WSGP: A Window-based Streaming Graph Partitioning Approach

Yunbo Li, Chuanyou Li, Anne-Cécile Orgerie, Philippe Parvédy

WSGP: A Window-based Streaming Graph Partitioning Approach

Yunbo Li*, Chuanyou Li†, Anne-Cécile Orgerie‡, Philippe Raipin Parvédy§
* *Shanghai Pudong Development Bank, Shanghai, China – Email: liyb14@spdb.com.cn*
† *Southeast University, China – Email: cyli@seu.edu.cn*
‡ *Univ Rennes, Inria, CNRS, IRISA, Rennes, France – Email: anne-cecile.orgerie@irisa.fr*
§ *Orange Labs, France – Email: philippe.raipin@orange.com*

***Abstract*—Graph partitioning, a preliminary step of distributed graph processing, has been attracting increasing attention in the last decade. A high quality graph partitioning algorithm should facilitate graph processing by minimizing the communication overhead and maintaining the load balancing among distributed computing units. Offline partitioning algorithms usually require the knowledge of a complete graph,and therefore, are not adaptive to handle massive graph-structured data. On the contrary, streaming partitioning algorithms take edges or vertices as a stream and make partitioning decisions on the fly. However, the streaming manner faces dilemmas from time to time because of a lack of knowledge. Furthermore, an unmindful partitioning decision in such a dilemma could significantly decrease the partition quality. In this paper, we propose a novel window-based streaming graph partitioning algorithm (WSGP). WSGP leverages a greedy-based heuristic to perform edge partitioning. When facing a decision dilemma, WSGP utilizes a size-bounded window to buffer the edges. When the window is fully filled, an edge is poped and assigned to a partition. The assignment is decided by knowledge obtained from both the edges already settled and the ones still cached in the buffer window. Our experiments take into account various real-world benchmark graphs. The experimental results demonstrate that WSGP consistently has a smaller replication factor than the state-of-the-art algorithms by up to 23%, at a limited cost in terms of memory and comprehensive running time.**

I. Introduction

Many complex systems such as social networks, road networks, or molecular networks, ranging from computer science to biology, can be naturally represented by graphs. Analyzing graph-structured data brings opportunities to explore underlying rich and complex relationships. This has been widely applied in recommendation, link prediction, and subgraph matching for instance. However, with the rapid growth of graph-structured data, it is no more adaptive to perform graph processing in a single computing unit. As an example, the graph from the online social network Orkut (used later in this work) comprises more than 3,000,000 vertices and more than 117,000,000 edges [1].

Processing graphs over distributed systems has attracted significant interests in the last decade, from both the research and industry communities. In particular, distributed graph processing systems, such as Pregel [2], GraphLab [3], Powergraph [4] and GraphX [5], have been proposed and succeeded to process billion-scale graphs. To perform distributed graph processing, it is required to partition the graph first. Edge partitioning currently constitutes the favored approach for very large graphs in order to balance the load among distributed computing units.

In this paper, we study the edge partitioning (vertex cut) problem, where each edge is assigned to a single partition, such that the graph is divided along vertices. Usually, vertex cut is modeled through vertex replication. For example, if two edges share a common vertex but are assigned to two different partitions, then the common vertex has two replicas, one on each partition. Such replication inevitably causes synchronization or communication overhead for distributed graph processing. Hence, a high quality edge partitioning approach should minimize the number of produced replicas. On the other hand, fair load distribution is highly desirable in distributed systems. Therefore, edge partitioning should also maintain load balancing by assigning edges evenly over partitions.

To perform edge partitioning, there are two basic approaches: offline partitioning and streaming partitioning. Offline partitioning relies on loading the complete graph into memory. This way allows for designing high quality partitioning heuristics, but is lacking of availability to process large-scale graphs. On the opposite side, streaming partitioning follows the online setting. Edges arrive as a stream and the partitioner assigns edges to partitions on the fly. Compared with offline partitioning, the streaming manner has a good scalability and a reasonable loss of partitioning quality.

In this paper, we systematically investigate the streaming edge partitioning problem, and propose a novel Window-based Streaming Graph Partitioning approach (*WSGP*). *WSGP* inherits the framework of Oblivious [4] and leverages new heuristic strategies. Once an edge arrives, Oblivious makes the assignment according to four rules. Yet, we notice that in Oblivious it is hard to make a good assignment for an edge whose pair vertices does not appear in the same partition. We consider such scenario as a dilemma, whose existence is due to the fact that the current partial knowledge is not sufficient. Aiming at such problem, our idea is to leverage a window to buffer this edge temporarily while continuing to assign the next incoming edges. The buffering

operation does provide new opportunities to break through the dilemma, as it allows to collect more knowledge of the graph. As soon as the window is fully filled, we pop a buffered edge and carry out the assignment. If this edge is still in the same dilemma, our new strategy selects the partition which targets to benefit the assignment of more edges still buffered in the window, *i.e.*, let more buffered edges not meet the dilemma again.

Our contributions are summarized as follows:

- We perform a comprehensive study on different streaming edge partitioning algorithms. We characterize the features and identify the key problems.
- We propose *WSGP* a new window-based streaming partitioning approach. *WSGP* enables the partitioner to synthetically use knowledge obtained from both the edges already settled and still cached, and this approach succeeds in avoiding bad assignments.
- We carry out the evaluation with different types of real-world graphs. The experimental results demonstrate that our new approach outperforms existing streaming edge partitioning algorithms in terms of the replication factor, at the cost of a acceptable processing time overhead and a limited additional memory consumption.

