Edge partitioning of large graphs

Yifan Li

Edge Partitioning of Large Graphs

Yifan Li

Supervisor: Prof. Cédric DU MOUZA
Prof. Camélia CONSTANTIN

Laboratoire d'informatique de Paris 6
University Pierre and Marie CURIE

This dissertation is submitted for the degree of
Doctorat en informatique

Sorbonne University December 2017

Dedicate this thesis to my loving parents.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Yifan Li
December 2017

Acknowledgements

Firstly, I would like to thank my family, they understood me and supported me in the tough but exciting path to pursue a Ph.D in computer science.
In the research work, I would like to express my most sincere gratitude to my supervisors, Dr. Camélia CONSTANTIN and Dr. Cédric DU MOUZA. I am so grateful for their instruction and guidance to my study in past years, what they have given to me are not only professional knowledge in this field, but also the enlightenment on how to be a qualified researcher. Moreover, in my hard time in work, I really appreciate their sustained care and help, especially the great tolerance!
Then, I have a lot of thanks to my team in Lip6, Amine, Anne, Benjamin, Bernd, Hubert and Stéphane. They provided so much precious help to my work and life, I never forget the amazing time we spend together.
Finally, I want to thank my friends, Bin, Quentin and Xin, who had offered great advices before and during the writing of my thesis.

Abstract

In this thesis I basically focus on the study of scalable data storage in distributed graph computation system. To handle this problem, my work is started from developing an edge partition method, also known as Vertex-Cut. We built a novel block-based technique, instead of the traditional vertex partition(Edge-Cut) in which most communication depends on those edges cut during partitioning. From this means, we can generate partitions with well-explored localities hidden in graphs to obtain an remarkable reduction on vertex replica factor(VRF), a general measurement of inter-(edge)-partition communication cost. What's more, we render detailed analysis into the characters of specific algorithm implementation, ie. Local Access Pattern(LAP) in Random Walks, one fundamental operation in many graph mining applications, which can bring valuable insights to help people to customise their own partitions, e.g. we stated a proof why the edge partitions with cluster structures facilitates the random walks with less communication requests. Finally, we extended our work from homogenous structured graphs to the heterogenous, i.e. multi-layer graph, for more complicated application scenario. In this kind of graphical model, the edges and/or vertices are allowed to have various types(labels). Thus, higher structural complexity behind those real-life graphs led to more difficulties to efficiently process graph data in a distributed context. For this reason, we developed several new tools to solve this problem, e.g. Blocks Profile Matrix, to acquire "locality" from not only graph structures, but also from label distribution over graph edges. Consequently, with our label-concerned partitions, we show that the inter-partition communication and time overhead can be decreased significantly compared to existing methods.

During evaluation to our partition approaches, I launched plenty of experiments over real world graphs(with up to 85 million edges and 2 million vertices), based on a wide range of benchmark graph mining algorithms, e.g. static and dynamic PageRanks, Shortest Path and Regular Expression. For the homogenous graph datasets, our method can achieve a communication saving between 30 and 60 percent compared to latest works in most cases, specifically an extra gain(5-10%) was won for random walks algorithm due to the block metric. For the label-concerned partition experiments to multi-layer graphs, our method also had distinguished performance in different scenarios, and can obviously enhance the

algorithms containing inter-label correlation calculation, correlated-label random walks for instance.

Table of contents

Chapter 1

Introduction

As real-world information sources become more diverse and rich, the volume of data stores created in last decades is becoming unexpectedly large. For instance, one of famous social medias, Facebook[1], has generated the massive social network with more than one billion users and hundreds of times more connections. The big Internet of Things(IoT) application has already been bond with ten billion objects in our life and World Wide Web(WWW) is still growing in an exponential speed which has consisted of trillions of documents, images and videos.

This unprecedent data explosion creates several industrial and research opportunities but also raises new challenges. As a consequence, many works have been recently presented and successfully applied to exploit the graphs en provide enhanced functionnalities, from online social network advertisement, public transportation optimisation to biological network analysis. In research community, many efforts have been emphasized on the graph mining algorithms such as clustering and searching, by measuring topological properties, analysing contents associated to vertices and/or edges, or both in the same time. However the proposals were generally not designed to face this volumetry and investigating solutions for processing these algorithms on very large real graphs is particularly challenging.

To achieve scalability when processing large real graphs, like for example the Twitter graph[2] social media graph or the human metabolic network represented in Figures 1.1 and 1.2 respectively, we must design first a scalable computation model and re-write algorithms according to this model. So there are a variety of solutions proposed in the past decades to handle scalable computations on large graphs, especially for graphs which are too large to keep in memory of a standalone machine. Along the large size of the graph, these proposals must consider the high dynamicity of the graphs. But how applications can handle the graph

[1] https://www.facebook.com/

[2] https://twitter.com/

inherent dynamicity? For instance, more than 250 million new photos were uploaded per day on Facebook[3], as shown in Figure 2.1. Moreovere, is it possible to design generic methods, with respect to graphs'properties, that can flexibly scale graph computations in a distributed context?

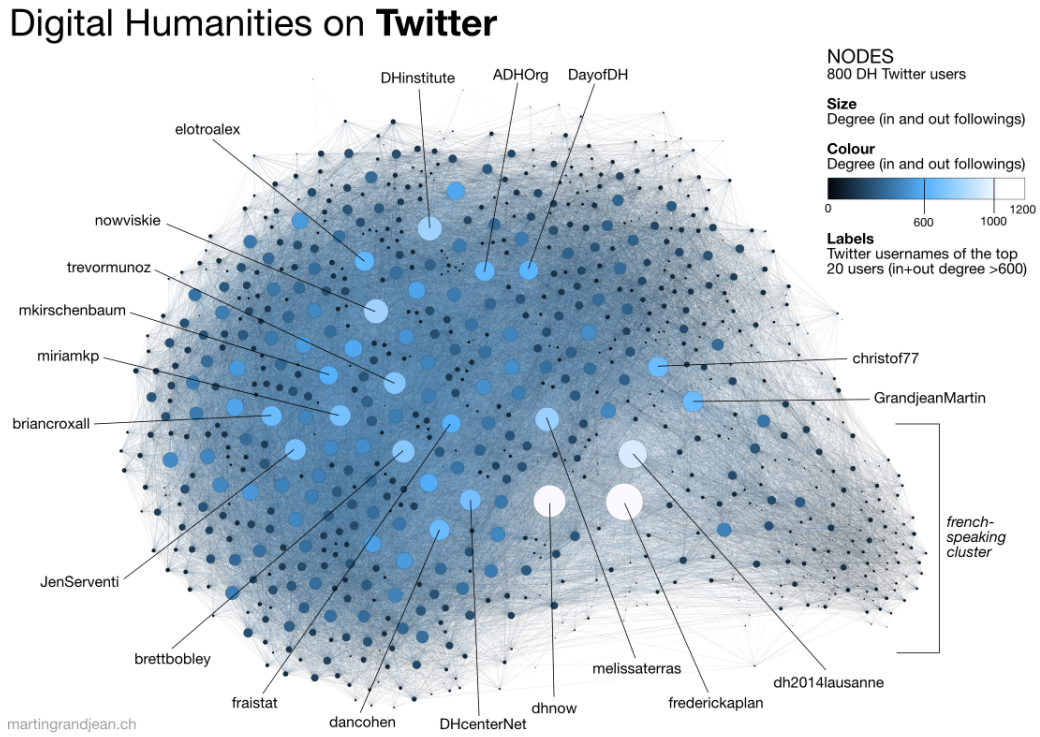


Fig. 1.1 A dense "following" network of 800 digital humanists on Twitter[Grandjean]

In order to answer these questions, we focus in this thesis on graph partitioning algorithms, which have been shown as efficient for distributed computation on real life graphs [18, 44, 118] with skewed power-law degree distribution[84, 14, 34]. Several popular graph computation systems based on this approach have emerged, such as PowerGraph (GraphLab2) [44] and GraphX [118]. Most existing works [8, 52, 61, 38, 82] use vertex partitioning approaches, also called (*edge-cut* partitioning). However this approach might not be suitable for distributed computation platforms on very large real graphs which try to keep the workload balanced between partitions and limit the inter-partition communication bottleneck. Recent results [18, 44, 118] prove that *edge partitioning* approaches (also known as *vertex-cut*) outperform *vertex partitioning* (*edge-cut*) approaches for computations on

[3]https://www.facebook.com/

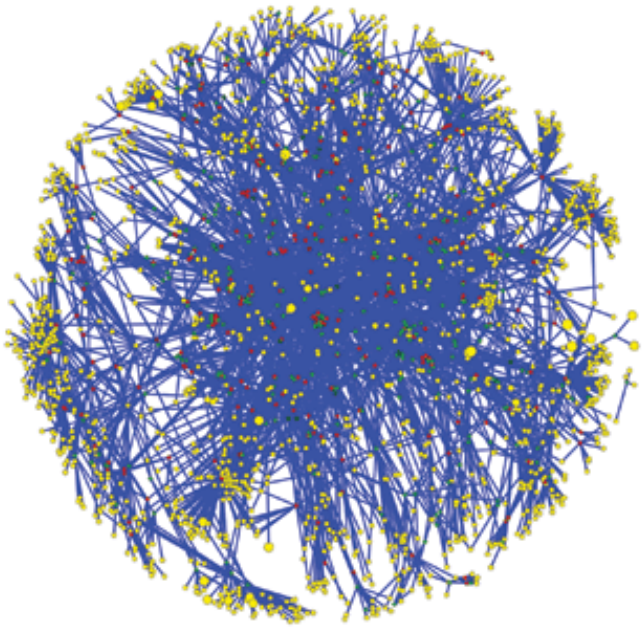


Fig. 1.2 A part of the human metabolic network, in which metabolites are shown as yellow nodes while the enzymes are displayed as red/green nodes[met]

large real-world graphs with skewed distribution, like *social networks*, since they generally avoid unbalanced computation due to the *power-law degree distribution* of graphs. However, these methods, like *evenly random assigning* [118] or *greedy assignment strategy* [44], are generic and do not consider any computation pattern for specific graph algorithm. These algorithms distribute edges evenly over partitions either randomly, *i.e.* a hash function of vertex ids in Giraph [Apache] and GraphX, or using a greedy or dynamic algorithm like in PowerGraph and GPS [96]. Besides, in contrast with *light-weight* algorithms like PageRank whose messages transmitted between vertices are only rank values, the simulation of *heavy-communication* algorithms, such as *fully (multiple) random walks* in this work, have a more important communication cost since (i) some extra path-related information of walks must also be delivered, and (ii) more than one message (walk) start from each vertex at one time. In this case, reducing communication cost is crucial for computation performance guarantee. Particularly in a large scale computing cluster with complex machine-to-machine network infrastructure, or in distributed *Computing-in-Memory* system like Spark where communication might be a bottleneck for reaching high overall performance.

Our work focuses on edge (instead of vertex) graph partitioning, hereafter called *vertex-cut* partitioning, where part of graph vertices are replicated between different partitions. The goal of our algorithms is to optimize the cost of inter-partition communication and to keep the workload balanced for Random Walks-type computations. Our approach also takes advantage

from the existence of communities. In [43] authors state that, due to the heavy-tailed degree distributions and large clustering coefficients properties in social networks, considering only the direct neighbors of a vertex allows to construct good clusters (communities) with low conductance. In [116] authors improve this method to detect communities over graph, but neither edge partitioning nor workload balancing problem is studied. Moreover, the overlapping communities approach for graph partitioning are not suitable to Pregel-like systems.

We also design partitioning algorithms for multi-layer graphs where edges are labeled by topics of interest.

We particularly study the underlying structures and properties of real social graphs, like the Twitter social network[26], the Pokec[4] and the LiveJournal[5] graphs to design *block-based edge partitioning* methods designed to efficiently process graph computations such as the PageRank and Shortest Path algorithms or regular expression queries. These computations can then be easily deployed in parallel processing frameworks like PowerGraph [45] and Spark [126]. Our methods were extended to *multi-layer* graphs, e.g. rumor spreading network(Higgs [28]) and gene association network(HomoGenetic [29]), that have more heterogenous properties.

1.1 Big Graph Processing and Mining

To face the "Volumetry" issue when processing computation in very large big graph, we can classify existing solutions from related work in three families:

- **Algorithm.** Several works[50, 88, 89] propose the design or an improvement of algorithms to perform classical graph mining tasks over massive graphs. For instance, in [130] they convert the pattern matching query into a distance-based multi-way join problem using a vector space via graph embedding techniques. To achieve a better scaling of correlation clustering similar items in big graphs, [87] uses concurrency control method to enforce serialisability of its parallel process. Shortest path(s) computation is another fundamental operation in graph theory. To achieve its scalability on large graphs, [49] proposes the construction of a scalable sketch-based index structure.

- **Data.** The "Divide and Conquer" paradigm which has been successfully applied in many big data problems, can be applied to partition directly the existing graph, such as in papers [61, 76, 70], for further parallel processing in a distributed platform. What's

[4] https://snap.stanford.edu/data/soc-pokec.html
[5] https://snap.stanford.edu/data/com-LiveJournal.html

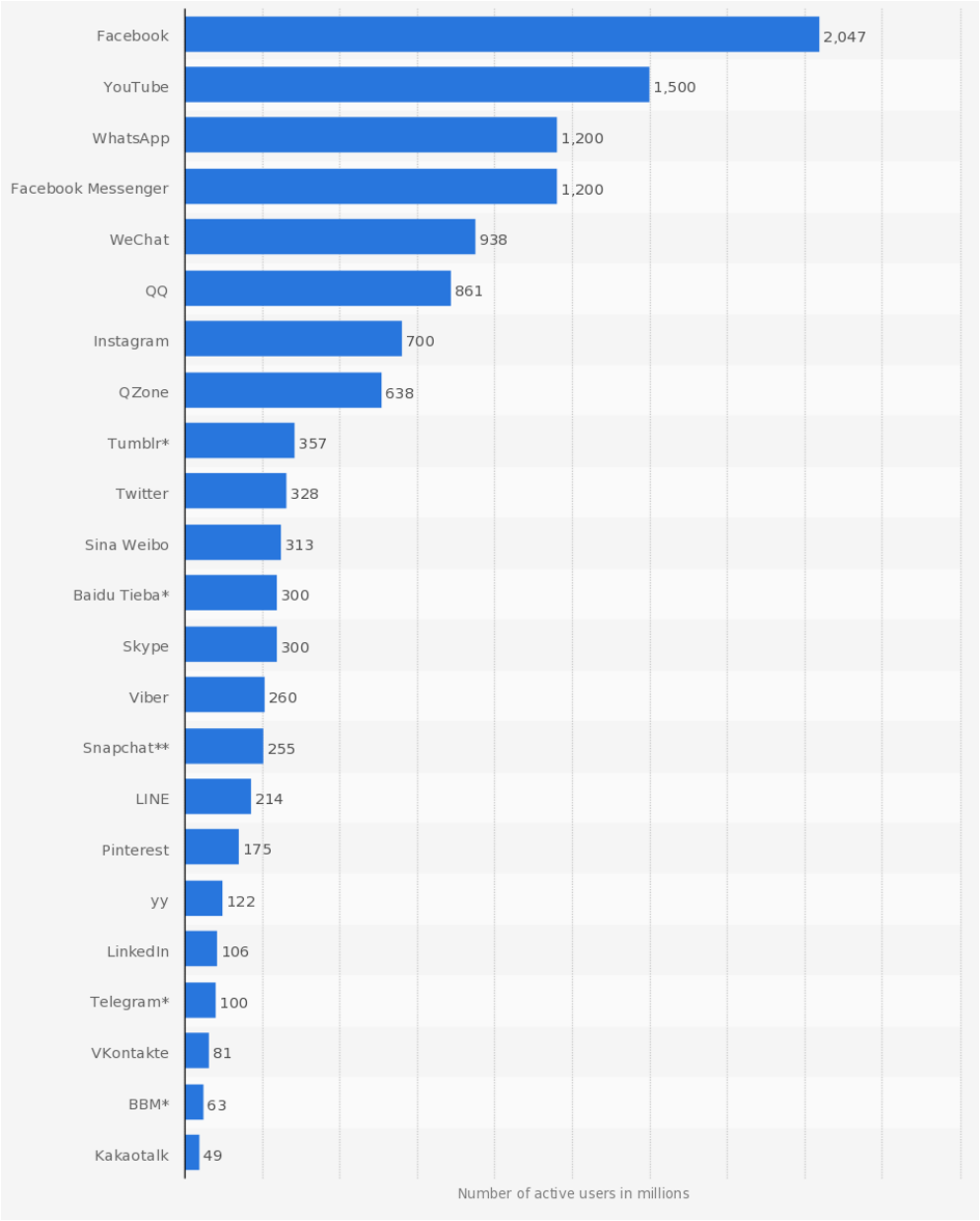


Fig. 1.3 Most famous social networks in the world, ranked by number of active users(in million) by August 2017

more, some other works propose to "reduce" the data size using various strategies for cutting down the computation overhead, for instance, [131, 3, 108] apply some metrics to decrease the search space with sampling techniques and [75, 57] propose to compress the graphs by compacting the similar vertices or clusters in original graph.

- **System.** In addition, several works go beyond traditional data-parallel solutions, like MapReduce(Hadoop) programming model, by proposing specific computation systems which enable more efficient processing for large graphs. To the best of my knowledge, the most initial work is Pregel[80] by Google, which is inspired by Bulk Synchronous Parallel(BSP) model [112]. After that, the Giraph [Apache] system was built for open-source implementation. In [77, 78], a system called GraphLab is established to provide asynchronous execution for iterative algorithms in machine learning. Unlike these Vertex-Centric models, Blogel [120] employs a Block, *i.e.* subgraph, -Centric method to deploy the graph algorithms and improve the system performance. In recent years, people start to use a new abstraction, Edge-Centric, also call GAS(Gather, Apply and Scatter) [44, 118], to perform distributed computation on large graphs, esp., trying to tackle the workload balance issue due to skew power-law degree distributions that many natural graphs have.

1.2 Thesis Overview

This thesis will describe my contribution on the large real graph partitioning issue. In Chapter 2, existing works on graph partitioning will be summarized to provide an overview of the current state of the art. This Chapter also introduces some basic but necessary preliminaries on this topic. Additionally I will discuss how the partitions can benefit local access pattern(LAP) algorithms, for example, the random walks with our block-based methods.
Thus, we first study the classical local access pattern algorithm, Random Walk, a fundamental operation in many graph mining algorithms, and then observe the impact of graph partitions on its performances. Basically we provide a proof that the partitions having good locality properties, *e.g.* with our block-based method, will permit to significantly reduce the communication cost and processing time.
Then I will particularly focused on how to partition edges(vertex-cut) graphs in unlabeled graph structure(Chapter 3) and heterogeneous graph structure, in other word multi-layer graph (Chapter 4), with consideration in both balancing the workload and reducing communication cost. Moreover, I present in Chapter 3 a tool with an interface which allows to have an intuitive understanding of the partition result and how it enhance the execution of graph algorithm. The main contributions of my thesis are consequently:

Block-Based Edge Partitioning of large Graphs The existing edge partitioning methods have good performance in workload balancing for many real life graphs and applications. We notice however that their communication costs are still important, which impacts the algorithms performances. Actually, since the number of messages transmitted during graph iterative computation is linear to the partitions' global vertex replica factor if we strip out the effect of user-defined algorithm operations[103], we propose to design a novel block-based edge partitioning method to decrease the global VRF. It onsists in building blocks to capture the local neighbourhood of important vertices, called seeds. These blocks are used as basic units for constructing final partitions. From our experiments, the communication cost, compared to existing methods, has decreased by around 30 to 60 percent for various benchmark graph algorithms deployed on real world graphs. Consequently, the time consumed for each computation is also reduced significantly. So our proposal would be of considerable benefit for applications which require repeatedly graph computations.
With our interface we develop for visualizing the result of different graph partitioning strategy, we can also check thanks to a graphical interface the characteristics of random walks performed on different graph partitions, like statistics about messages sent during processing.

Scalable Approach to Heterogeneous Graphs In most existing works on graph partitioning, independently of their approach,edge-cut or vertex-cut, the graph is considered as unlabeled, so a unified structure, in which the vertices and edges have a common type. However several real life graphs, present more complicated and heterogeneous structures. For example, we have not only "friends" but also "following/followed" and "like/unlike" relationships(linkages) in social networks, and a biological network could consist of elements of different types and have several kinds of connections between them, such as physical association and chemical interaction. For this reason, we investigate the partitioning of multi-layer graphs, which assume the existence of different edge types. There are several papers which study how to launch traditional graph mining jobs, like clustering/classification, over this type of graphs, but to the best of our knowledge, this is he first proposition of an edge partitioning approach for achieving scalability when processing algorithms on these graphs. The main idea is that 1) we consider the multi-layer graph as a graph with multiple labels on edge, 2) we compute labels "similarity" w.r.t their distribution and edges "closeness" via several metrics, 3) finally we partition the edges grouping those which have fair similarity in labels and closeness in structure. The experiment results have shown that it can significantly reduce communication cost for various graph mining algorithms, like regular expression and shortest paths, executed on single label, and also for the correlated-label random walks with considering different labels together.

Chapter 2

Preliminary and Existing Work

This chapter introduces some preliminary concepts and existing distributed graph processing systems and models as well as standalone partitioning algorithms.

The graph partitioning approaches can be mainly divided into categories, namely edge-oriented algorithms (*e.g* Edge-Cut) or vertex-oriented algorithms (*e.g* Vertex-Cut). These two approaches are described in the following.

2.1 The large graphs partitioning issue

In discrete mathematics, a graph is a set of vertices (also called points or nodes) and edges (arcs or lines), in which the vertices are used to represent objects and the edges for "related" sense between them. Basically we can define a graph as $G = (V, E)$ where V denotes the vertices set V and E the edges set. Some graphs may also present more information, *e.g.* weights or attributes of vertices or of edges.

Graph partitioning methods have been applied extensively in many studies, such as community detection, VLSI design, transportation optimization, and image segmentation. Recently, with the rapid development of computer science and technology, we have entered into the time of information and knowledge explosion. From online music rating records to mobile device location data, many of them can be organized in graph structured models to enable efficient management and exploiting explicit knowledge behind them. For instance, nowadays we have built social networks on Internet with more than one billion users (see Figure 2.1 [1]), IoT (Internet of Things) application for ten billion objects and WWW with trillions of documents and pictures. Undoubtedly, these datasets are highly valuable with

[1] https://www.sec.gov/Archives/edgar/data/1326801/000119312512034517/d287954ds1.htm

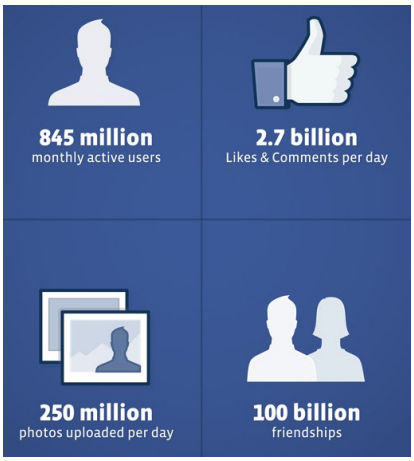


Fig. 2.1 By January 2012, Facebook has 845 million monthly active users and more than 100 billion friend connections. In particular, its size will increase exponentially if other objects, e.g. comments, photos and videos, are also included.

huge potential in improving human's life, but how to cope with and even utilize so large and complicated data, *i.e.* graphs, is still challenging to academics and industry.

2.2 "Thinking Like a Vertex"

Traditional graph partitioning methods from 2-way cut by local search to multi-level approaches, like Kernighan-Lin [62], PageRank Vectors [7] and METIS-based [60] algorithms, follow a vertex-partitioning (edge-cut) strategy. They propose partitionings which assign (almost) evenly vertices between partitions while minimizing the number of edges cut (edges between two partitions). These algorithms are efficient for small graphs, which are not sensitive to workload-unbalanced computation. However for real world graphs the large size and the power-law distribution lead to an unbalanced load over edge-cut partitions.

