Comparison of Edge Partitioners for Graph Processing

Hlib Mykhailenko, Fabrice Huet, Giovanni Neglia

Comparison of Edge Partitioners for Graph Processing

Hlib Mykhailenko*[1], Fabrice Huet†[2], and Giovanni Neglia‡[1]

[1]Université Côte d'Azur, Inria
[2]Université Côte d'Azur, CNRS, I3S

November 23, 2016

Abstract

Deploying graph on a cluster requires its partitioning into a number of subgraphs, and assigning them to different machines. Two partitioning approaches have been proposed: vertex partitioning and edge partitioning. In the edge partitioning approach edges are allocated to partitions. Recent studies show that, for power-law graphs, edge partitioning is more effective than vertex partitioning. In this paper we provide an overview of existing edge partitioning algorithms. However, based only on published work, we cannot draw a clear conclusion about the relative performances of these partitioners. For this reason, we compare all the edge partitioners currently available for GraphX. Our preliminary results suggest that `Hybrid-Cut` partitioner provides the best performance.

Keywords **- vertex-cut, edge partitioning, GraphX, Spark**

1 Introduction

There are many frameworks for distributed graph processing, such as GraphX [20], Pregel [16], PowerGraph [9], GraphLab [15], GraphBuilder [11], Giraph [3], etc. These frameworks require the graph to be divided in a number of different partitions, each assigned to a different machine of the computational cluster. Different partitioning algorithms are used to this purpose and in the rest of this paper we refer to them as partitioners.

A common approach to partition a graph is to assign a subset of vertices to each partition, and then as a consequences an edge may be *cut* when its vertices

*hlib.mykhailenko@inria.fr
†fabrice.huet@unice.fr
‡giovanni.neglia@inria.fr

belong to different partitions. This approach is called vertex partitioning or also edge-cut partitioning.

Recently edge partitioning approach (also called vertex-cut partitioning) was proposed in [9]. In this case the partitioner assigns edges to partitions, and then a vertex may be cut into several pieces (from 2 till N pieces, where N is the number of partitions), where the number of vertex pieces equals the number of different partitions its edges are assigned. The presence of cut vertices obliges the different partitions to exchange periodically information during the computation to keep aligned the values for the vertices cut. The recent study [5] advocates that edge partitioning is better for power-law graphs (most of the real-world graphs are power-law graphs). Several graph processing frameworks rely on edge partitioning, such as GraphX [20], PowerGraph [9], and GraphBuilder [11]. However, there are very few studies dedicated to edge partitioners.

The first contribution of our paper is to provide an overview of all the edge partitioners we are aware of. Beside describing the specific algorithms proposed, we focus on existing comparisons of their relative performance. It is not easy to draw conclusions about which are the best partitioners, because research papers often consider a limited subset of partitioners, different performance metrics and different computational frameworks. For this reason a second contribution of this paper is a preliminary evaluation of all the edge partitioners currently implemented for GraphX, one of the most promising graph processing frameworks. Our current results suggest that `Hybrid-Cut` is probably the best choice, achieving significant reduction of the execution time with limited time required to partition the graph.

The paper is organized as follows. In Section 2 we present the notation used across the paper and we describe partitioning metrics. Section 3 presents all existing edge partitioning algorithms and is followed by a discussion in Section 4 about what can be concluded from the literature. Our experiments with GraphX partitioners are described in Section 5. Finally, Section 6 presents our conclusions.

2 Metrics

Let us denote the input directed graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of edges. An edge partitioner splits the set of edges into N disjoint subsets, $E_1, E_2, \ldots, E_N$. With some abuse of terminology, we can say that a vertex belongs to the partition E_i if it is the source or the destination of an edge in E_i. Using this convention, an edge can only belong to one partition, but a vertex is at least in one partition and at most in N partitions. Let $V(E_i)$ denote the set of vertices that belong to partition E_i. The vertices that appear in more than one partition are called frontier vertices. Each frontier vertex has thus been cut at least once. Let $F(E_i)$ denote the set of frontier vertices that are inside partition E_i and $\bar{F}(E_i)$ the set of vertices in partition E_i that are not cut. Hence we have $\bar{F}(E_i) = V(E_i) - F(E_i)$. $\mathcal{N}_{in}(v)$ is the set of source vertices for the edges incoming to vertex v.

