many applications, such as register allocation, sparse matrix computation or jobs scheduling, there are not many applications

on large-scale graphs. Therefore, the algorithms on this section are interesting mostly for their applications on parallel systems, while in distributed implementation the innate costs of setting up a distributed system for relatively small graphs may negate the gains in efficiency.

The most successful thread of research for parallel algorithms follows the approach used by Luby et al. algorithm to find the maximal independent set [69] and use a different color for each discovered set. Luby's algorithm starts by assigning a random value to each node; it then selects those nodes that have the largest value in their neighborhood and insert them in the output set. All nodes that have been selected are removed with their neighborhood from the graph, and the approach is repeated until the graph is empty. Since all decisions are made by looking only at the neighborhood of a node, it is possible to implement this algorithm using any of programming models introduced in Chapter 8. To compute the graph coloring, the user needs to color all nodes retrieved by Luby's algorithm with a unique color and remove them from the graph. Each run of the algorithm will find another independent set and a new color to be used.

Boman et al. [21] use a divide-and-conquer approach: the graph is divided in partitions and each process will color a partition independently. The processes will then coordinate to find all incorrectly colored nodes and re-color them using one of the offered strategies. While this approach has been tested on reasonably powerful clusters, it has not been subjected to an extended study on large graphs, since the biggest of the graphs included in the experiments had only a few million edges. A thorough study of implementations showed that many of the common known techniques can be implemented in systems such as HADOOP [40], but their experimental results are still inconclusive because of the small size of the cluster and the limited scale of the graphs used.

10.2.5 Subgraph matching

We conclude this chapter with the problem of finding subgraph matching: given a graph G and a query graph Q, find all subgraphs of G that are similar to Q. The notion of similarity can be captured by the concept of isomorphism or by more relaxed definition such as graph simulation. From the definition it becomes clear that this is a different type of problem from others included in this chapter, as it does not ask to compute a property about the graph or the nodes, but to retrieve the part of a graph that conforms to the given specification. This type of problems is much easier to be solved in graph databases than general graph processing system, since the query graph Q could be converted into a query written in the language of that database.

Sun et al. [107] tackle the problem of subgraph matching on billion-scale graphs in distributed memory. Since algorithms based on joins can be inefficient with this scale,

their proposed approach uses an improvement of the query decomposition idea: the main query is decomposed into smaller sub-queries, but they are not computed independently: subsequent sub-queries are started from already matched sub-queries. While the first call to the sub-query matching procedure might return many possible answers, all subsequent steps will prune them. The paper also introduces a few query optimizations to help the scalability of the approach.

The usage and optimization of sub-queries is a topic that has seen much research. An alternative optimization [123] notices that many sub-queries have parts in common and their output could be reused to avoid waste of resources. The proposed approach decomposes each subgraph in trees and uses a prefix tree to keep track of which of them have already been processed. By using a parallel approach also in the joining phase the algorithm can scale to billion edges graphs.

Fard et al. [34] introduce a vertex-centric approach that can be implemented in graph processing systems that offer PREGEL's programming model. The query graph is broadcast to all processes which will check, for each vertex, to which vertex of the query graph it can be mapped to. During each superstep all vertices communicate its possible matches to the neighbors, check if they are consistent with the proposed match and update their possible matching. When all nodes have voted to halt, the algorithm has terminated. The algorithm is implemented using GPS, and tested successfully on billion edges graphs from the web.

Chapter 11

Graph Clustering for Word Sense Induction

In this chapter we introduce a case study that uses graph technology for two language-based problems: word sense induction and word sense deduction.

Language has been ambiguous from its inception. Not only words have several meanings, but even given names (assigned to individuals, companies, or even cities) can be ambiguous. While Ulysses was able to use this feature to his advantage, ambiguity has generally created more harm than good. This problem has important consequences for web intelligence companies that want to extract public opinions and reaction to news and products from massive data sets acquired by mining social web. Are users complaining about Apple's new phone or about apples that they ate for lunch? Companies do not want to see noise and irrelevant information caused by homonyms and misspelling in their data.

Companies have similar issues when they need to re-conciliate their own data with user-input data or different data sources. Understanding the correct meaning for ambiguous words can help in cleaning their datasets, in correcting typos and small mistakes and assigning them to the right category. Because of this application this problem can be seen also as part of pre-processing step for successive data analysis and illustrates the fact that graph systems can be useful at any point of the pipeline.

In this chapter we propose an approach, based on the work of Rahimian et al. [92]:

- We take as input a set of documents to be disambiguated, where the important *words* (nouns, verbs, adjectives) have been identified; one target word is *ambiguous* and needs to be disambiguated (Section 11.1).
- We build a co-occurrence graph where words are nodes and two nodes share an edge if they occur in the same document (Section 11.3), as proposed in [92]

- Using our novel distributed graph clustering algorithm, TOVEL, we cluster together the documents that refer to the same ambiguous word (Section 11.4).
- All these steps are incrementally executed on incoming data by continuously adding nodes and edges on the ever-evolving graph without having to restart TOVEL from scratch(Section 11.6).

As shown in Section 11.7, TOVEL obtains very good results in terms of precision and recall, and outperforms existing approaches in terms of F1-score. Since in TOVEL all nodes act independently, our approach can be scaled to huge quantities of data by implementing it in one of the many distributed large-scale graph processing frameworks, such as GIRAPH, GRAPHX or GRAPHLAB. We thus demonstrate the scalability of TOVEL using GRAPHX in Section 11.7.

11.1 Problem statement

The problem of finding the correct meaning of a word can be defined in different ways, according to the specific requirements of the problem. The most common definition for *word sense disambiguation* asks for identification of the correct meaning of a word from a given set of possible meaning. In this chapter we solve another, related problem: *word sense induction*, in which words usages must be grouped according to their shared meaning. Our approach can be extended to word sense disambiguation by using a semi-supervised approach where a few data points are already disambiguated.

We can define the *word sense disambiguation* more formally: given a single *ambiguous word* W, a collection of possible *senses* or *meanings* m_1, m_2, ..., m_M associated with W and a set of *documents* $D = \{d_1, d_2, \ldots, d_N\}$ containing mentions to W, this task asks to understand which of the documents in D refer to the different sense of W. The number of documents N is usually larger than the number of senses M. Our approach is based on the following sub-problems:

#	Sentence
1	Many doctors would recommend eating **apples** for breakfast
2	Technology like **apple**'s watches could help doctors monitor their patients
3	**Apple** produces many distinct technologies, from smartphones to watches
4	Before eating an **apple**, I always check which is its variety.

Table 11.1: Simple example: 4 documents, *Apple* is the target ambiguous word with two different senses (the fruit and Apple Inc.)

- *Clustering*: compute a clustering of D such that documents that refer to the same meaning appear in the same cluster.
- *Disambiguation*: assign each document in D to one of the meaning of W, according to clustering obtained in the previous step.

If the algorithm stops after the Clustering phase, the problem solved is *word sense induction*, but the addition of the mapping step solves the *word sense disambiguation*. Table 11.1 contains an illustrative toy instance of our problem. The word apple can refer to at least two different meaning: the fruit apple and the company. To solve the clustering subproblem would mean to recognize that documents 1 and 4 refer to one meaning, while 2 and 3 refer to a different one. To complete the disambiguation subproblem we also need to map each of the document to a specific meaning (Apple Inc. for documents 2 and 3, apple the fruit for documents 1 and 4). Note that other possible meanings of Apple (such as the Beatles' multimedia corporation) are irrelevant to this instance of the problem, since there are no documents that refer to it.