One of the most popular computational paradigms integrated into distributed graph frameworks is the vertex-centric programming model that was firstly introduced by *Pregel*[80]. *Pregel* [80] has become a popular distributed graph processing framework due to the facilities it offers to the developers for large-graph computations, especially compared with other data-parallel computation systems, *e.g.* Hadoop. The input graph is divided into partitions, the user programs are implemented from the perspective of a vertex rather than a graph, the user-defined programs are executed iteratively over vertices. More recent partitioning proposals in Pregel-like systems, such as Giraph, GPS, Gelly and Chaos[92] shard the graph using an *edge-cut* strategy which also generates unbalancing for power-law graphs, as introduced in [44]. Other large-scale graph processing frameworks such as GraphLab[77], PowerGraph[45],

GraphX[118] and Gigraph++[109] have also adopted this paradigm. Whereas these systems use Edge-Cut graph partition techniques, it has been shown [77] that Vertex-Cut partitioning is more effective for scalable and balanced computation over graphs with billion nodes.

2.2.1 Pregel model

Pregel is inspired by *Bulk Synchronous Parallel* [113] computation model. For a given input graph where vertices/edges are labeled with attributes, graph computations are performed iteratively as a sequence of superstep iterations updating the whole graph, which are separated by global synchronization points. During each superstep, the system invokes the the same operation(vertex program) on every vertex in parallel. Each vertex first receives all the messages which were addressed to it by other vertices in the previous super-step.

Each vertex performs the actions defined by user-specific function, i.e. *vertex.compute()* [96] or *vertex.program()* [44], in parallel, using the updated values received in the messages. More precisely, the function *Compute()* invoked in each superstep might consist of the following operations:

- receive the messages that where sent to the corresponding vertex in the previous step;
- send messages to connected vertices that will be read during the next superstep;
- modify the state of the node and of its outgoing edges;
- modify the graph topology by adding/deteting/modifying vertices or edges.

Then each vertex may decide to halt computing or to pass to other vertices the messages to be used in next super-step. When there is no message transmitted over graph during a super-step (*i.e.* every vertex has decided to halt) the computation stops.

The framework provides two mechanisms, *Combiner* and *Aggregator*, to reduce the communication cost from local and global aspects respectively. For example, in the computation of PageRank, we don't consider one single rank value but the sum of them, thus a combiner can be enabled to sum up the values sent to same target. Consequently the total number of messages transmitted and buffered will decrease. For aggregator, users can define it to make statistics and global coordination to reduce the input values from supersteps.

Due to *Pregel* success, several optimizations have been recently proposed in literature like the function *Master.compute()* [96] to incorporate global computations or *Mirror Vertices* [79] to reduce communication.

```
interface GASVertexProgram(u) {
  // Run on gather_nbrs(u)
  gather(D_u, D_(u,v), D_v) → Accum
  sum(Accum left, Accum right) → Accum
  apply(D_u, Accum) → D_u^new
  // Run on scatter_nbrs(u)
  scatter(D_u^new, D_(u,v), D_v) → (D_(u,v)^new, Accum)
}
```

Fig. 2.2 The gather, sum, apply, and scatter functions in PowerGraph [45].

Unlike the Pregel model, GraphLab proposes to use an asynchronous distributed shared-memory abstraction, instead of synchronous and message-passing mechanism. The vertex-program will even have more power, e.g., the ability to schedule neighboring vertex-programs to be executed in the future and can directly access the data stored in neighboring vertices. More details can be found in [77].

2.2.2 GAS model

In [45], Gonzalez etc. present a new framework, *GAS*, to implement the vertex program in a different way for more explicit distinction between vertex and edge specific computation. They divide the vertex program into three phases, Gather, Apply, and Scatter, to "combines the best features from both Pregel and GraphLab"[45]. Generally speaking, this model makes the graph computation and communication among vertices more focused on edges, which can be considered as a "edge-centric" graph processing model in practice. Its programming abstraction[45] can be simulated as shown in figure2.2:

where D_u, D_v and $D_{(u,v)}$ are the data, e.g. intermediate values or meta-data, associated to vertices u, v and the edge between them $e_{(u,v)}$. To *gather* function, these information from adjacent vertices and edges in the pre-defined direction(s) are collected and passed through a commutative and associative *sum* operation(similar to that Combiner in Pregel), then a temporary accumulator with user-defined type will be returned. After gather operation, the *apply* function take the final accumulator for calculating a new value for vertex u. During *scatter* phase, the function will be invoked in parallel on the edges adjacent to vertex u, in the pre-defined direction(s). It produces the new edge value $D^{new}_{(u,v)}$ to $e_{u,v}$ that will be written back to graph in the end of current superstep. In addition they conducted the first formal analysis of challenges on computing real life power-law graphs, and limitations in existing solutions, e.g. the data layout. To solve this problem, a balanced p-way vertex-cut data distribution algorithm was developed in their work to provide more adapted graph partition results w.r.t the power-law characteristic in graphs. Another similar graph processing abstraction,

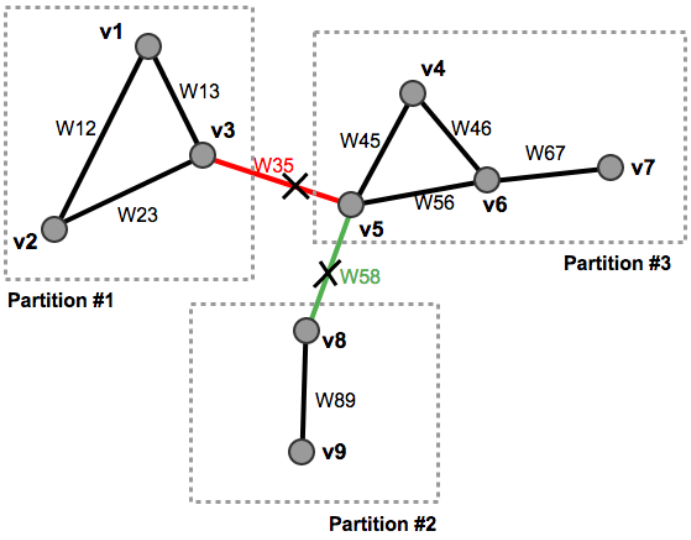


Fig. 2.3 A simple undirected graph is divided into 3 partitions $1-3$ via cutting edges $E_{3,5}$ and $E_{5,8}$ at the cost of $W35+W58$, which are the weights on them respectively.

GraphX, was presented in [118], which is also based on the vertex-cut graph partitions. It is built on top of *Apache Spark*[2], a known distributed data-flow system for its in-memory computing feature. They followed the GAS decomposition concept but also made some extensions, e.g. allow the direct communication between non-adjacent vertices.

2.3 Edge-Cut

The first approaches for large graph partitioning proposed to partition a graph $G(V,E)$ by dividing it by vertices, which means a partition p is designed as a subset of vertices $p \subseteq V$. The intuition is to group vertices w.r.t specified sizes and minimizing the number of edges between partitions. An example of such an approach is depicted in Fig.2.3 where the graph G consists of 9 vertices, $v1$ - $v9$, and has been split into 3 partitions, $\{v1, v2, v3\}$, $\{v4, v5, v6, v7\}$ and $\{v8, v9\}$. We notice that the edges $E_{3,5}$ and $E_{5,8}$ are cut since they rely vertices from distinct partitions. This explains why such kind of graph partition methods is called "Edge-Cut", or "Vertex Partition".

Then we can define the Edge-Cut graph partitioning problem in a general way,

[2]https://spark.apache.org/

Definition 1 (Edge-Cut Partitioning)**.** *Given a graph $G = (V, E, W_V, W_E)$, where W_V and W_E denotes the weights of the vertices and edges respectively. Its vertex n-partitioning $P = \{p_1, p_2, ..., p_n\}$ can be defined as:*

$$\bigcup_{i=1}^{n} p_i = V \text{ and } \{p \cap p' | \forall p \in P, \forall p' \in \{P - p\}\} = \varnothing, \text{ such that}$$

$$\begin{cases} min \sum_{e \in E_{cut}} w_e, w_e \in W_E \text{ and} \\ min \sum_{p_i \in P} cost(p_i, V) \end{cases}$$

where n is the number of partitions and p_i is one partition(subset of V) from partitioning P, $|p_i|$ is the number of vertices of p_i. Besides, E_{cut} is the set of edges cut in partitioning, w_e and w_v are the weights on edge e and vertex v respectively. $cost(|p_i|, |V|/n)$ is a cost function to measure the balance of partitioning, we can use, for example, $(\sum_{v \in p_i} w_v - \sum_{v \in V} w_v/n)^2/n$ to calculate it.

Here we notice that the Edge-Cut partitioning process can be translated into an optimization problem: how to split the graph in a balanced way and at the same time cutting as few as possible edges, preferably whose weights are as low as possible. This problem is known as NP-hard problem, even for $k = 2$ partitions, which is also called the Minimum Bisection Problem, it is still NP-Complete[8]. For this reason a variety of works have been proposed to provide approximations or heuristics. For instance ones of the first works on minimum bisection problem were proposed in [63] and [39]. Besides, several approximated algorithms were presented based on achieving different improvements of approximation ratio in [37, 97, 36].

In addition, to estimate the quality of graph partitioning or clustering results, a metric called *Graph Conductance* have been widely used [58, 100, 76]. The basic idea is that a partition with stronger connection inside and weaker connection to external would intend to have lower conductance to other ones. The conductance is formally defined as follows:

Definition 2 (Graph Conductance)**.** *Given a graph $G = (V, E)$, the conductance of one of its vertex partition p, $\tau(p)$, can be defined as:*

$$\tau(p) = \frac{cut(p)}{min\{\sum_{v \in p} deg(v), \sum_{v \notin p} deg(v)\}}$$

where $cut(p) = |e_{v,u} \in E : v \in p, u \notin p|$. Note that the minimization of $\tau(p)$ to $p \subset V$ is known to be NP-hard[58].

2.4 Vertex Partition Methods

From last century, a number of graph vertex partition (Edge-Cut) methods have been proposed to solve the various large scale problems like compute a good ordering for the parallel factorization of sparse matrices, design a good telephone network, decompose data structures for parallel computation and place the components of an electronic circuit.
In the following I introduce some of the major contributions based on the edge-cut approach.

2.4.1 Kernighan-Lin Heuristic

In [62], Kernighan and Lin developed the famous graph partitioning heuristic called *Kernighan-Lin(KL) Heuristic* which had deeply impacted lots of algorithms after it. This algorithm has been improved in [39]. Its general idea can be described as follows:

1. arbitrarily divide the graph vertices into two groups p_1, p_2 of equal size,
2. select two never-chosen-before vertices $v_1 \in p_1$ and $v_2 \in p_2$ such that the cost reduction is maximum when swapping them,
3. swap v_1 and v_2, and let C_i be the current partition cost of p_1' and p_1'
4. return p_1', p_1' as the partitions with smallest cost C_i observed,
5. if the cost can be reduced still, repeat step 2-4, else end.

Obviously this bisection method is easy to be extended to multiple partitions($k > 2$), in which k initial partitions are constructed to meet required sizes and then apply the KL heuristic over every pair of partitions until no improvement can be obtained. The KL heuristic procedure has been proven to be effective and employed in many graph partitioning approaches and applications [52, 19, 121]. However this algorithm consumes considerable computation time and can not be adopted for very large graphs.

2.4.2 Spectral Methods

Another series of methods[111, 23, 17, 91] are inspired by the work of Miroslav Fiedler[40, 41] on bisections of irreducible matrices and algebraic connectivity measurement in algebraic graph theory. Basically the graph two-way partition (also called spectral bisection) problem can be converted to the calculation of the eigenvector (Fiedler vector) v_2 corresponding to the smallest non-zero eigenvalue λ_2 of the graph Laplacian.
Here I will introduce the general idea and computing process briefly. Two important matrices,

incidence matrix $In(G)$ and Laplacian matrix $L(G)$ must be first introduced for a clear understanding:

Definition 3 (Incidence Matrix)**.** *Given a graph $G = (V,E)$, its incidence matrix $In(G)$ is an $|V| \times |E|$ matrix, with one row for each vertex and one column for each edge. For edge $e = (i, j)$, the column e of $In(G)$ is 0 except for the i-th and j-th entries, which are +1 and -1, respectively.*

Definition 4 (Laplacian Matrix)**.** *Given a graph $G = (V,E)$, its Laplacian matrix $L(G)$ is an $|V| \times |V|$ symmetric matrix, with one row and column for each vertex, and $L(G)_{i,j}$ is defined:*

$$L(G)_{i,j} = \begin{cases} deg(v_i) & i = j \\ -1 & i \neq j \text{ and } e_{i,j} \in E \\ 0 & otherwise \end{cases}$$

Consequently we have $In(G).(In(G))^T = L(G)$ and $L(G) \times v = \lambda.v$ where v is an eigenvector and λ is an eigenvalue of $L(G)$. In addition we obtain that the eigenvalues are non-negative, $0 = \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$ and that the number of connected components in G is equal to the number of λ_i with a zero value. In particular we have $\lambda_2 \neq 0$ if and only if G is connected.

Based on Fiedler's theorem on algebraic connectivity of graph G, more precisely $\lambda_2(L(G))$, the basic bisection algorithm of vertices can be performed as: 1) allocate vertex i to partition N^- if $v_2(i) < 0$, 2) otherwise, allocate i to partition N^+.
This method is known to its capability in building good partitions for many graph model applications. However the calculation of Fiedler vector is still costly in both time and communication[59], even if we speed it up via parallelising the execution[17, 102].
For instance, in [102] Simon established a framework with several new decomposition algorithms for unstructured computational domains, esp, one is focused on computation of eigenvector of the Laplacian matrix associated with graph. In detail, he implemented the recursive spectral bisection(RSB) using a modified Lanczos algorithm for Fiedler vector execution. The general procedure can be described as follows:

1. compute the Fiedler vector(FV) using Lanczos algorithm,
2. sort the vertices according to the sizes of components of FV,
3. allocate half the vertices to each subdomain,
4. repeat steps 1-3 on each subdomain until obtain the expected number of partitions.

2.4.3 Multilevel Scheme

The above two classes of methods are used to solve most of real world graph partitioning problems that range from scientific computation to industry products development, despite the expensive time-consuming to find out the optimal partitioning cut, using for instance iterated multiplication of large size matrices in Spectral Methods. However the sizes of many real-life graphs have become too massive and these methods reach their limits. Therefore a novel scheme, called *Multilevel*, was proposed to approximate the graph partition process in a more flexible and high-parallelism computing form, which can significantly reduce the time cost during partitioning.
Since the computation of recursive spectral bisection(RSB) is almost dominated by its first step (see section *Spectral Methods*), the FV calculation, Barnard and Simon developed in [16] a multilevel method, closely related to multi-grid methods, to compute the Fiedler vector in parallel for RSB. The basic idea is starting the search of FV from a small matrix which can approximate the larger graph (predecessor or original graph) and whose Fiedler vector FV' can be figured out in an acceptable time. Then this vector FV' is interpolated into the next higher level for producing a high-grade approximation to final FV for the Laplacian matrix. The process would be repeated until the final FV is found out.

Three elementary operations [16] are necessary for their multilevel computation:

- Contraction: construct a succession of smaller graphs that can keep the global structure of original graph.
- Interpolation: given one Fiedler vector obtained in Contraction phase, interpolate it to the predecessor (larger) graph to facilitate the computation of the following larger FV.
- Refinement: given a FV of graph, refine its vector values.

In [52], Hendrickson and Leland showed that the multilevel schemes can produce better partitions than the spectral ones at lower cost. In particular they first construct the coarse graph of quite small size to approximate original graph using a maximal matching algorithm (coarsening phase), and partition the small graph via spectral method. Then they uncoarsen the graph level by level until the desired partition is obtained (uncoarsening phase). During the uncoarsening phase, to determine the best partition of the coarse graph for its uncoarsened counterpart, they used a local refinement scheme which was originally proposed in the Kernighan-Lin heuristic(refinement phase).
The same year, Karypis and Kumar[59] proposed the first version of their multilevel graph partition algorithm whose implementation, *METIS*, has been applied extensively in many

areas so far. There are two main significant contributions they made to solve this problem: 1) a novel technique, called *Heavy Edge Matching (HEM)* introduced in the coarsening phase(described in [52]) to minimize the number of coarsening levels using a greedy method, instead of random matchings, and 2) two metrics, boundary greedy refinement(BGR) and boundary Kernighan-Lin refinement(BKLR), applied in graph uncoarsening phase with respect to the observation that most of vertices moved are along the boundary of the cut. In particular they proposed to combine these schemes into a hybrid scheme(namely BKLGR) that can adaptively launch appropriate refinement algorithm w.r.t the volume of target graph, to different levels. In other words, the BKLR would be applied to small graphs and BGR to large graphs. Using this hybrid scheme the time required for refinement can be reduced.

2.4.4 Geometric Methods

Many works like [51, 83, 33, 4] are dedicated to the optimal cut (also called separator) on graphs with geometrical information. However they can hardly be applied in a wide range of applications. For instance, in [83], the authors developed a geometric partition approach for a specific class of graphs, *k-overlap graphs*, such as planar and k-nearest neighbor graphs, which both have the required geometric properties of embeddings. But for many real-life graph-structured data, their geometric information doesn't exist or is not easy to capture. Besides, the another limit results from its relatively lower quality of partitions, compared to those obtained by spectral methods. It must be underlined that it runs however faster than spectral methods.

2.5 Vertex-Cut

While there exists a large literature and several implementations for vertex-partitioning, few recent works propose edge-partitioning (*vertex-cut*). The two principal ones are GraphX [118] and PowerGraph [44]. However GraphX only offers random/hash partitioning where edges are evenly allocated over partitions with some constraints of communication between nodes.The underlying graph property, like *local communities in social networks*, is not properly explored. Unlike the hash-like partitioning, PowerGraph uses a heuristic partitioning method, *Greedy Vertex-Cuts*, which has shown significant better performance than random placement in any cases [44]. However, it also ignores the graph topological property and only focus on how to minimize the future communication on previous partitioning situation during edges distribution among partitions. Additionally, GraphX and PowerGraph partitionings can not be updated dynamically with graph evolution.

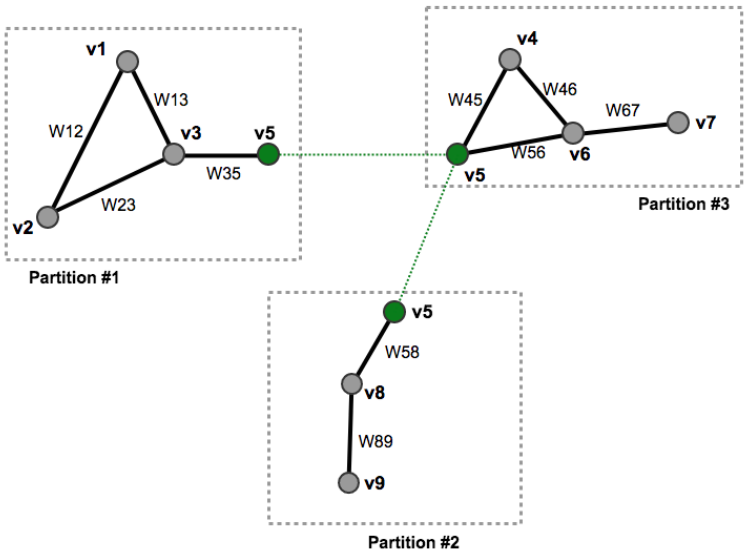


Fig. 2.4 A simple undirected graph is divided into 3 partitions via cutting vertex $v5$ for its 2 replicas in another two partitions, and the green dotted lines indicate where the communication happens during graph algorithms execution, whose cost consequently will be linear to global vertex replica factor(VRF), i.e. 2 in this partitioning.

The vertex cut partitioning strategy consists in splitting the graph into edge subsets, also call edge partitions. For instance, in Fig.2.4, we suppose we have obtained 3 edge partitions $\{ E_{1,2}, E_{2,3}, E_{1,3}, E_{3,5} \}$, $\{ E_{5,8}, E_{8,9} \}$ and $\{ E_{4,5}, E_{5,6}, E_{4,6}, E_{6,7} \}$. We see that the vertex $v5$ exists in all three partitions at once, so $v5$ has been "cut" into three parts, each per partition. This is the reason why we call it "Vertex-Cut".
This method has been applied in several latest graph processing models and systems. For example, in Graphx/Spark [118], they distribute the graph edges to machines in a random way or using some hash functions on vertex ids, which could offer fast graph partitioning but still present an important inter-partitions communication cost. Another proposal, PowerGraph [45] is able to build better partitions in most cases using a greedy algorithms based on several edge-allocation strategies, but with the same limitation concerning the communication costs.

The definition of this problem of determining the best partitioning will be discussed in Sec.3. In particular, it has been proven to be NP-hard in [127].

2.6 Edge Partition Methods

In the following paragraphs, I will introduce several recent study results, including open-source systems, models and heuristics, on this *graph edge partition* problem, based on different types of metrics.

2.6.1 Greedy Approaches

This kind of edge partition methods presented for instance in [44, 90, 94] are somehow similar to the Kernighan-Lin(KL) heuristic introduced in Section 2.4. Indeed, both approaches try to partition the graphs in one pass over graph elements(vertices in KL and edges in edge-partitioning approaches) via measuring the costs or gains in a greedy-algorithm strategy during elements' allocation. Inspired by the Pregel [80] and GraphLab [77] graph parallel computation abstractions, Joseph et al. developed a new "edge-centric" graph processing model, called GAS(Gather, Apply, and Scatter), in which they propose to handle the scalability of computing massive natural graphs, with highly skewed power-law degree distributions, using a balanced p-way vertex-cut. Generally, speaking, they use a sequential greedy heuristic which distribute the next edge e to a partition that minimizes the conditional expected replication factor(RF).
Particularly, to assign an edge $e_{u,v}$, the heuristic can be translated in the following rules:

- Case 1: assign e to $A(u) \cap A(v)$ if $A(u) \cap A(v) \neq \varnothing$ where $A(u)$ is the partitions(also called machines in [44]) having replica of vertex u and similarly for $A(v)$ with v.
- Case 2: assign e to the smallest partition in $A(u) \cup A(v)$ if neither $A(u)$ nor $A(v)$ is empty and $A(u) \cap A(v) = \varnothing$.
- Case 3: If only either $A(u)$ or $A(v)$ is empty, then assign e to the partition of which is non-empty.
- Case 4: If both $A(u)$ and $A(v)$ are empty, then assign e to the less loaded partition.

This graph edge partitioning is fast and can produce better partitions than randomized methods, but two problems still exist in practice, 1) there is no guarantee for reaching a "good" global result, which means it is difficult to predict the quality of final partitioning, moreover its quality may depends on the order with which we process the edges, 2) the coordination between partitions/machines required to calculate the $A(x)$ of vertex x which could lead to a high overhead in computation and in communication cost.
In [90], Petroni et al. build a stream-based partitioning approach, namely HDRF, that

mostly focused on the cut of high-degree vertices in power-law graphs. Similarly, it needs to maintain a global data structure keeping the degrees of vertices observed from input at runtime, therefore the time consuming during degrees updating/searching and communication between machines are considerable in a distributed computing context.