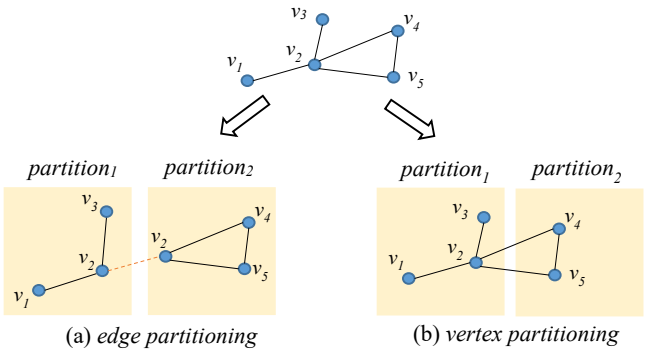
The rest of this paper is organized as follows. Section II sketches out the related work. Section III introduces the notations and gives a formal problem definition. In Section IV, a new window-based streaming partition algorithm is proposed and analyzed. The experimental setup and results are discussed in Section V. Finally, Section VI concludes the paper.

II. Related work

Graph partitioning is a well-known fundamental problem in many parallel and distributed applications [2], [6], [7]. There are two graph partitioning variants: edge partitioning (vertex cut) and vertex partitioning (edge cut). Edge partitioning results in partitions that are edge disjoint while vertex partitioning results in partitions that are vertex disjoint. Figure 1 depicts an example showing their differences. We can see that both partitioning variants bring in additional costs, *e.g.*, in Figure 1(a) vertex v_2 is replicated and in Figure 1(b) two edges are cut. In another aspect, load balancing also affects the efficiency of downstream tasks execution due to Liebig's law, such that graph partitioning in a general not only targets to minimize the partitioning costs, but also to maintain load balancing. For optimizing both objectives, the two variants are known to be NP-hard problems [8]–[10]. In the following, we first sketch typical vertex partitioning approaches and then introduce the latest development on edge partitioning.

A. Vertex Partitioning

Pregel [2] and *GraphLab* [3] are two distributed graph processing systems that leverage a vertex-program which is executed on each vertex and can interact through shared-memory or messages. Both *Pregel* and *GraphLab* depend on vertex partitioning to minimize communication costs and ensure load balancing. However, in the case of natural graphs (*e.g.*, power-law graphs), *Pregel* and *GraphLab* are forced to resort to hash-based partitioning which can extremely decrease the system performance.



(a) *edge partitioning* (b) *vertex partitioning*

Figure 1: Edge Partitioning vs. Vertex Partitioning

LDG [11], [12] and *Fennel* [7] also follow the vertex partitioning paradigm but take a stream of vertices as input. *LDG* uses a greedy heuristic that tries to assign neighboring vertices to the same partition. *Fennel* leverages a heuristic which combines locality-centric measures with load balancing factors [12]. Both *LDG* and *Fennel* require the total number of vertices as an a priori. Hence, both of them could be considered as bounded streaming algorithms.

J.E. Gonzalez *et al.* [4] argued that edge partitioning is more adaptive for partitioning natural graphs. They also pointed out that for a given edge-cut with g replicas, any vertex cut along the same partition boundary has strictly fewer than g mirrors. In recent years, more researches has focused on studying edge partitioning.

B. Edge Partitioning

C. Zhang *et al.* [13] proposed *NE* an edge partitioning heuristic whose key idea is neighborhood expansion. *NE* selects edge gradually to fully fill each partition and this approach performs quite well. M. Hanai *et al.* [14] proposed a follow-up study. They reformed *NE* to distribute its approach, this proposition can process trillion-edge graphs and achieves better performance in terms of running time. However, both of them depend on the knowledge of the complete graph.

Streaming based edge partitioning does not rely on knowing the complete graph. Z. Abbas *et al.* [15] did an experimental study of several typical streaming edge partitioning methods including *Hash*, *Grid* [16], *PDS* [16], *DBH* [17], *Oblivious* [4] and *HDRF* [18]. *Hash* is an intuitional method that decides the partition by simply computing a hash function. *Hash* is adept at maintaining load balancing but can lead to a large number of vertex replicas. *Grid* and *PDS* are

also hash based methods. The difference is that they require additional constrains. For example, *Grid* needs all partition IDs arranged in a square matrix, while *PDS* involves Perfect Difference Sets which requires p^2+p+1 partitions (p being prime). Existing studies [15] showed that only relying on hash function is difficult to have a good partitioning result. *DBH* prioritizes cutting the vertices having highest degree and employs hash for partitioning. With its heuristic cutting high degree vertices, *DBH* outperforms pure hash based methods. Different from *Hash*, *Grid*, *PDS* and *DBH*, our new proposed *WSGP* does not depend on any hash function to assign the edges.

Oblivious [4] and *HDRF* [18] are closer to our work. As a general approach, *Oblivious* uses greedy heuristic to make the current optimal assignment according to the known partial knowledge. When *Oblivious* meets a dilemma, it simply performs the assignment by only considering load balancing. Different from *Oblivious*, *HDRF* prefers to cut vertices with high degrees when meeting a dilemma. After deciding the cutting preference, *HDRF* makes the final assignment in order to maintain load balancing. Our *WSGP* also uses a greedy algorithm. Different from *Oblivious* and *HDRF*, *WSGP* leverages a bounded window to buffer the edge temporarily when facing an assignment dilemma. When processing a buffered edge, *WSGP* does the assignment aiming at producing less replicas for the successive assignments of edges still buffered in the window.

Our work is also related *ADWISE* [19] as the common idea consists in involving a window to buffer edges. However, both ways of window utilization differ. *ADWISE* puts every edge into the window. Once the window is fully filled, a greedy strategy integrating a balancing factor, a degree-awareness factor and a clustering factor is performed to decide the assignment. According to our study, this strategy that buffer each vertex is not good for partitioning efficiency. Hence, in our approach, we consider to buffer only the edges facing a special dilemma. Meanwhile, for the way to use information collected by the window, we propose a new greedy strategy aiming at producing less replicas, which also differ from *ADWISE* strategy.

III. Notations and Problem Formulation

In this section, we first provide necessary notations and then formulate the streaming edge partitioning problem.

A. Notations

We consider $G = (V, E)$ the undirected graph waiting to be processed, where V is a set of vertices and E is the set of edges. To facilitate our presentation, we usually use lowercase like u (or v, x, y, ...) to express a vertex and use $e_{u,v}$ to represent an edge connecting two vertices u and v. A partition of edges $P = (P_1, P_2, ..., P_k)$ is defined as a family of pairwise disjoint edge sets, *i.e.*, $\forall i \neq j, P_i \cap P_j = \emptyset$ and $P_1 \cup P_2 \cup ... \cup P_k = E$. Each partition P_i $(i = 1, ..., k)$ is a subset of edges denoted as $P_i = \{e_{x,y} | e_{x,y} \in E \wedge x, y \in V\}$.

