A Block-Based Edge Partitioning for Random Walks Algorithms over Large Social Graphs

Yifan Li, Camelia Constantin, Cedric Du Mouza

A Block-Based Edge Partitioning for Random Walks Algorithms over Large Social Graphs

Yifan LI[1,2], Camelia Constantin[1], and Cedric du Mouza[2]

[1] LIP6, University Pierre et Marie Curie
Paris, France
firstname.lastname@lip6.fr
[2] CEDRIC Lab., CNAM
Paris, France
dumouza@cnam.fr

Abstract. Recent results [5, 9, 23] prove that *edge partitioning* approaches (also known as *vertex-cut*) outperform *vertex partitioning* (*edge-cut*) approaches for computations on large and skewed graphs like *social networks*. These vertex-cut approaches generally avoid unbalanced computation due to the *power-law degree distribution* problem. However, these methods, like *evenly random assigning* [23] or *greedy assignment strategy* [9], are generic and do not consider any computation pattern for specific graph algorithm. We propose in this paper a vertex-cut partitioning dedicated to random walks algorithms which takes advantage of graph topological properties. It relies on a blocks approach which captures *local communities*. Our split and merge algorithms allow to achieve load balancing of the workers and to maintain it dynamically. Our experiments illustrate the benefit of our partitioning since it significantly reduce the *communication cost* when performing *random walks*-based algorithms compared with existing approaches.

1 Introduction

Random walks-based algorithms, such as personalized PageRank (PPR) [10] and personalized SALSA [4] 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 [3]. We call this intensive computation Fully Multiple Random Walks(FMRWs).

Graph partitioning is a key area of distributed graph processing research, and plays an increasingly important role in both vertex-centric computation, like in *Pregel* model, and query evaluation. Recent results exhibit that *edge partition(vertex-cut)* turned out to be more efficient [5, 9] than traditional vertex partitioning(edge-cut) for computation on real-world graphs like social networks. As a consequence, several popular graph computation systems based on this approach have emerged, such as PowerGraph (GraphLab2) [9] and GraphX [23]. However their graph partitioning strategies are generic and do not depend on

the algorithms performing the different computations. So they distribute edges evenly over partitions either randomly, *i.e.* a hash function of vertex ids in Giraph [2] and GraphX, or using a greedy or dynamic algorithm like in PowerGraph and GPS [19]. Due to the power-law nature of the Web and social network graphs, this edge allocation may lead to an important workload imbalance between the resources. 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 paper, 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.

We propose in this paper 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. 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. Finally, the experiments show that our partitioning made significant improvements on both communication cost and time overhead.

Contributions

In summary, our contributions are:

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 random methods over large real social graphs.

After the related work introduced in Section 2, Section 3 presents our block building strategy while Section 4 describes our blocks merge and refinement algorithms. Section 5 presents our experimental results and Section 6 concludes and introduces perspectives.

2 Related work

Pregel [16] 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. Pregel is inspired by *Bulk Synchronous Parallel* [21] computation model where computations on a graph consists of several iterations, also called *super-steps*. During a super-step, 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, namely *vertex.compute()* [19] or *vertex.program()* [9], in parallel, using the updated values received in the messages. 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. Due to *Pregel* success, several optimizations have been recently proposed in literature like the function *Master.compute()* [19] to incorporate global computations or *Mirror Vertices* [14] to reduce communication.

Traditional methods from 2-way cut by local search to multi-level approaches, like Kernighan-Lin [12], PageRank Vectors [1] and METIS-based [11] 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, using in-memory computation. However for real world graphs the large size and the power-law distribution lead to an unbalanced load over edge-cut partitions. More recent partitioning proposals in Pregel-like systems, such as Giraph, GPS, Gelly and Chaos[18] shard the graph using an *edge-cut* strategy which also generates unbalancing for power-law graphs, as introduced in [9].

While there exists a large literature and several implementations for vertex-partitioning, few recent works propose edge-partitioning. The two principal ones are GraphX [23] and PowerGraph [9]. 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 [9]. 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, unlike our proposal, GraphX and PowerGraph partitionings can not be updated dynamically with graph evolution.

Our approach also takes advantage from the existence of communities. In [8] 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 [22] 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.

3 Block-Based Graph Partitioning

3.1 Principle

Most existing edge partitioning methods, like random [23] or greedy [9] 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 super-step. Consequently, Vertex Replication Factor(VRF) firstly defined in [9] is often used as a communication measurement. So, given an edge partitioning, the communication cost is generally estimated in Pregel, as

$$cost_{Comm} = O(L \times (VRF \times |V|)) \tag{1}$$

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 [24] 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.