2.6.2 Random/Hash Methods

Given an edge $e_{i,j}$, which edge partition $p(t)$ (with $1 \leq t \leq m$ the partition's index among m partitions) it should be assigned to? To answer this question, several works [118, 117, 22] adopted a randomized or hash-based method, which can build partitions with a very small time cost but provide worst performances (processing time and/or communication time) compared to greedy or block-based methods in most scenarios.
In [118], which present one of the most popular graph partitioning system, four graph edge partition algorithms have been implemented as follows:

- RandomVertexCut: assigns an edge $e_{i,j}$ to partition $p(t)$ by passing i and j to a defined hash function such that $h(i, j) = t$, and meantime enabling the co-location of all edges with same orientation between i and j.
- CanonicalRandomVertexCut: similar to RandomVertexCut, but with a different hash function that can co-locate all edges between i and j *regardless* of orientation.
- EdgePartition1D: assigns the edges with same source to a common partition.
- 2D-Hash Partition: also called *EdgePartition2D* in GraphX, is developed in [22] to provide a two-dimentional partitioning of sparse edge adjacency matrix which can guarantee a $2 * \sqrt{m}$ bound on vertex replication factor.

To cope with the complexity brought into graph partitioning and computation due to the power-law peculiarity in many real-world graphs, [117] provided a new graph edge partition method, called *degree-based hashing(DBH)*, which attempts to perform effective exploration of the skewed degree distributions for graph partitioning. Roughly speaking, their approach is twofold:

1. Master-Vertex Assignment: vertex-cut partitioning model requires a master-replica for each replicated vertex to updating data for all mirrors. In DBH a randomized hash function $vertex_hash(i)$ is applied to uniquely assign the master replica of vertex v_i to one of partitions with equal probability.
2. Edge Assignment: assign the edge $e_{i,j}$ to one of the partitions by using a hash function $edge_hash(i, j)$.

For the second step *Edge Assignment*, they calculate it using the following function which originates from an intuition that "we observe that in power-law graphs the replication factor, which is defined as the total number of replicas divided by the total number of vertices, will be smaller if we cut vertices with relatively higher degrees."[117].

$$edge\_hash(i,j) = \begin{cases} vertex\_hash(i) & if\ deg(v_i) < deg(v_j), \\ vertex\_hash(j) & otherwise. \end{cases}$$

where $deg(v_i)$ is the degree of v_i. This is why they call their partitioning method *degree-based*. Especially, they also theoretically prove its ability to provide both low communication cost and good workload balance.

2.6.3 Transfer Partitioning

In [18], Bourse et al. made a theoretical study on the expected cost of vertex/edge partitioning with and without aggregation of messages. Particularly they demonstrated a polynomial time $O(d_{max}\sqrt{logk\ logn})$-approximation algorithm to the graph edge partition problem with message aggregation(note that d_{max} is the maximum degree of a vertex for any graph with $m = \Omega(k^2)$ edges). Moreover, given a sparse graph, they showed a heuristic for transferring a "good" graph vertex partitioning to an edge partitioning. It relies on the following idea:
Suppose we already have the vertex partitions $\{p_i\}$, where $i \in \{1,...,k\}$, and they are balanced with regard to degree weights. Now for each arriving edge $e_{u,v}$, we do one of the following operations with same probability in some way: 1) assign $e_{u,v}$ to edge partition p'_i such that $u \in p_i$, 2) assign $e_{u,v}$ to edge partition p'_i such that $v \in p_i$. As the edges were placed randomly, we can obtain an edge partitioning keeping a max-load ratio close to the previous, and will reach a lower communication cost than that original vertex partitioning. In the experiments with real-world graph datasets, they evaluated the hypothesis that we can obtain an edge partitioning from the graph vertex partition(with degree-weighted vertices) using a one-pass streaming algorithm, it will outperform the random edge placement algorithm for communication cost reduction but be worse than the least marginal cost strategy.
I will call this kind of approaches in the following , *"Transfer Partitioning"*, which provides a class of heuristics to acquire expected edge partitions from existing vertex partition solution.

2.6.4 Dynamic-Graph Challenge

In the context of large graph processing, a number of works [10, 95, 71] are dedicated to another important issue, *dynamism*, which is an intrinsec property of real life graph

applications. In social network applications, for instance, the graph evolves with additions or deletions of millions of users and connections(friendships, etc.) per day. Thus the difficulties in graph computation are not only from their massive volumes, but also from their constant evolution.
Traditional methods (see [10, 71]) to handle dynamic graphs are almost following this principle:

- avoid the re-computation of the whole graph when its topology has changed, since the cost is too expensive because of their large size.
- design the metric to maintain pre-computed values over graphs.
- update only a small part of vertices and/or edges values if their is a graph modification, *e.g.* when adding a vertex or deleting an edge. Actually the elements which are updated are those which are near the added/deleted element.

The similar idea has also been applied in graph mining applications, for example, the updating of PageRank in an evolving graph is considered as a crawling over small portion of whole graph in [13]. Additionally some work [25] demonstrated that many changes occurred in real world graphs are located in a predictable range, determined for instance by geographic and social constraints.

2.7 Local Access Pattern

Given a graph application, how to evaluate its communication cost in a distributed computing environment, especially. with a specific partitioning deployment?
To simplify this problem most existing works on graph partitioning have ignored the variety of operations of different algorithms applied on graph to provide a general solution instead of thinking about the quality of partitions themselves. More precisely, they assumed the algorithm's implementation would match a pattern like "label propagation" over whole graph. Thus almost every edge in graph is made active in each super-step/iteration for message exchange and the number of messages it receives will depend on its degree. So the overall number of messages will increase linearly to Vertex Replica Factor.
In some cases, however, the pattern will be quite different, e.g. the Random Walks are more likely to stay in local area and not every edge in graph will be chosen for message transmission during one super-step. In some other situations, *e.g.* Breadth-First Search, we might want to control the range of search rather than a global search in whole graph because only approximated results are needed as less computation time is used. In both above

examples, it would not be a good choice to just apply the general metric of partitioning, *e.g.* VRF, to measure the communications occurred in various applications/algorithms. Therefore, in following sections, we first propose to introduce the local access pattern(LAP) for a certain kind of algorithms, and to have a special focus on random walk-based algorithms for detailed discussion about the gain ensures by our partition method compare to other existing works on reducing communication cost.

2.7.1 Concepts

Firstly, I would like to outline some basic concepts in below:

Random Walks

A random walk [65] $w_s^t = [s, v^1, v^2, ..., v^t], s, v^j \in V$ on a graph is an ordered sequence of t visits, starting from vertex s, to vertices $v^1, ..., v^{t-1}$, and terminates at v^t. Besides, the random variable W_s^t is used to indicate that in a walk each visit $W_s^t(j+1)$, here to v, is chosen randomly according to some transition probability $P(v|W^t(j) = v')$ based on the previous hop.

Local Access Pattern

In most real-world graphs, like social networks, there exist many clusters (communities). Our objective is to take advantage of this topological characteristic in our block construction. *Local Access Pattern(LAP)* is described in [124] for first time as one of three kinds of query workload in graphs. We propose to rely on its principles and analysis when proposing our edge-partitioning strategy for random walks-based algorithms considering graph communities to reduce communication costs. Particularly for those which access graph locally, their tracks always tend to stay in a graph community(partition), *e.g.* Breadth-First Search(BFS), Triangle Counting(TC), and short random walk (RW). Consequently an edge partitioning strategy which considers graph local-communities feature, could benefit to algorithms which follow this local access pattern in reducing communication cost, and even in memory usage and in time-consuming.

Communication: a Bottleneck for Edge Partitions

In most existing edge partitioning methods, like Random [119] or greedy[45] approaches, balanced workload can be obtained, which means each partition has the same number of

edges. So, our main contribution is more focused on how to minimize the communication required during graph computation.

As introduced above, we know that in *Edge Partitions* a vertex might be allocated to multiple partitions by keeping its replications(mirror vertices) in different edges on those partitions. Thus, given an edge partitioning, we can use VRF to make an estimation of communication cost, as

$$O(L*(VRF*|V|)) \tag{2.1}$$

where L is the number of supersteps (iterations) during graph computation in Pregel model, and *VRF* is the Vertex Replication Factor of graph's edge partitions.

However, with this measurement, there is no consideration for the characteristics of the algorithms in graph computation. In other words, with regard to the simulation of a specific algorithm, like Fully Random Walks in this chapter, we need a more detail analysis of communication cost. In this way, we propose a more accurate expression for estimating communication cost of an algorithm *algo* during computing graph *G*:

$$Com_{algo}(G) \equiv \sum_{v \in V} NV(v) * Rep(v) \tag{2.2}$$

Where *NV(v)* is the number of visits to vertex *v* during algorithm simulation, and *Rep(v)* is the replication factor of *v*. Note that in this formulation, the size of message for communication is not considered.

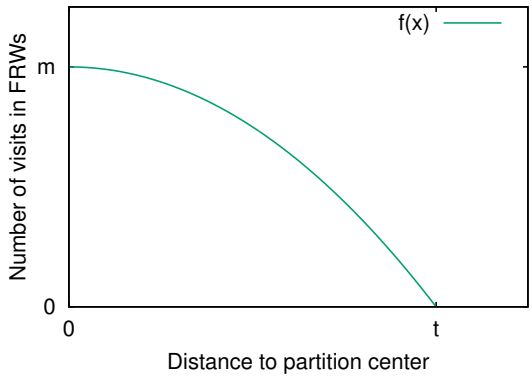
2.7.2 Random Walk with Local Access Pattern

We present the following property to justify why the LAP-based partitioning approach can facilitate FRWs over power-law graph on communication reduction.

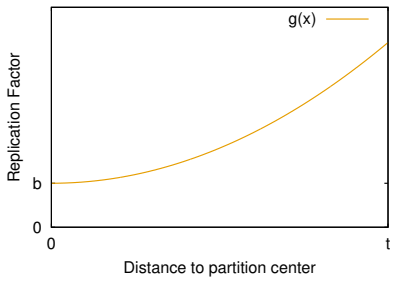
Property 2.7.1. *Consider the power-law graph G(V, E), where V is the set of vertices and E is the set of edges in G. When simulating fully random walks on LAP-based edge partitions of G, it requests less communication than on randomly distributed-based partitions of G, even if the latter's VRF is equal to the former's.*

Proof. An edge partition p is a subgraph of G, and for each vertex x in $p(V)$, there must exist at least one edge $x->y$ or $y->x$ in $p(E)$ where y is also in $p(V)$.

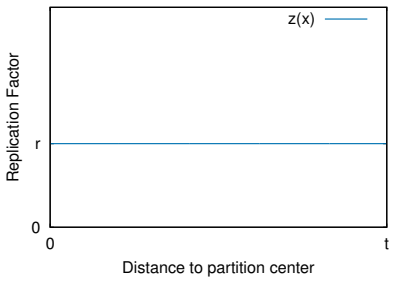
Given a power-law graph, it contains many communities which cover most of edges and vertices. Thus in our block-based approach which is a LAP-based edge partitioning method, p is a community, and $p(V)$ is a cluster which is a part of one of our blocks in Graph G, since



(a) The number of visits to each vertex in fully random walks(FRWs)



(b) Replication factors of vertices in LAP-based edge partition



(c) Replication factors of vertices in randomly distribution-based edge partition

Fig. 2.5 Given an edge partition (whose vertices form a cluster), to characterize every vertex's state during (after) fully random walks simulation: (a) Most walks starting from this partition will remain in this partition. (b) If the partition is obtained using LAP-based method, the vertices close to partition's center have small replication factors. (c) If the partition is constructed using random method, the vertices inside have similar replication factors.

it is constructed by some close edges. In previous discussion, we know that random walks starting from this community are more likely to stay inside of this community itself. From this observation, we can depict the relation between vertex's position (length of shortest path to partition center, so the seed in our proposal) and the number of visits to it in random walks, as in Fig. 2.5 (a). Note that here we assume that:

- the diameter of the partition (community) is large enough, thus we can approximately model that relation with a quadratic function
- the visits crossing boundary of p are rare, which would lead to $f(t) = 0$

More precisely, we represent it by

$$f(x) = m - nx^2$$

where $m > 0$ and $n > 0$ are coefficients and $m - nt^2 = 0$ when t is the boundary of p.

Since in LAP-based edge partitioning method, we allocate close edges together in a common partition, then most vertices from these edges do not have replications (mirrors) in other partitions. It means that the closer to p's center the vertex is, the small its replication factor is. We illustrate this relevance using a function $g(x)$ as in Fig. 2.5 (b),

$$g(x) = ax^2 + b$$

where $a > 0$ and $b > 0$.

In Section 2.7.1, we give the formula 2.2 to calculate the exact communication overhead during graph computation. Here the functions $f(x)$ and $g(x)$ indicate $NV(v)$ and $Rep(v)$ respectively. So, we can figure out the exact overall communication $Com_{FRWs}(p, g(x))$ during the processing of FRWs on LAP-based p by

$$\int_0^t f(x)g(x)$$

Second, we can prove that LAP-based edge partitionings guarantee a gain in communication cost compared to random partitioning methods whenprocessing FRWs simulation over power-law graph. Indeed in formula 3.1, the partitionings of same VRF will request equal communication cost, regardless of the algorithm conducted over graph.

Thus we assume that the randomly-distributed partitioning, see $z(x)$ in Fig. 2.5 (c), also obtains same VRF to the one in Fig. 2.5 (b), which can be described as

$$\int_0^t g(x) = \int_0^t z(x)$$

and

$$z(x) = r$$

where $r > 0$. Then we deduce the following equation from above.

$$r - b = \frac{a}{3}t^2 \tag{2.3}$$

Finally we prove that the exact communication on LAP-based edge partition $Com_{FRWs}(p, g(x))$ is less than that on randomly-distribution edge partition $Com_{FRWs}(p, z(x))$:

$$\int_0^t f(x)g(x) < \int_0^t f(x)z(x)$$

Based on this result, we deduce that

$$\frac{a}{5}t^2 < r - b$$

which can be derived from equation 2.3. □

We experimentally verify this result. We launched FRWs experiments on 200 partitions of LiveJouranl graph[123], where for each vertex it conducts two random walks of length four.

The result in Table 3.2 shows that our LAP-based edge partitioning approaches can even save a greater proportion of communication cost, compared to the proportion of VRF reduced.

2.8 Graph Processing Systems

In this section, I will briefly introduce several recent graph processing systems, and in particular some of them which propose specific techniques w.r.t graph partition problem.

Gemini [129] is a distributed graph processing system which aims at achieving scalability on top of efficiency. In this work, they first analyze the behavior of several existing systems to identify their design limits to multi-core shared-memory models, to figure out a way to bridge the gap between efficient shared-memory and scalable distributed systems. So they design a sparse-dense signal-slot model which controls the message passing/updating process between nodes, derived from GAS model in PowerGraph.
To enhance the computation scalability, similar to our block-based method, they presented a Chunk-Based partitioning technique which is developed to provide more efficient distributed graph stores via the use of the "locality" information hidden in graph, *e.g.* the geo-locations in social networks or the lexicographical URL ordering of web pages, however bad ordering IDs of input graph will significantly affect its effectiveness, unless an extra locality-"recover" operation is performed before.
For instance, in Fig. 2.6, a small graph is split into three vertex chunks, $\{0,1\}$, $\{2,3\}$ and $\{4,5\}$. The author propose to partition the graph vertices into contiguous subsets(chunks) with the assumption that the vertices order has preserved localities. Then they set the vertices, e.g. $\{0,1\}$ in $Node_0$, as masters to differentiate them from mirrors which will be added later. By expanding these master vertices with their corresponding dense-mode (very similar to "outgoing" mode) edges, the mirrors (black vertices) are also created for remote vertices. Generally speaking, they presented a low-overhead and easily-implemented partitioning

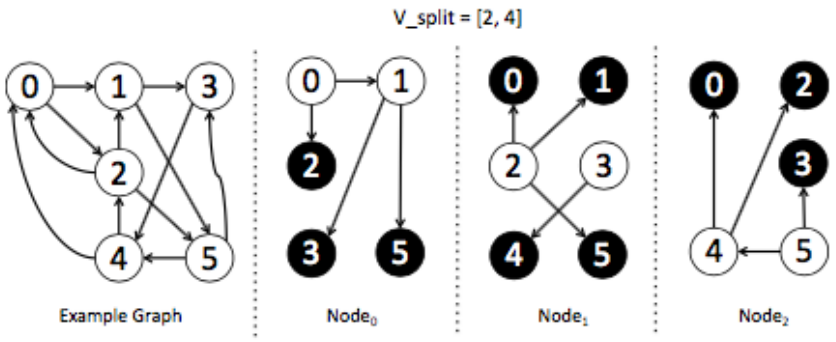


Fig. 2.6 An example of chunk-based partitioning presented in Gemini system [129].

method with taking into account the locality (more precisely, *adjacency*) information, which strongly relies on the ordering of input).
Along with the chunk-based partition algorithm, they propose a new metric $\alpha \cdot |V_i| + |E_i^D|$ for measuring load balance, instead of using only the number of edges, where both the number of vertices and amount of work on edges are taken into account. Note that α is a configurable parameter which was empirically set to $8 \cdot (p-1)$ in their experiments and p is the number of chunks.

Sheep is a novel input-independent edge partitioning method described in [81]. It generate an upper-bounded communication volume[18] partitioning of the original graph. The algorithm can be described in the following steps,

1. building an elimination tree from input (undirected-) graph using the elimination game algorithm, based on a distributed reduction,
2. partitioning the tree with an upper bound on the communication volume given by the tree depth,
3. finally translating the partitions of elimination tree into graph partitions using the mapping between them.

In particular the elimination tree is a directed acyclic graph that can express the reachability of original graph. As shown in Fig. 2.7, the right tree T is an elimination tree of left graph G, where the adjacent vertices (connected by edge), *e.g.* 2 and 7, should share an ancestor-descendant relation in T. Thus an element z in T and its reachable ancestor, namely $supr(z)$, will be a separator of its subtrees. For instance, $supr(5) = \{5,7\}$ is a vertex separator for 3 and 4.

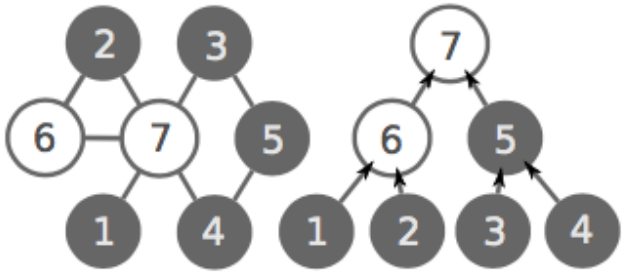


Fig. 2.7 An example of elimination tree shown in Sheep [81].

Then the authors present how the bounded communication edge partitioning of G can be obtained from a (cut-minimised) vertex partitioning on T. They employed a dynamic program [64] to find a minimal cost partitioning on T with given maximal subtree weight. Because each subtree in T can be translated into an edge partition in G, the upper bound on the communication volume could be consequently calculated from the sum of $|supr()|$ for each cut.

The experimental results show that Sheep makes edge partitions an order of magnitude faster than some existing approaches, e.g., METIS[61] and Fennel[110]. Besides, its partition quality is competitive with METIS for a small number of partitions and with Fennel for larger numbers of partitions.

GraphGrind [103] is a recent work from Sun et al., which proposes a specific NUMA-aware graph analytics system that focuses on reducing the performance impact of graph partitioning to overall system. They launched this work form several observations they found: 1) the extra overhead caused by multiplying partitions could quickly dominate performance gains in a distributed context, and 2) "the heuristic to balance CPU load between graph partitions by balancing the number of edges is inappropriate for a range of graph analyses" [103]. For these reasons, they rely on an optimized graph representation which can produce partitions which fit the characteristics of the algorithm, particularly specified to Non-Uniform Memory Access(NUMA) computing platform.

In practical, their graph representation is based on Compressed Sparse Rows (CSR) [93] and Compressed Sparse Columns (CSC), which are designed to compress sparse matrix for efficient storage and fast traversal during computation. Obviously most of real world graph follow the sparse property in matrix. For instance, the CSR maintains two arrays keeping the structural(adjacent) information in graph: one destinations array (DA) for storing the ids of destination vertices from each vertex's out-going edges (incoming edges for CSC), and one index array (IA) which keeps the starting place of vertex's destinations in DA. Thus normally their lengths are $|E|$ and $|V|$ respectively. For GraphGrind, some modifications are

proposed to lessen the side effects on system performances coming from graph partitioning: 1) a vertex-ID array is added to each partition to record the ids of vertices belonging to it, and 2) the index array will only store the information of vertices with non-zero degree, to facilitate faster sequential edge traversal during the iteration over index array. In addition, they applied some strategies to optimize the NUMA-awareness and made an extension to the Cilk parallel programming language for enhancing NUMA-aware scheduling in parallel programs.

Chaos [92] is a "low-cost" streaming partitioning method. It obtains sequential access to secondary storage since it proposes to save more pre-processing time by distributing graph data uniformly and randomly across the cluster, instead of achieving locality in graph. Besides, it uses a work stealing approach to handle the computational load balance problem. However this work is based on a debatable assumption that "machine-to-machine network bandwidth exceeds the bandwidth of a storage device and that network switch bandwidth exceeds the aggregate bandwidth of all storage devices in the cluster" [92]. This could not be applicable in all situations because it depends on multiple factors, *e.g.* the real capacities of network transmission and disk-access, network load/traffic in cluster and data location. Moreover, in some cases such as when we force the graph stored and computed in distributed shared memory, the communication bottleneck would exist between network and memory.

Chapter 3

Locality Exploration in Building Graph Partitions

We present in this chapter a novel block-based, workload-aware graph(-edge) partitioning strategy which provides a balance edge distribution and reduce the communication costs for random walks-based computations. Random walks-based algorithms, such as personalized PageRank (PPR) [56] and personalized SALSA [12] have proven to be effective in personalized recommender systems due to their scalability. Some recent proposals rely on multiple random walks started from *each vertex* on graph, *e.g.* Fully personalized PageRanks computation using Monte Carlo approximation [11]. We call this intensive computation Fully Multiple Random Walks(FMRW).

Our vertex-cut partitioning method is dedicated to random walks algorithms and take advantage of graph topological properties to capture *local communities* in graphs. To the best of our knowledge, this is the first time a partitioning strategy dedicated to fully multiple random walks algorithms is proposed in Pregel model.

We propose split and merge algorithms to achieve load balancing of workers and to maintain it dynamically. Our experiments illustrate the benefit of our partitioning since it significantly reduce the *communication cost* and time overhead when performing *random walks*-based algorithms compared with existing approaches.

In summary, our contributions are the following:

1. a *block-based* partitioning strategy which considers graph algorithms specificities and the topological properties of real-world large graphs along with a seeds selection algorithm for building the blocks;

2. algorithms for merging and splitting blocks to achieve a dynamical load-balancing of the partitions;

3. an experimental comparison of our partitioning approach with several existing methods, over large real social graphes.

The work presented in this chapter has been published in one international conference [74], a national journal [73] and an international demonstration paper [72].

Section 3.1 presents our block building strategy while Section 3.2 describes our blocks merge and refinement algorithms. Section 3.3 presents our experimental results and in Section 3.4 our visual interface tool for partitioning will be introduced in detail. At last, Section 3.5 concludes and outlines some perspectives.

3.1 Block-Based Graph Partitioning

This Section first introduces the main idea of our block-based graph partitioning strategy and then gives details about the partition building.

3.1.1 Principle

Most existing edge partitioning methods, like random[118] or greedy[44] approaches achieve a balanced workload, which means each partition has the same number of edges. Our objective is to go beyond workload balancing and to lower graph processing time by reducing the communication between partitions during graph computation. In *edge-partitioning* approach, a vertex is possibly allocated to multiple partitions and communications between partitions occur when updating the different replicas (mirror vertices) at each Pregel superstep. Consequently, Vertex Replication Factor(VRF) firstly defined in [44] is often used as a communication measurement. So, given an edge partitioning, the communication cost is generally estimated in Pregel, as

$$cost_{Comm} = O(L * (VRF * |V|)) \tag{3.1}$$

where L is the number of supersteps (iterations) during graph computation.