Let $A(u)$ be the set of partitions ($A(u) \subseteq P$) where each vertex $u \in V$ is replicated. The replication factor RF is defined to be the average number of replicas per vertex [20]:

$$RF = \frac{1}{|V|} \sum_{u \in V} |A(u)| \tag{1}$$

A partitioning of edges P is σ balanced if:

$$\max_{P_i \in P} |P_i| < \sigma \frac{|E|}{|P|}, \tag{2}$$

where $|E|$ is the total number of edges of G, $|P| = k$ represents the total number of partitions, and $\sigma \geq 1$ is the factor describing the maximum acceptable imbalance.

B. Problem Formulation

In streaming edge partitioning, a sequence of edges $e_{u,v}$, $e_{x,y}$, ... (that could be in any order) arrive in a streaming manner. An edge partitioning algorithm claimed to be streaming should not foreknow the sequence order and has to assign every edge to a given partition P_i on the fly.

In this paper, we focus on a k-way σ balanced streaming edge partitioning problem which divides the edge set E into k disjoint parts and aims at minimizing the replication factor RF, meanwhile, maintaining the load of each partition σ balanced. The formulation is given in the following:

$$minimize\ RF; \tag{3}$$

$$s.t.,\ \max_{P_i \in P} |P_i| < \sigma \frac{|E|}{|P|} \tag{4}$$

IV. Streaming Edge Partitioning

A. Analysis

Our algorithm *WSGP* follows the framework of *Oblivious* [4] which leverages a greedy approach to assign edges on the fly. Once an edge $e_{(u,v)}$ arrives, *Oblivious* decides the assignment for $e_{(u,v)}$ according to the following four cases:

1) Case 1: If $A(u) \cap A(v) \neq \emptyset$, $e_{(u,v)}$ is assigned to the minimum loaded partition in $A(u) \cap A(v)$.
2) Case 2: If $A(u) = \emptyset$ and $A(v) \neq \emptyset$, $e_{(u,v)}$ is placed to the minimum loaded partition in $A(v)$. And vice versa, if $A(u) \neq \emptyset$ and $A(v) = \emptyset$, $e_{(u,v)}$ is placed to the minimum loaded partition in $A(u)$.
3) Case 3: If $A(u) = \emptyset$ and $A(v) = \emptyset$, $e_{(u,v)}$ is placed to the minimum loaded partition in $|P|$.
4) Case 4: If $A(u) \neq \emptyset$, $A(v) \neq \emptyset$ and $A(u) \cap A(v) = \emptyset$, $e_{(u,v)}$ is assigned to the minimum loaded partition in $A(u) \cup A(v)$.

Case 1 and Case 2 apply reasonable greedy strategies to selected the optimal partition according the current state. In case 3, both vertices u and v appear for the first time, *i.e.*, there is no known information to assign $e_{(u,v)}$ towards RF minimization. Hence, the assignment can be done by only

```
1: Input e_(x,y);
2: if |w| = MaxSize then
3:     e_(u,v) ← w.pop();
4:     if A(u) ∩ A(v) ≠ ∅ then
5:         j ← arg min_{P_i ∈ A(u)∩A(v)}(|P_i|);
6:     else
7:         j ← score(u, v);
8:     end if
9:     P_j ← P_j ∪ e_(u,v);
10:    N(u) ← N(u)\v;
11:    N(v) ← N(v)\u;
12: end if
13: w ← w ∪ e_(x,y);
14: N(x) ← N(x) ∪ {y};
15: N(y) ← N(y) ∪ {x};
```

Figure 2: Window-based Edge Allocation (*WEA*)

```
1: Input u, v;
2: for each P_i ∈ A(u) do
3:     rank(P_i) = Σ_{x∈N(v)} g(x, P_i);
4: end for
5: m_1 ← max rank(P_i), where P_i ∈ A(u);
6: S_1 ← {P_i | P_i ∈ A(u) ∧ rank(P_i) = m_1};
7: for each P_i ∈ A(v) do
8:     rank(P_i) = Σ_{x∈N(u)} g(x, P_i);
9: end for
10: m_2 ← max rank(P_i), where P_i ∈ A(v);
11: S_2 ← {P_i | P_i ∈ A(v) ∧ rank(P_i) = m_2};
12: S ← S_1 ∪ S_2;
13: id ← arg max{rank(P_i) + P_bal(P_i)}, where P_i ∈ S;
14: return id;
```

Figure 3: The pseudo code of method $score$

considering load balancing. In Case 4, both vertices u and v appeared before. However u and v do not have replicas in the same partition as $A(u) \cap A(v) = \emptyset$. We consider Case 4 as an assignment dilemma, since no matter how to do the partition, the partitioner either increases u's replicas by one or increases v's replicas by one. In such a dilemma, Oblivious considers the minimum loaded partition, *i.e.*, and totally ignores RF in this case. Different from Oblivious, to break the symmetry, *HDRF* [18] chooses to sacrifice the one with the larger degree.

Our idea is different from both of these approaches: we consider taking advantage of memory to cache edges like $e_{(u,v)}$ temporarily. When we try to assign $e_{(u,v)}$ for the second time, the situation is different because we can utilize information from both the already processed edges and the ones still cached to break the symmetry. According to this key idea, we propose a novel Window-based Streaming Graph Partitioning algorithm (*WSGP*).

B. Window-based Algorithm

WSGP leverages a window to cache those edges whose partitions are hard to be decided yet. In a nutshell, once a new edge $e_{(u,v)}$ arrives, there are four cases to deal with $e_{(u,v)}$:

1) Case 1, Case 2, Case 3 inherits from *Oblivious*;
2) Case 4: If $A(u) \neq \emptyset$, $A(v) \neq \emptyset$ and $A(u) \cap A(v) = \emptyset$, call the method *WEA*($e_{(u,v)}$) to tackle $e_{(u,v)}$.