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 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 [8, 22], we propose to compute a set of K blocks by exploring the graph. An edge is allocated to a block based on its connectivity score 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 connectivity score with respect to all blocks. When the exploration step ended, we allocate the edges to the closest block.

3.2 Connectivity score 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 connectivity score measures is to detect cluster in graph (see Section 3.3). 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 [1] 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 [20] which describes how to obtain these partitions by conducting random walks.

For our edge-partitioning approach, we propose here to estimate the *connectivity score* between an edge and a *query* vertex, e.g. the seed in our paper. We adapt the inverse P-distance [10] used for connectivity score computation between two vertices.

Vertex to vertex connectivity score

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

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

$$conn_v(i, j) = \sum_{p \in P_{ij}} S(p) \tag{2}$$

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

According to the idea of inverse P-distance score, we introduce the concept of "reachability" into connectivity score 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, ..., $v_{(k-1)}$ with length k, $S(p)$ can be defined by:

$$S(p) = (1 - \alpha)^k \cdot \prod_{i=0}^{k-1} \frac{1}{outDeg(v_i)} \tag{3}$$

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 connectivity score

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

Definition 1 (edge connectivity score).
The connectivity score $conn_e(a, b)$ from a vertex a to an edge $b = (i, j)$ is:

$$conn_e(a, b) = \theta(conn_v(a, i), conn_v(a, j))$$

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.

3.3 Edges allocation algorithm

Based on our edge connectivity score 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) connectivity score 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 blocks. 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 [22] to detect communities on graph or in [7] 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 [22]), 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) connectivity score to other existing seeds. Our seeds selection algorithm is:

1. first we sort the vertices in ascending 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 connectivity score 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 to 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 because:

– 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 VRF;
– the more blocks we pre-computed, the higher the level of reusability our partitioning will be.

Connectivity score computation

For the second step of our algorithm, we compute first the distance scores of each vertex to all seeds. To perform this connectivity score computation efficiently in our Pregel-like architecture, we proceed to a parallel BFS exploration starting from each seed. Consider a set of seeds $\mathcal{S} = (s_1, s_2, \ldots, s_N)$. We maintain for each vertex ν a connectivity score vector $conn(\nu) = (d_1, d_2, \ldots, d_N)$ where $d_i = conn_v(s_i, \nu)$ is connectivity score 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 depth of 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, 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 [17], we observe that the *radius of block*, *i.e.*, the connectivity score 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 [13] 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 connectivity score vector for each edge in the graph. Consider an edge $e(\nu, \nu')$ and the connectivity score vectors for its vertices $conn(\nu) = (d_1, d_2, \ldots, d_N)$ and $conn(\nu') = (d'_1, d'_2, \ldots, d'_N)$. Based on Definition 1 we compute the edge connectivity score vector $conn(\varepsilon) = (D_1, D_2, \ldots, D_N)$ as:

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

Edges allocation

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

Example 1. We illustrate the *edge allocation* process with the example in Fig. 1. We assume we have already computed the vertex connectivity score vectors for vertices i and j, considering three seeds s1, s2 and s3. Notice that the '*' value

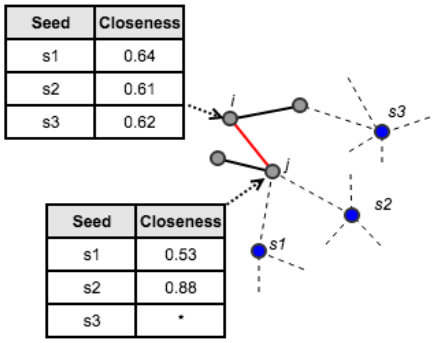

Fig. 1: Example of edge allocation

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 connectivity score vector for $e(i,j)$: $conn(i,j) = (0.64 + 0.53, 0.61 + 0.88, 0.62 + 0.0) = (1.17, 1.49, 0.62)$. 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 connectivity score vectors and for the edge connectivity score vector computation. For instance we can avoid keeping all connectivity score 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 $k \leq |\mathcal{S}|$. Of course the larger k is, the more precise our final result is.

4 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 definition 2.
The first 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. The second part of the definition means the partitioning $\mathcal{A}$ is the one which minimizes the *Vertex Replication Factor*(VRF). The VRF measure adopted for instance in [9] 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. With respect to Definition 2 we can proceed to the final partitioning based on the different blocks we built.