11.2 Related work

Our approach is based on using graph clustering algorithms for word sense induction and disambiguation. Therefore, the related work introduces both traditional word sense disambiguation algorithms and an overview of graph clustering algorithms.

11.2.1 Word sense induction and disambiguation

There is a large body of work on word sense disambiguation in the NLP community [77]. The problem is typically seen as a classification task, where different senses of a word are classified into different generic classes. One of the most well-known approach is by Lesk et al. [65], which computed the size of overlap between the glosses of the target and the context words, as an indication for classification. Since then various efforts has been made to extend the original Lesk algorithm [62, 11, 12]. An inherent limitation in the Lesk algorithm, however, is that the results are highly sensitive to the exact wordings of the definition of senses [77]. To overcome this problem, some solutions computed the overlap differently. For example, Chen et.al [26] used tree-matching, and recently various solutions used vector space models [85, 1, 96]. They, however, struggle with the problem at large scale. TOVEL computes the overlap between various context words using a graph model, without having to concern about grammatical or syntactical properties, while it addresses the scalability problem with a highly parallel algorithm.

After deciding what information to use, the main task of classification starts. The solutions are either supervised [125, 88, 103], unsupervised [2, 25, 116], or semi-supervised. Supervised methods generally produce reasonably accurate results, but the training data is usually not available in the real-world applications. Therefore, semi-supervised and unsupervised methods have gained lots of attention. While unsupervised solutions exploit the dictionary entries or taxonomical hierarchies like WordNet [74], the semi-supervised solutions start with some labeled data as seeds [82].

11.2.2 Graph clustering

The research in graph community detection itself has produced numerous works [38]. The problem, which is known to be NP-Hard, has been addressed through various heuristic solutions. A few approaches use the spectral properties of the graph for clustering [81], by transforming the initial set of nodes/edges into a set of points in the space, whose coordinates are the element of the eigenvectors of the graph adjacency matrix. The intuition is that nodes that belong to the same community structure have similar components in their eigenvectors. The problem with spectral clustering is that computing eigenvectors for large graphs is non-trivial, thus this solution is not applicable in large scale.

Some other solutions aim for maximizing the modularity metric. This approach, which was first introduced by Girwan and Newman [80], involves iteratively removing links with the highest betweenness centrality, until the maximum modularity is achieved. The problem with this technique is that computing the betweenness centrality is a complex task and requires global knowledge of the graph, thus, it cannot scale in large graphs. Many others extended the modularity optimization idea and made an effort to make it faster and more efficient [28, 18]. However, [39] has shown that the modularity has a resolution limit, and therefore, these solutions sometimes show a poor performance, particularly if there is a large graph with lots of small communities.

There are also solutions based on random walks in graphs [89]. The intuition is that random walks are more likely to get trapped in the densely connected regions of a graph, which correspond to the community structures. Other ideas, which are similar to random walk in nature, include the diffusion [72] and label propagation [117] [17] techniques. These solution have shown reasonable performance in general, and for word sense disambiguation in particular [92, 82], while having a relatively low complexity. They can be applied to large graphs, thanks to their parallel nature. TOVEL has the closest resemblance to this family of solutions.

11.3 Graph construction

The first step of our approach is the construction of the word graph from the input documents. The graph construction follows the same principles of [92] and is illustrated in Algorithm 8. For each document d, we add a distinct *ambiguous node* representing the ambiguous mention of the target word W, identified by the document id; we create a new node for each context word w extracted from d, unless it already exists in the graph. The nodes representing the words contained in the document (either created or found through getNode()) are added to the set S; then, undirected edges are created between all pair of distinct nodes contained in S, with the effect of creating a clique among them.

For the sake of simplifying the notation and save space, in this chapter G is a multi-graph where parallel edges between pair of nodes correspond to stronger links between them. In the real implementation, edges are weighted and the interactions among nodes connected by edges are proportional to such weights.

The strategy for extracting context words from the document depends on the specific application area. In our current approach on text documents, we just select every noun and adjective from surrounding sentences, converting everything to lowercase. Note that this step is needed only when we need to disambiguate plain text. In some applications, such as the Spotify scenario presented in Section 11.7, the documents are already provided as a bag of context words.

In Table 11.1, all context words are underlined, while mentions of our target word are in bold. Figure 11.1 shows the graph constructed from this example. Aside from

Algorithm 8: Graph construction

```
G = new GRAPH()
foreach DOCUMENT d do
    NODE ambNode = new NODE(d.id)
    G.addNode(ambNode)
    SET S = {ambNode}
    foreach WORD w ∈ d do
        if not G.contains(w) then
            G.addNode(new NODE(w))
        S = S ∪ G.getNode(w)
    foreach u, v ∈ S, u ≠ v do
        G.addEdge(u, v)
        G.addEdge(v, u)
```

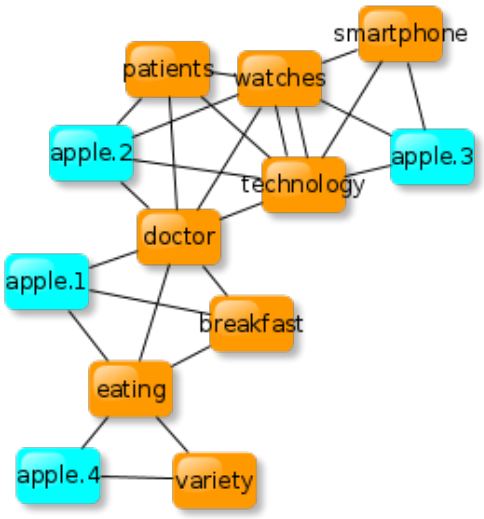


Figure 11.1: Sample graph created from Table 11.1. Blue nodes are ambiguous.

the blue nodes, one for each document, all other nodes are shared across the documents. Note that context words watches and technology co-occur in two different documents, thus increasing the strength of their connection. In a real implementation, we would have created an edge of weight 2 to connect them.

11.4 Tovel: a Distributed Graph Clustering Algorithm

Our novel clustering algorithm, TOVEL, is inspired by the cycle of water. Water of different colors is generated at initialization and competes for control of the graph through a cycle of diffusion, evaporation and rain. Some colors will disappear from the graph, while others will survive by following the shapes of the underlying clusters. The resulting clusters represent the different meanings of the ambiguous word.

Using the input graph constructed as in Section 11.3, we create one color for each node that represents the ambiguous word. This means that, initially, the number of colors will be equal to the number of documents. This choice is appropriate because we are not expecting more meanings than documents.

Note that, since the algorithm uses information about ambiguous nodes only at initialization, TOVEL could also be used on a general graph by starting with a reasonable number of colors from random starting position.

The algorithm can be summarized as following: during each iteration, each node diffuses its water to its neighbors. Each node will then decide independently its dominant color according to information in the neighborhood. All water of non-dominant colors

is evaporated and sent to the appropriate cloud, one for each color in the graph. Each cloud will then try to send back POUR of its water to all nodes with its dominant color, if enough water is present in it. The remaining water will be kept in the clouds for the following iterations. Once the algorithm has converged to a solution, all nodes with the same dominant color will form a cluster.

11.4.1 Data structures

Every node u contains a variable $u.color$ that represents the *dominant* color of that node and a variable $u.water$ containing the amount of water of color $u.color$ contained in u. Furthermore, a map H will be used to collect colors diffused from other nodes.

Apart from the set of nodes, we create a set of *clouds* $C_1, C_2, \ldots, C_n$, one for each color and thus one for each document. Each cloud C_i contains a amount of water of color i. As shown in Algorithm 9, each ambiguous node starts with a fixed amount of a unique color, identified by the unique id of the node. The non-ambiguous nodes and the clouds start empty. This is the only point in the algorithm where water is created.