There are different metrics which can help measure the quality of a partitioning. The most straightforward ones are execution metrics like the *partitioning time*, i.e. the time to partition the graph, the number of *rounds* performed by the partitioner to converge (for iterative partitioners), the *execution time* of a particular graph algorithm once the graph is partitioned, the *network communication* overhead, i.e. the amount of messages or bytes exchanged during the partitioning or graph algorithm execution. A limitation of execution metrics is that they are tightly coupled to specific applications and execution environments, making them unsuitable for general comparisons. Moreover, they can be costly to evaluate or measure.

For this reason, specific partitioning metrics have been proposed. These metrics try to evaluate the quality of the partition produced, assuming that certain characteristics of the partition will affect the final execution metrics. In the remaining of this section we list various partitioning metrics that are often used in the literature.

2.1 Partitioning metrics

Partitioning metrics can be split into two groups. In the first group there are the metrics that quantify how homogeneous the partitions' sizes are. The underlying idea is that if one partition is much larger that the others, the computational load on the corresponding machine is higher and then this machine can slow down the whole execution. The metrics in the second group quantify how much overlap there is among the different partitions, i.e. how many vertices appear in multiple partitions. This overlap is a reasonable proxy for the amount of inter-machine communication that will be required to merge the results of the local computations. The first two metrics below belong to the first group and we call them *balance metrics*, all the other ones are *communication metrics*.

2.1.1 Balance

It is measured as the ratio of the maximum number of edges in a partition to the average number of edges across all the partitions.

$$\text{Balance} = \frac{\max_{i=1,\dots N} |E_i|}{|\mathcal{E}|/N}.$$

2.1.2 Standard deviation of partition size (STD)

It is the normalized standard deviation of the number of edges in each partition.

$$\text{STD} = \sqrt{\sum_{i=1}^{N} \left(\frac{|E_i|}{|\mathcal{E}|/N} - 1 \right)^2 \frac{1}{N}}$$

2.1.3 Replication factor

It is the ratio of the number of vertices in all the partitions to the number of vertices in the original graph. It measures the overhead, in terms of vertices, induced by the partitioning.

$$\text{Replication Factor} = \sum_{i=1}^{N} |V(E_i)| \frac{1}{|\mathcal{V}|}$$

2.1.4 Communication cost

It is defined as the total number of frontier vertices.

$$\text{Communication cost} = \sum_{i=1}^{N} |F(E_i)|$$

2.1.5 Vertex-cut

This metric measures how many times vertices were cut. For example, if a vertex is cut in 5 pieces, then its contribution to this metric is 4.

$$\text{Vertex-cut} = \sum_{i=1}^{N} F(E_i) - |V| + \sum_{i=1}^{N} \bar{F}(E_i)$$

2.1.6 Normalized vertex-cut

It is a ratio of the *vertex-cut* metric of the partitioned graph to the expected *vertex-cut* of a randomly partitioned graph.

2.1.7 Expansion

Originally *expansion* was introduced in [12, 8] in the context of vertex partitioning. In [14] *expansion* was adapted to edge partitioning approach. It measures the relative contribution of a partition to the number of frontier vertices.

$$\text{Expansion} = \max_{i=1,\dots N} \frac{|F(E_i)|}{|V(E_i)|}$$

2.1.8 Modularity

As *expansion*, *modularity* was proposed in [12, 8] and later adapted to edge partitioning approach in [14] as follows:[1]

$$\text{Modularity } = \sum_{i=1}^{N} \left(\frac{|V(E_i)|}{|\mathcal{V}|} - \left(\sum_{j \neq i} \frac{|F(E_i) \cap F(E_j)|}{|\mathcal{V}|} \right)^2 \right)$$

Intuitively, the higher the modularity is, the smaller is the fraction of vertices shared among different partitions.

[1] Both for expansion and modularity could also be defined normalized to the number of edges rather than to the number of vertices.

3 Algorithms

In this section we will first provide a classification of the partitioners. We already mentioned the main distinction between **edge partitioning** and **vertex partitioning**, but the focus of this paper is on the former type.

Then, we can classify partitions by the amount of information they require; they can be **online** or **offline** [11]. **Online** partitioners decide where to assign an edge on the basis of local information (e.g. about its vertices, and their degrees). For this reason they can be easily distributed. On the contrary, in offline partitioners (like METIS [13]) the choice depends in general on the whole graph and multiple iterations may be needed to produce the final partitioning.