Before elaborating Window-based Edge Allocation (*WEA*), let us introduce the data structure for constructing the window w. Generally, w buffers each edge $e_{(u,v)}$ coming from Case 4. The cardinality $|w|$ of w is defined as the number of edges it has buffered. For easily obtaining information from w, we also maintain a partial neighbor list for each vertex appearing in w. More precisely, if $e_{(u,v)}$ has been buffered in w, we then maintain two sets $N(u)$ and $N(v)$. Take $N(u)$ as an example, it records u's neighbors that also appears in w. For example, in Figure 4, there is a window buffering of six edges. At a given time depicted on top of the Figure 4, $N(u)$ includes three vertices v, x, and y, and $N(v)$ includes one vertex u.

The pseudo code of *WEA* is provided in Figure 2. *WEA* takes an edge $e_{(x,y)}$ as its input (line 1). If w still has an empty space to accommodate one more edge, *WEA* buffers $e_{(x,y)}$ in w (line 13) and updates $N(x)$ and $N(y)$ (lines 14-15) immediately. Otherwise, *WEA* pops out an edge $e_{(u,v)}$ (line 3) to compute its partition (lines 4-11) and then buffers $e_{(x,y)}$. For the popped edge $e_{(u,v)}$, *WEA* first tests whether Case 1 is satisfied (line 4). If so, $e_{(u,v)}$ is assigned to the minimum loaded partition in $A(u) \cap A(v)$ (line 5). Note that when $e_{(u,v)}$ arrives in the system, Case 4 is satisfied, such that $e_{(u,v)}$ is buffered in w. The buffering of $e_{(u,v)}$ actually changes the order of edge assignments, such that when $e_{(u,v)}$ is popped, we can have Case 1 that is satisfied. If the test in line 4 is not passed, the method $score$ is called to return the partition for $e_{(u,v)}$ (line 7). Once the assignment is decided, $e_{(u,v)}$ is settled (line 9) and then the dual sets $N(u)$, $N(v)$ are updated (lines 10-11).

The pseudo code of the method $score$ is provided in Figure 3. $score$ takes the pair vertices of $e_{(u,v)}$ as the inputs (line 1). As now $A(u)$ and $A(v)$ have no intersection, we shall choose the best partition from $A(u) \cup A(v)$ to be the target partition. Hence, we scan each partition in $A(u) \cup A(v)$ to get their ranks (lines 2-4, 7-9). We pick a partition P_i from $A(u)$ as an example. P_i's rank is counted by the number of vertices that appear both in P_i and $N(v)$, which could be formulated by $g(x, P_i)$ as follows:

$$g(x, P_i) = \begin{cases} 1, & \text{if } P_i \in A(x); \\ 0, & \text{otherwise.} \end{cases}$$

Note that such a heuristic aims at getting benefits from using the window scheme: once $e_{(u,v)}$'s partition is decided, it should facilitate as much as possible edges also buffered in the window to get their partitions, since they are not in a dilemma situation anymore.

m_1 records the highest rank of partitions from $A(u)$ (line 5). Then all the partitions from $A(u)$ ranked by m_1 are logged in to an auxiliary set S_1 (line 6). Lines 7-11 perform

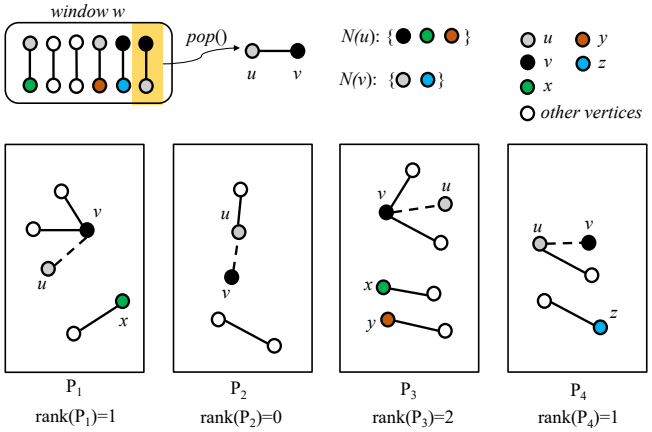


Figure 4: An Example of *WEA*

a similar ranking process for partitions from $A(v)$. For all partitions selected (logged in S, line 12), we then combine them with the balance factor P_{bal} to make the final decision (line 13). The id of the target partition is returned (line 14).

$$P_{bal}(P_i) = \lambda \times \frac{\max_{P_j \in P} |P_j| - |P_i|}{\max_{P_j \in P} |P_j| - \min_{P_j \in P} |P_j| + \epsilon} \quad (5)$$

The above equation (5) calculates $P_{bal}(P_i)$, where ϵ is a constant used to avoid the denominator equals to 0 and λ is used to define the importance of load balancing.

Figure 4 depicts an example showing the process after an edge $e_{(u,v)}$ is popped. In this example, w contains six edges, from which we can get $N(u) = \{v, x, y\}$ and $N(v) = \{u, z\}$. Right now $A(v) = \{P_1, P_3\}$ and $A(u) = \{P_2, P_4\}$, such that $e_{(u,v)}$ will be located on one of the four partitions. In Figure 4, we use a dash line to express $e_{(u,v)}$'s four possible assignments. Considering that $e_{(u,v)}$ is assigned to P_1. We can see that one vertex x appears in both $N(v)$ and P_1. According to our strategy, $rank(P_1) = 1$. Analogously, we can get that $rank(P_2) = 0$, $rank(P_3) = 2$ and $rank(P_4) = 1$. Therefore, in line 12, $S = \{P_2, P_3, P_4\}$ is obtained in this case. If the balance factor P_{bal} does not play a major role, then $id = 3$ is obtained in line 13.

Note that if there are no more edges arriving, *WSGP* successively pops edge from w and then carries out the assignment by directly executing lines 4-12 in Figure2.