Algorithm 9: Initialization phase

```
{ Executed by node u ∈ V }
if isAmbiguous() then
    u.color = u.id
    u.water = 1.0
else
    u.color = Nil
    u.water = 0.0
H = new MAP()
```

11.4.2 Main cycle

TOVEL is organized in consecutive rounds, each of them subdivided in three independent phases. In the first phase (*diffusion*), each node diffuses the water of its dominant color by sending it through each of its edges, divided equally among all of them. In the second phase (*evaporation*), each node computes a new dominant color and evaporates all the water of non-dominant colors by sending it to the clouds. In the third phase (*raining*), all cloud sends their water back to the nodes with their dominant color. An upper bound POUR is used to limit the rate of rain in each round.

Algorithm 10: Diffusion phase

```
{ Executed by node u ∈ V }
foreach NODE v ∈ u.neighbors do
    real amount = u.water/u.degree
    send ⟨u.color, amount⟩ to v
H.clean()
foreach NODE v ∈ u.neighbors do
    receive ⟨v.color, amount⟩ from v
    H[v.color] = H[v.color] + amount
```

Diffuse Each node in the graph sends all its water to its neighbors, divided equally among the (multi)edges connecting them. For example, if a node has 0.8 of red water and 4 outgoing edges, it will send 0.2 of red water along each of them. If two edges among those are pointing to the same node, that node will receive 0.4 of red water. The strengths of connections will influence the behavior of the diffusion process. Nodes will then wait until they receive messages from each of their neighbors, and aggregate the amount of water received in a fresh map indexed by colors.

Algorithm 11: Evaporation phase

```
{ Executed by node u ∈ V }
foreach NODE v ∈ u.neighbors do
    send⟨H⟩ to v
MAP H' = H.copy()
foreach NODE v ∈ u.neighbors do
    receive ⟨H_v⟩ from v
    foreach COLOR c ∈ H_v do
        H'[c] = H'[c] + H_v[c]
u.color = argmax(H'(c))
u.water = H[color]
H[u.color] = 0
foreach COLOR c do
    send ⟨H[c]⟩ to C_c
```

Evaporation Each node recomputes its dominant color by summing all the maps of its neighbors and choosing the color with the highest amount. In case of ties, nodes will choose the color with the lowest id. We chose this simple and deterministic heuristic since other approaches, such as breaking ties randomly, did not improve the quality of the clustering in our experiments. Computing the dominant color is the most computationally expensive step, but it allows us to better understand the shape of the cluster by looking at our neighbors neighbors [71]. By collecting the colors of our neighbor we can glimpse at what we will receive in the following iteration and choose how to interact with the clouds accordingly. Each node then sends all water of non-dominant color to the appropriate clouds. If a node has chosen red as its dominant color, it will send all of its blue color to the blue cloud.

Algorithm 12: Rain phase

```
{Executed by cloud C_c}
foreach NODE u ∈ V do
    receive ⟨amount⟩ from u
    C_c.water = C_c.water + amount
SET S = {u : u ∈ V ∧ u.color = c}
if |S| > 0 then
    rain = min(POUR, C_c.water/|S|)
    foreach NODE u ∈ S do
        send ⟨rain⟩ to u
    C_c.water = C_c.water − rain ∗ |S|
{Executed by node u}
receive ⟨amount⟩ from C_{u.color}
u.water = u.water + amount
```

Rain Each cloud receives water sent by the nodes in the graph and sums it with all water it kept since the previous iteration. It will then send some water back to the nodes following this procedure: the cloud of color c will compute the set of nodes that have c as dominant color; it will then try to send at most POUR amount of water to them. If the cloud does not contain the necessary amount of water, it will just divide it equally between the nodes with that color. If there are no nodes of color c in the graph, the corresponding cloud will not be able to send water to any node and the color will thus disappear from the graph.

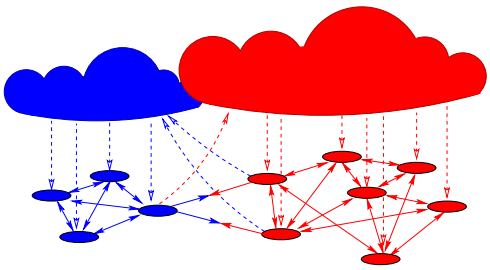


Figure 11.2: Illustration of the water cycle in TOVEL

11.4.3 Convergence criterion

Each node will vote to halt the algorithm if its dominant color has not changed for a sufficient number of iterations. The algorithm is stopped when all nodes vote to terminate. In our experiments, we call this parameter IDLE and set its value to 5. A larger value does not increase significantly the quality of our clustering and come at the cost of a much larger iteration count.

While this simple approach has been sufficient to converge with real-world graphs, there are corner cases in which convergence is never reached because of a flickering effect in which a few nodes continuously switch between colors.

11.4.4 Rationale

TOVEL is an heuristic approach to an NP-complete problem. In this section we provide an overview of the rationale behind the different phases for our approach, while Section 11.5 presents a more analytical study of the quality of the algorithm and its relation to the conductance of the graph.

Figure 11.2 illustrates the cycle of red water in the algorithm. The red cluster will diffuse the red water along both red and blue edges. All water that is sent towards blue nodes will go out of the red cluster and, if it does not convince those blue nodes to change their color, will be evaporated and sent back to the red cloud. The cloud will then rain some of the red water back to the cluster.

We call the *leakage* of a cluster the amount of water lost by that cluster via the process of diffusion and evaporation, and the *precipitation* of that cluster the amount of water that it has received from the cloud.

It is easy to see that if the leakage is larger than the precipitation, the total amount of water in the cluster will decrease. This process will also decrease the average amount of water in the cluster, thus decreasing the leakage in the following iterations. If we assume that there is an infinite amount of water in the cloud, the precipitation will stay constant if the cluster does not change and the leakage will eventually be equal to the precipitation.

V_c	set of nodes with dominant color c
deg_n	degree of node n
cut_n	edges between node n and nodes of a different color
W_c	total amount of water c in the graph
$\overline{W_c} = W_c/\|V_c\|$	average amount of water c in the nodes of its cluster

Table 11.2: Notations used in the analysis of TOVEL (Section 11.5)

If the leakage is smaller than the precipitation, the inverse process will appear and the leakage will grow until it reaches the precipitation.

The core property of the algorithm is the following: the smaller is the fraction of outgoing edges of a cluster (close to its conductance), more water will each node of the cluster have at the converged state. This property is crucial in making sure that well-connected clusters survive the competition, while badly connected clusters will be weaker, get invaded more easily and eventually disappear.

The total quantity of water in TOVEL is fixed, thus it is possible that a cloud will not be able to send the full POUR to the nodes of the clusters. This event becomes more likely once a cluster gets bigger, since it will spread its fixed amount of water on a larger set of nodes. By choosing the correct POUR, we can thus control the desired sizes of the clusters. In our experiments, we used an heuristic based on the ratio of distinct context words over total context words, as illustrated in Section 11.5

11.5 Analysis

In this section we analytically study the behavior of TOVEL and show that it will tend to favor well-structured subgraphs of the desired size. The notation used is defined in Table 11.2.