Finally, some partitioners can refine the partitioning during the execution of a graph processing algorithm (e.g. partitioners in Giraph [3]). For example, after several iterations of the `PageRank` algorithm, the graph can be re-partitioned, based on the current execution time of the `PageRank` algorithm itself.

Most of the non-iterative algorithms have $\mathcal{O}(|E|)$ time complexity, and iterative algorithms usually have $\mathcal{O}(k|E|)$ time complexity, where k is the number of iterations of the partitioner.

In the rest of the section we introduce 17 partitioners. For them we propose the following classification based on their approach to partition. Table 2 provides a list of references where the following algorithms were first described or implemented.

3.1 Random assignment

Random assignment approach includes partitioners that randomly assign edges using a hash function based on some value of the edge or of its vertices. Sometimes, the input value of the hash function is not specified (e.g. [18]). Because of the law of large numbers, the random partitions will have similar sizes and then these partitioners achieve good balance.

3.1.1 RandomVertexCut

It assigns each edge using a hash function based on the pair of values source vertex id and destination vertex id.

3.1.2 CanonicalRandomVertexCut

It works is similar to previous one but first it orders the two ids.

3.1.3 EdgePartition1D

It was implemented in GraphX [20] and it uses only the source vertex id in the hash function.

3.1.4 Randomized Bipartite-cut (BiCut)

This partitioner relies on the idea that random assigning the vertices of one of the two independent sets of a bipartite graph may not introduce any replicas. In particular `BiCut` selects the largest set of independent vertices, and then splits it randomly into N subsets. Then a partition is created from all the edges connected to the corresponding subset.

3.2 Segmenting the hash space

This approach complements random assignment with a segmentation of the hash space using some geometrical forms as grids, torus, etc. It maintains the good balance of the previous partitioners, while limiting at the same time the maximum replication factor.

3.2.1 Grid-based Constrained Random Vertex-cuts (Grid)

It partitions edges in a shard-grid G by using a simple hash function. For example the *EdgePartition2D* partitioner in [20] maps the source vertex to a column of G and the destination vertex to a row of G. The edge is then placed in the shard at the column-row intersection and the shards are finally assigned to partitions in a round robin fashion. This partitioner guarantees that the *replication factor* is upper-bounded by $2\sqrt{n} - 1$. A variant of this algorithm is proposed in [11], where two different column-row pairs are selected for the source and the destination and then the edge is randomly assigned to one of the shard belonging to both the pairs.

3.2.2 Torus-based Constrained Random Vertex-cuts (Torus)

It is similar to the `Grid` partitioner considered in [11] but it relies on a 2D torus. Each vertex is mapped to one column and to $\frac{1}{2}R + 1$ shards of a given row, where R is the number of shards in a row. Then, as in previous partitioner, it considers the shards at the intersection of the two sets identified for the two vertices, and it randomly selects one of them. In this way, the *replication factor* has an upper bound equal to $1.5\sqrt{n} + 1$.

3.3 Greedy approach

These partitioners assign edges greedily among the partitions, minimizing the current *replication factor* at each step.

3.3.1 Greedy Vertex-Cuts

To place the i^{th} edge, this partitioner considers where the previous $i - 1$ edges have been assigned. Basically, it tries to place an edge in a partition which already contains the source and the destination of this edge. If it is not possible, then it tries to allocate the edge to a partition which contains at least one vertex

of this edge. If no such a partition can be found, the edge is assigned to the partition that is currently the smallest one. The authors also described a distributed implementation called `Coordinated Greedy Vertex-Cuts` (`Greedy-Coordinated` in what follows). This implementation requires communication about the different instances of the partioner. The authors also introduced a relaxed version called `Oblivious Greedy Vertex-Cuts` (`Greedy-Oblivious` in what follows), where no communication is required because each instance of the partitioner greedily assign an edge as described above but only considering its previous choices.

3.3.2 Grid-based Constrained Greedy Vertex-cuts (Grid Greedy)

As the `Grid` partitioner proposed in [11] it relies on a shard grid, but the final shard is selected (from intersection) using the same greedy criterium in 3.3.1.

3.3.3 Torus-based Constrained Greedy Vertex-cuts (Torus Greedy)

The algorithm modifies the `Torus` algorithm in the same spirit the `Grid Greedy` partitioner modifies the `Grid` one.