Definition 2 (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 $\mathcal{B}$ and a number of servers P. The* balanced edge partitioning $\mathcal{A}(\mathcal{B}, P)$ *is defined as:*

$$\mathcal{A}(\mathcal{B},P) \in 2^{\mathcal{B}},\ such\ that \begin{cases} \forall i \in [1..P], \eta\frac{|E|}{P} \leq |Edge(p_i)| \leq \lambda\frac{|E|}{P} \\ \\ \forall \mathcal{A}' \in 2^{\mathcal{B}}\ satisfying\ above, \\ \frac{1}{|V|}\sum_{v\in V}|alloc(v,\mathcal{A})| \leq \frac{1}{|V|}\sum_{v\in V}|alloc(v,\mathcal{A}')|, \\ otherwise,\ relax\ \eta\ and(or)\ \lambda\ to\ find\ the\ \mathcal{A}. \end{cases}$$

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

Block split

Since the edges allocation to blocks is only based on a connectivity score criterium some blocks may not fit the maximum size allowed for a partition (second part of Definition 2). 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 2). 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 partitions.

Considering this latter approach, we exhibit the following drawbacks: (1) there is an exponential complexity for finding the best blocks allocation considering this criterion, (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 criterion, to achieve 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 $O(|\mathcal{B}|)$ where $\mathcal{B}$ 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 $\mathcal{B} = \{b_1, \ldots, b_n\}$ of blocks, a set $\mathcal{P} = \{p_1, \ldots, p_m\}$ of partitions
output: each block is allocated to a $p_j \in \mathcal{P}$

```
// 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 connectivity score vectors of the two vertices of the edge if both vertices were already present in the graph to compute its edge connectivity score vector. Then we allocate the edge to the block, and consequently to the partition, with the highest connectivity score score. If one of the vertices is new, we have first to perform the BFS exploration from that vertex and compute its vertex connectivity score 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.

5 Experiments

This section presents experiments on our block-based partitioning strategy. We compare it with existing edge partitioning methods: the hash-based approaches [23] and greedy algorithm [9].

5.1 Setting

Computation are performed using GraphX [23] APIs in Spark [25] (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).

Data Sets. We validate our approach on two datasets: LiveJournal [6] with 4.8M vertices and 68.9M edges, and Pokec [15] with 1.6M vertices and 30.6M edges. These datasets can be downloaded from *SNAP* [3].

Competitors. *Hash Partitioning*. There are four wide used random(hash)-like partitioning methods[4], 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.
- 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.

5.2 Communication

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

[3] https://snap.stanford.edu/data/index.html
[4] see details and implementations at http://spark.apache.org/docs/latest/api/scala/index.html

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 Figure 2. We observe that, as observed in [9], 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.

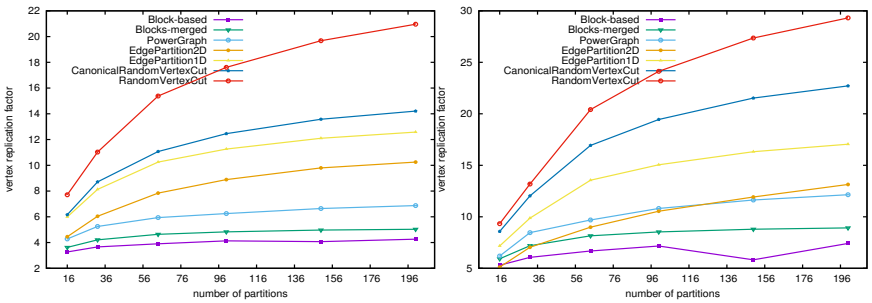

Fig. 2: VRF w.r.t. edge partitioning methods on LiveJournal (left) and Pokec (right)

Number of Messages

VRF is a general criterion 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 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. 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 the farthest, high-replicated, vertices. Similar results are obtained from experiments on Pokec.