11.5.1 Quality at convergence

To help our analysis, we will analyze the behavior of TOVEL once it has reached the steady state and the clusters are fixed. Each cluster will diffuse some of its water to neighbors of a different color, who will then send it back to the cloud. The total amount of water c that evaporate to the cloud in each round can be computed as in Equation 11.1. Each node of the cluster has a certain amount of water of that color and a fraction of it will be sent to neighbors of a different color. Assuming that the distribution of water inside each cluster is uniform we can continue the analysis and obtain Equation 11.2. During each iteration the cluster also receives some water from the clouds. If there is sufficient water

there, each node will receive exactly POUR of water of its dominant color. We reach the steady state when the precipitation and the leakage are the same. How much water will there be in each node of the cluster at that point? By solving Equation 11.4 we obtain a value for the average amount of water as in Equation 11.5

$$Leakage_c = \sum_{n \in V_c} W_n(c) \frac{cut_n}{deg_n} \tag{11.1}$$

$$Leakage_c = \sum_{n \in V_c} \overline{COL_c} \frac{cut_n}{deg_n} = \overline{COL_c} \sum_{n \in V_c} \frac{cut_n}{deg_n} \tag{11.2}$$

$$Precipitation_c = |V_c| * \text{POUR} \tag{11.3}$$

$$|V_c| * \text{POUR} = \overline{COL_c} \sum_{n \in V_c} \frac{cut_n}{deg_n} \tag{11.4}$$

$$\overline{COL_c} = \frac{\text{POUR} \cdot |V_c|}{\sum_{n \in V_c} \frac{cut_n}{deg_n}} \tag{11.5}$$

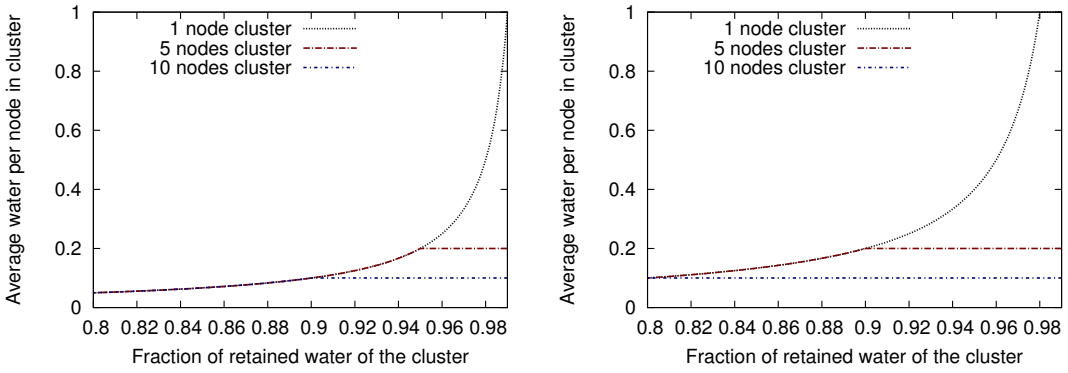
This formula shows that the average amount of water contained in a node in the steady state depends on the average fraction of water that evaporates during each iteration. This measure is very close to the conductance of the cluster, since the fewer cut edges there are, the bigger will be the average amount of color in the nodes of that cluster. This is not only true at the steady state, but also during the execution of the algorithm. Badly formed clusters will see its color evaporate much faster and will be made easier to invade by the other clusters.

11.5.2 Sizes at convergence

The feature that allows us to control the sizes of the partitions is the fixed amount of water in the system. The clusters are discouraged from becoming too large because the average amount of color cannot be more than $\frac{1}{|V_C|}$.

Figure 11.3 illustrates the effect of different values of POUR. It shows the average amount of water in each node of the cluster, a measure of the strength of the cluster in our algorithm, against the fraction of water that is kept during each iteration by the cluster. If a cluster contains 0.8 of water and loses 0.2 because of evaporation, it means that it keeps 0.75 of its water.

Figure 11.3b shows that clusters of size smaller than 5 and quality higher than 0.9 will keep more color than clusters of size 5 with the same quality. The higher the size, the more of its clusters will be penalized. Choosing a different POUR, as shown in Figure 11.3a, changes the strength of that effect. Fewer clusters of size 5 will be penalized, but the



(a) POUR = 0.01 (b) POUR = 0.02

Figure 11.3: Average color per node in a cluster at steady state, for different quality and values of POUR, in a graph with 1000 nodes

behavior of clusters of size 10 is now close to the behavior of clusters of size 5 with the previous value of the cap. By choosing and tuning the value of this parameter we can control the desired sizes of the clusters and encourage smaller but less connected clusters or larger and better connected clusters.

11.5.3 Computing pour for word sense induction

While in a general graph we do not have much information about the underlying clusters, in the word sense induction scenario we start from a better position. The graph is created as a collection of cliques, one for each document, and is thus possible to estimate how much dense is the graph and thus how conservative the algorithm should be in creating clusters. If the graph is not dense, then POUR can be set to a lower value, while if the graph is very dense, an higher POUR might allow us to find clusters with lower quality, but still meaningful, thus avoiding finding only one huge cluster.

In this subsection we present an heuristic to set POUR in a meaningful way, while still allowing users the freedom to control the degree of resolution of the cluster. In Section 11.7 we show that this approach allow us to obtain very good results with graphs of wildly different characteristics.

Be A the set of ambiguous nodes in the graph and $\overline{A} = N \setminus A$ the set of non-ambiguous nodes in the graph. The following measure is an indication of the density of the graph:

$$d = \frac{\text{total context words extracted from documents}}{\text{distinct context words extracted from documents}} = \frac{\sum_{n \in A} deg_n}{|\overline{A}|}$$

Note that each ambiguous node represents a document, and its degree is equal to the

Algorithm 13: Classifier

```
Input: Sentence S
Input: Colored graph G
M = new Map();
foreach Word w ∈ S do
    if w ∈ G then
        color = G.getNode(w).color;
        amount = G.getNode(w).water;
        M(col) = M(col) + amount;
return arg max_c(M(c));
```

number of context words associated to that document. The numerator of the fraction is thus equal to the total number of context words in the dataset. The denominator is instead equal to the number of distinct context words, since for each of them we create only one word (see Section 11.3).

$$\text{POUR} = \frac{d}{|N|}$$

This metric is strongly related to the eventual size of the clusters that will be found by our approach. If the graph is very sparse, then POUR will be equal to $\frac{1}{|N|}$, thus indicating that a full cloud is able to serve clusters size at most $|N|$, the largest cluster possible. If, instead, the graph is very dense, each ambiguous node will be connected to the same context words and POUR will be equal to $\frac{|A|}{|N|}$. A full cloud is thus able to serve a clusters of size $\frac{|N|}{|A|}$, which is the average size of a cluster if we create one different cluster for each ambiguous node. Since we do not care about clusters that do not contain ambiguous nodes for this specific application, this is the smallest cluster size we are interested in.

11.6 Extensions for word sense induction and disambiguation

Incremental addition of batches of documents Since our approach should be run on huge quantities of documents, it is infeasible to run it from scratch every time there is an update in the dataset. For this reason, both the graph construction and the graph clustering algorithms can be adapted to work in an incremental scenario.

If we assume that the new batch of sentences to be added does not introduce new meanings (does not change the number of clusters in the ground truth), then it is possible to extract the context words from the new sentences according to Section 11.3 and add any newly created node to the graph without any water, while keeping the old distribution

Dataset statistics						TOVEL results				
Name	Source	Docs	Nodes	Edges	Senses	Prec	Rec	F1	F05	Rounds
Apple	Wikipedia	369	5046	258798	2	91.61	85.52	88.46	90.33	42
Mercury	Wikipedia	2921	15111	816744	4	86.52	80.40	83.35	85.22	40
Orange	Wikipedia	1447	9546	489736	5	74.23	61.21	67.09	71.20	46
CA	Recorded Future	1182	2045	16700	8	97.92	68.83	80.84	90.29	43
Kent	Spotify	124	160	1654	6	95,76	97,64	96,69	96,13	10

Table 11.3: Datasets used in evaluation

of colors in the rest of the graph. TOVEL will converge extremely fast since it will start from a state already close to the desired result.