3.3.4 Distributed Funding-based Edge Partitioning (DFEP)

This partitioner was proposed in [10] and implemented for both Hadoop [4] and Spark [21]. In the `DFEP` algorithm each partition receives initially a randomly assigned vertex and then tries to progressively grow by including all the edges of the vertices currently in the partition that are not yet assigned. In order to avoid a dishomogeneous growth of the partitions, a virtual currency is introduced and each partition receives an initial funding that can be used to bid on the edges. Periodically, each partition receives additional funding inversely proportional to the number of edges it has.

3.4 Cut hubs

Most of the real-world graphs are power-law graphs, where a relatively small percentage of nodes (hubs) concentrate most of the edges. This approach moves then from the observation that in any case hubs need to be cut to maintain balance among the partitions. These partitioners prefer then to cut hubs, trying to spare the large share of nodes that have a small degree.

3.4.1 Hybrid-Cut

The `Hybrid-cut` [7] partitioner considers that a vertex is not a hub if its in-degree is below a given threshold. ISn such case all the incoming edges are partitioned based on the hash value of this vertex and then they are placed in the same partition. Otherwise, all incoming edges are partitioned based on their source vertices. The algorithm was implemented for GraphX [1].

3.4.2 Ginger

As `Hybrid-cut` this partitioner [7] allocates the edges incoming to a no-hub vertex (v) to the same partition. The difference it that the partition P_i is selected on the basis of the following heuristic:

$$i = \underset{i=1,\dots N}{\operatorname{argmax}} \Big\{ |\mathcal{N}_{in}(v) \cap V(E_i)| - \tfrac{1}{2}(|V(E_i)| + \tfrac{|V|}{|E|}|E_i|) \Big\}.$$

The underlying idea is that the partitioner selects the partitions where there are already neighbors of node v, as far as this partition is not too large.

3.4.3 HybridCutPlus

It works as `Hybrid-Cut` whenever possible (i.e. if one vertex of an edge is a hub and another vertex is not a hub), otherwise it works as a `Grid` partitioner. It was implemented for GraphX [1].

3.4.4 Aweto

It is a greedy implementation of `BiCut` using a heuristic similar to the one in the `Ginger` partitioner.

3.5 Iterative approach

While the previous partitioners assign once and for all a link to a partition, other partitioners may iterate multiple times on the partitioning, reassigning an edge to a different partition.

3.5.1 DFEPC

It is an iterative variant of `DFEP` that compensates for the possible negative effect of starting from an initial vertex that is poorly connected to the rest of the graph. `DFEP` is changed as follows: a partition whose size is by a given factor smaller than the current average partition size can also bid for edges that were already bought by other partitions, leading to their reassignments.

3.5.2 JA-BE-JA-VC

It was proposed in [18], and it is a vertex-cut version of an existing edge-cut partitioner `JA-BE-JA` [19]. The `JA-BE-JA-VC` algorithm starts by randomly assigning edges to partitions. Then each iteration of the algorithm is executed on each vertex in parallel. Two vertices are allowed to exchange the partition their edges belong to, provided that the exchange leads to a better cut of the graph. The potential benefit of an exchange is evaluated through two different heuristics (referred to as Dominant Color or Edge Utility). Simulated annealing can also be used to escape local minima.

In [18] the authors also consider a simple way to adapt `JA-BE-JA` [19] to produce an edge partitioning. First, `JA-BE-JA` is executed to produce vertex

partitions. Then the edges that are cut are randomly assigned to one of the two partitions their vertices belong to.

4 Discussion

Table 1 shows which partitioners were directly compared and which metrics were considered in the comparison ("I" denotes execution metrics, while "II" denotes partitioning metrics). Bold fonts indicate that the comparison was performed by us (see Sec. 5). The table shows how many direct comparisons are missing (e.g. `Hybrid-Cut` was not compared to `Torus`, `JA-BE-JA-VC`, etc.) and our experiments in this paper contribute to provide a more complete picture. Even when a direct comparison is available, results are not necessarily conclusive. For example, in [18] the authors show that `JA-BE-JA-VC` outperforms `DEFP`, `DFEPC`, and the modified version of `JA-BE-JA` in terms of *STD*. However in terms of *normalized vertex-cut*, the modified version of `JA-BE-JA` is the best.