C. The costs of WSGP

We analyze the time complexity of *WSGP* through a comparison with *Oblivious*, *i.e.*, focusing on the extra overhead for settling each edge. *Oblivious* is a one-pass streaming algorithm guaranteeing that each edge is processed only once. *WSGP* could be seen as a two-pass streaming algorithm as each buffered edge needs to be processed again. Hence, the extra overhead of *WSGP* comes from the operations after an edge is popped. Let us take edge $e_{(u,v)}$ as an example. After $e_{(u,v)}$ is popped, the major time consumption arises from computing $rank$ for each candidate partition (lines 2-3, 7-9, Figure 3), which could be modeled by $|A(u)||N(v)|+|A(v)||N(u)|$. Therefore, the extra overhead for settling each edge is upper-bounded by $O(|w||P|)$, as $|A(u)| \leq |P|$, $|A(v)| \leq |P|$, $|N(v)| \ll |w|$ and $|N(u)| \ll |w|$.

V. Evaluation

This section presents the experimental results for *WSGP* approach compared with other streaming edge partitioning (SEP) algorithms. The evaluation was performed on different degree distribution graphs.

A. Datasets

Many graph datasets are used in previous work [13], [15], [19], [21]. In order to build a fair and comparable result, we selected the four most common real-world graphs that capture different characteristics for degree distribution. We also used two additional graphs in order to compare with the results extracted from other articles. Table. I shows the statistics of the 4 datasets.

Table I: Statistics for real-world datasets

Dataset	$\|V\|$	$\|E\|$
dblp [1]	317,080	1,049,866
web-google [23]	875,713	4,322,051
com-liveJournal [22]	3,997,962	34,681,189
com-orkut [1]	3,072,441	117,185,083

Figure 5 provides the degree of vertices, the number of vertices with the same degree, the percentage of each degree and a regression line of degree distribution. We observed that DBLP and WEB-GOOGLE datasets have many low degree vertices. Compared with DBLP and WEB-GOOGLE, COM-LIVEJOURNAL and COM-ORKUT have less low degree vertices and follow a heavy-tailed distribution.

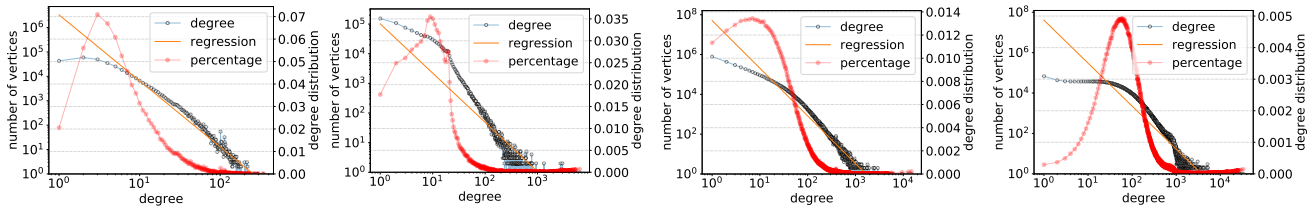
B. Partition tool and Environment setup

In order to study the performance of the state-of-the-art SEP algorithms and compare them to our approach, we build a low cost window streaming partitioner *WStool* based on the approach provided in [21]. The motivation of developing such a stand-alone partition tool is to run the SEP algorithms in an isolate environment with efficient implementation, not only avoiding any extra overhead of resource management and scheduling from the downstream graph computing systems [2], [5], [22], but also minimizing the running time.

Table II: Preprocessing time of *VGP* (seconds)

dblp	web-Google	com-LiveJournal	com-Orkut
1	7	81	327

There are three major differences compared with the *HDRF* partitioner *VGP* [21]. Firstly, *VGP* needs to load



(a) dblp (b) web-google (c) com-livejournal (d) com-orkut

Figure 5: Characteristics for real-world datasets

the full graph into memory such that it is not suitable to process the large-scale graphs. Table II shows the preprocessing time of *VGP*. Our implementation *WStool* loads and processes each edge on the fly. Hence, compared with *VGP*, *WStool* removes the preprocessing time and reduces memory footprint. Secondly, we implemented a more efficient data-structure (*e.g.* bitset) to store the information of replicas that allows to further reduce memory footprint and speedup the calculation. Lastly, we also eliminated the lock mechanism which is used for parallel processing. In this paper, all experiments used one thread.

The number of partitions is set to 30 except the experiments dedicated to varying this parameter. Figure 6 plots the running time (without preprocessing time) of *WStool* and *VGP* partitioner for a variety of real-world datasets.

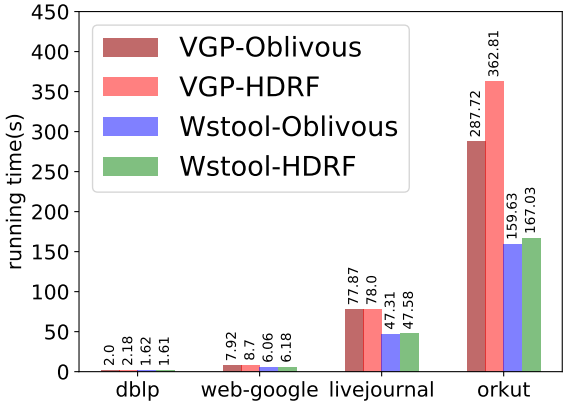


Figure 6: Graph partitioning time of *VGP* and *WStool*

As expected, *WStool* processes much faster than *VGP*. In this paper, all experiments are deployed on a single server with the following specifications: 2 $\times$ Intel(R) Xeon(R) CPU @ 3.2GHz, 96GB of RAM, Ubuntu 18.04. *WStool* is implemented with Java 1.8.

C. Metrics

We used two metrics to measure the quality of graph partitioning algorithms: Replication Factor (RF) and Load Balancing Index (LBI).

Replication Factor: this metric is defined in Equation 1 (in Section III.A). RF describes the average number of replicas per vertex. RF corresponds to the communication overhead for synchronization in graph computing phase. Generally, graph computing phase is composed of multiple iterations. Each vertex exchanges information by message passing or memory sharing in each iteration [4]). So the higher the RF is, the more messages arise over the network, which slows down the whole computing phase.