Colored graph as a classifier The incremental algorithm can be used when we need to continuously update our inner model, but in some cases we might want to run our approach only once on a large dataset and then use that model to answer queries on single input sentences independently. By following this approach, we get huge gains in efficiency and scalability, since the colored graph can be accessed independently for each query in "read-only mode", but we lose the capability of use the input sentences as part of our dataset.

Given the colored graph, we assign each color to a meaning of the target word by manually disambiguating a few sample sentences in the dataset. Once we have a mapping between the colors and the meanings, we store for each non-ambiguous word both its color and the amount of water of that color. For each input sentence we extract its context words and, if they exists in our colored graph, collect all water that they contain. The input sentence will be classified according to the most popular color, computed following Algorithm 13.

11.7 Experimental results

In this study we used datasets collected from different sources. *Apple*, *Mercury* and *Orange* are taken from Wikipedia. For each meaning of that word, we extracted sentences with outlinks toward that page. The context words are automatically extracted by selecting all adjectives and nouns using the Stanford NLP parser tool. *CA* contains a set of documents pertaining different "Chris Andersen" as collected by Recorded Future, a web intelligence company. The words are extracted from each document using their proprietary techniques. *Kent* was constructed by Spotify from a collection of albums with the field "Artist Name" equal to "Kent". The context words extracted from the albums are other fields such as the recording company, the country and the language. This graph is much smaller but

shows the feasibility of our approach in scenarios different from text disambiguation, such as artist disambiguation. Both *CA* and *Kent* have been confidentially given to us by the respective companies. In the case of *Kent*, our algorithm is already used as a pre-processing phase in Spotify's system.

To evaluate the quality of our clustering we use the B-cubed approach to compute the precision, recall and F1-score of the ambiguous nodes, as presented in [10]. High precision means clusters that are clean and contain nodes that have the same meaning, while high recall means having clusters that contain most nodes of that meaning. The F05-score gives twice the importance to precision than recall. In Table 11.3 we show the performance of our approach.

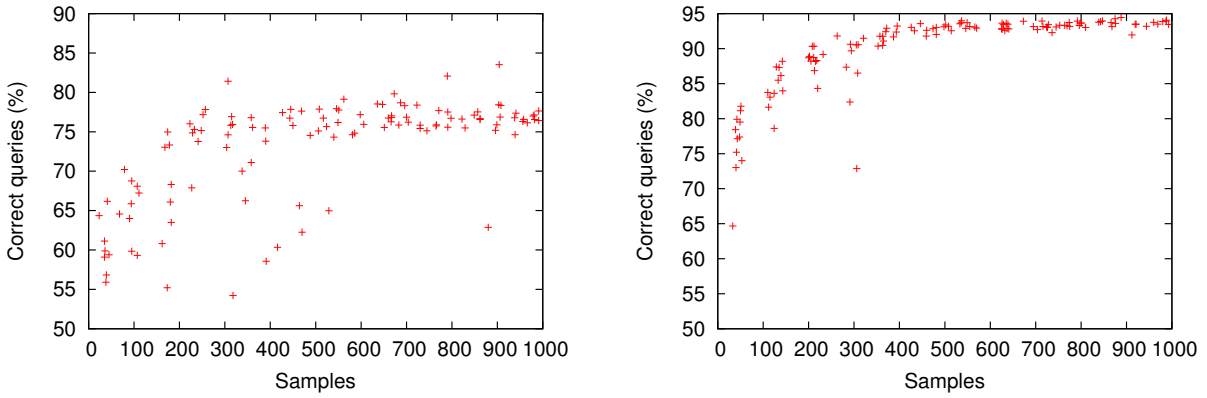
Service	**Prec**	**Rec**	**F1**	**F05**	
textrazor	100.00	79.17	88.38	95.00	Apple
dbpedia	98.94	76.07	86.01	93.33	
wikimeta	96.29	69.41	80.67	89.37	
combined	98.45	54.15	69.87	84.61	
TOVEL	91.61	85.52	88.46	90.33	
textrazor	74.23	27.10	39.71	55.08	Mercury
dbpedia	75.26	28.35	41.19	56.55	
wikimeta	73.56	27.23	39.74	54.88	
combined	97.29	53.51	69.04	83.61	
TOVEL	86.52	80.40	83.35	85.22	

Table 11.4: Comparison of our approach against online disambiguation services

Clustering comparison with disambiguation services To give a comparison of the quality of the clustering of our approach, we used the NERD API to run different disambiguation services on our own datasets. Table 11.4 shows that our approach reaches results comparable with the leading disambiguation services, without using any external source.

Disambiguation with colored graph To simulate our approach in a realistic scenario, we follow the model from Section 11.6. From a random subset of our dataset we build the graph, run the clustering algorithm and assign each cluster to a meaning. We then use this classifier to process the remaining sentences. As Figure 11.4 illustrates, our approach can disambiguate with high precision once the size of the learning dataset is large enough.

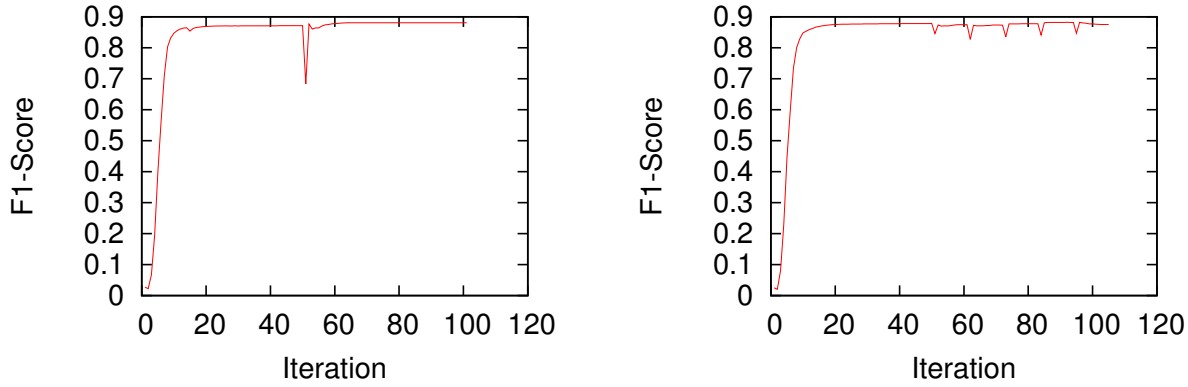
Incremental results Figure 11.5 shows the behavior of our algorithm in presence of incremental updates in the graph. We test two different scenarios: in the first scenario our approach is run to convergence on 80% of the document and the remaining 20% are added in a single batch after 50 iterations. In the second scenario our approach is run on 50% of the graph and 5 batches of 10% each are added at specific interval. In both cases



(a) Mercury dataset

(b) Apple dataset

Figure 11.4: Percentage of queried sentence correctly classified, against the number of sampled sentences used in the model construction.



(a) Starts with 80% of documents, adds 20% after 50 iterations

(b) Starts with 50% of documents, after 50 iterations adds 10% each, 5 times

Figure 11.5: F-score against iteration

the algorithms takes only a few iterations to recover and reach convergence.

Scalability TOVEL is extremely scalable, since each vertex and each cloud can work in parallel in each of the different stages of TOVEL. Figure 11.6 shows the running time of our prototype implementation in SPARK on the EC2 cloud, using different number of nodes.