However, in most real graphs, like social networks, there exist many clusters (communities). Our objective is to take advantage of this topological characteristic in our block construction. *Local Access Pattern(LAP)* is described in [124] for first time as one of three kinds of query workload in graphs. We propose to rely on its principles and analysis when

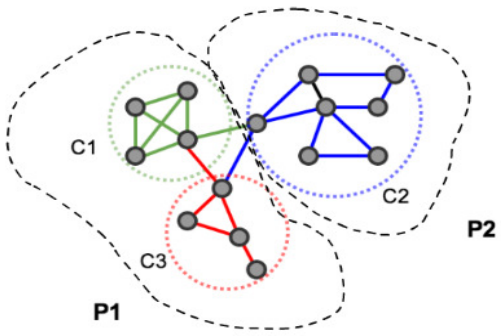


Fig. 3.1 Example of blocks and partitions

proposing our edge-partitioning strategy for random walks-based algorithms considering graph communities to reduce communication costs.

As a consequence we consider that, while VRF is a good estimator of communication cost for some graph algorithms, it is not suitable for the random walks-based algorithms which follow a LAP, since the number of visits of each vertex is different for these algorithms in one super-step. In other words, communications are conducted unevenly on graph. So our objective is to design a new edge partitioning strategy dedicated to random walks-based algorithms which takes into consideration both the power-law topology of the graph and the LAP characterizing these algorithms.

Our approach

A block corresponds to a tightly knit cluster in graph, *e.g.* a community in social network. In the *Pregel* approach, we consider the block as a set of edges which are "close" one to another, and these blocks become the component units of each partition in computation, but also the allocation units for workload over machines. Similar to the methodology adopted in vertex partitioning [43, 116], we propose to compute a set of K blocks by exploring the graph. An edge is allocated to a block based on its distance from this block. We start a breadth-first search exploration (BFS) from a pre-defined set of K seeds. For each edge encountered we update its distance with respect to all blocks. When the exploration step ended, we allocate the edges to the closest block.

Example 1. *Figure 3.1 presents an example which illustrates our 2-step approach. First, we group edges based on their distances to the different seeds (here three seeds). We get three communities(edge blocks): c1, c2 and c3. Then we merge blocks to get partitions with similar size. Here we build partition P1, composed of c1 and c3, and partition P2 corresponds to the single block c2.*

3.1.2 Distance of an edge

In graph computation, how to measure the closeness between a pair of nodes is a fundamental question and it has been studied in many existing works. One interest of these distance measures is to detect cluster in graph (see Section 3.1.3). But based on the observation that for several graph algorithms like random walk, nearest neighbors, breadth-first search, etc, the communications during computations mainly occur between vertices belonging to the same cluster, several approaches extended this cluster detection to perform graph partitioning. For instance [7] proposed a PageRank vector method to find a "good" partition w.r.t. an initial vertex and several pre-set configurations. Besides, there are some proposals like [98] which describes how to obtain these partitions by conducting random walks.

For our edge-partitioning approach, we propose here to estimate the distance between an edge and a *query* vertex. We adapt the inverse P-distance[56] used for distance computation between two vertices.

Vertex to vertex distance

Inverse P-distance captures the connectivity: the more numerous and short paths between two vertices, the closer they are in graph topology.

So, the distance $dist_v(i, j)$ from vertex i to vertex j in a directed graph G can be calculated by the paths between them, as follows:

$$dist_v(i, j) = \sum_{p \in P_{ij}} S(p) \tag{3.2}$$

where the P_{ij} denotes the set of paths from i to j. *S(p)* is the inverse distance value of path p defines below.

According to the idea of inverse P-distance, we introduce the concept of "reachability" into distance computation between vertices. The reachability means the probability for a random walk starting from i to arrive at j. So, for path p: v_0, v_1, ..., v_k with length k, *S(p)* can be defined by:

$$S(p) = (1-\alpha)^k \cdot \prod_{i=0}^{k-1} \frac{1}{outDeg(v_i)} \tag{3.3}$$

where $\alpha \in (0,1)$ is the teleporting probability, *i.e.*, the probability to return to the original vertex, and $outDeg(v_i)$ is the out-degree of vertex v_i.

Vertex to edge distance

Based on the vertex to vertex distance introduced above, we define a vertex to edge distance. We adopt the following definition:

Definition 5 (edge distance)**.** *The distance $dist_e(a,b)$ from a vertex a to an edge $b = (i,j)$ is:*

$$dist_e(a,b) = \theta(dist_v(a,i), dist_v(a,j))$$

where θ is an aggregation function.

In our experiment we choose the average function for θ but other functions like *min* or *max* may also be considered.

3.1.3 Edges allocation algorithm

Based on our edge distance we can now design an edge allocation algorithm. Our algorithm can be decomposed into three steps:

i) selection of a subset of vertices, namely *seeds*

ii) distance computation from each edge to all the seeds

iii) edges allocation to the different blocks

Seeds selection

We consider for our block-partitioning a seed-expansion strategy: we select a vertex as seed for each block and add each edge to one of the existing block. Obviously the result of the partitioning, in term of size-balancing or communication during the computation, is highly dependent on the choice of the seeds. This problem has been studied in literature for instance in [116] to detect communities on graph or in [27] where authors propose and experiment for the pre-computation step of their recommendation algorithm several landmark selection strategies.

Here we adopt the simple but efficient seeds selection procedure, based on *Spread Hubs* method (see [116]), which can be easily deployed on existing graph processing systems. There are two main measurements we used in seeds selection: 1) vertex degree, and 2) distance to other existing seeds. Our seeds selection algorithm is:

1. first we sort the vertices in descending order, according to their global (in+out) degrees;
2. then we scan the sorted list of vertices, and check if the current one is not *too close* to any existing seed, otherwise we discard it.

The rationale for this algorithm is that a vertex with a large global degree is a vertex with a centrality property and its connected vertices are likely to join its block. Moreover observing a minimum distance between seeds allows a better distribution of the seeds within the graph. Since BFS is efficiently implemented in Pregel, we use it to measure the *distance* between seeds. So we start a BFS from the seed candidate and report the number of hops required to reach the first existing seed. We observe experimentally that we achieve a good partitioning with this algorithm even when the depth of each seed's BFS is set 1 (so a new seed is not allowed to be the direct neighbor of an existing seed).

Number of seeds In our approach, each seed will determine a block which implies to have at least as many seeds as the number of final partitions. However we argue that we can achieve a better partitioning when setting this number to a larger value. The reasons are:

- the *expansion* of each block can be processed independently, thus can be deployed easily on Pregel-like architecture;
- the combination of small blocks needs *much less* overhead cost than splitting (*i.e.*, refinement) of large blocks when trying to minimize the replication factor;
- the more blocks we pre-computed, the higher the level of reusability of our partitioning will be.

Distance computation

For the second step of our algorithm, we compute first the inverse p-distance of each vertex to all seeds. To perform this distance computation efficiently in our Pregel-like architecture, we proceed to a parallel BFS exploration starting from each seed. Consider a set of seeds $\mathscr{S} = (s_1, s_2, \ldots, s_N)$. We maintain for each vertex v a distance vector $dist(v) = (d_1, d_2, \ldots, d_N)$ where $d_i = dist_v(s_i, v)$ is the inverse p-distance to the seed s_i. This vector is updated for each vertex encountered during the BFS exploration.

Since the BFS exploration in large graphs is very costly, we propose to limit the distance of exploration during the BFS. Indeed we observe in most of the large graphs (like social graphs) a community phenomenon which we capture by selecting the seeds among the vertices with the largest degrees, representing the center of these communities. Intuitively,

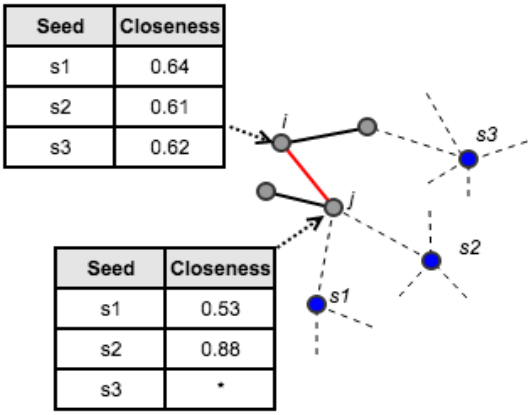


Fig. 3.2 Example of edge allocation

the distance from the community center is short to other vertices inside the community. Actually, from our experiment results and "Six Degrees of Separation" theory [85], we observe that the *radius of block*, *i.e.*, the distance from seed to potential community members is small and consequently the BFS depth can be set to a small value.

For instance, during the experiment on Livejournal [69] social network, we found the vertex/edge coverages of 200 seeds can reach around 88 percent and 96 percent by limiting the BFS only to 3 and 4 hops respectively.

Finally we compute a distance vector for each edge in the graph. Consider an edge $e(v,v')$ and the distance vectors for its vertices $dist(v) = (d_1, d_2, \ldots, d_N)$ and $dist(v') = (d'_1, d'_2, \ldots, d'_N)$. Based on Definition 5 we compute the edge distance vector $dist(\varepsilon) = (D_1, D_2, \ldots, D_N)$ as:

$$\forall i \in [1..N],\ D_i = dist_e(s_i, \varepsilon)$$

Edges allocation

Finally we can allocate the different edges to the blocks according to their edge distance vector. We decide that an edge belongs to the block whose seed is the *closest* to this edge. For edges without any distance value (which means its end vertices have not been reached by any seed during the BFS step), we allocate them in an extra-block.

Example 2. *We illustrate the* edge allocation *process with the example in Fig. 3.2. We assume we have already computed the vertex distance vectors for vertices i and j, considering three seeds s1, s2 and s3. Notice that the '*' value means that the current vertex can not be reached by the seed s_3 in our BFS exploration step. We sum (or make the average) the two vectors to determine the edge distance vector for $e(i,j)$: $dist(i,j) =$*

$(0.64+0.53, 0.61+0.88, 0.62+0.0) = (1.17, 1.49, 0.62)$. *Here we can clearly point out that the edge e should be allocated to s2 since it has maximum closeness value to this seed.*

Observe that some optimizations are possible for storing the vertex distance vectors and for the edge distance vector computation. For instance we can avoid keeping all distance values to every seed, since in this *edge allocation* step, only the maximum value is used to allocate an edge to a block. So we could keep only a *top-k* values for each vertex, with $k \leq |\mathscr{S}|$. Of course the larger k is, the more precise our final result is.

3.2 Blocks merge and refinement algorithms

Our block partitioning respects the topological properties of the (social) graph, *e.g.* local communities and power-law degree distribution to significantly reduce the communication costs compare to a random allocation strategy.

Given a number of servers P, we must determine how to allocate the different blocks to these servers considering two criteria:

- minimizing the global communication cost;
- balancing the storage and computation workload between servers.

These conditions can be captured by the following definition.

Definition 6 (Balanced edge partitioning)**.** *Consider a graph G(V,E) where V is the set of vertices and E the set of edges, a set of blocks $\mathscr{B}$ and a number of servers P. The* balanced edge partitioning $\mathscr{A}(\mathscr{B},P)$ *is defined as:*

$$\mathscr{A}(\mathscr{B},P) \in 2^{\mathscr{B}},\ such\ that \begin{cases} \forall \mathscr{A}' \in 2^{\mathscr{B}}, \frac{1}{|V|}\sum_{v\in V} |alloc(v,\mathscr{A})| \\ \leq \frac{1}{|V|}\sum_{v\in V} |alloc(v,\mathscr{A}')| \\ \\ \forall i \in [1..P], \eta\frac{|E|}{P} \leq |Edge(p_i)| \\ \leq \lambda\frac{|E|}{P} \end{cases}$$

where p_i is a partition (server) and $Edge(p_i)$ the edges it contains, $alloc(e,\mathscr{A})$ is the set of partitions to which edge e *is assigned with the partitioning $\mathscr{A}$ (more than one if the vertex is replicated) and $(0 \leq \eta \leq 1 \leq \lambda)$ are small constants to control the storage in each partition.*

The first part of the definition means the partitioning $\mathscr{A}$ is the one which minimizes the *Vertex Replication Factor*(VRF). The VRF measure adopted for instance in [44] means the less partitions the vertex span on average, the less communication across partitions the system initiates for vertices synchronization before running into the next superstep. The second part of the definition allows to control the size of a partition to fit the server capacity and to have an almost balanced edges distribution.

With respect to Definition 6 we can proceed to the final partitioning based on the different blocks we built.

Block split

Since the edges allocation to blocks is only based on a distance criterium some blocks may not fit the maximum size allowed for a partition (second part of Definition 6). Consequently we propose a simple split strategy. Assume that the size of a partition p_i is $(\beta-1)\lambda\frac{|E|}{P} \leq |Edge(p_i)| < \beta\lambda\frac{|E|}{P}$. We then apply our block building algorithm to the partition p_i with β seeds to split it into β sub-blocks. We potentially iterate the process for any of the sub-blocks which exceeds the partition size.

Blocks merge

Our block building may also result in producing some blocks whose size is lower than the minimal size (*i.e.* $\eta\frac{|E|}{P}$, see Definition 6). For such a block we re-allocate its edges without considering its seed anymore. Observe that this may lead in turn to some block splits.

Block allocation

We assume that, possibly after some required splits, the size of all blocks respect the partition size limit. To allocate the blocks to the different partitions, two strategies may be considered: based only on the balancing of the partition sizes, or on minimizing the replication factor between partition.

Considering this latter approach, we exhibit the following drawbacks: (1) there is an exponential complexity for finding the best blocks allocation considering this criterium, (2) the final size of each partition may highly differ one from another, (3) reducing the global replication factor will not reduce that much the cost of the random-walks algorithms since a path starting in one block and finishing in another is unlikely (according to our blocks building) and finally (4) this partitioning could not evolve dynamically and the partitioning must be re-built when many edges are added or removed.

Consequently we decide to adopt a blocks allocation considering only the size criterium, to achieved a balanced partitioning. We propose a simple but efficient *greedy* algorithm. We allocate the largest block to the partition with the smallest size, and we iterate this strategy until all blocks are allocated. Consequently this allocation is in $O(|\mathscr{B}|)$ where $\mathscr{B}$ represents the set of blocks.

The whole algorithm is presented in Algorithm 1 where *split* refers to a function which proceeds to the block split introduced above, *sortSize* is a function which sorts a set of blocks according to their size, from the largest to the smallest one, and *first* returns the first element from an ordered set.

Algorithm 1: Block allocation algorithm

input : a set $\mathscr{B} = \{b_1, \ldots, b_n\}$ of blocks, a set $\mathscr{P} = \{p_1, \ldots, p_m\}$ of partitions
output : each block is allocated to a $p_j \in \mathscr{P}$

```
// Initialization to avoid large blocks
B' = ∅
foreach b_i in the B do
    if b_i.size > λ |E|/n then
        B' = B' ∪ split(b_i)
    end
    B' = B' ∪ b_i
end
// Sort the set of blocks in descending size order
B' = sortSize(B')
b = first(B'); while B' ≠ ∅ do
    p_i = smallest(P);
    p_i = merge(p_i, b); //merge b with the smallest partition
    B' = B' − {b};
    b = first(B');
end
Return P ;
```

Managing graph dynamicity

Large graphs, especially for social network applications, are often characterized by a high dynamicity. One important aspect of our partitioning algorithm is its ability to manage this dynamicity. Indeed when adding a new edge (for instance when adding a friend on Facebook or an url on a Website) we simply have to aggregate the two vertex distance vectors of the two vertices of the edge if both vertices were already present in the graph to compute its edge

distance vector. Then we allocate the edge to the block, and consequently to the partition, with the highest distance score. If one of the vertex is new, we have first to perform the BFS exploration from that vertex and compute its vertex distance vector. Potentially this edge allocation may lead to a block split which can be handled with our split algorithm. Oppositely when removing an edge, the size of a block may become too small and we proceed to our block merge algorithm.

3.3 Experiments

This section presents experiments on our block-based partitioning strategy. We compare it with existing edge partitioning methods: the hash-based approaches [118] and greedy algorithm [44].

3.3.1 Setting

Computation are performed using GraphX [118] APIs in Spark [125] (version 1.3.1), on a 16 nodes cluster. Each machine has 22 cores with 60 GB RAM running Linux OS. For our experiments we set teleporting probability α to a classical value 0.15. The depth of the BFS exploration (*i.e.*, the maximum length considered for paths from seed to other vertices) is set to 4. So we intend to compute only the nearby vertices to current seed, rather than to every vertices in graph, according to *Six Degrees of Separation* theory[86].

Data Sets.

We validate our approach on two datasets: LiveJournal [24] with 4.8M vertices and 68.9M edges, and Pokec [107] with 1.6M vertices and 30.6M edges. These datasets can be downloaded from *SNAP* [1].

Competitors.

Hash Partitioning. There are four wide used random(hash)-like partitioning methods[2], introduced in GraphX:

- RandomVertexCut: allocates edges to partitions by hashing the source and destination vertex IDs.
- CanonicalRandomVertexCut: allocates edges to partitions by hashing the source and destination vertex IDs in a canonical direction.

[1] https://snap.stanford.edu/data/index.html

[2] see details and implementations at http://spark.apache.org/docs/latest/api/scala/index.html

- EdgePartition1D: allocates edges to partitions using only the source vertex ID, co-locating edges with the same source.
- EdgePartition2D: allocates edges to partitions using a 2D partitioning of the sparse edge adjacency matrix.

Greedy Vertex-Cuts. PowerGraph proposes a greedy heuristic for edge placement process which relies on the previous allocation of vertices to determine the partition next edge should be assigned.

Graph Algorithms.
Our partitioning is tailored to radom walks-based algorithms which lead to important communication costs between partitions. So we decided to implement two such algorithms for validation: PageRank and Fully Multiple Random Walks. However we also intend to prove that this approach also offers good performances for other graph algorithms. consequently we also perform experiments with a connected components computation algorithm. So, in detail, we compare different partitioning strategies for the following algorithms:

- Fully Multiple Random Walks(FMRW): several algorithms for *ranking* and *recommendation* proposed in literature (like the fully Personalized PageRank(PPR) computation presented in [11, 99]) simulate multiple random walks from each vertex for their computation, as described in 2. So for our experiments we decide to perform two independent random walks of length 4 starting from each vertex. Observe that for longer walks, we can obtain them by combining short ones like in [11, 99]. Thus for this computation we have 4 supersteps(iterations) in our Pregel-like implementation. For each superstep, every walk at that vertex will be randomly delivered to one of its direct neighbors, here note that the *restart* operation is not introduced in. Finally we collect $|V| * 2$ random walks when all supersteps are completed.
- PageRank: the famous recommendation algorithm where the messages transmitted between vertices consist only of score values. Thus the I/O overhead is not very high since the *size* of message is small even if the number of messages is large. Particularly in our experiments, we conduct both static and dynamic versions of PageRank. The number of supersteps is fixed in former, to control the overall time-consuming of graph computation. The dynamic version relies on a convergence tolerance, traditionally 0.005 or 0.001, which provides more accurate results.
- Connected Components(CCs): an algorithm to compute the connected components which can typically be used to approximate the cluster structures in graph. In other

words, we can consider it as a locality-sensitive algorithm. For instance, in its implementation on Pregel model, we aim to label each vertex with the lowest vertex id of the component they belong to. When starting the algorithm we set the initial label of each vertex as its own id. At each superstep, the vertex will receive messages(labels) from its neighbors if they have lower label. Then each vertex updates its label using the lowest label value received during this superstep (or keeping its label if it receives no message). The computation ends after a pre-set number of supersteps, or when there is no more message to be sent. Finally, the vertices with same label compose a connected component.

- Strongly Connected Components(SCC): the directed-graph version of CC.
- Shortest Paths(SP): a classic graph algorithm to compute the shortest path distance from every vertex to landmark(s). Before we start to run the algorithm, a set of landmarks is selected among the vertices as the targets. Then, during each superstep, the vertex will receive the neighbours' distance information to each landmark to determine if its value should be updated. The computation terminates when no new message is transmitted over the graph. In general, it needs more communication at the first supersteps since almost every vertex acquires information from neighbours and updates its distance, but it reaches convergence quickly due to less and less updates on vertices.

Algorithm 2: Fully multiple random walks algorithm

input : A directed graph G = (V,E), number of random walks starting at each vertex *wn*, length of walk θ

output : *wn* segments, for each vertex in the graph

```
1  Initialization:
2  For each u ∈V, S[u, i] ← u.getRandomNeighbor(), where 1 ≤ i ≤ wn ;
3  for k ←2 to θ do
4      for i ←1 to wn do
5          foreach u in the V do
6              l = S[u, i].lastNode;
7              e = l.getRandomNeighbor();
8              S[u, i] = S[u, i].append(e);
9          end
10     end
11 end
12 Return S;
```

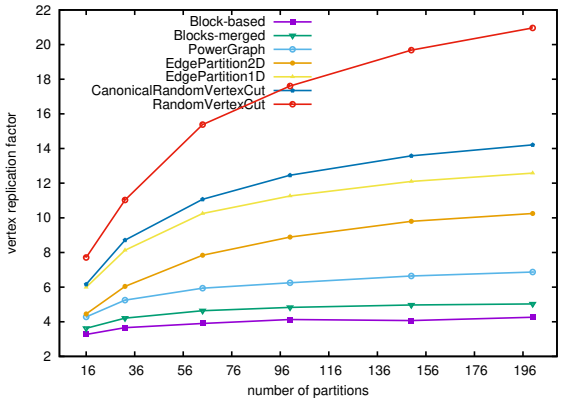


Fig. 3.3 VRF w.r.t. edge partitioning methods on LiveJournal

Table 3.1 Messages transmitted in FMRW (LiveJournal)

#Partitions	Random-Vertex-Cut[118]		Block-Based Partitioning			
	VRF	real mess.	VRF	real mess.	expected mess.	ratio
64	15.38	303.5m	3.90	55.3m	76.9m	0.72
100	17.61	381.9m	4.13	61.8m	89.4m	0.69
150	19.68	464.8m	4.07	70.6m	96.1m	0.73
200	21.12	525.6m	4.26	76.0m	106.0m	0.72

3.3.2 Communication

Our approach aims at reducing the runtime graph processing thanks to a significant reduction of the communication costs.

Vertex Replica Factor (VRF)

VRF is the traditional way to compare two partitionings regarding the communication costs, independently of the algorithm executed. We compare the VRF of our *Block-based* partitioning with the one of the competitors for different numbers of partitions. Results are depicted on Figures 3.3 and 3.4. We observe that, as observed in [44], partitioning strategies based on topology outperform as expected hash-based methods: VRF decreased by 30-60% (resp. 60-80%) for PowerGraph (resp. our block strategy) compare to the strategies used in GraphX. This experiment also illustrates the benefit of our global approach for edge allocation compare to a greedy approach with on average a 40%-lower VRF.

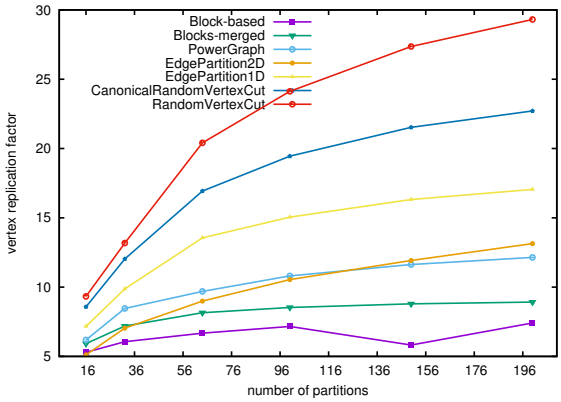


Fig. 3.4 VRF w.r.t. edge partitioning methods on Pokec

We also observe that the VRF exhibits a sub-linear growth wrt the number of partitions for all strategies. Indeed a low number of partitions generally leads to a large number of vertices which are shared by two edges from different partitions. When increasing the number of partitions, we also increase the number of vertices replicated, but our community-based clustering allows to limit this number. Observe that the growth remains however more important for the GraphX strategies.