Load Balancing Index: this metric is defined as follows:

$$LBI = \frac{\max |P_i|}{\frac{|E|}{|P|}} \tag{6}$$

A partition can be seen as an isolate computing node in computing phase. A node hosts a portion of edges received from the partitioning procedure. The node having the largest number of edges decides the max processing time.

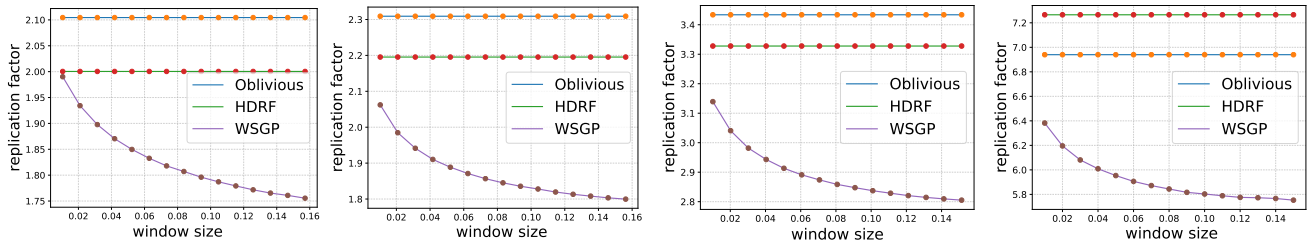
D. Input Order

The streaming partitioning algorithms have a common feature that their performance is particularly sensitive to the input edge order, such that it is difficult to propose an "one-fit-for-all" streaming partitioning algorithm. For example, if the input edges arrive according to a BFS (Breadth First Search) or a DFS (Depth First Search) order, the greedy-based algorithms that prefer to assign successive edges into a single partition, can provide a bad RF. Enlarging the balance factor [21] is a possible way to break the tie. However, we believe that for a particular order of edges, the best choice is to design a particular partitioning method. To avoid the ordering problem discussed above, we shuffled each dataset randomly before running the experiments. Moreover, we tested several random orders and the results show little difference.

E. Window size

In this section, we evaluate the proposed *WSGP* algorithm with different window sizes and compare its performance with other SEP algorithms on real-world graphs.

Replication factor: Figure 7 shows the replication factor of *Oblivious*, *HDRF* and *WSGP* algorithms in different window size settings (the x-axis is the percentage of the buffered edges). *Oblivious* and *HDRF* have different partitioning qualities in the four datasets. In general, *HDRF* outperforms



(a) dblp (b) web-google (c) com-livejournal (d) com-orkut

Figure 7: Replication factor in different windows sizes

Oblivious. However, *Oblivious* has a lower replication factor than *HDRF* on COM-ORKUT.

As for *WSGP*, when the window size approaches to 0, *WSGP* shows a similar behavior to *Oblivious*. By increasing the window size, *WSGP* is able to get more information that helps to make good partitioning decisions. In general, the larger the window size is, the better performance *WSGP* achieves. Note that, the slope of the replication factor curve is slowly decreasing. In our experiments, when the window size is around in average $25\%|E|$, the slope of the replication factor curve approaches to 0.

Load balancing index: Table III reports the results of load balancing index. All algorithms yield a good load balancing result. For our algorithm *WSGP*, the load balancing index increases slightly when the window size grows. In Table III, we gives the result of *WSGP* by setting $|w| = 15\%|E|$. In [21], the authors concluded that the value of parameter λ(Equation 5) affects the load balancing result. Note that $\lambda \to \infty$, the $P_{bal}(P_i)$ in Equation 5 actually dominates the score(3 line 13), our algorithm becomes meaningless. According to our study, we assign $\lambda = 1.1$.

Table III: Load balancing index

	dblp	web-google	com-livejournal	com-orkut
Oblivious	$\leq 1‰$	$\leq 1‰$	$\leq 1‰$	$\leq 1‰$
HDRF	$\leq 1‰$	$\leq 1‰$	$\leq 1‰$	$\leq 1‰$
WSGP	$\leq 1‰$	$\leq 1‰$	$\leq 1‰$	$\leq 8‰$

Partitioning time: Figure 8 shows the average throughput (edges/millisecond) for all algorithms (the x-axis is the percentage of the buffered edges). We can see that as the window size increases, the throughput of *WSGP* slowly drops due to more computation needed. Oblivious and HDRF do not impact by window size and have a better throughput than *WSGP*. We consider this as a reasonable tradeoff between throughput and RF. Many previous work [13], [15], [19], [21] demonstrated that the communication overhead in the graph computation phase is strongly affected by the replication factor achieved: even RF is increased by a little, the computing time, due to thousands of graph computing iterations, increases a lot. In particular, reducing communication cost is rather important in geo-graphically processing. Therefore, we believe that in the graph partitioning stage it is worth to have a one time cost to minimize RF: the extra overhead of *WSGP* can be well compensated in the following graph computing phase.

F. The number of partitions

In this section, we evaluated the performance of increasing the number of partitions on a wide range from 2 to 64. The window size $|w|$ is fix to $3\%|E|$.

Replication factor: Figure 9 shows the results in different number of partitions. The replication factor of all algorithms show a slowly increasing trend when increasing the number of partitions. *WSGP* outperforms all the comparison algorithms. Since *WSGP* leverages a bounded window to collect more information from the incoming edges, it greedily maximizes the potential data locality. In particular, increasing the number of partitions, it provides more opportunities for *WSGP* to buffer edges in the window. As shown in Figure 9, the gap between *WSGP* and oblivious/hdrf are getting larger when the partition number increases.