11.8 Future work

We intend to study TOVEL in a more general graph clustering scenario. In such a setting we do not have information about ambiguous nodes, therefore we need different approaches to initialize the distribution of water in the graph. Starting with a random sample of

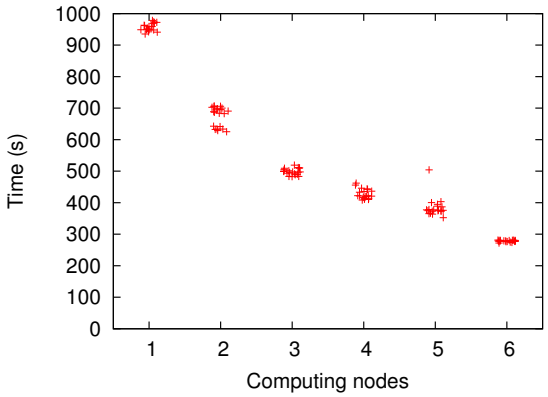


Figure 11.6: Running time against number of machines of SPARK implementation

nodes might be enough, provided that the number of nodes is reasonably larger than the expected number of communities to be found in the graph.

Our SPARK implementation of TOVEL is still a proof of concept that needs to be polished to scale efficiently to truly large datasets. Since there are few distributed algorithm to compare with, there is also the option of creating a faster, parallel program to compare its efficiency with other parallel clustering algorithm.

Part V

Concluding Remarks

Chapter 12

Conclusions

In this thesis we introduced the many steps that form the graph processing pipeline: the original data is processed to allow the construction of a graph, the graph is loaded into appropriate systems and specific algorithms are then run to solve problems of interest to the analyst.

Stand-alone tools that solve specific graph problems, while useful for research, are not that useful in practical applications. Since companies will need more than a single tool to solve all the different problems on their data, tools that live in established ecosystems will be more visible and gain more acceptance. There are some exceptions: if what the system does is so novel that cannot be reproduced inside established frameworks, then it may gain enough visibility to allow it to survive. If the tool is just more efficient than similar tools inside larger ecosystem, the gain in efficiency will have to be very large to justify the costs in adoption.

This is a pattern that appear in all stages of the pipeline: convenience and ease of use trumps efficiency. For example, while there are cases in which spending more time on partitioning the graph can be useful, there are many more cases where having an option for a quick partitioning is crucial. Users will often want to test their algorithm as soon as possible and only then, if there is a need, try a more complex partitioning algorithm.

Flexibility is extremely important: systems should allow users to choose their preferred programming model. Edge-centric programming models are more efficient on scale-free graphs, but vertex-centric algorithms can be easier to develop and have seen more applications. Similar points can be done on synchronous and asynchronous approaches: a system that offers a simple, synchronous programming model and allows for asynchronous optimizations are at advantage with respect to purely asynchronous systems. Users prefer to start from the simplest interface and are more likely to stay with a system if they are able to eventually optimize their approach without changing the underlying infrastructure.

These observations lead us to better define what will be the future steps of this study.

There are three novel approaches that are presented in this thesis: a novel edge partitioning algorithm (DFEP), a partition-centric graph processing system (ETSCH) and a distributed clustering algorithm with applications on word sense disambiguation and induction (TOVEL).

At the moment, all three approaches have been implemented in stand-alone tools and therefore are quite limited in scope and possible applications in the industry. While these tools allowed us to show the promise of the underlying ideas, to allow their usage by the general public would need a huge amount of work. Instead, we plan to implement these approaches inside mature frameworks with a already defined user base, to allow these ideas to spread more quickly and have more impact in the industry.

On the side of basic research, while Chapter 10 showed a overview of different graph-related problems, a more complete study is needed to really understand the impact of different programming models. Grouping these problems according to how easy are they to be solved using the programming models would allow a construction of a taxonomy of graph problems that suggests the best programming model for when a user starts working on a specific problem.

The research on TOVEL has moved toward a promising direction. A collaboration with a local company has started on the analysis of a corporate network that connects Italian companies and workers using shares and positions. Computing clusters on this graph would allow understanding the correct categories for the companies, and running the clustering algorithm for different levels of definition could construct a hierarchical taxonomy of categories to improve upon the government defined categories.

Publications

- **Distributed estimation of global parameters in delay-tolerant networks**, Alessio Guerrieri, Iacopo Carreras, Francesco De Pellegrini, Alberto Montresor and Daniele Miorandi, Proceedings of the 3rd IEEE WoWMoM Workshop on Autonomic and Opportunistic Communications (AOC'09)
- **Distributed estimation of global parameters in delay-tolerant networks**, Alessio Guerrieri, Iacopo Carreras, Francesco De Pellegrini, Daniele Miorandi, and Alberto Montresor, Computer Communications 33 (2010), no. 13, 1472-1482
- **DS-Means: Distributed Data Stream Clustering**, Alessio Guerrieri and Alberto Montresor, Euro-Par, Lecture Notes in Computer Science. Springer, 2012
- **Top-k Item Identification on Dynamic and Distributed Datasets**, Alessio Guerrieri, Alberto Montresor and Yannis Velegrakis, In Euro-Par, Lecture Notes in Computer Science. Springer, 2014
- **DFEP: Distributed Funding-based Edge Partitioning**, Alessio Guerrieri and Alberto Montresor, In Euro-Par, Lecture Notes in Computer Science. Springer, 2015
- **Tovel: Distributed Graph Clustering for Word Sense Disambiguation**, Alessio Guerrieri, Fatemeh Rahimian, Sarunas Girdzijauskas, and Alberto Montresor, Submitted for publication
- **ETSCH: Partition-centric Graph Processing**, Alessio Guerrieri, Alberto Montresor and Simone Centellegher, Submitted for publication

Bibliography

[1] Khaled Abdalgader and Andrew Skabar. Unsupervised similarity-based word sense disambiguation using context vectors and sentential word importance. *ACM Transactions on Speech and Language Processing (TSLP)*, 9(1):2, 2012.

[2] Eneko Agirre, David Martínez, Oier López de Lacalle, and Aitor Soroa. Two graph-based algorithms for state-of-the-art WSD. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 585–593, 2006.

[3] Akka. `http://www.akka.io`.

[4] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74(1):47, 2002.

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal - International Journal on Very Large Data Bases*, 23(6):939–964, 2014.

[6] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the* 16^{th} *annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 120–124, 2004.

[7] Sabeur Aridhi, Philippe Lacomme, Libo Ren, and Benjamin Vincent. A mapreduce-based approach for shortest path problem in large-scale networks. *Engineering Applications of Artificial Intelligence*, 41:151 – 165, 2015.

[8] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit Santa Clara*, 2011.

[9] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. Approximating betweenness centrality. In *Algorithms and Models for the Web-Graph*, pages 124–137. Springer, 2007.

[10] Amit Bagga and Breck Baldwin. Algorithms for scoring coreference chains. In *Proceedings of the Linguistic Coreference Workshop at The 1st International Conference on Language Resources and Evaluation (LREC'98)*, pages 563–566, 1998.

[11] Satanjeev Banerjee and Ted Pedersen. An adapted Lesk algorithm for word sense disambiguation using WordNet. In *Computational Linguistics and Intelligent Text Processing*, pages 136–145. Springer, 2002.

[12] Satanjeev Banerjee and Ted Pedersen. Extended gloss overlaps as a measure of semantic relatedness. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 805–810. Morgan Kaufmann Publishers Inc., 2003.

[13] Judit Bar-Ilan. Data collection methods on the web for infometric purposes - a review and analysis. *Scientometrics*, 50(1):7–32, 2001.