Number of Messages

VRF is a general criterium to compare two partitioning strategy independently from the algorithms, but we expect our partitioning to exhibit even better results for random walks-based algorithms. Consequently to estimate the benefit of our approach we simulate fully multiple random walks (FMRW) and we measure the number of messages exchanged between partitions. From each vertex we perform 2 random walks of length 4 and we report experimental results in Table 3.1. We observe that our method reduces significantly the number of messages exchanged between partitions. For instance with 100 partitions, 61.8 million messages are necessary for processing the FMRW with our method while 381.9 million are transmitted with Random-Vertex-Cut method, so a drop of 84%. This result was expected since the VRF is 3-4 times lower with our method than with Random-Vertex-Cut. But we notice that if the reduction of the number of messages and of the VRF were proportional, the system should exchange 89.4 million message(see the underlined in Table 3.2). This 30% gain in the number of messages transmitted validates our intuition that random walks intend to stay in the local cluster(community). So low-replicated vertices (close to the seed in block) are accessed more times, and oppositely few random walks reach

Table 3.2 Exact and expected numbers of messages transmitted in FMRW w.r.t corresponding VRF(LiveJournal)

#Partitions	Random-Vertex-Cut	Block-based
64	303,5m:15,38	55,3m(76,9m):3,90
100	381,9m:17,61	61,8m(89,4m):4,13
150	464,8m:19,68	70,6m(96,1m):4,07
200	525,6m:21,12	76,0m(106,0m):4,26

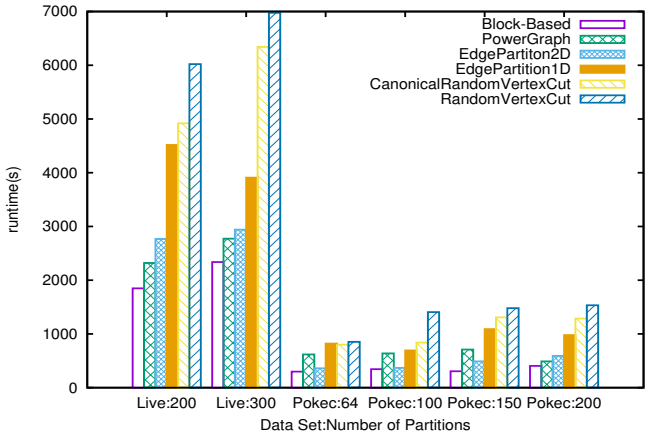


Fig. 3.5 Runtimes for FMRW with different partitionings for LiverJournal and Pokec

the farthest, high-replicated, vertices. Similar results are obtained from experiments on Pokec.

3.3.3 Runtimes

We propose to evaluate how the runtime of different graph processing algorithms benefits our partitioning, compared to other methods. First, we launch FMRW, a heavy-communication algorithm, on LiveJournal and Pokec datasets respectively, with 3 random walks of length 4 started from each vertex. From the results in Figure 3.5, we see that our partitioning can save up between from 20 to 65 percent of runtime, compared with other partitionings, for both datasets. We also notice that as expected runtime increases with the number of partitions, since processing the algorithm will require more communication. Regarding the competitors, we observe that PowerGraph's performances improve with the number of partitions, oppositely to hash partitioning methods from GraphX. Indeed PowerGraph greedy heuristic is based on reducing communication costs so the more partitions we have, the more important the gain is compare to GraphX.

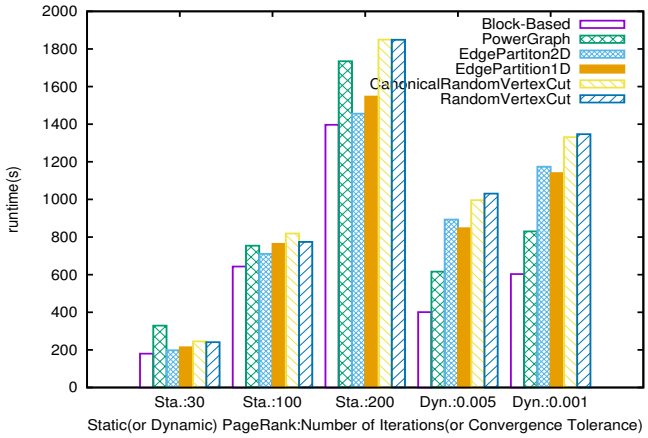


Fig. 3.6 Runtimes for static and dynamic PageRank for LiverJournal

We also test our method with traditional PageRank algorithm. We consider the static (fixed number of iterations) and dynamical (with convergence and a threshold value) approaches. We consider there are 200 partitions and we proceed to resp. 30, 50 and 100 iterations for static PageRank and to dynamical PageRank with resp. 0.005 and 0.001 convergence factor. Figures 3.6 and 3.7 depict results for respectively LiveJournal and Pokec datasets, and confirm that our partitioning method outperforms other ones. While we observe a small 5-20% gain for the static implementation of PageRank, we reach a 20-55% gain for the dynamical implementation.

So for static PageRank, oppositely to FMRW while we also proceed to a fix number of supersteps, performances of hash-based strategies are close to the ones got with PowerGraph and with our method. The main differences with FMRW is the size of the messages exchanged between partitions and the importance of the processing performed. While a message for FMRW contains the different random walks we are building, a message in PageRank only contains a score value and a node only proceeds to a scores aggregation. For dynamical PageRank (with convergence threshold), our method provides a significant gain for the performance. These experiments illustrate we provide a partitioning which follows existing communities in the graph. Indeed the random walks are more likely staying inside a given partition, leading to a faster convergence of the scores.

Besides, we propose to use the locality-sensitive algorithm, Connected Components construction, to check whether our partitioning also provide good performances on other algorithms which are not based on random walks. Both normal version and *strongly* version to directed graphs are conducted in experiments. The results in Figure 3.8 and 3.9 show that our partitioning method also outperforms other approaches for this algorithm. The rationale is that our partitioning groups edges based on graph topology and more precisely

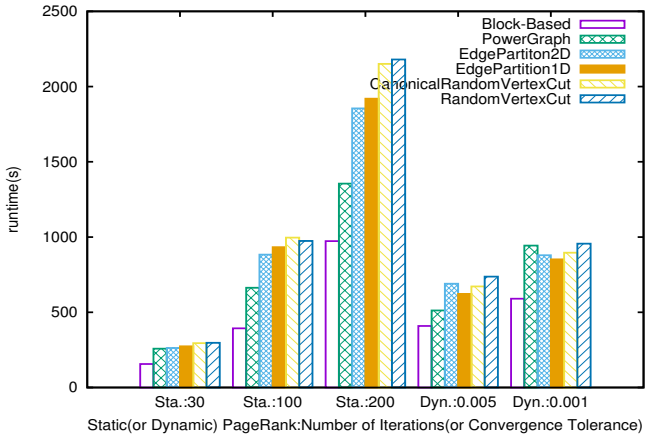


Fig. 3.7 Runtimes for static and dynamic PageRank for Pokec

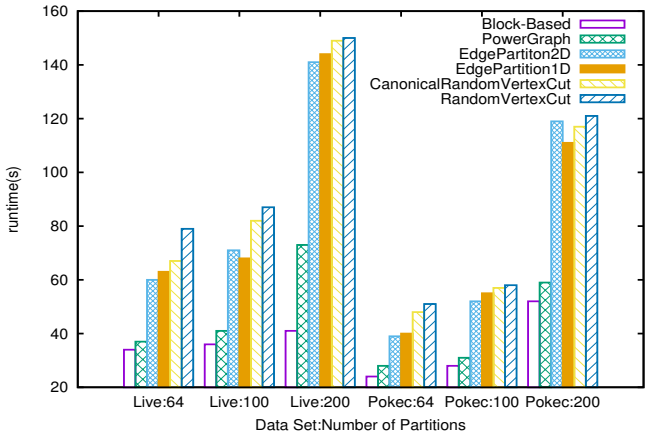


Fig. 3.8 Runtimes for Connected Components(CCs) computation with different partitionings for LiverJournal and Pokec

on connectivity and proximity. So with our approach the messages between partitions are more unlikely than with other strategies.

We also performed another classic not-random walks-based algorithm: Shortest Paths(SP) calculation algorithm over our partitions. We execute the different partitioning strategies mentioned in this chapter over both datasets of various partitions number, as shown in Figure 3.10. Our results confirm that our approach presents better runtime statistics compared to other ones due to the significant reduction of communication during graph computation.

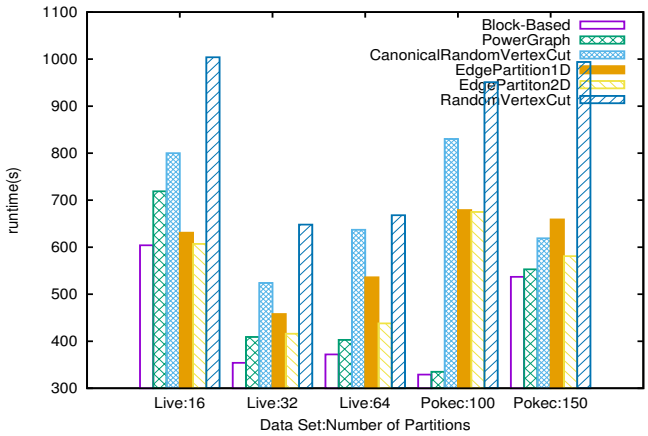


Fig. 3.9 Runtimes for Strongly Connected Components(SCC) computation with different partitionings for LiverJournal and Pokec

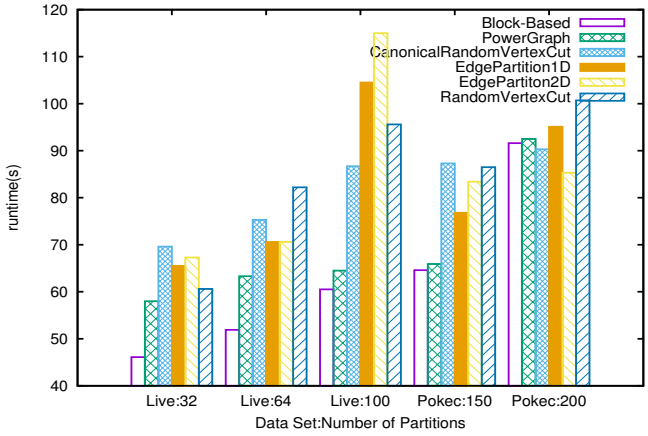


Fig. 3.10 Runtimes for Shortest Paths(SP) computation with different partitionings for LiverJournal and Pokec

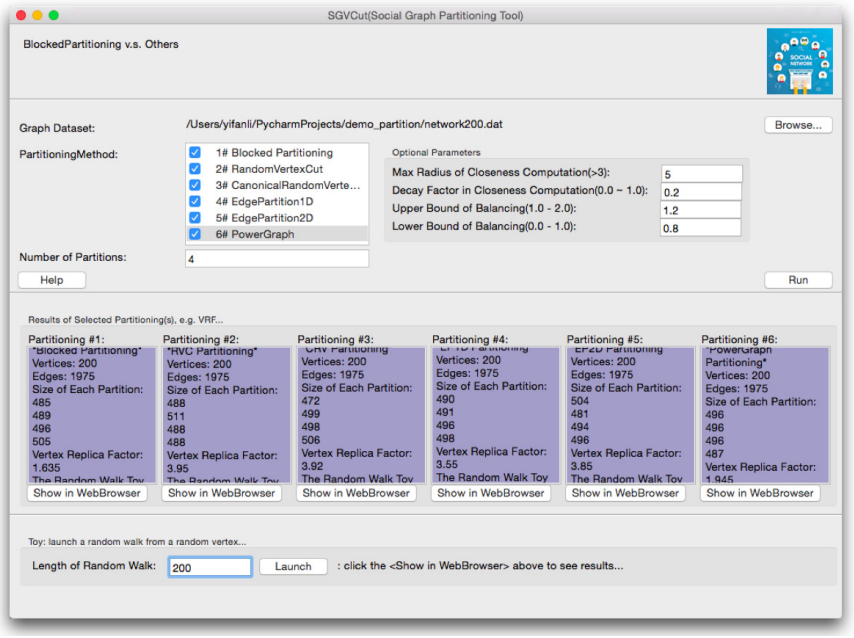


Fig. 3.11 Screenshot of main interface

3.4 Visualization

Figure 3.3 illustrates the benefit of our approach regarding the vertex replication factor (VRF) for LiveJournal dataset [24]. We build the partitions with our algorithm and compare the VRF with the ones obtained with existing algorithms. Our algorithm achieves a significant gain, between 0.5 and 0.18 compared to PowerGraph partitioning and the GraphX random vertex cut, respectively. When simulating fully multiple random walks (FMRW), we observe that our approach requires 6.25 times less messages exchanged between partitions than GraphX for instance (see experiments in Chapter 3.3.2).

Even to a large extent this experimental results can verify our assumption that partitions with effective LAP exploration, e.g. our block method, can provide more benefit to Random Walk-based applications on communication reduction, it still makes a lot of sense to observe this situation with more intuitive experiences. Thus in following sections, we propose an visualised interface to show what will happen during the graph algorithm execution over different partitions, which can explain the various performances of them.

3.4.1 Introduction

The visualization interface we introduced includes two windows, one is developed in wx-Python and the other is in HTML. We can directly deploy the required partitioning tasks

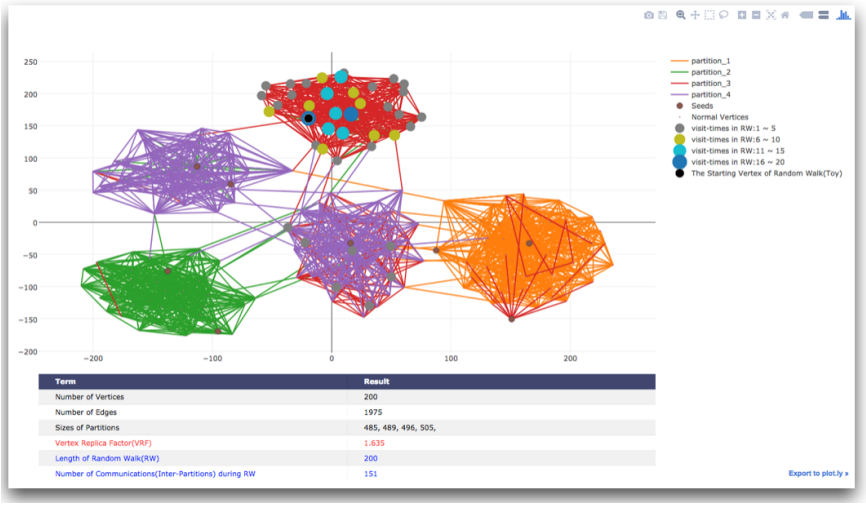


Fig. 3.12 Screenshot of HTML page

and graph mining applications, e.g. Random Walks(RWs) with pre-set length. The main window allows to set the different parameters and to execute our partitioning algorithm along with other partitioning algorithms. The second window permits to visualize the partitioning obtained with the different algorithms and the nodes visited during random walks. Particularly the track of RWs that spans multiple partitions can be easily output and visualized in a HTML page, see fig.3.13.

3.4.2 Building partitions

Figure 3.11 presents a capture of the main window. The first step consists in selecting the graph dataset that we intend to partition. For verifying its functionality we propose to use one of our real datasets, *i.e.* LiveJournal with 4.8M vertices and 68.9M edges, and Pokec with 1.6M vertices and 30.6M edges [3], or synthetic graphs produced by the benchmark graph generator in [66]. Then the partitioning strategies would be selected to perform and to compare to each other. We implement 6 different algorithms: our block-based algorithm, the four partitionning algorithms proposed in GraphX [119], *i.e. Random Vertex Cut(RVC), Canonical Random Vertex Cut(CRV), Edge Partition 1D(EP1D) and Edge Partition 2D(EP2D)* and *PowerGraph Partitioning(PowerG)* [45].

Next, we can set the number of partitions expected. In addition, some optional parameters are proposed to fix some block/partitions properties, like the *max radius* of a block when we require the partitions enforce this closeness property, or the *upper* and *lower balance bounds*

[3] SNAP, http://snap.stanford.edu/

which fix the maximum imbalance accepted for a partition. Finally the *decay factor* allows to control the effect of distance for partitioning based on connectivity computation.

After the validation of the different settings, the result of each selected partitioning method is displayed in this window (see purple frames on Figure 3.11). we could check the size of the partitions obtained with the different algorithms. For instance for the dataset used in Figure 3.11 we can verify that all algorithms achieve a similar balanced partitioning (each partition contains between 472 and 506 edges). However we also display the vertex replication factor to show the audience the benefit of our approach. Here our algorithm achieves to get a VRF equal to 1.635 while others produce a VRF between 3.55 and 3.95. We could test the impact of the different parameters on the partitioning algorithms and verify that our proposal provides for all settings the best VRF.

We can visualize the partitioning obtained with each algorithm through a Web interface (see Figure 3.12) realized using *plotly* [4] python library. Each color represents a partition. The seeds for the block construction are also displayed with purple dots. We see for instance that our algorithm produces partitions which correspond to very connected subgraphs with few edges between partitions. This interface allows also to underline the low VRF we obtain. When we click on a vertex we display the set of partitions it belongs to. In this way, We can demonstrate that less vertices are replicated and in less partitions, than with other methods.

3.4.3 Performing random walks

So far we only propose to run a random walk simulation with a chosen length from the main window, to compare the communication costs when performing this random walk with the different partitionings. The communication costs are displayed in their respective frame in the main interface. Observe that the initial vertex for the random walk is randomly selected. Then we propose to visualize the result of the random walks on the Web interface. The starting node corresponds to the black dot. We choose to depict the number of visits for a given vertex with dots, the larger diameter they have, the more numerous visits they got (see Figure 3.12). Finally we can check that the majority of visited vertices, especially the most visited ones, belong to the same partition as the initial vertex of the random walk. It illustrates the main property of our algorithm which takes advantages of the *local access pattern(LAP)* in social graphs, *i.e.* a reduced communication cost for random walk-based algorithms.

Moreover, the tracking of random walks over different partitions can also be shown in this interface, e.g. in Fig.3.13, a random walks of length 200 was launched on every kind of

[4] https://plot.ly/python/

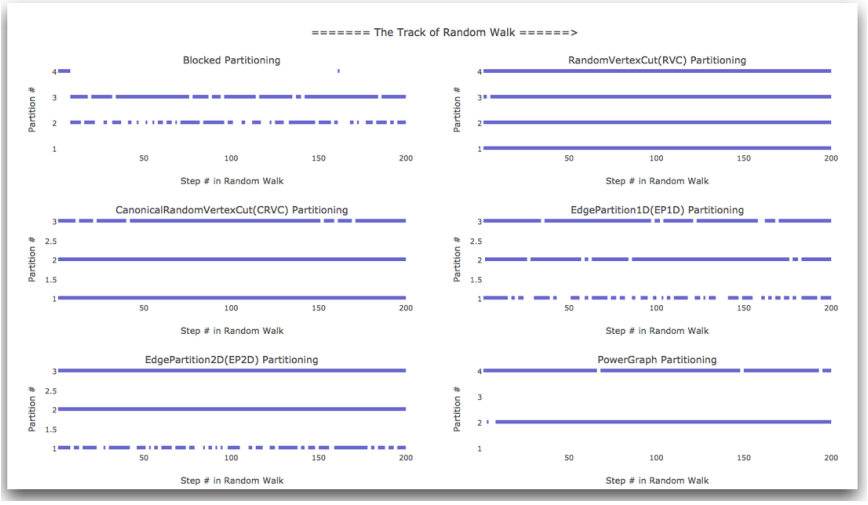


Fig. 3.13 Tracking results of random walks(length=200) over different partitions(k=4) based on Blocked, RVC, CRVC, EP1D, EP2D and PowerGraph partitioning methods

partitions(k=4) we have. For each hop in random walk, one of neighbours is chosen, then the partitions where its master or replica(s) exists will be coloured in purple. Therefore we can evaluate the communication cost during this random walks via the number of "purples" and obviously in this example our method performed best compared to others.

3.5 Conclusion

We present in this article a vertex-cut partitioning for random-walks-based algorithms relying on the topology to build blocks which respect local communities. We propose *split* and *merge* algorithms to get and to maintain the final partitioning. We experimentally demonstrate that our proposal outperforms existing solutions.

As future work we plan to investigate different seeds selection algorithms. While this problem has been studied in different context (see [116, 27]) we believe that the nature of the graph algorithms, here random walks-based algorithms, must be considered when selecting the seeds. We also intend to study the 5-10% of vertices which are not reached by the BFS exploration issued at seeds. They are located on the periphery of social graph and are poorly connected. While we currently place them to an extra-block, we will design a strategy to allocate them to existing blocks.

Chapter 4

Partitioning Large Multi-Layer Graphs

4.1 Introduction

An important number of applications are based on top of heterogenous graph where vertices and edges are labelled whith different type of attributes [101, 104, 20]. We focus here on *Multi-layer Graphs* [32, 30] with different connection types, such as the Like/Follow/Friend interactions in social networks [128], the multi-labels on edge in user-interest graphs [26, 6], or the different airlines between same pair of cities in air transportation networks [21]. For instance, in Fig. 4.1, Layer 0 represents the Air France company, which provides flights between cities a/b/c and b/c/d. Layer 1 corresponds for instance to the KLM company that has flights between cities a and d, and overlapped flight between c and d. Layer 2 corresponds to the Ryanair company that only offers flights between a and d.

The previously presented block-based partitioning method respects the topological properties of the (social) graph, *e.g.* the local communities and power-law degree distribution to significantly reduce the communication costs compared to a random allocation strategy. However our strategy can not be applied for multi-graphs where edges represent different types of relationship. For the existing works of heterogenous network partitioning, for instance in [5], Dan, etc. proposed to solve the streaming min-max hypergraph partitioning problem via a greedy items, i.e. vertices, allocation strategy to recover the hidden co-clusters of items in probabilistic inputs. In their proposal, however, only vertices specified with topics are considered for clustering without concerning the edges, and the structural property, like *locality*, underlying the graph has not been studied well during placement process of items. Interestingly, Lee and Liu developed a heterogenous graph partitioning method [68] in the context of cloud computing, based on components(vertex blocks)-grouping technique, which is similar to ours in last chapter, but their model only uses number of hops instead of more comprehensive exploration of locality during blocks construction and lacks the analysis

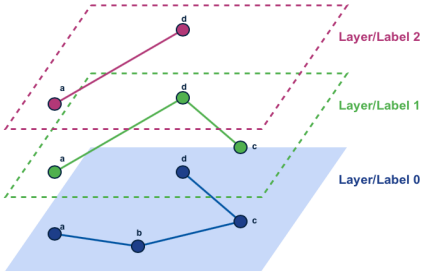


Fig. 4.1 Multi-Layer Graph.

of relations between blocks for efficient blocks merging. To address the data placement challenge in recent RDF stores(of typically heterogenous graphs), Janke, etc. performed an experimental analysis [54] on relative strategies w.r.t query processing and memory usage, and devised several measurements, e.g. they observed that a well distributed workload might be more essential than minimal data transport for efficient query processing.
What's more, many more general methods based on traditional objects clustering, eg. the clustering of authors in heterogenous DBLP bibliographic networks [105], transformation into multiple compatible bipartite graphs from original heterogenous information networks [122] and clustering with distinct semantics using paths of relations(meta-path) between objects [106], have been presented in this context, but even these approaches have obtained competitive performances in some particular applications, much extra effort are still required to transferring these vertex clustering/partitioning methods to edge partitioning, with specific restricts such as inter-partition communication cost and workload balance.
Hence in the following we propose a new edge partitioning strategy for heterogeneous multi-graphs. Our contributions are the following:

- starting from our partitioning method that only depends on the graph structural properties, we propose a new block-based partitioning method for heterogeneous graphs that takes into consideration the diversity of the edge labels.
- we propose new metrics to measure the aggregation and distribution of edges with common or similar labels in graph;
- we propose an optimised block merging algorithm that builds the required partitions.