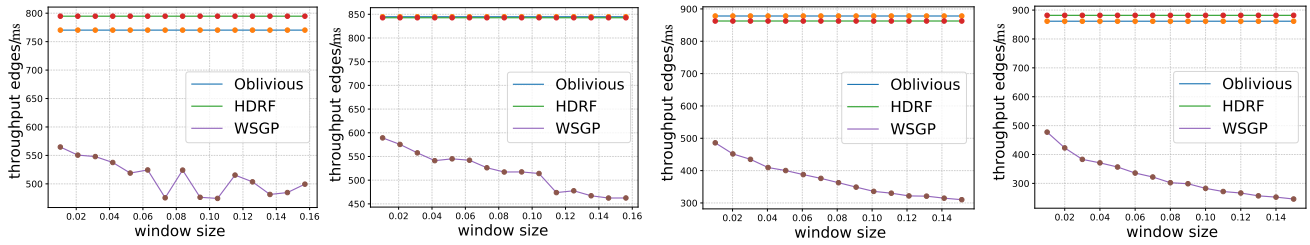
Furthermore, we show another example that how the parameter λ affects RF. We added TWITTER a larger social graph. Figure 10 shows the results of RF for all algorithms. With increasing the value of λ, the value of RF of *Oblivious* increases significantly, the RF of *HDRF* and *WSGP* stay invariant. We also extends the test to other graphs. The results consistently report that *WSGP* is less sensitive to parameter λ.

Load balancing index: All algorithms yield almost perfectly balanced partitions($LBI \leq 1‰$).

Partitioning time: Figure 11 shows the average throughput (edges/millisecond). All algorithms slow down as the number of partitions increases. *Oblivious* and *HDRF* have better performance than *WSGP* as they perform less computation.

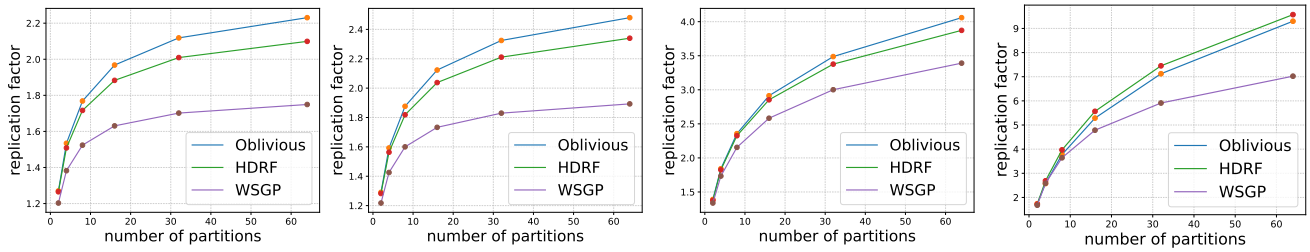
G. ADWISE

In this section, we compare with another window-based streaming graph partitioning ADWISE [19], *WSGP* is sometime better in RF but with much less resources. We used the same datasets BRAIN [24] and COM-ORKUT from article [19] as the input stream for fair comparison. The BRAIN dataset



(a) dblp (b) web-google (c) com-livejournal (d) com-orkut

Figure 8: Throughput in different window sizes



(a) dblp (b) web-google (c) com-livejournal (d) com-orkut

Figure 9: Replication Factor in different number of partitions

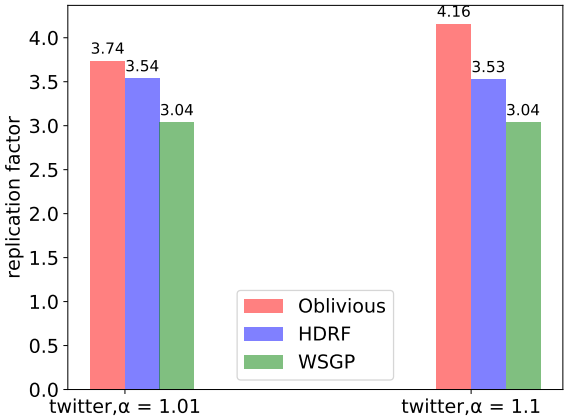


Figure 10: Twitter dataset

consists of 734,561 vertices and 165,916,089 edges, it is extremely skewed and follows a power-law degree distribution. Our experiments follow the configuration provided in [19] such that the number of partitions $|P| = 32$ and $\lambda = 1.1$.

Since [19] did not mention the window size, we tested RF under different throughput values. We then picked out the results which are the closest to the one provided in [19]. Figure 12 shows the results with respect to RF of *WSGP* and *ADWISE*. The results of *ADWISE* are plotted according to [19] (from Figures 7 and 8 in Section IV in [19]).

Replication factor: *WSGP* is worse than *ADWISE* on BRAIN and outperforms *ADWISE* on COM-ORKUT graph.

Load balancing index: Although there is no exact LBI in [19] for each dataset, the authors indicated $LBI \leq 5\%$. *WSGP* provides well balanced partitions where $LBI \leq 1\%$.

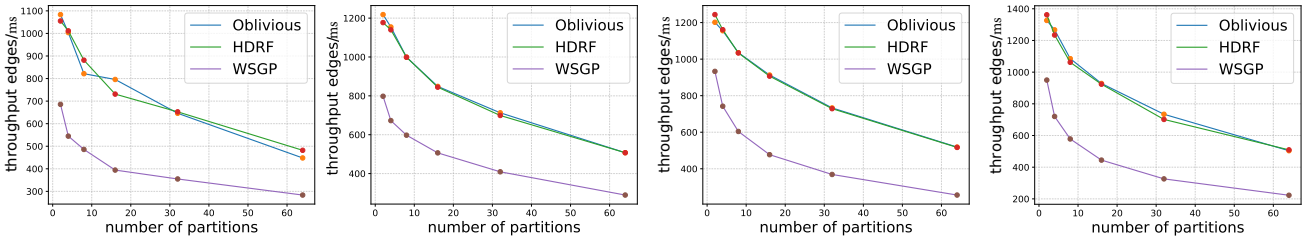
Partitioning time: *ADWISE* spent 281s on BRAIN and 329s on COM-ORKUT. *WSGP* required 306s on BRAIN and 349s on COM-ORKUT. Yet, it should be highlighted that the results of *ADWISE* employ 8 nodes with 8 Intel(R) Xeon(R) CPU cores (3.0GHz, 6144 KB cache) and 32GB RAM per node, while *WSGP* algorithm is processed by a single core on a single node with only 8.4G memory consumption (see Section V-B). Thanks to adopt the efficient implementation *WStool* and lower complexity of *WSGP*, the computation overhead and memory consumption of *WSGP* is much lower than *ADWISE* with a similar running time, which implies *WSGP* is more adaptive to process large-scale graphs with much less resources. Moreover, by increasing the window size, *WSGP* can further reduce the RF to 3.88 on BRAIN with an acceptable time cost (408s).