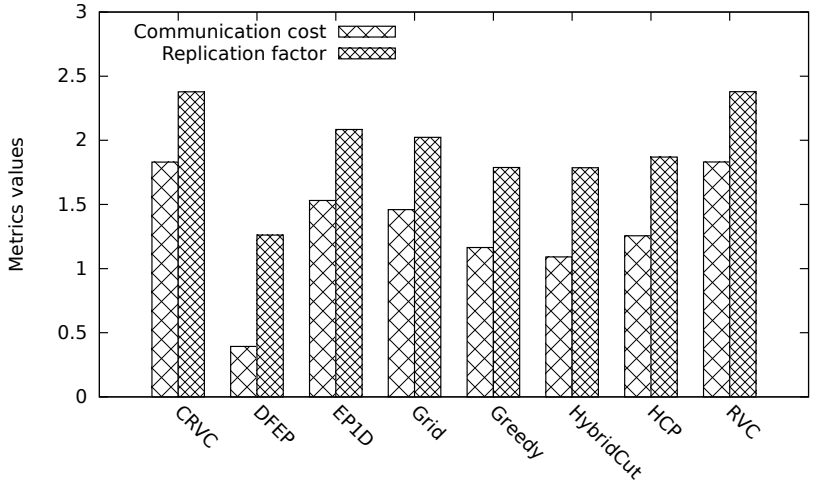
Table 2 indicates which specific metrics were considered in each paper. We can observe that in many cases execution metrics have not been considered due to their cost in terms of computation time. In particular, the lack of information about the partitioning time is problematic because in our experiments we have observed that it can vary by more than one order of magnitude across partitioners and contribute the most to the total execution time. The table also shows that it is difficult to perform an indirect comparison among partitioners using results from different papers, because there is often no common set of metrics considered across them.

We showed that it is hard to reach any conclusion regarding the benefits of existing partitioners for multiple reasons. First, not all partitioners have been compared. Second, execution metrics are not always provided and are tied to a specific computing environment. Finally, there is no study which links partitioning metrics to execution metrics, but for [17]. Motivated by these considerations, we decided to conduct experiments using the GraphX framework. Our results partially complete the pairwise comparison in Table 1 and are presented in the next section.

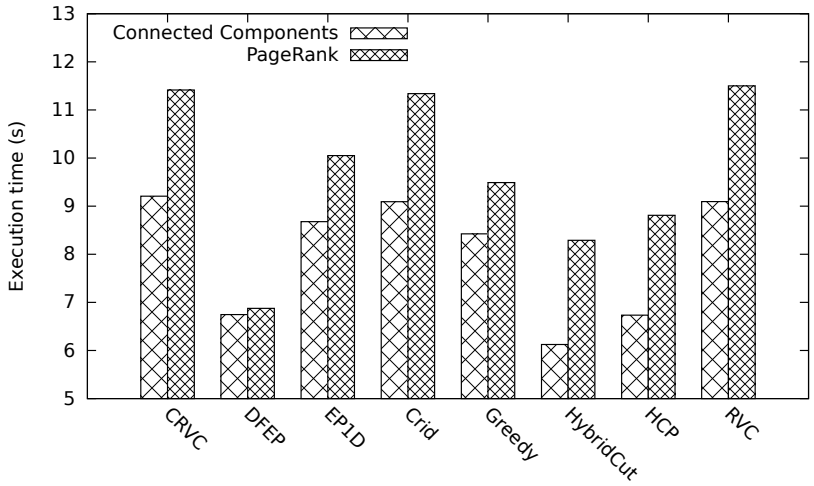
5 Experiments

We conducted our experiments on a computational cluster of nodes with dual-Xeon E5-2680@2.80GHz, 192GB RAM and 20 cores. We used Spark version 1.4.0 in standalone cluster mode. Our cluster configuration had eleven machines, one for master and ten for executors. We configured 4 Spark properties: *spark.executor.memory* (4 GB), *spark.driver.memory* (10 GB), *spark.cores.max* (10) and *spark.local.dir* (set to a local directory).

We measured *replication factor*, *communication cost*, *balance*, *STD* as partitioning metrics, and the *partitioning time* and the *execution time* of `Connected Components` and `PageRank` (10 iterations) algorithms as execution metrics.



(a) Communication metrics.



(b) Execution time of algorithms.