Table 4.1 Summary of the Notation

Notation	Description
G	a graph, of multi-layer if no specification
P	a set of partitions
B	a set of blocks
S	a set of seeds
L	a set of labels(layers)
m	the number of seeds/blocks, $\|S\|$ or $\|B\|$
n	the number of labels in G, $\|L\|$
v_i	a vertex of id i in G
$e^l_{i,j}$	an edge in G, from vertex i to j and with label l
$label^l(es)$	the number of edges having label l in edges es
$adjacent(v_i)$	the adjacent edges to vertex v_i
$Deg^l(v_i)$	the number of adjacent edges to vertex v_i having label l, $label^l(adjacent(v_i))$
$neighbor(v_i)$	the adjacent vertices to vertex v_i

4.2 Block Construction

We present in the following the the *vertex-cut* partitioning of graph that groups the edges of G to several subsets, P, where the vertices might be duplicated across partitions. Table 4.1 lists the notations that we used in this chapter.

4.2.1 Data model

Definition 7 (Multi-graph)**.** *A multi-graph is a triple $G(V,E,\Lambda)$ where $E \in V \times V$ denotes a set of edges, V a set of vertices and $\Lambda : E \to 2^{\mathscr{L}} \setminus \emptyset$, with $\mathscr{L}$ the set of edge labels.*

In other word, a multi-graph is a graph whose edges are labelled with at least one label put potentially several ones. Conceptually a multi-graph may be considered as a set of graphs, one for each label, which share the same vertices. We define the restriction of a multi-graph G to a given label l as:

Definition 8 (Multi-graph restriction)**.** *Let $G(V,E,\Lambda)$ be a multi-graph and a label $l \in dom(\Lambda)$. The restriction of G to l, denoted G_l, is the graph $G_l(V_l,E_l)$ where*

$$V_l \subseteq V \wedge E_l \subseteq E \wedge \forall e \in E_l, l \in \Lambda(e) \wedge \forall e' \in E \setminus E_l, l \notin \Lambda(e')$$

So the multi-graph G_l is the graph, possibly not-connected, which contains all edges from G labbeled with l.

Block definition. A block corresponds to a tightly knit cluster in graph, *e.g.* a community in social network. In our approach, we consider the block as a set of edges which are "close" one to another, and these blocks become the component units of each partition in computation, but also the allocation units for workload over machines. More formally, a block is defined as follows.

Definition 9 (Block)**.** *Consider a graph $G(V,E)$, a set $\mathscr{S} \subseteq V$ of vertices called seeds, and an allocation function $alloc : E \times \mathscr{S} \rightarrow boolean$. A block b is a couple (s,E') where $s \in \mathscr{S}$ is a node called the* block seed *and $E' \subseteq E$ is a subset of edges such as $\forall e \in E'$, $alloc(e,s) = true$ and $\forall s' \in \mathscr{S} - \{s\}, alloc(e,s) = false$.*

The $alloc$ function allows to allocate an edge to the seed which is *"close"*, considering both the topology (*locality*) and the labels (*similarity*). Consequently a block groups a set of edges close to each other in the graph and sharing some common topics.

To take into consideration the locality of edges we propose a *Connectivity Computation*.

Connectivity Computation

To preserve the locality inside a block, we adapt the *Inverse P-distance* to capture the connectivity, *i.e.*, closeness, from a seed to a given vertex. The idea is that the more and shorter paths between two vertices, the more connected (closer) they are.
We compute the connectivity score $con_v^l(i,j)$ between vertex i and j for a given label l in multi-layer graph G:

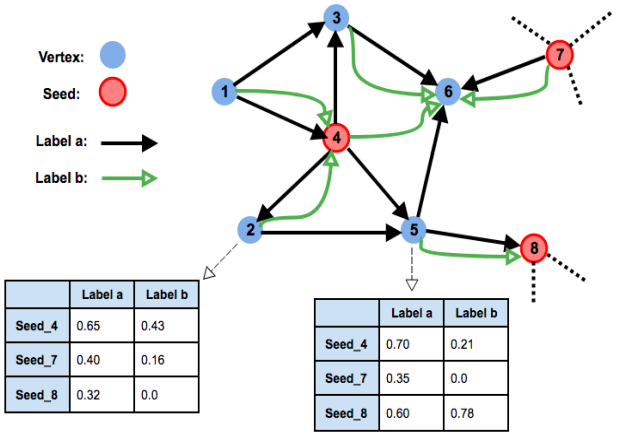
$$con_v^l(i,j) = \sum_{p \in P_{ij}^l} S(p^l) \tag{4.1}$$

where P_{ij}^l is the set of paths between i and j in the multi-graph restriction G_l (*i.e.* the set of l-labeled paths in the multi-graph G), $S(p^l)$ is the inverse distance value calculated on path p^l, $\{(v_0, v_1, ..., v_k)\}$ as:

$$S(p^l) = (1-\alpha)^k \cdot \prod_{i=0}^{k-1} \frac{1}{outDeg^l(v_i)} \tag{4.2}$$

where $\alpha \in (0,1)$ is the decay factor used in distance calculation, and $outDeg^l(v_i)$ is the number of out-edges with label l from vertex v_i.

Based on our *vertex-to-vertex* connectivity score we define an *edge-to-vertex* connectivity score which captures how "close" an edge is from a given vertex.



	Label a	Label b
Seed_4	0.65	0.43
Seed_7	0.40	0.16
Seed_8	0.32	0.0

	Label a	Label b
Seed_4	0.70	0.21
Seed_7	0.35	0.0
Seed_8	0.60	0.78

Fig. 4.2 An example of connectivity scores calculation over multi-layer graph.

Definition 10 (Edge Connectivity)**.** *The connectivity score* $con_e^l(k,e')$ *between a vertex* k *and an edge* $e' = (i,j)$ *is:*

$$con_e^l(k,e') = \theta(con_v^l(k,i), con_v^l(k,j))$$

where θ *is an aggregation function.*

In the experiments we will chose for the aggregate function the average function but any other aggregate function may be selected.

Example 3. *Figure 4.2 depicts a 2-layer graph with labels* a *and* b *and three seeds (vertices 4, 7 and 8). For the vertices 2 and 5 we compute the connectivity scores to the different seeds for the different labels. For instance, considering the label a, we compute on the restriction* G_a *the connectivity scores between the vertex 2 and the seeds 4, 7 and 8. We get respectively the connectivity scores* $con_v^a(v_2,v_4) = 0.65$, $con_v^a(v_2,v_7) = 0.40$ *and* $con_v^a(v_2,v_8) = 0.32$. *Similar computation for the vertex 5 on* G_a *gives respectively the scores* 0.70, 0.35 *and* 0.60. *Consider the aggregate* $\theta = sum$, *we obtain for the edge* $e_{2,5}$ *the edge connectivity scores* $con_e^a(v_4,e_{2,5}) = 1.35$, $con_e^a(v_7,e_{2,5}) = 0.75$ *and* $con_e^a(v_8,e_{2,5}) = 0.92$. *Observe that we have some vertex connectivity scores equals to zero. It means that there exists no path between the vertex and the corresponding seed.*

Since for each label, we have for each vertex (resp. edge) a score to each seed, we can adopt the following matrix representation.

Definition 11 (Vertex and Edge Connectivity Scores Matrices)**.** *Consider the set* $S = \{s_1,s_2,\ldots,s_\kappa\}$ *of seeds and the set* $\mathscr{L} = \{l_1,l_2,\ldots,l_\lambda\}$ *of labels. The vertex connectiv-*

ity score matrix for a vertex k, denoted V_k, is a matrix with size $|S| \times |\mathscr{L}|$ where

$$\forall i \in [1..\kappa], \forall j \in [1..\lambda], V_k(i,j) = con_v^j(k, s_i).$$

From this definition we can deduce the definition of the edge connectivity score matrix E_ε for an edge $\varepsilon = (k,k')$ as:

$$\forall i \in [1..\kappa], \forall j \in [1..\lambda], E_\varepsilon(i,j) = \theta(V_k(i,j), V_{k'}(i,j))$$

where θ is an aggregation function.

Observe that the computation of these two matrices can be implemented by performing a *Breadth-First Search(BFS)* from each seed in S for each label in L, which can be easily deployed in *Pregel*-like model systems. This computation has a space and time complexity of $O(|S| \times |L| \times (r^q))$, where $|S|$ denotes the number of seeds, $|L|$ the number of labels and r the average degree of vertices. q corresponds to the depth of the BFS performed, generally a small value, *e.g.* 5 in our experiments.

In the matrix E_ε each column corresponds to a label, and the highest score in a column points out the topologically closest seed for this edge in the corresponding graph restriction. However the candidate seed proposed for the edge allocation is likely to be different from one label to another. To determine the seed, so the associate block, to allocate the edge, we must consider all the labels in the same time. To achieve this, we first build a *block profile* for the different blocks.

Example 4. *Consider the example in Figure 4.2. The left table (resp. right table) corresponds to the matrix V_2 of vertex 2 (resp. V_5 of vertex 5) for the seed set $S = \{4,7,8\}$ and the label set $\mathscr{L} = \{a,b\}$.*
According to Definition 10, with the choice of addition as aggregation function θ, we obtain the connectivity scores matrix E_ε from the edge $\varepsilon = (2,5)$ to every seed for each label in multi-layer graph:

$E_\varepsilon =$

$$\begin{bmatrix} \mathbf{1.35} & 0.64 \\ 0.75 & 0.16 \\ 0.92 & \mathbf{0.78} \end{bmatrix}$$

We observe that for the label a (first column), the edge ε got the best score with the seed S_4 while for the label b the best score is obtained with the seed S_8.

Block Profiles

The block profile is a summarization of the interests of the users within a block. Different content-based profiles for communities are proposed in literature [35, 2]. We propose a simple block profile construction based on the number of occurrences of the different labels of the edges of the block, but more advanced strategies may be used.

So assume that for the set of edges E of a graph G we have a set $E' : \{E_1, E_2, ..., E_k\}$ of blocks, such that $\bigcup_{1 \leq i \leq} E_i \subseteq E$ and $\forall i, \forall j, i \neq j \Rightarrow E_i \cap E_j = \emptyset$. Also assume the existence of the function $occ : 2^E \times \mathscr{L} \rightarrow N$ which returns the number of occurrences of a given label in a set of edges. We define the label score of a block as:

Definition 12 (label score of a block)**.** *The label score for a block E and a label l is:*

$$score(E,l) = \frac{occ(E,l)}{\sum_{l_i \in \mathscr{L}} occ(E,l_i)}$$

The block profile of a given block E consists of the set of the different label scores computed for each label $l \in \mathscr{L} = \{l_1, l_2, \ldots, l_\lambda\}$. So we can represent the different block profiles for a set of blocks $E' : \{E_1, E_2, ..., E_k\}$ as a matrix P that we denote the *blocks profile matrix*:

$$P = (p_{ij})_{1 \leq i \leq \lambda, 1 \leq j \leq k} \; with \; p_{ij} = score(E_i, l_j)$$

Label correlation. The precedent definition assumes that the labels are independent one from another. In many applications, there exist however correlations between labels. For instance, we expect to group the edges(vertices) with labels *IT* and *Computer* together, or those with labels *Politics* and *Polls*, since they are more like to be queried or processed concurrently in applications.

So we assume the existence of a correlation matrix of labels $C(\mathscr{L}) \in Q^{|\mathscr{L}| \times |\mathscr{L}|}$, with $\forall i \forall j, c_{ij} = c_{ji}$ is the correlation score from label l_i to l_j. Observe that if the different labels are independent the matrix $C(\mathscr{L})$ is equals to $I^{|\mathscr{L}|^2}$.

Finally we adapt the definition of the block profile matrix to take into consideration the label correlation:

Definition 13 (blocks profile matrix)**.** *Consider the set of blocks* E' : $\{E_1, E_2, ..., E_k\}$ *and the set of labels* $\mathscr{L} = \{l_1, l_2, \ldots, l_\lambda\}$*, the blocks profile matrix P is the matrix*

$$P = P' \times C(\mathscr{L}) with \; P' = (p'_{ij})_{1 \leq i \leq \lambda, 1 \leq j \leq k} \; and \; p'_{ij} = score(E_i, l_j)$$

Edge Allocation

Intuitively, starting a random walk-based algorithm in a block with a small number of labels shared by its different edges will lead to less inter-block communication and consequently a faster execution time than a block with a large number of different labels with a uniform distribution. So our goal is to build such *topic focused* blocks, which represent basically blocks of users sharing similar interests, *i.e.* a community. To achieve this, we allocate an edge based on topology and on the different block's profiles.

Proposition 0.1. *Edge allocation Consider an edge ε and blocks profile matrix P. ε is allocated to the seed s_i determined by:*

$$argmax_{x\in\{1,\dots,m\}}\{a_x | a_x \in A = (E_\varepsilon \odot P) \times 1^n\}$$

where $\odot$ denotes the entrywise product*, and 1^n is a vector with all values equals to* 1.

Example 5. *Consider again the example in Figure 4.2. Suppose that labels a and b are independent, and the blocks matrix profile P for the blocks corresponding to the seeds 4, 7 and 8 is as below:*

$$\begin{bmatrix} 0.8 & 0.2 \\ 0.75 & 0.25 \\ 0.5 & 0.5 \end{bmatrix}$$

Then $A = (E_\varepsilon \odot P) \times 1^n$ is:

$$A = \left(\begin{bmatrix} \mathbf{1.35} & 0.64 \\ 0.75 & 0.16 \\ 0.92 & \mathbf{0.78} \end{bmatrix} \odot \begin{bmatrix} 0.8 & 0.2 \\ 0.75 & 0.25 \\ 0.5 & 0.5 \end{bmatrix} \right) \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1.35*0.8 & 0.64*0.2 \\ 0.75*0.75 & 0.16*0.25 \\ 0.92*0.5 & 0.78*0.5 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1.208 \\ 0.6025 \\ 0.85 \end{bmatrix}$$

*So from above result, edge $e_{2,5}$ should be allocated to block of seed-*4.

4.3 Seeds Selection

The choice of the seeds may largely impact the quality of the resulting partitioning since the edge allocation highly depends on the topological and thematic proximity of the seeds.

We propose in the following a strategy to select the seeds based on the topology and on the "profiles" of the seeds.

We first propose the *Variance of Edges*, $VOE(es)$, to measure how the labels from $\mathscr{L}$ are spread out over a set of edges $E' \subseteq E$. Intuitively, the more common edge labels and the fewer of them are involved, the higher VOE for E'.
Formally, given a set of edges $E' \subseteq E$, and set of labels $L \in \mathscr{L}$, the $VOE(es)$ is defined as the mean of the squared deviation from the mean of $label(E')$ for each label in L:

$$\frac{\sum_l^L (label^l(E') - \mu)^2}{|L|},$$

where

$$\mu = \frac{\sum_l^L label^l(E')}{|L|}.$$

Based on this measure, we can now propose criteria to select the seeds for our block construction.

Definition 14 (Seed)**.** *A vertice $s \in V$ belongs to the seed set S, which satisfies:*

i) $\forall s' \in S, s \notin neighbor(s')$

ii) $\forall v \in V \setminus S, \alpha \cdot Deg^{\mathscr{L}}(s) + (1-\alpha) \cdot VOE(adjacent(s)) \geq \alpha \cdot Deg^{\mathscr{L}}(v) + (1-\alpha) \cdot VOE(adjacent(v))$

where $Deg^{\mathscr{L}}(x) = \sum_{l \in \mathscr{L}} label^l(adjacent(x))$ *and* α *is an ad-hoc parameter in* $[0,1]$.

The first criterion avoid to select adjacent seeds which allows a better coverage of the large graph. Observe that we can decide to extend this criterion by requiring a distance of 2 or 3 between two seeds (but remember that recent results exhibit an average path length of 3.7 for Twitter [48]). The second criterium means that the seeds are selected among the most popular account (*i.e.* accounts with the largest number of labels on adjacent edges) but also those which federates a community around them with common/similar interest (*i.e.* labels), that will lead to a good block/partition construction. Indeed by selecting seeds with high degree and VOE we expect to build dense blocks with common labels, leading to lower communication, thus processing time, when querying or mining the graph.

Observe that the size of the seed set is a preset value, lower than the number of requested partitions.

The detailed procedure of selection is described in Algorithm 3.
Example 6 illustrates the seed selection algorithm.

Algorithm 3: Seeds selection algorithm

input : the graph G, labels L
output : a set of seeds S
parameter : the number of seeds m, the number of seed-candidates t and the importance factor of VOE ϕ

$S \leftarrow \emptyset$;
$V \leftarrow G(V)$; //the vertices of G
$C \leftarrow \emptyset$; //the seed-candidates
$N \leftarrow \emptyset$; //the neighbour vertices of existing candidates
while $|C| < t$ **do**
    $Deg(x) \leftarrow \sum_{l \in L} Deg^l(x)$;
    $v = argmax_{x \in V} Deg(x)$; //the vertex with max "Degree"
    if $v \notin N$ **then**
        $C+ = \{v \rightarrow (Deg(v), VOE(Adjacent(v)))\}$;
        $N+ =$ `neighbor(v)` ;
        $V- = v$;
    else
        $V- = v$;
    end
end
$C' \leftarrow$ `norm(C)`; //through all the elements in C, to normalize their Deg() and VOE() values respectively
foreach $c \in C'$ **do**
    $(deg, voe) = c$.value() ;
    c.valueUpdate($deg * (1 - \phi) + voe * \phi$) ;
end
$S \leftarrow C'$.maxByValue(m); //to select the candidates with first m max values
Return S;

Example 6. *Suppose we have a 2-layer graph G', with labels $L' = \{a,b\}$, and 3 candidate seeds, $\{c_1,c_2,c_3\}$, available.*
The statistic of labels on their adjacent edges are
$(label^a(adjacent(c_1)), label^b(adjacent(c_1))) = (6,4)$,
$(label^a(adjacent(c_2)), label^b(adjacent(c_2))) = (2,6)$ *and*
$(label^a(adjacent(c_3)), label^b(adjacent(c_3))) = (4,4)$.
Thus, we can calculate the $Deg^{\mathscr{L}}(c_i)$ of each candidate, represented here as a vector $dv = [6+4,\ 2+6,\ 4+4]^T = [10,\ 8,\ 8]^T$. We normalize this scores and get the vector $dv' = [10/26,\ 8/26,\ 8/26]^T$.
At the same time, their VOEs can be figured out via formulas above. For instance, for c_1, its VOE is $((6-5)^2+(4-5)^2)/2 = 1$. We compute similarly the values for c_2 and c_3 and find 4 *and* 0 *respectively. We also represent them as a vector $ov = [1,\ 4,\ 0]$.*
After normalisation we get the vector $ov' = [1/5,\ 4/5,\ 0/5]^T$.
We assume in the following that the ad-hoc parameter α is set to 0.5. We get consequently the final suitability *score for each candidate, as $(1-0.5)dv' + 0.5ov' = [0.292,\ 0.554,\ 0.154]^T$. We conclude that candidate c_2 is most suitable for becoming a seed among these three candidates since it has the maximum value.*

4.4 Block Refinement and Merging

Our edge allocation algorithm ensures the construction of blocks which present a topics homogeneity but which may leads to a block size heterogeneity with small blocks or oppositely very large blocks. To make blocks fit the expected partition size we propose to proceed in two steps: first we split the oversized blocks into smaller ones and second we merge the different blocks to get the final partitioning.

For the first step, splitting a block into several sub-blocks (*i.e.* re-allocating edges to different sub-blocks) in a random way would reduce the benefit of our block constructions where locality and topic similarity were considered. So we propose to iterate recursively the edge allocation algorithm by selecting
$\lceil size(B)/max\_size\_block \rceil$ seeds within the oversized block B. The block construction process stops when all blocks have a size lower than the given parameter *max_size_block*.

Finally to get the final partitions and to respect their maximum size we perform block merging. Relying only on block sizes to decide which blocks to merge would result in partitions composed by sub-blocks topologically and/or thematically distant. So we propose to exploit the block locality to decide the blocks to merge. To achieve this we consider the *block meta-graph*, a weighted directed graph G', where the vertex v'_i represents the block b_i

and the weight w_{b_i,s_j} on edge $e'_{i,j}$ is an vector which is the sum of edge connectivity scores from every edge in b_i to seed s_j(of block b_j), in other words, $\oplus_{e\in b_i}(E^e_\varepsilon)_j$, then we can merge the blocks according to Algorithm 4.

Algorithm 4: Meta-Block allocation algorithm

input : A set of blocks B of size n, a set of partitions P of size m, the labels L, the partition maximal size z
output : Each block is allocated to a $p_j \in P$

Initialisation to avoid large blocks;
$B' \leftarrow \emptyset$;
foreach b_i *in the* B **do**
    $b_i.size \leftarrow \sum_l^L label^l(b_i)$
    if $b_i.size > z$ **then** $B' \leftarrow B' \cup split(b_i)$;
    else $B' \leftarrow B' \cup b_i$;
end
while $B' \neq \emptyset$ **do**
    $p_i =$ `smallest(`P`)`//retrieve the smallest partition;
    $V = B'$`.multiple(`$P^L(p_i), w^T_{b_i,s_j}$`)`; //P^L: blocks profile matrix
    $b \leftarrow$ `largest(`V`)`; //to select the block with largest value in V
    $p_i =$ `merge(`p_i, b`)`; //merge b with the smallest partition
    $B' = B' - \{b\}$;
end
Return P ;

4.5 Experiments

To evaluate our partitioning method on multi-layer graphs, we propose to conduct three graph-basic fundamental operations, *Regular Expression*, *Shortest Paths* and *Random Walk*, over our real-world datasets.
We will conduct the experiments on a Spark(1.6.1)[125] platform which is deployed on a 16 nodes cluster representing 100 cores and each node being a 30GB RAM machine.
As the workload, *i.e.*, the number of edges, is balanced between partitions for each partitioning strategies used in experiments, we focus on the communication cost of each method to evaluate their performance.

4.5.1 Settings

Datasets

Three distinct datasets are used in our experiments:

Table 4.2 Datasets

	Edges	Vertices	Layers
Twitter	85,062,587	2,141,325	17
Higgs	15,450,464	456,630	4
HomoGenetic	154,354	18,208	7

1. An excerpt of the Twitter social network presented in [26]. It contains around 2 million vertices and 85 million edges, which represent the users(accounts) and their *follow* relationships between them. In particular, a label l on an edge (u,v), is to depict the interest of u for the posts (tweets) of v. 17 labels are extracted from a supervised classification model: *leisure, technology, entertainment, labor, social, disaster, sport, life, environment, religion, law, politics, business, health, education, climate, war* to tag the different edges.

2. The Higgs[28] dataset was built from the original graph derived from Twitter during the discussion and interaction on the news of discovery of elusive Higgs boson on 4th July 2012. 4 different interactions between users are extracted in this graph and are used to tag the edges: *friendship, replying, mentioning and retweeting*.

3. HomoGenetic[29] is a network obtained from a public database to describe various interactions between genes for organisms. In detail, there are 7 types of correlation which are used to tag the edges, *direct interaction, physical association, suppressive genetic interaction defined by inequality, association, colocalization, additive genetic interaction defined by inequality, synthetic genetic interaction defined by inequality*.

Table 4.2 summarizes the main features of our three datasets.