Table 1: Messages transmitted in FMRW (LiveJournal)

	Random-Vertex-Cut[23]		Block-based partitioning			
#Partitions	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

5.3 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, we see that our partitioning can save up between from 20 to 65 percent of runtime, compared with other partitionings, for both datasets.

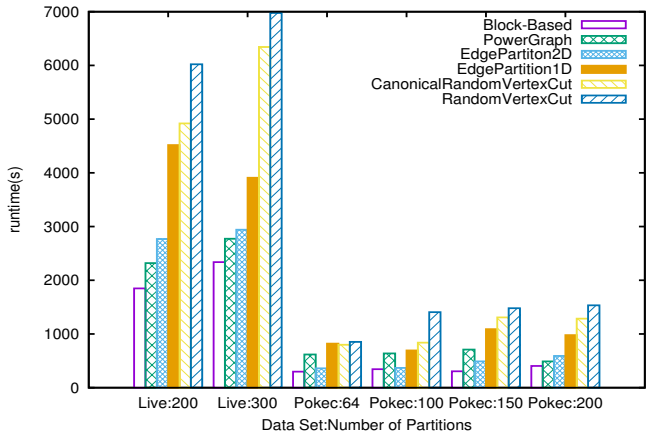

Fig. 3: Runtimes for FMRW with different partitionings for LiverJournal and Pokec

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. Figure 4 depicts results and confirms 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.

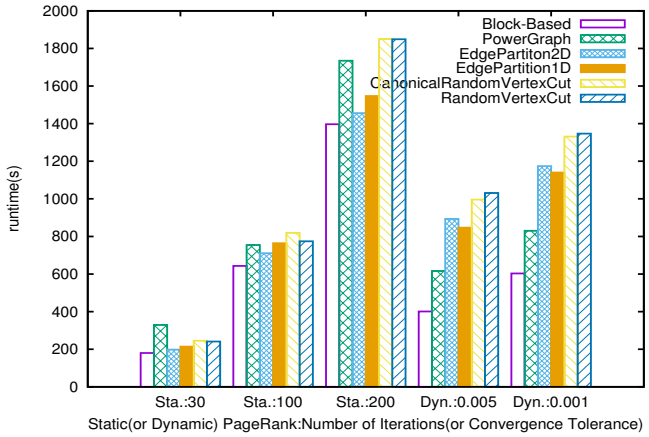

Fig. 4: Runtimes for static and dynamic PageRank for LiverJournal

6 Conclusion and Future Work

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 contexts (see [22, 7]) 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.

References

1. R. Andersen, F. Chung, and K. Lang. Local Graph Partitioning Using PageRank Vectors. In *FOCS*, pages 475–486, 2006.
2. Apache. Giraph. `http://giraph.apache.org`.
3. B. Bahmani, K. Chakrabarti, and D. Xin. Fast Personalized PageRank on MapReduce. In *SIGMOD*, pages 973–984, 2011.
4. B. Bahmani, A. Chowdhury, and A. Goel. Fast Incremental and Personalized PageRank. *Proc. VLDB Endow.*, 4(3):173–184, 2010.
5. F. Bourse, M. Lelarge, and M. Vojnovic. Balanced Graph Edge Partition. In *SIGKDD*, pages 1456–1465, 2014.

6. F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On Compressing Social Networks. In *SIGKDD*, pages 219–228, 2009.
7. R. Dahimene, C. Constantin, and C. du Mouza. RecLand: A Recommender System for Social Networks. In *CIKM*, pages 2063–2065, 2014.
8. D. F. Gleich and C. Seshadhri. Vertex Neighborhoods, Low Conductance Cuts, and Good Seeds for Local Community Methods. In *SIGKDD*, pages 597–605, 2012.
9. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *OSDI*, pages 17–30, 2012.
10. G. Jeh and J. Widom. Scaling Personalized Web Search. In *WWW*, pages 271–279, 2003.
11. G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1998.
12. B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.
13. J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters, 2008.
14. Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *VLDB Endow.*, 5(8):716–727, 2012.
15. M. Z. Lubos Takac. Data Analysis in Public Social Networks. *Present Day Trends of Innovations*, pages 1–6, 2012.
16. G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, and G. Inc. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD*, pages 135–146, 2010.
17. M. Newman, A.-L. Barabasi, and D. J. Watts. *The Structure and Dynamics of Networks: (Princeton Studies in Complexity)*. Princeton University Press, 2006.
18. A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *SOSP*, pages 410–424, 2015.
19. S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, pages 22:1–22:12, 2013.
20. P. Sarkar and A. W. Moore. Fast Nearest-neighbor Search in Disk-resident Graphs. In *SIGKDD*, pages 513–522, 2010.
21. L. G. Valiant. A Bridging Model for Multi-core Computing. *J. Comput. Syst. Sci.*, 77(1):154–166, 2011.
22. J. J. Whang, D. F. Gleich, and I. S. Dhillon. Overlapping Community Detection Using Seed Set Expansion. In *CIKM*, pages 2099–2108, 2013.
23. R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *GRADES*, pages 1–6, 2013.
24. S. Yang, X. Yan, B. Zong, and A. Khan. Towards Effective Partition Management for Large Graphs. In *SIGMOD*, pages 517–528, 2012.
25. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, pages 2–2, 2012.