Figure 1: Results for *com-Youtube* graph (CRVC - `CanonicalRandomVertexCut`; EP1D - `EdgePartition1D`; Greedy - `Greedy-Oblivious`; HCP - `HybridCutPlus`; RVC - `RandomVertexCut`).

As input we used the undirected *com-Youtube* graph (1,134,890 vertices/2,987,624 edges) from the SNAP project [2].

As a set of partitioners, we used all the GraphX built-in partitioners, i.e. `RandomVertexCut`, `CanonicalRandomVertexCut`, `EdgePartition1D`, `Grid`, the partitioner `DFEP` (whose code was gently made available by the authors of [10]),

Table 1: Partitioners pairwise comparisons, considering execution metrics (I) or partitioning metrics (II).
Bold fonts indicate experiments we conducted.

	RandomVertexCut	CanonicalRandomVertexCut	EdgePartition1D	Greedy-Coordinated	Greedy-Oblivious	Grid	Grid Greedy	Torus	Torus Greedy	DFEP	DFEPC	JA-BE-JA-VC	Hybrid-Cut	Ginger	HybridCutPlus	BiCut	Aweto	JA-BE-JA
RandomVertexCut	-	**I+II**	**I+II**	I+II	**I+II**	**I+II**	I+II	II	I+II	**I+II**			**I+II**		**I+II**			
CanonicalRandomVertexCut		-	**I+II**	I+II	I+II	**I+II**				**I+II**			**I+II**		**I+II**			
EdgePartition1D			-		**I+II**	**I+II**				**I+II**			**I+II**		**I+II**			
Greedy-Coordinated				-	I+II	I+II	I+II	II	I+II	II	II		I+II	I+II				II
Greedy-Oblivious					-	I+II				**I+II**	II		I+II	I+II	**I+II**			II
Grid						-	II	II	II	**I+II**			I+II	I+II	**I+II**	I+II	I+II	
Grid Greedy							-	II	I+II									
Torus								-	II									
Torus Greedy									-									
DFEP										-	I+II	II	**I+II**		**I+II**			I+II
DFEPC											-	II						I+II
JA-BE-JA-VC												-						II
Hybrid-Cut													-	I+II	**I+II**			
Ginger														-				
HybridCutPlus															-			
BiCut																-	I+II	
Aweto																	-	
JA-BE-JA																		-

`Greedy-Oblivious` (implemented by us), `Hybrid-Cut` and `HybridCutPlus` (both implemented by Larry Xiao [1]).

In each experiment we first randomly selected a partitioner and a graph processing algorithm. Then, we applied the selected partitioner to the input graph. Finally, we executed the selected graph processing algorithm. We repeated every combination at least 30 times to obtain 95% confidence intervals for the execution time whose relative amplitude is always less than 10%.

Our experiments confirm that, as expected, the best *balance*, and *STD* metrics are obtained by random partitioners like `RandomVertexCut` and `CanonicalRandomVertexCut`. In terms of communication metrics, Fig. 1a shows that `DFEP` outperforms the other partitioners. This improvement comes at the cost of a much larger partitioning time, indeed in our setting `DFEP` partitions the graph in a few minutes, larger than the other partitioners considered by a factor 20. Moreover, these partitioning metrics are not necessarily good predictors for the final execution time. Indeed, Fig. 1b shows that while `PageRank` achieves the shortest execution time when the graph is partitioned by `DFEP`, `Hybrid-Cut` provides the best partitioning for `Connected Components` and `HybridCutPlus` performs almost as `DFEP`. This result questions the extent to which partitioning metrics can be used to predict the actual quality of a partition. From the practical point of view, `Hybrid-Cut` appears to achieve the best trade-off between partitioning time and reduction of the execution time.

Table 2: Metrics used to evaluate partitioners grouped by papers

References	Partitioners	Execution metrics	Partitioning metrics
[9]	CanonicalRandomVertexCut	Partitioning and Execution time (PowerGraph)	Replication factor
	Greedy-Coordinated		
	Greedy-Oblivious		
[11]	RandomVertexCut	Execution time (GraphBuilder)	Replication factor
	Greedy-Coordinated		
	Grid Greedy		
	Torus Greedy		
	Grid	-	
	Torus	-	
[10]	DFEP	Execution time (Hadoop and GraphX) and rounds	Balance, communication cost, STD
	DFEPC		
	JA-BE-JA	Rounds	
	Greedy-Coordinated	-	Balance and communication cost
	Greedy-Oblivious		
[18]	Random	-	Vertex-cut, normalized vertex-cut, STD
	DFEP		
	DFEPC		
	JA-BE-JA		
	JA-BE-JA-VC		
[7]	Hybrid-Cut	Execution time (PowerLyra)	Replication factor
	Ginger		
	Greedy-Coordinated		
	Greedy-Oblivious		
	Grid		
[6]	BiCut	Execution time and normalized network traffic (GraphLab)	Replication factor
	Aweto		
	Grid		
[20]	RandomVertexCut	-	-
	CanonicalRandomVertexCut		
	EdgePartition1D		
	Grid		