Competitors

We compare our algorithm with most popular edge-partitioning algorithms, *i.e.* the ones proposed in *GraphX* and in *PowerGraph*. More precisely we compare in our experiments the following partitioning strategies:

- Random edges allocation methods [119], such as *RandomVertexCut, CanonicalRandomVertexCut, EdgePartition1D and EdgePartition2D*.
- Greedy method introduced in PowerGraph [45].
- Our block-based method, esp., the *Meta-Block* version has been employed in all following experiments as it always outperform *4/3 allocation* method.

4.5.2 Regular Expression

For the first experiment, we compare the impact of the partitioning strategies when performing a frequently-used query in graph-structural dataset from the basic graph mining operation, *regular expression* [15]. In this experiment we focus more particularly on the frequent regular expression query

$$(v_1, l', *, l', x)$$

which means, we want to find all the vertices connected to it a given vertex v_1 by a path of two edges with common label l'.
For instance, in social network analysis, this query is used to facilitate the crime detection [15] if label l' corresponds to "lucrative deal", and potential client search in recommendation system if l' represents the user's interest.
For our experiments we run the different graph-partitioning algorithms to built a set of partitionings with different sizes for each method. We consider for our method that we initially partition the graph in ten times more blocks than the final requested partitions. Then we conduct the regular expression query on these different partitionings and compare the resulting performances based on two measures: the total runtime and the number of messages transmitted among partitions (communication cost).
In Figure 4.3, we observe that for the Twitter graph the communication cost linearly increases with the number of partitions for all partitionings. However our partitioning strategy largely reduces the communication cost. For instance with 100 partitions, we obtain a gain of 0.84, 0.3 and 0.33 for the random, EdgePartition2D or PowerGraph partitionings respectively. Regarding the runtime, our proposal also outperform other algorithms since reducing the communication cost in a distributed system directly impacts the execution time.

We perform similar experiments with Higgs and HomoGenetic graphs and the results are depicted in Fig. 4.4 and Fig. 4.5. We observe that our method also outperforms existing ones for these two datasets with a similar gain which enlightens that our approach remains efficient for graphs with a small number of distint labels (Higgs) or for small graphs (HomoGenetic). Indeed, our partitions present a topic-homogeneity, in other words few distinct labels, whatever the number of existing labels is. In plus we group edges based on topological proximity. Consequently the search for this regular expression on the graph is mainly processed within each partition, with few communication between partitions, whatever the size of the graph and/or the number of label are.

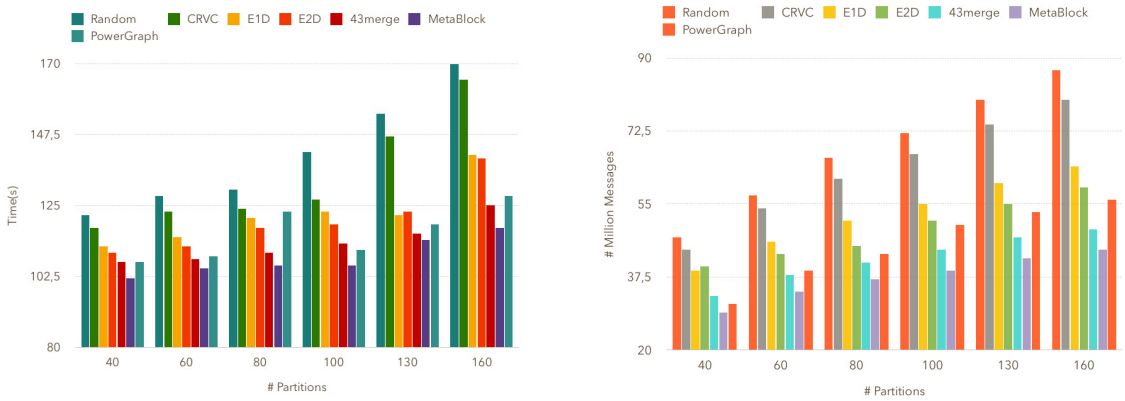


Fig. 4.3 Runtime and communication cost for the regular expression query on Twitter graph for 100,000 randomly selected vertices

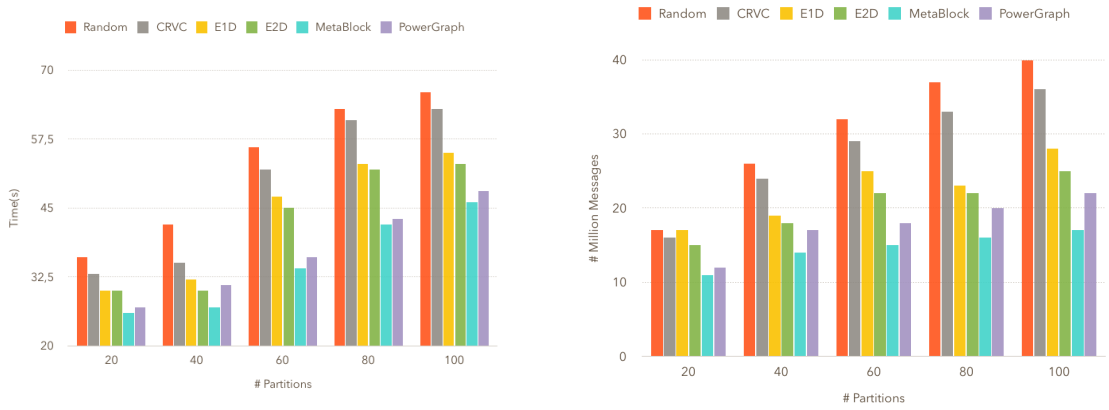


Fig. 4.4 Runtime and communication cost for the regular expression query on Higgs graph for 100,000 randomly selected vertices

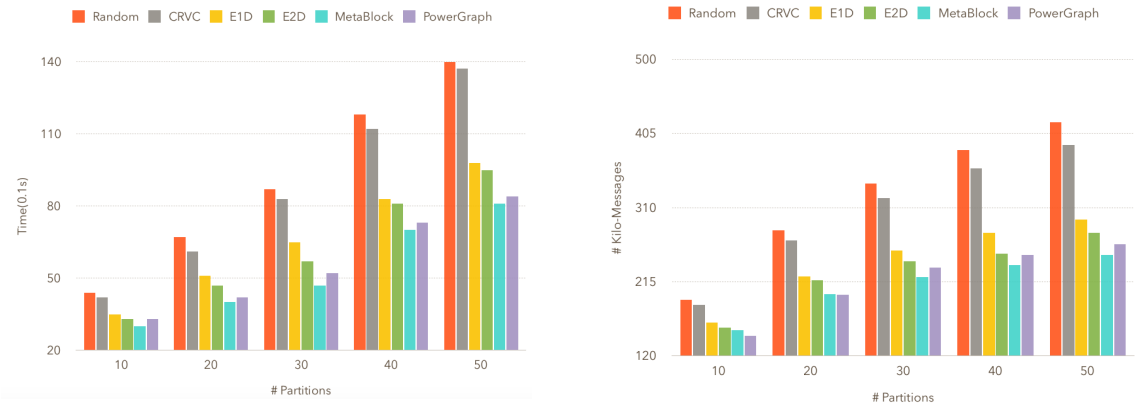


Fig. 4.5 Runtime and communication cost for the regular expression query on HomoGenetic graph for 10,000 randomly selected vertices

4.5.3 Shortest Path

For this second experiment we study another classic algorithm in graph mining, *shortest path*, used in many applications such as route recommendation in navigation and similarity calculation in graph clustering. For this experiment, in our multi-graph assumption, we consider that a *shortest path(SP)* search corresponds to the following problem: for a graph G, given some landmarks $L \in V$ and a specific label t, we want to find for every vertex $v \in V$ the shortest path to each reachable $l \in L$ in the graph restriction G_t. The result of this search is a vector of shortest path values to each landmark for each vertex. Observe that since the algorithm can be implement using recursive search of neighbors (BFS), the implementation can be naturally deployeded in Pregel-like systems, such as PowerGraph and GraphX.
Similarly to the previous experiment we construct 40-160 partitions for Twitter graph and a set of vertices are randomly selected as landmarks. The results are presented in Fig. 4.6. We notice that there is a decrease in the communication cost from 10 to 60 percent, compared to other methods. The rationale is that our partitioning considers the connectivity and the labels for building blocks. So the resulting partitions usually present large subgraphs extracted from restriction graphs. Since the shortest paths are computed on the restricted graph of a chosen label, the computation is more likely to stay within a partition. With other partitioning algorithms the vertices which belongs to the same restriction graph are allocated to much more partitions. So the vertex replication factor will be higher and more messages between partitions are required when performing a computation. Regarding the runtime, with a large number of partitions, *e.g.* 130 and 160, the runtime of other methods will consequently consume long time.

We make similar observations for the two other datasets (see 4.8 and 4.7), so our partitioning strategies is also efficient for graphs with less labels and/or smaller size.

4.5.4 Random Walk

Random-walks are the building blocks for numerous widespread algorithms in different areas such as Personalized PageRank [42] or TrustWalker [53] for recommandation, SimRank [55] for similarity computation, ItemRank [46] for ranking, or [67] for knowledge inference. So in this Section we conduct an experiment to illustrate how random walk-based algorithms may benefit from our partitioning algorithm. Algorithms on labeled graphs generally consider edge label as independent during the computation. However the layers or labels of multi-graph are not independent in a variety of applications, see for instance [114, 115, 31].

So we propose two types of random walk implementations for our experiments:

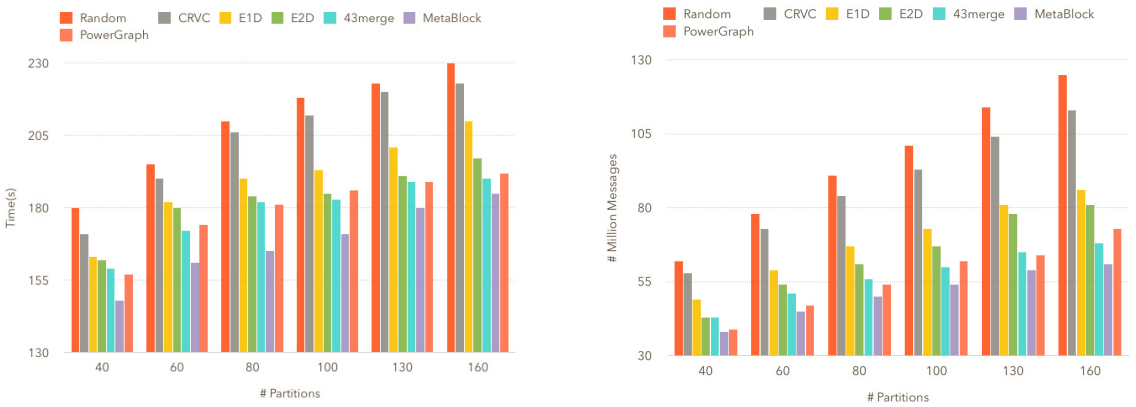


Fig. 4.6 Runtime and communication cost for the shortest path search on Twitter graph for 20 randomly selected landmarks

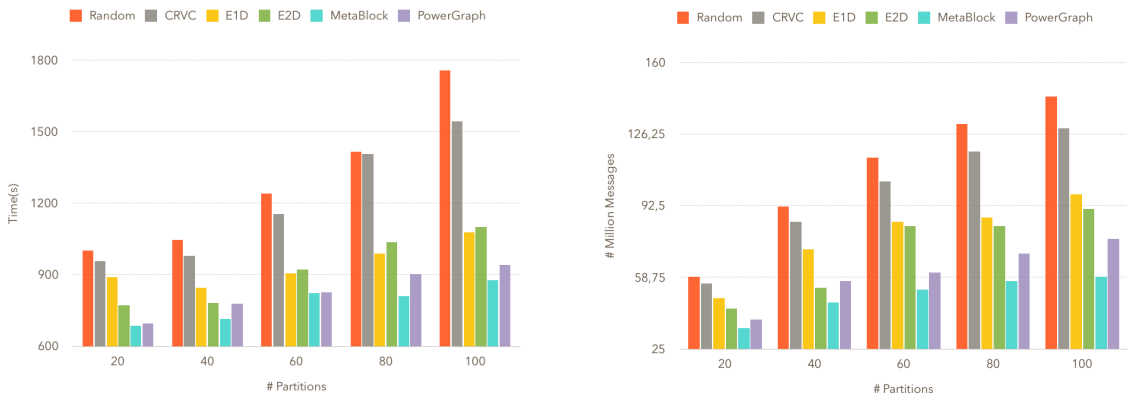


Fig. 4.7 Runtime and communication cost for the shortest path search on Higgs graph for 50 randomly selected landmarks

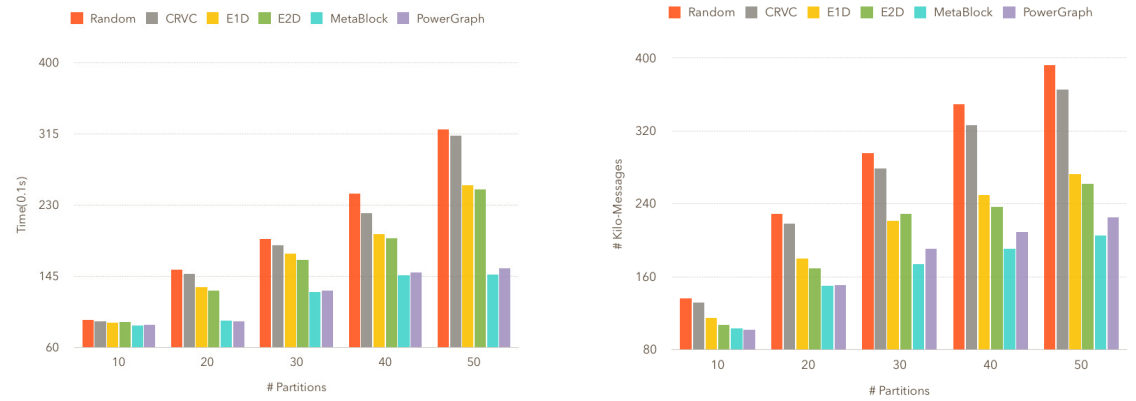


Fig. 4.8 Runtime and communication cost for the shortest path search on HomoGenetic graph for 200 randomly selected landmarks

Independent-label Random Walks,

where the random walks are conducted over only edge with queried label λ, *i.e.*, the graph restriction G_λ.

Correlated-label Random Walks,

where we rely on the *Correlation Matrix of Labels L, $C(L)$,* in Sec.4.2.1 into random walks execution. Thus the transitions for random walks consider all outgoing edges but with a weight which depends on the correlation between its label and the requested label. More precisely the transition probabilities are defined as:

$$P(v_i \to v_j) = \begin{cases} \frac{P(v_i \to v_j|\lambda, C(L))}{\sum_{e_{i,k} \in E} P(v_i \to v_k|\lambda, C(L))} & if\ e_{i,j} \in E, \\ 0 & otherwise. \end{cases}$$

where E is the edges of multi-layer graph G, λ is the label to query and $C(L)$ is the correlation matrix of labels L. In above formula, we propose to calculate the overall probabilities of each transition via linearly summarising all conditional ones. For the conditional probability $P(v_p \to v_q|\lambda, C(L), e_{p,q} \in E)$, it can be estimated as:

$$P(v_p \to v_q|\lambda, C(L), e_{p,q} \in E) = \sum_{l \in Labs(e_{p,q})} C_{\lambda,l}(L)$$

where the $Labs(e_{p,q})$ is the set of labels for the edge $e_{p,q}$.

In Fig. 4.9, we see the benefit of our partitioning algorithm when performing independent-label random walks on Twitter dataset. Regarding the communication cost, we observe a 30%-40% decrease compare to executions on partitions produced by GraphX strategies, and around 5-10% decrease for GraphLab. The rationale is that with our partition building the random walk is likely to remain in the same partition since our partition algorithm considers graph distance and labels. Regarding runtime, our method allows to achieve a 70% benefit compare to GraphX approaches.

Fig. 4.10 presents the results for correlated-label random walks on Twitter dataset. We notice that our method, *MetaBlock_hasCML*, always provides the best results, both for communication cost and runtime. The gain is even more important. This is due to our building which groups edges considering the topics and their correlation.

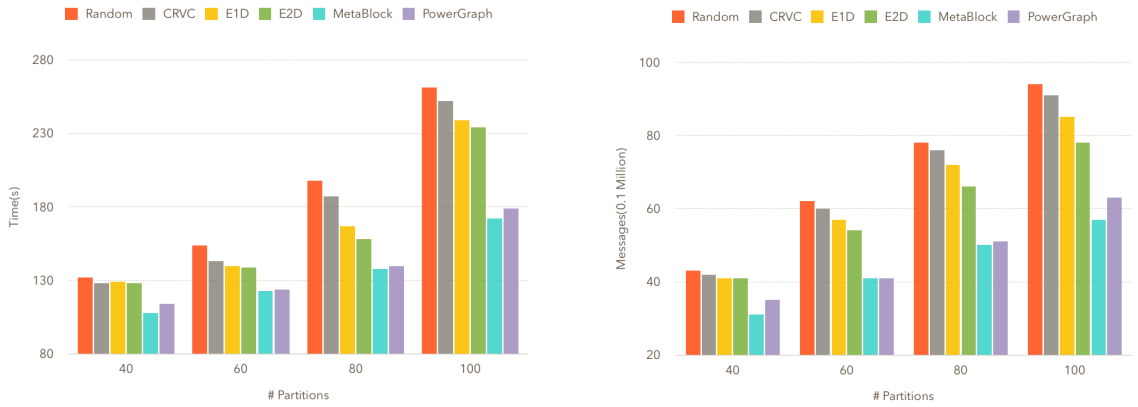


Fig. 4.9 Runtime and communication cost for the independent-label random walks on Twitter graph

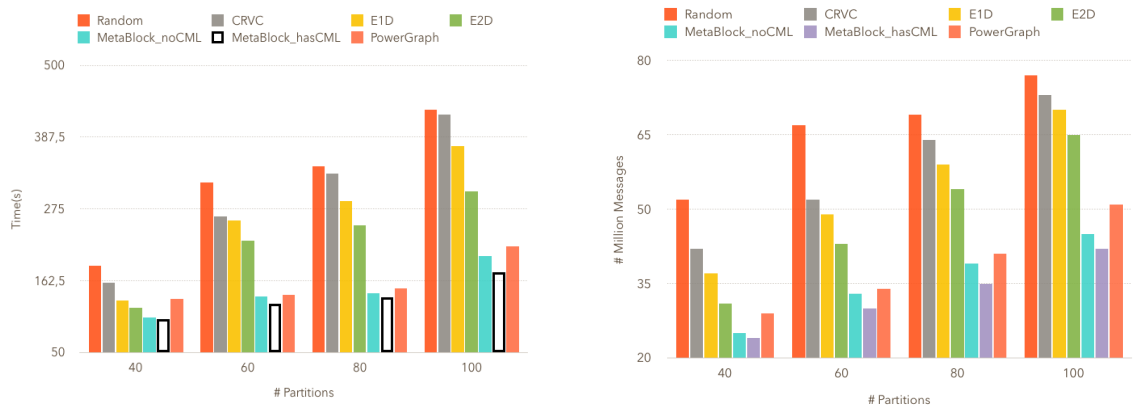


Fig. 4.10 Runtime and communication cost for the correlated-label random walks on Twitter graph

Chapter 5

Conclusions

In this thesis, I have described the details of our proposals for graph partitioning which permit to achieve significant gains when performing random walk-based algorithms on very large graph. Based on the observation of the existence of a *Local Access Pattern* for random walks in real large graphs we propose edge allocation strategies for unlabeled graphs in Chapter 3 and for labeled graphs in Chapter 4. This chapter summarizes the main contributions of the thesis and introduces several perspectives.

5.1 Summary

Our work was motivated by proposing a solution for the fundamental problem of *Graph Partitioning* which is crucial in both theoretical studies and in real-life applications. Finding the best partitioning over a graph data structure is a well-known NP-hard problem, and the fast development of Internet, but also other data-intensive disciplines, have raised a large number of new challenges. For instance, to process these very large graphs, modern distributed graph processing systems (*Spark-based GraphX*, *GraphLab*, *Neo4j*, etc) are frequently used. But to achieve good performances, workload must be balanced. Moreover, in most real scenarios the speed of main memory or even of disk is faster than the communication between machines, especially with cloud computing when the network is shared by multiple systems. This situation will make the heavy inter-machine communication a performance bottleneck to applications deployed on them.

First we aim at efficiently partition the graphs with basic structural properties, *i.e.*, with a single kind of relationship between vertices, so represented by a unlabeled graph, with requirements to both balanced workload and optimised communication cost. To deal with workload problem, we introduced a novel approach, *Vertex-Cut* based on PowerGraph abstraction[44], which has been proved to be effective for distributed process graphs in a

computation-balanced way. In particular, it is based on the observation that many real-world graphs, such as social network, have very skew vertex degree distributions,*e.g. power-law* or *long-tail* distributions. We propose to split a graph by its edges, instead of vertices, which avoid too have to many vertices of high degree co-located on different machines. As in many graph mining tasks that are deployed on modern platforms, the calculation that each vertex has to performed relies on results computed at previous steps by its neighbors, so connected by an edge. Consequently the workload of this kind of application over graphs can be measured by the number of their edges.

Existing works on this topic can be classified into two kinds of approaches: the random/hash methods and greedy algorithms for edges allocation among partitions. The former kind of approach relies on a random way or similar hash functions, whose input are edge's id(s), to perform the edge partitioning, which is easy to be deployed on existing system and can complete quickly the partitioning. The latter kind of approaches proposes to allocate edges in turn to partitions via a greedy mechanism. This method provides better partitions than the former, w.r.t communication costs, but requires more pre-processing time on partitioning and possibly reaches local convergence results. Both of them perform well on workload balancing but do not reduce the communication cost in a satisfactory way. For this reason, we built a workload-balanced, block-based edge partition method that can significantly decrease the communication overhead, compared to other proposals. Motivated by the existence of Local Access Pattern(LAP) existing in many graph computing algorithms, we build blocks, tight small subsets of edges which are based on the locality properties of the underlying. With this technique, we can obtain benefit for performing algorithms which respect LAP. These blocks are then merged into partitions as expected, based on some rules, *e.g.* their sizes or other strategies such as the meta-block we have introduced. In addition, our method can also be applied to cope with evolving graphs, whose structures could vary with time, for instance the friend/unfriend action in social network.
Nevertheless, there are two main shortcomings in this approach, 1) the partitioning phase is time consuming and 2) the graphical model we employed is homogeneous which means that vertices are of same type as well as the edges. The real world graphs are produced with much more complicated structural properties, unfortunately. For the first problem, the user has to make a trade-off between fast deployment and long-term benefit to his graph application. For instance, the random method could be a good choice if user intends to build a distributed graph storage very quickly and use it only a few times, otherwise our method would be a better option because of its sustainable gain on graph computation. Our multi-layer graph partitioning approach aims at facing the second problem by considering different labels on edges.

Our intuition is that people are connected in social network because they share same interest. Based on this intuition, numerous graph mining algorithms have been proposed in literature, which consider vertices/edges with specific properties, w.r.t the application context. For this reason, we extend our work to support Multi-Layer Graphs, in which the edges have different labels or types. The basic idea is that during graph partitioning we will not only explore its local structure but also study the distribution of labels in partitions(blocks). More precisely, we introduced several metrics, such as the Blocks Profile Matrix and Label Score, in partitioning process to ensure that the edges allocated in a block shared the same, or correlated, labels. With our approach we successfully improved the performance of algorithms on labeled-graphs.

While we get promising results with our approach, we identify different tracks of improvement as future work.

5.2 Future Work

This thesis illustrates how we handle the scalable graph computation problem via edge partition approach, for homogeneous (unlabeled) and multi-layer graphs respectively. We especially focus on one fundamental graph operation, random walks, and propose a partitioning strategy which takes benefit from the LAP feature existing in many graph computations. I introduce in the following several possible extensions to our current work.