H. Summary of results

We summarize the results of our *WSGP* algorithm for the studied aspects. Using *WSGP* effectively reduces the number of replicas. By varying the size of windows and the number of partitions, *WSGP* consistently gives a good load balancing for all graph datasets. The above results rely on a reasonable window size with an acceptable time cost.

VI. Conclusion and Future Work

Distributed graph partitioning for very large-size graphs targets both load balancing among computing units and runtime minimization. Replicating the high-degree vertices



(a) dblp (b) web-google (c) com-livejournal (d) com-orkut

Figure 11: Throughput in different number of partitions

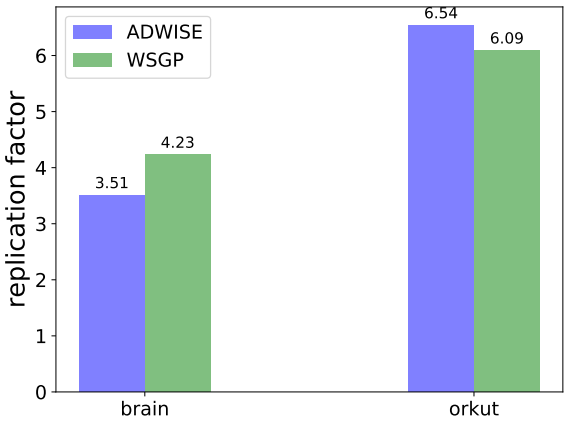


Figure 12: Replication factor of WSGP and ADWISE

is a good heuristic and existing work has shown that it is useful for graph partitioning problem. In this paper, we demonstrated that there is an alternative way to achieve the same goal. With the low cost partitioner *Wstool*, the proposed *WSGP* algorithm leveraged a size-bounded window to optimize the decision for edge assignment efficiently. Our experimental results show that *WSGP* consistently achieves a smaller replication factor than the state-of-the-art algorithms (up to 23%), with a limited impact on memory utilization and runtime. We consider implementing a parallel version of our algorithm and integrating it to a distributed graph computing system as our future work.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant 61902063, and by the Provincial Natural Science Foundation of Jiangsu, China under Grant BK20190342. This work is also supported by China Postdoctoral Science Foundation under Grant 2020M671669. Early experiments that help us to understand partitioning problem were partially carried out using the Grid'5000 testbed (supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations, see https://www.grid5000.fr).

REFERENCES

[1] J. Yang and J. Leskovec, "Defining and Evaluating Network Communities Based on Ground-Truth," in *IEEE International Conference on Data Mining*, 2012, pp. 745–754.

[2] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD), Indianapolis, Indiana, USA, June 6-10*, 2010, pp. 135–146.

[3] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (UAI), Catalina Island, CA, USA, July 8-11*, 2010, pp. 340–349.

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

[5] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 599–613.

[6] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[7] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM international conference on Web search and data mining (WSDM)*, 2014, pp. 333–342.

[8] M. Kim and K. S. Candan, "Sbv-cut: Vertex-cut based graph partitioning using structural balance vertices," *Data and Knowledge Engineering*, vol. 72, pp. 285–303, 2012.

[9] U. Feige, M. Hajiaghayi, and J. R. Lee, "Improved approximation algorithms for minimum weight vertex separators," *SIAM Journal on Computing*, vol. 38, no. 2, pp. 629–657, 2008.

[10] K. Andreev and H. Räcke, "Balanced graph partitioning," in *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2004, pp. 120–124.

[11] I. Stanton, "Streaming balanced graph partitioning algorithms for random graphs," in *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Portland, Oregon, USA, January 5-7*, 2014, pp. 1287–1301.

[12] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), Beijing, China, August 12-16*, 2012, pp. 1222–1230.

[13] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li, "Graph edge partitioning via neighborhood heuristic," in *Proceedings of the 23rd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD), Halifax, NS, Canada, August 13-17*, 2017, pp. 605–614.

[14] M. Hanai, T. Suzumura, W. J. Tan, E. S. Liu, G. Theodoropoulos, and W. Cai, "Distributed edge partitioning for trillion-edge graphs," *Proc. VLDB Endow.*, vol. 12, no. 13, pp. 2379–2392, 2019.

[15] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: An experimental study," *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1590–1603, 2018.

[16] N. Jain, G. Liao, and T. L. Willke, "Graphbuilder: scalable graph etl framework," in *Proceedings of the First International Workshop on Graph Data Management Experiences and Systems (GRADES), New York, NY, USA, June 24*, 2013, p. 4.

[17] C. Xie, L. Yan, W. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS), December 8-13, Montreal, Quebec, Canada*, 2014, pp. 1673–1681.

[18] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM International Conference on Information and Knowledge Management (CIKM), Melbourne, VIC, Australia, October 19 - 23*, 2015, pp. 243–252.

[19] C. Mayer, R. Mayer, M. A. Tariq, H. Geppert, and K. Rothermel, "ADWISE: Adaptive Window-Based Streaming Edge Partitioning for High-Speed Graph Processing," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018.

[20] B. Florian, L. Marc, and V. Milan, "Balanced graph edge partition," in *Proceedings of the 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2014, pp. 1456–1465.

[21] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "Hdrf: Stream-based partitioning for power-law graphs," in *Proceedings of the 24th ACM International Conference on Information and Knowledge Management (CIKM), Melbourne, VIC, Australia, October 19-23*, 2015.

[22] A. Pacaci and M. T. Özsu, "Experimental analysis of streaming algorithms for graph partitioning," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1375–1392.

[23] 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," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2008.

[24] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com