6 Conclusion

In this paper we provided an overview of the existing edge partitioners. An analysis of the related literature shows they have been evaluated considering a disparate set of metrics and, moreover, they have been limitedly compared against each other. We presented some preliminary experimental results comparing a large number of edge partitioners for GraphX framework. We observed that partitioning metrics are not always suited to predict which partitioner will provide the shortest execution time and that simple and fast partitioners like `Hybrid-Cut` can outperform more sophisticated ones.

References

[1] Hybrid-Cut and HybridCutPlus partitioners. `https://github.com/larryxiao/spark/`

[2] Stanford Large Network Dataset Collection. `https://snap.stanford.edu/data/`

[3] Avery, C.: Giraph: Large-scale graph processing infrastructure on hadoop. Proceedings of the Hadoop Summit. Santa Clara 11 (2011)

[4] Bialecki, A., Cafarella, M., Cutting, D., O'Malley, O.: Hadoop: a framework for running applications on large clusters built of commodity hardware. Wiki at http://lucene. apache. org/hadoop 11 (2005)

[5] Bourse, F., Lelarge, M., Vojnovic, M.: Balanced graph edge partition. In: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 1456–1465. ACM (2014)

[6] Chen, R., Shi, J.X., Chen, H.B., Zang, B.Y.: Bipartite-oriented distributed graph partitioning for big learning. Journal of Computer Science and Technology 30(1), 20–29 (2015)

[7] Chen, R., Shi, J., Chen, Y., Chen, H.: Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In: Proceedings of the Tenth European Conference on Computer Systems. p. 1. ACM (2015)

[8] Flake, G.W., Tarjan, R.E., Tsioutsiouliklis, K.: Graph clustering and minimum cut trees. Internet Mathematics 1(4), 385–408 (2004)

[9] Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). pp. 17–30 (2012)

[10] Guerrieri, A., Montresor, A.: Dfep: Distributed funding-based edge partitioning. In: European Conference on Parallel Processing. pp. 346–358. Springer (2015)

[11] Jain, N., Liao, G., Willke, T.L.: Graphbuilder: scalable graph ETL framework. In: First International Workshop on Graph Data Management Experiences and Systems. p. 4. ACM (2013)

[12] Kannan, R., Vempala, S., Vetta, A.: On clusterings: Good, bad and spectral. Journal of the ACM (JACM) 51(3), 497–515 (2004)

[13] Karypis, G., Kumar, V.: METIS–unstructured graph partitioning and sparse matrix ordering system, version 2.0 (1995)

[14] Kim, M., Candan, K.S.: SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. Data & Knowledge Engineering 72, 285–303 (2012)

[15] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment 5(8), 716–727 (2012)

[16] Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146. ACM (2010)

[17] Mykhailenko, H., Neglia, G., Huet, F.: Which Metrics for Vertex-Cut Partitioning? In: Proceedings of the 11th International Conference for Internet Technology and Secured Transactions (2016)

[18] Rahimian, F., Payberah, A.H., Girdzijauskas, S., Haridi, S.: Distributed Vertex-Cut Partitioning. In: 4th International Conference on Distributed Applications and Interoperable Systems (DAIS). vol. LNCS-8460, pp. 186–200. Berlin, Germany (2014)

[19] Rahimian, F., Payberah, A.H., Girdzijauskas, S., Jelasity, M., Haridi, S.: Ja-be-ja: A distributed algorithm for balanced graph partitioning (2013)

[20] Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: A resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems. p. 2. ACM (2013)

[21] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (2012)

7 Acknowledgment

This work was partly funded by the French Government (National Research Agency, ANR) through the "Investments for the Future" Program reference #ANR-11-LABX-0031- 01. We thank the authors of [10] to have provided us their implementation of *DFEP* for Spark.