[14] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[15] Casey Battaglino, Pienta Pienta, and Richard Vuduc. Grasp: distributed streaming graph partitioning. In *Proceedings of the 1st High Performance Graph Mining workshop*, 2015.

[16] Andrzej Bialecki, Mike Cafarella, Doug Cutting, and Owen OMalley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *http://lucene.apache.org/hadoop*, 2005.

[17] Chris Biemann. Chinese whispers: an efficient graph clustering algorithm and its application to natural language processing problems. In *Proceedings of the 1st Workshop on Graph-based Methods for Natural Language Processing*, pages 73–80, 2006.

[18] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10), 2008.

[19] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International World Wide Web (WWW'11)*. ACM Press, 2011.

[20] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW'04)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[21] Erik G Boman, Doruk Bozdağ, Umit Catalyurek, Assefaw H Gebremedhin, and Fredrik Manne. A scalable parallel graph coloring algorithm for distributed memory computers. In *Euro-Par 2005 Parallel Processing*, pages 241–251. Springer, 2005.

[22] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[23] Sergey Brin and Lawrence Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 56(18):3825–3833, 2012.

[24] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. *Computer Networks*, 33(1):309–320, 2000.

[25] Samuel Brody and Mirella Lapata. Bayesian word sense induction. In *Proceedings of the 12th Conference of the European Chapter of the ACLs*, pages 103–111, 2009.

[26] Ping Chen, Wei Ding, Chris Bowes, and David Brown. A fully unsupervised word sense disambiguation method using dependency knowledge. In *Proceedings of the 2009 Annual Conference of the North American Chapter of the ACLs*, pages 28–36, 2009.

[27] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 85–98. ACM, 2012.

[28] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[29] J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science & Engineering*, 11(4):29–41, 2009.

[30] A. Das Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*, pages 401–410. ACM, 2010.

[31] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[32] Paul Erdös and Alfréd Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[33] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741.

[34] Arash Fard, M Usman Nisar, Lakshmish Ramaswamy, John Miller, Matthew Saltz, et al. A distributed vertex-centric approach for pattern matching in massive graphs. In *Proceedings of 2013 IEEE International Conference on Big Data*, pages 403–411. IEEE, 2013.

[35] Paolo Ferragina and Ugo Scaiella. Tagme: on-the-fly annotation of short text fragments (by wikipedia entities). In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pages 1625–1628. ACM, 2010.

[36] Charles M Fiduccia and Robert M Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Conference on Design Automation*, pages 175–181. IEEE, 1982.

[37] Philippe Flajolet and G Nigel Martin. Probabilistic counting. In *Proceedings of 24th Annual Symposium on Foundations of Computer Science*, pages 76–82. IEEE, 1983.

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

[39] Santo Fortunato and Marc Barthélemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

[40] Nishant M Gandhi and Rajiv Misra. Performance comparison of parallel graph coloring algorithms on bsp model using hadoop. In *Proceedings of Computing, 2015 International Conference on Networking and Communications*, pages 110–116. IEEE, 2015.

[41] Joachim Gehweiler and Henning Meyerhenke. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *Proceedings of the Workshops and Phd Forum on Parallel & Distributed Processing (IPDPSW)*, pages 1–8. IEEE, 2010.

[42] John R Gilbert and Earl Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, 16(6):427–449, 1987.

[43] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. Practical recommendations on crawling online social networks. *IEEE Journal on Selected Areas in Communications*, 29(9):1872–1892, 2011.

[44] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[45] Alessio Guerrieri and Alberto Montresor. DFEP: Distributed funding-based edge partitioning. In *Proceedings of the 21st European Conference on Parallel Processing (Euro-Par'15)*, Lecture Notes in Computer Science. Springer, 2015.

[46] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[47] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, page 1. ACM, 2014.

[48] Michael Hardey. The city in the age of Web 2.0 a new synergistic relationship between place and people. *Information, Communication & Society*, 10(6):867–884, 2007.

[49] Amr Hassan and Christopher J Fluke. Scientific visualization in astronomy: towards the petascale astronomy era. *Publications of the Astronomical Society of Australia*, 28(2):150–170, 2011.

[50] Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, pages 355–364. ACM, 2012.

[51] Doug Howe, Maria Costanzo, Petra Fey, Takashi Gojobori, Linda Hannick, Winston Hide, David P Hill, Renate Kania, Mary Schaeffer, Susan St Pierre, et al. Big data: The future of biocuration. *Nature*, 455(7209):47–50, 2008.

[52] Salim Jouili and Valentin Vansteenberghe. An empirical comparison of graph databases. In *Proceedings of the 2013 International Conference on Social Computing (SocialCom)*, pages 708–715. IEEE, 2013.

[53] Sepandar Kamvar, Taher Haveliwala, Christopher Manning, and Gene Golub. Exploiting the block structure of the web for computing pagerank. *Stanford University Technical Report*, 2003.

[54] U Kang, Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. Hadi: Mining radii of large graphs. *ACM Transaction on Knowledge Discovery from Data*, 5(2):8:1–8:24, February 2011.

[55] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 9th IEEE International Conference on Data Mining*, pages 229–238. IEEE, 2009.

[56] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[57] George Karypis and Vipin Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical report, University of Minnesota, 1995.

[58] George Karypis and Vipin Kumar. hmetis 1.5: A hypergraph partitioning package. Technical report, Technical Report, University of Minnesota, 1998.

[59] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[60] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis. Technical report, Technical Report, University of Minnesota, 2003.

[61] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 1970.

[62] Adam Kilgarriff and Joseph Rosenzweig. Framework and results for english SENSEVAL. *Computers and the Humanities*, 34(1-2):15–48, 2000.

[63] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, volume 12, pages 31–46, 2012.

[64] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6:70, 2001.

[65] Michael Lesk. Automatic sense disambiguation using machine readable dictionaries: how to tell a pine cone from an ice cream cone. In *Proceedings of the 5th International Conference on Systems Documentation*, pages 24–26. ACM, 1986.

[66] Jure Leskovec. Stanford large network dataset collection. *http://snap.stanford.edu/data/index.html*, 2011.

[67] Foad Lotfifar and Matthew Johnson. A multi–level hypergraph partitioning algorithm using rough set clustering. In *Euro-Par 2015: Parallel Processing*, pages 159–170. Springer, 2015.

[68] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[69] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[70] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010.

[71] Gurmeet Singh Manku, Moni Naor, and Udi Wieder. Know thy neighbor's neighbor: the power of lookahead in randomized P2P networks. In *Proceedings of the 36th ACM Symposium on Theory of Computing*, pages 54–63. ACM, 2004.

[72] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *Journal of Parallel and Distributed Computing*, 69(9):750–761, 2009.

[73] Stanley Milgram. The small world problem. *Psychology Today*, 2(1):60–67, 1967.

[74] George A Miller. WordNet: a lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[75] Jayanta Mondal and Amol Deshpande. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, pages 1335–1346. ACM, 2014.

[76] Lifeng Nai, Yinglong Xia, I Tanase, Hyesoon Kim, and Ching-Yung Lin. Graphbig: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, volume 15, 2015.

[77] Roberto Navigli. A quick tour of word sense disambiguation, induction and related approaches. In *SOFSEM 2012: Theory and practice of computer science*, pages 115–129. Springer, 2012.

[78] Neo4j. `http://neo4j.com/`.

[79] Mark EJ Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[80] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical Review*, 69(2):026113, 2004.

[81] Andrew Y Ng, Michael I Jordan, Yair Weiss, et al. On spectral clustering: Analysis and an algorithm. *Advances in Neural Information Processing Systems*, 2:849–856, 2002.