5.2.1 More Complicated Graph Structure

Our study on the multi-layer graph model can be extended to consider more heterogeneous graph models, *e.g.* graphs where vertices also have different types or attributes/labels. This work could be significant to large and complex networks, *e.g.* IoT(Internet of Things) and Semantic Web, for pursuing efficient processing w.r.t specific graph algorithms, in a distributed computing context. Generally speaking, the more "complicated" the graph structure is, the more opportunities we have for improving its execution performance.

5.2.2 Parameter Training

In the experiments for our block-based graph partitioning method, we observed that, within a certain scope, the more blocks we initially produce, the best partitions we can build at last. Thus it would be interesting and relevant to study how to set this parameter, the number of blocks. Similarly, the range of "locality" exploration in block construction and graph mining

applications also need further investigation to determine their values which provide the best performances. Generally speaking, most parameter values are strongly connected to the structural properties of underlying graphs and the computational cost during partitioning. For this reason, I propose to employ a graph sketch method to get the basic profiles and characteristics of given graphs before partitioning. This approach should be helpful to determine approximations of the optimal values (or variation trends) for the parameters.

5.2.3 Self-Adaptive Block Allocation

We build edge partitions via merging blocks obtained in pre-processing. Consequently the merging/allocation procedure of blocks is crucial to the final quality of partitioning, w.r.t the applications deployed on it. For this reason, I intend to develop a self-adaptive means to allocate blocks for partitions building by considering the specific characteristics of the application and the system requirements to achieve more appropriate partitions. For example, with the wide use of Cloud-Computing techniques, more and more systems and applications are deployed on cloud, which is in practice a hybrid computing environment. Thus we can not simply suppose that computing nodes have unified capacity and stable communication. For this reason, a self-adaptive block assignment could be more efficient to reach good global performance, in both workload balance and communication cost.

5.2.4 Evolving Graphs

The dynamicity of real life graph of many applications is a big challenge for maintaining a partitioning which ensures workload balancing and low communication costs when performing graph processing. What we have mentioned in Chapter 3.2, is a straightforward strategy based on our blocked edge partition method, however we still need to figure out a more comprehensive solution to solve this problem in a general case. For instance, we should have a formal lower/upper-bound to measure the changes on given graph, via theoretical study and analysis, to present some heuristics to decide when, where and how the updates need to be conducted.

5.2.5 Complexity Analysis

To the best of my knowledge, the complexity analysis of distributed graph processing has not been dealt correctly. For instance, the workload measurement of computation in distributed context is discussed in Gemini [129] as a simple linear combination of vertex numbers ($|V_i|$) and amount of work on edges ($|E_i^D|$). However, due to the variety of graph

applications and underlying data stores, in current work they still prefer to characterize their communication and/or computation overhead in an simple and uniform model, which ignores several peculiarities in the given applications. Therefore I plan to design a more powerful model with a more precise complexity analysis in distributed graph computation context by considering the factors introduced in this thesis.

References

[met] Systems biology: Looking beyond the genome. http://www.biokemi.org/biozoom/issues/514/articles/2285.

[2] Abdelbary, H. A., ElKorany, A. M., and Bahgat, R. (2014). Utilizing deep learning for content-based community detection. In *2014 Science and Information Conference*, pages 777–784.

[3] Ahmed, N. K., Duffield, N. G., Neville, J., and Kompella, R. R. (2014). Graph sample and hold: A framework for big-graph analytics. *CoRR*, abs/1403.3909.

[4] Alber, J. and Fiala, J. (2004). Geometric separation and exact solutions for the parameterized independent set problem on disk graphs. *J. Algorithms*, 52(2):134–151.

[5] Alistarh, D., Iglesias, J., and Vojnovic, M. (2015). Streaming min-max hypergraph partitioning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pages 1900–1908, Cambridge, MA, USA. MIT Press.

[6] Amjad, T., Ding, Y., Daud, A., Xu, J., and Malic, V. (2015). Topic-based heterogeneous rank. *Scientometrics*, 104(1):313–334.

[7] Andersen, R., Chung, F. R. K., and Lang, K. J. (2006). Local Graph Partitioning using PageRank Vectors. In *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 475–486.

[8] Andreev, K. and Räcke, H. (2004). Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA. ACM.

[Apache] Apache. Giraph.

[10] Aridhi, S., Montresor, A., and Velegrakis, Y. (2017). BLADYG: A Graph Processing Framework for Large Dynamic Graphs. *Big Data Research*.

[11] Bahmani, B., Chakrabarti, K., and Xin, D. (2011). Fast Personalized PageRank on MapReduce. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 973–984.

[12] Bahmani, B., Chowdhury, A., and Goel, A. (2010). Fast Incremental and Personalized PageRank. *Proc. of the VLDB Endowment (PVLDB)*, 4(3):173–184.

[13] Bahmani, B., Kumar, R., Mahdian, M., and Upfal, E. (2012). Pagerank on an evolving graph. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 24–32, New York, NY, USA. ACM.

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

[15] Barceló, P., Libkin, L., and Reutter, J. L. (2011). Querying graph patterns. In *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, pages 199–210, New York, NY, USA. ACM.

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

[17] Bertrand, A. and Moonen, M. (2013). Distributed computation of the fiedler vector with application to topology inference in ad hoc networks. *Signal Process.*, 93(5):1106–1117.

[18] Bourse, F., Lelarge, M., and Vojnovic, M. (2014). Balanced Graph Edge Partition. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1456–1465.

[19] Bui, T., Heigham, C., Jones, C., and Leighton, T. (1989). Improving the performance of the kernighan-lin and simulated annealing graph bisection algorithms. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, DAC '89, pages 775–778, New York, NY, USA. ACM.

[20] Cai, D., Shao, Z., He, X., Yan, X., and Han, J. (2005). Mining hidden community in heterogeneous social networks. In *Proceedings of the 3rd International Workshop on Link Discovery*, LinkKDD '05, pages 58–65, New York, NY, USA. ACM.

[21] Cardillo, A., Gómez-Gardeñes, J., Zanin, M., Romance, M., Papo, D., Pozo, F. d., and Boccaletti, S. (2013). Emergence of network features from multiplexity. *Scientific Reports*, 3:1344 EP –.

[22] Çatalyürek, U. V., Aykanat, C., and Uçar, B. (2010). On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Sci. Comput.*, 32(2):656–683.

[23] Chan, T. F., P. Ciarlet, J., and Szeto, W. K. (1997). On the optimality of the median cut spectral bisection graph partitioning method. *SIAM Journal on Scientific Computing*, 18(3):943–948.

[24] Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., and Raghavan, P. (2009). On Compressing Social Networks. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 219–228.

[25] Cho, E., Myers, S. A., and Leskovec, J. (2011). Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1082–1090, New York, NY, USA. ACM.

[26] Constantin, C., Dahimene, R., Grossetti, Q., and Du Mouza, C. (2016). Finding Users of Interest in Micro-blogging Systems. In *International Conference on Extending Database Technology, EDBT*, Bordeaux, France.

[27] Dahimene, R., Constantin, C., and du Mouza, C. (2014). RecLand: A Recommender System for Social Networks. In *Proc. of the ACM International Conference on Conference on Information and Knowledge Management (CIKM)*, pages 2063–2065.

[28] De Domenico, M., Lima, A., Mougel, P., and Musolesi, M. (2013). The anatomy of a scientific rumor. 3:2980 EP –.

[29] De Domenico, M., Porter, M. A., and Arenas, A. (2015). Muxviz: a tool for multilayer analysis and visualization of networks. *Journal of Complex Networks*, 3(2):159–176.

[30] Dong, X., Frossard, P., Vandergheynst, P., and Nefedov, N. (2012a). Clustering with multi-layer graphs: A spectral perspective. *IEEE Transactions on Signal Processing*, 60(11):5820–5831.

[31] Dong, X., Frossard, P., Vandergheynst, P., and Nefedov, N. (2012b). Clustering with multi-layer graphs: A spectral perspective. *Trans. Sig. Proc.*, 60(11):5820–5831.

[32] Dong, X., Frossard, P., Vandergheynst, P., and Nefedov, N. (2014). Clustering on multi-layer graphs via subspace analysis on grassmann manifolds. *IEEE Transactions on Signal Processing*, 62(4):905–918.

[33] Eppstein, D., Miller, G. L., and Teng, S.-H. (1993). A deterministic linear time algorithm for geometric separators and its applications. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, pages 99–108, New York, NY, USA. ACM.

[34] Faloutsos, M., Faloutsos, P., and Faloutsos, C. (1999). On power-law relationships of the internet topology. *SIGCOMM Comput. Commun. Rev.*, 29(4):251–262.

[35] Fani, H., Zarrinkalam, F., Bagheri, E., and Du, W. (2016). *Time-Sensitive Topic-Based Communities on Twitter*, pages 192–204. Springer International Publishing, Cham.

[36] Feige, U. and Krauthgamer, R. (2000). A polylogarithmic approximation of the minimum bisection. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 105–115.

[37] Feige, U., Krauthgamer, R., and Nissim, K. (2000). Approximating the minimum bisection size (extended abstract). In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 530–536, New York, NY, USA. ACM.

[38] Feldmann, A. E. (2012). *Fast Balanced Partitioning Is Hard Even on Grids and Trees*, pages 372–382. Springer Berlin Heidelberg, Berlin, Heidelberg.

[39] Fiduccia, C. M. and Mattheyses, R. M. (1982). A linear-time heuristic for improving network partitions. In *19th Design Automation Conference*, pages 175–181.

[40] Fiedler, M. (1975). A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4):619–633.

[41] Fiedler, M. (1989). Laplacian of graphs and algebraic connectivity. *Banach Center Publications*, 25(1):57–70.

[42] Fogaras, D. and Racz, B. (2004). Towards Scaling Fully Personalized PageRank. In *WAW*, pages 105–117.

[43] Gleich, D. F. and Seshadhri, C. (2012). Vertex Neighborhoods, Low Conductance Cuts, and Good Seeds for Local Community Methods. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 597–605.

[44] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012a). PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30.

[45] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012b). PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *OSDI*, pages 17–30.

[46] Gori, M. and Pucci, A. (2007). Itemrank: A random-walk based scoring algorithm for recommender engines. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 2766–2771, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Grandjean] Grandjean, M. The digital humanities network on twitter: Following or being followed? http://www.martingrandjean.ch/digital-humanities-network-twitter-following/.

[48] Grossetti, Q., Constantin, C., du Mouza, C., and Travers, N. (2018). An Homophily-based Approach for Fast Post Recommendation in Microblogging Systems. In *International Conference on Extending Database Technology, EDBT*, Wien, Austria.

[49] Gubichev, A., Bedathur, S., Seufert, S., and Weikum, G. (2010). Fast and accurate estimation of shortest paths in large graphs. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 499–508, New York, NY, USA. ACM.

[50] Han, J. and Wen, J.-R. (2013). Mining frequent neighborhood patterns in a large labeled graph. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 259–268, New York, NY, USA. ACM.

[51] Heith, M. T. and Raghavan, P. (1992). A cartesian parallel nested dissection algorithm. Technical report, Champaign, IL, USA.

[52] Hendrickson, B. and Leland, R. (1995). A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA. ACM.

[53] Jamali, M. and Ester, M. (2009). *TrustWalker*: a Random Walk Model for Combining Trust-based and Item-based Recommendation. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 397–406.

[54] Janke, D., Staab, S., and Thimm, M. (2017). On data placement strategies in distributed rdf stores. In *Proceedings of The International Workshop on Semantic Big Data*, SBD '17, pages 1:1–1:6, New York, NY, USA. ACM.

[55] Jeh, G. and Widom, J. (2002). Simrank: A measure of structural-context similarity. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 538–543, New York, NY, USA. ACM.

[56] Jeh, G. and Widom, J. (2003). Scaling Personalized Web Search. In *Proc. of the International World Wide Web Conference (WWW)*, pages 271–279.

[57] Kang, U., Tong, H., Sun, J., Lin, C.-Y., and Faloutsos, C. (2012). Gbase: An efficient analysis platform for large graphs. *The VLDB Journal*, 21(5):637–650.

[58] Kannan, R., Vempala, S., and Vetta, A. (2004). On clusterings: Good, bad and spectral. *J. ACM*, 51(3):497–515.

[59] Karypis, G. and Kumar, V. (1995). Multilevel graph partitioning schemes. In *Proc. 24th Intern. Conf. Par. Proc., III*, pages 113–122. CRC Press.

[60] Karypis, G. and Kumar, V. (1998a). A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Scientific Computing*, 20(1):359–392.

[61] Karypis, G. and Kumar, V. (1998b). Multilevel algorithms for multi-constraint graph partitioning. In *Supercomputing, 1998.SC98. IEEE/ACM Conference on*, pages 28–28.

[62] Kernighan, B. W. and Lin, S. (1970a). An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291–307.

[63] Kernighan, B. W. and Lin, S. (1970b). An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307.

[64] Kundu, S. and Misra, J. (1977). A linear tree partitioning algorithm. *SIAM Journal on Computing*, 6(1):151–154.

[65] Kyrola, A. (2013). DrunkardMob: Billions of Random Walks on Just a PC. In *RecSys*, pages 257–264.

[66] Lancichinetti, A., Fortunato, S., and Radicchi, F. (2008). Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4):046110.

[67] Lao, N., Mitchell, T. M., and Cohen, W. W. (2011). Random Walk Inference and Learning in A Large Scale Knowledge Base. In *EMNLP*, pages 529–539.

[68] Lee, K. and Liu, L. (2013). Efficient data partitioning model for heterogeneous graphs in the cloud. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12.

[69] Leskovec, J., Lang, K. J., Dasgupta, A., and Mahoney, M. W. (2008). Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *CoRR*, abs/0810.1355.

[70] Li, L., Geda, R., Hayes, A. B., Chen, Y., Chaudhari, P., Zhang, E. Z., and Szegedy, M. (2017a). A simple yet effective balanced edge partition model for parallel computing. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1):14:1–14:21.

[71] Li, R. H., Yu, J. X., and Mao, R. (2014). Efficient core maintenance in large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering*, 26(10):2453–2465.

[72] Li, Y., Constantin, C., and du Mouza, C. (2017b). Sgvcut: A vertex-cut partitioning tool for random walks-based computations over social network graphs. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, pages 39:1–39:4, New York, NY, USA. ACM.

[73] Li, Y., Constantin, C., and du Mouza, C. (2017c). Un partitionnement d'arêtes à base de blocs pour les algorithmes de marches aléatoires dans les grands graphes sociaux. *Ingénierie des Systèmes d'Information*, 22(3):89–113.

[74] Li, Y., Constantin, C., and Mouza, C. (2016). A block-based edge partitioning for random walks algorithms over large social graphs. In *Proceedings of the 17th International Conference on Web Information Systems Engineering - Volume 10042*, WISE 2016, pages 275–289, New York, NY, USA. Springer-Verlag New York, Inc.

[75] Lim, Y., Kang, U., and Faloutsos, C. (2014). Slashburn: Graph compression and mining beyond caveman communities. 26:3077–3089.

[76] Lim, Y., Lee, W.-J., Choi, H.-J., and Kang, U. (2017). Mtp: discovering high quality partitions in real world graphs. *World Wide Web*, 20(3):491–514.

[77] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., and Hellerstein, J. M. (2012a). Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727.

[78] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. (2010). Graphlab: A new framework for parallel machine learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, UAI'10, pages 340–349, Arlington, Virginia, United States. AUAI Press.

[79] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. (2012b). Distributed GraphLab: A Framework for Machine Learning in the Cloud. *Proc. of the VLDB Endowment (PVLDB)*, 5(8):716–727.

[80] Malewicz, G., Austern, M. H., Bik, A. J. C., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2009). Pregel: a System for Large-scale Graph Processing. In *Proc. of the ACM Symposium on Principles of Distributed Computing (PODC)*, page 6.

[81] Margo, D. and Seltzer, M. (2015). A scalable distributed graph partitioner. *Proc. VLDB Endow.*, 8(12):1478–1489.

[82] Meyerhenke, H., Monien, B., and Sauerwald, T. (2009). A new diffusion-based multilevel algorithm for computing graph partitions. *J. Parallel Distrib. Comput.*, 69(9):750–761.

[83] Miller, G. L., Teng, S.-H., and Vavasis, S. A. (1991). A unified geometric approach to graph separators. In *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 538–547, Washington, DC, USA. IEEE Computer Society.

[84] Muchnik, L., Pei, S., Parra, L. C., Reis, S. D. S., Andrade Jr, J., Havlin, S., and Makse, H. A. (2013). Origins of power-law degree distribution in the heterogeneity of human activity in social networks. 3:1783 EP –.

[85] Newman, M., Barabasi, A.-L., and Watts, D. J. (2006a). *The Structure and Dynamics of Networks: (Princeton Studies in Complexity)*. Princeton University Press.

[86] Newman, M., Barabasi, A.-L., and Watts, D. J. (2006b). *The Structure and Dynamics of Networks: (Princeton Studies in Complexity)*. Princeton University Press.

[87] Pan, X., Papailiopoulos, D., Oymak, S., Recht, B., Ramchandran, K., and Jordan, M. I. (2015). Parallel correlation clustering on big graphs. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pages 82–90, Cambridge, MA, USA. MIT Press.

[88] Pattabiraman, B., Patwary, M. M. A., Gebremedhin, A. H., Liao, W.-k., and Choudhary, A. (2013). *Fast Algorithms for the Maximum Clique Problem on Massive Sparse Graphs*, pages 156–169. Springer International Publishing, Cham.

[89] Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 701–710, New York, NY, USA. ACM.

[90] Petroni, F., Querzoni, L., Daudjee, K., Kamali, S., and Iacoboni, G. (2015). Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 243–252, New York, NY, USA. ACM.

[91] Pothen, A., Simon, H. D., and Liou, K.-P. (1990). Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452.

[92] Roy, A., Bindschaedler, L., Malicevic, J., and Zwaenepoel, W. (2015). Chaos: Scale-out Graph Processing from Secondary Storage. In *Proc. of the Symposium on Operating Systems Principles (SOSP)*, pages 410–424.

[93] Saad, Y. (1994). Sparskit: a basic tool kit for sparse matrix computations - version 2.

[94] Sajjad, H. P., Payberah, A. H., Rahimian, F., Vlassov, V., and Haridi, S. (2016). Boosting vertex-cut partitioning for streaming graphs. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 1–8.

[95] Sakouhi, C., Aridhi, S., Guerrieri, A., Sassi, S., and Montresor, A. (2016). Dynamicdfep: A distributed edge partitioning approach for large dynamic graphs. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, IDEAS '16, pages 142–147, New York, NY, USA. ACM.

[96] Salihoglu, S. and Widom, J. (2013). GPS: a Graph Processing System. In *Proc. of the Conference on Scientific and Statistical Database Management (SSDBM)*, pages 22:1–22:12.

[97] Saran, H. and Vazirani, V. V. (1995). Finding k cuts within twice the optimal. *SIAM J. Comput.*, 24(1):101–108.

[98] Sarkar, P. and Moore, A. W. (2010). Fast Nearest-neighbor Search in Disk-resident Graphs. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 513–522.

[99] Sarma, A. D., Gollapudi, S., and Panigrahy, R. (2008). Estimating PageRank on Graph Streams. In *Proc. of the ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 69–78.

[100] Schaeffer, S. E. (2007). Survey: Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64.

[101] Shi, C., Li, Y., Zhang, J., Sun, Y., and Yu, P. S. (2017). A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering*, 29(1):17–37.

[102] Simon, H. (1991). Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135 – 148. Parallel Methods on Large-scale Structural Analysis and Physics Applications.

[103] Sun, J., Vandierendonck, H., and Nikolopoulos, D. S. (2017). Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 16:1–16:10, New York, NY, USA. ACM.

[104] Sun, Y. and Han, J. (2012). *Mining Heterogeneous Information Networks:Principles and Methodologies*.

[105] Sun, Y., Han, J., Zhao, P., Yin, Z., Cheng, H., and Wu, T. (2009). Rankclus: Integrating clustering with ranking for heterogeneous information network analysis. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 565–576, New York, NY, USA. ACM.

[106] Sun, Y., Norick, B., Han, J., Yan, X., Yu, P. S., and Yu, X. (2012). Integrating meta-path selection with user-guided object clustering in heterogeneous information networks. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1348–1356, New York, NY, USA. ACM.

[107] Takac, L. and Zabovsky, M. (2012). Data Analysis in Public Social Networks. *Present Day Trends of Innovations*, pages 1–6.

[108] Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. (2015). Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, pages 1067–1077, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee.

[109] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. (2013). From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204.

[110] Tsourakakis, C., Gkantsidis, C., Radunovic, B., and Vojnovic, M. (2014). Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 333–342, New York, NY, USA. ACM.

[111] Urschel, J. C. and Zikatanov, L. T. (2014). Spectral bisection of graphs and connectedness. *Linear Algebra and Its Applications*, 449(Complete):1–16.

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

[113] Valiant, L. G. (2008). A Bridging Model for Multi-core Computing. In *Proc. of the Annual European Symposium Algorithms - ESA* , pages 13–28.

[114] Wan, X. and Yang, J. (2008). Multi-document summarization using cluster-based link analysis. In *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '08, pages 299–306, New York, NY, USA. ACM.

[115] Wang, Y., Lin, X., and Zhang, Q. (2013). Towards metric fusion on multi-view data: A cross-view based graph random walk approach. In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 805–810, New York, NY, USA. ACM.

[116] Whang, J. J., Gleich, D. F., and Dhillon, I. S. (2013). Overlapping Community detection Using Seed set Expansion. In *Proc. of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 2099–2108.

[117] Xie, C., Yan, L., Li, W.-J., and Zhang, Z. (2014). Distributed power-law graph computing: Theoretical and empirical analysis. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 1673–1681. Curran Associates, Inc.

[118] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013a). GraphX: a Resilient Distributed Graph System on Spark. In *Proc. of the International SIGMOD Workshop on Graph Data Management Experiences and Systems (GRADES)*, page 2.

[119] Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013b). GraphX: A Resilient Distributed Graph System on Spark. In *GRADES*, pages 2:1–2:6.

[120] Yan, D., Cheng, J., Lu, Y., and Ng, W. (2014). Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow.*, 7(14):1981–1992.

[121] Yan, J.-T. and Hsiao, P.-Y. (1995). A new fuzzy-clustering-based approach for two-way circuit partitioning. In *Proceedings of the 8th International Conference on VLSI Design*, VLSID '95, pages 359–, Washington, DC, USA. IEEE Computer Society.

[122] Yang, J., Chen, L., and Zhang, J. (2015). Fctclus: A fast clustering algorithm for heterogeneous information networks. *PLoS ONE*, 10(6):e0130086.

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

[124] Yang, S., Yan, X., Zong, B., and Khan, A. (2012). Towards Effective Partition Management for Large Graphs. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 517–528.

[125] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28.

[126] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA. USENIX Association.

[127] Zhang, C., Wei, F., Liu, Q., Tang, Z. G., and Li, Z. (2017). Graph edge partitioning via neighborhood heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 605–614, New York, NY, USA. ACM.

[128] Zhou, Y. and Liu, L. (2013). Social influence based clustering of heterogeneous information networks. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 338–346, New York, NY, USA. ACM.

[129] Zhu, X., Chen, W., Zheng, W., and Ma, X. (2016). Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 301–316, GA. USENIX Association.

[130] Zou, L., Chen, L., and Özsu, M. T. (2009). Distance-join: Pattern match query in a large graph database. *Proc. VLDB Endow.*, 2(1):886–897.

[131] Zou, R. and Holder, L. B. (2010). Frequent subgraph mining on a single large graph using sampling techniques. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 171–178, New York, NY, USA. ACM.