[82] Zheng-Yu Niu, Dong-Hong Ji, and Chew Lim Tan. Word sense disambiguation using label propagation based semi-supervised learning. In *Proceedings of the 43rd Annual Meeting on ACLs*, pages 395–402. ACL, 2005.

[83] Orientdb. `http://orientdb.com/orientdb/`.

[84] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. Comparison and classification of nosql databases for big data. In *Proceedings of International Conference on Big Data, Cloud and Applications*, 2015.

[85] Siddharth Patwardhan and Ted Pedersen. Using WordNet-based context vectors to estimate the semantic relatedness of concepts. In *Proceedings of the EACL 2006 Workshop Making Sense of Sense-Bringing Computational Linguistics and Psycholinguistics Together*, volume 1501, pages 1–8, 2006.

[86] Zdzisław Pawlak. *Rough sets: Theoretical aspects of reasoning about data*, volume 9. Springer Science & Business Media, 2012.

[87] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 243–252. ACM, 2015.

[88] Mohammad Taher Pilehvar and Roberto Navigli. A large-scale pseudoword-based evaluation framework for state-of-the-art word sense disambiguation. *Computational Linguistics*, 40(4):837–881, 2014.

[89] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *Computer and Information Sciences (ISCIS 2005)*, pages 284–293. Springer, 2005.

[90] Daniel Presser, Lau Cheuk Lung, and Miguel Correia. Greft: Arbitrary fault-tolerant distributed graph processing. In *Proceedings of the 2015 IEEE International Congress on Big Data*, pages 452–459. IEEE, 2015.

[91] Rami Puzis, Polina Zilberman, Yuval Elovici, Shlomi Dolev, and Ulrik Brandes. Heuristics for speeding up betweenness centrality computation. In *Proceedings of the 2012 International Confernece on Social Computing*, pages 302–311. IEEE, 2012.

[92] Fatemeh Rahimian, Sarunas Girdzijauskas, and Seif Haridi. Parallel community detection for cross-document coreference. In *IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technologies*, volume 2, pages 46–53. IEEE, 2014.

[93] Fatemeh Rahimian, Amir H Payberah, Sarunas Girdzijauskas, and Seif Haridi. Distributed vertex-cut partitioning. In *Distributed Applications and Interoperable Systems*, pages 186–200. Springer, 2014.

[94] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *In Proceedings of 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO'13)*, pages 51–60. IEEE, 2013.

[95] Lakshmish Ramaswamy, Bugra Gedik, and Ling Liu. A distributed approach to node clustering in decentralized peer-to-peer networks. *Parallel and Distributed Systems, IEEE Transactions on*, 16(9):814–829, 2005.

[96] Ariel Raviv, Shaul Markovitch, and Sotirios-Efstathios Maneas. Concept-based approach to word-sense disambiguation. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2012.

[97] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[98] Gert Sabidussi. The centrality index of a graph. *Psychometrika*, 31:581–603, 1966.

[99] M Sagharichian, H Naderi, and M Haghjoo. Expregel: a new computational model for large-scale graph processing. *Concurrency and Computation: Practice and Experience*, 2015.

[100] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[101] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of the VLDB Endowment*, 7(7):577–588, 2014.

[102] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Proceedings of the 2nd International Conference on Cloud Computing Technology and Science*, pages 721–726. IEEE, 2010.

[103] Hui Shen, Razvan Bunescu, and Rada Mihalcea. Coarse to fine grained sense disambiguation in wikipedia. *Proceedings of the Conference on Lexical and Computational Semantics (SEM)*, pages 22–31, 2013.

[104] Spark partitioning strategies. `https://spark.apache.org/docs/0.9.1/api/graphx/index.html#org.apache.spark.graphx.PartitionStrategy`.

[105] Sparksee: high-performance graph database. `http://www.sparsity-technologies.com`.

[106] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1222–1230. ACM, 2012.

[107] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient subgraph matching on billion node graphs. *Proceedings of the VLDB Endowment*, 5(9):788–799, 2012.

[108] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web*, pages 607–614. ACM, 2011.

[109] James Joseph Sylvester. Chemistry and algebra. *Nature*, 17:284, 1878.

[110] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[111] Titan: Distributed graph database. `http://thinkaurelius.github.io/titan`.

[112] Aleksandar Trifunovic. *Parallel algorithms for hypergraph partitioning.* PhD thesis, University of London, 2006.

[113] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342. ACM, 2014.

[114] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846. ACM, 2009.

[115] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[116] Tim Van de Cruys and Marianna Apidianaki. Latent semantic word sense induction and disambiguation. In *Proceedings of the 49th Annual Meeting of the ACLs: Human Language Technologies-Volume 1*, pages 1476–1485. ACL, 2011.

[117] Fei Wang and Changshui Zhang. Label propagation through linear neighborhoods. *IEEE Transactions on Knowledge and Data Engineering*, 20(1):55–67, 2008.

[118] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*, 2013.

[119] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement: Scalable and programmable analytics over very large graphs on a single pc. In *Proceedings of the Usenix Annual Technical Conference*, pages 387–401, 2015.

[120] Duncan J Watts and Steven H Strogatz. Collective dynamics of small-world networks. *Nature*, 393(6684):440–442, 1998.

[121] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *1st International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

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

[123] Ye Yuan, Guoren Wang, JefferyYu Xu, and Lei Chen. Efficient distributed subgraph similarity matching. *The VLDB Journal*, 24(3):369–394, 2015.

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

[125] Zhi Zhong and Hwee Tou Ng. It makes sense: A wide-coverage word sense disambiguation system for free text. In *Proceedings of the ACL 2010 System Demonstrations*, pages 78–83, 2010.

[126] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the Usenix Annual Technical Conference*, pages 375–386, 2015.

Acknowledgments

I want to start by thanking all those who read the entire thesis before skipping to this chapter, but do not worry if you got bored after the introduction! I'll do my best to keep my acknowledgments spoiler-free so that you will be able to read the rest of the thesis whenever you will feel like it and still get surprised.

I want to thank prof. Montresor, who followed me for many years and helped me write more theses than I thought was possible. If I had to write another thesis (I truly hope not), he would still be my first choice for an advisor. Thanks also to all the researchers with whom I collaborated through my PhD studies and in particular to my coauthors. Special mention to my colleagues who did not think it was possible to get a PhD in Computer Science without drinking coffee. I would like to claim that I did it to show that life without caffeine is possible, but in reality I just do not like the taste of coffee.

In these years over 700 students had to solve my carefully prepared projects and suffer through my weird metaphors and stories. Teaching them algorithmic concepts helped me understand them better and, most importantly, allowed me to build an extremely long list of anecdotes about bad coding practices. Thanks for the laughs and for the brief moments in which I felt that I was doing a good job.

As always, I thank prof. Naismith for convincing his class that launching balls in peach baskets could be fun. I am also grateful to Asimov, Le Guin, Tolkien, Kerr, Swanwick, Bujold and Pratchett for keeping my mind fresh and my imagination in good shape.

I have been lucky to find many friends even in the real world. My thanks go to all of them: the old and the new, the geeky and the deeply geeky. Thanks for the good times we had in the past and those that we will have in the future (an unnamed time traveler told me that they will happen, and I always trust unnamed time travelers).

Lastly, I would (literally) not be here without my family. It does not matter if you live in Trento, in Borgia, in Paris, in Rome, or if you left us for the dark desert: I am still thankful for everything you did for me. I promise that at one point I will actually explain to you what my work is about, but this is not the right place: after all I promised that I would keep this chapter spoiler-free...