Distributed Vertex-Cut Partitioning

Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Seif Haridi

Distributed Vertex-Cut Partitioning

Fatemeh Rahimian[1,2], Amir H. Payberah[1]
Sarunas Girdzijauskas[2], and Seif Haridi[1]

[1] Swedish Institute of Computer Science {fatemeh,amir,seif}@sics.se
[2] KTH - Royal Institute of Technology sarunasg@kth.se

Abstract. Graph processing has become an integral part of big data analytics. With the ever increasing size of the graphs, one needs to partition them into smaller clusters, which can be managed and processed more easily on multiple machines in a distributed fashion. While there exist numerous solutions for edge-cut partitioning of graphs, very little effort has been made for vertex-cut partitioning. This is in spite of the fact that vertex-cuts are proved significantly more effective than edge-cuts for processing most real world graphs. In this paper we present Ja-be-Ja-vc, a parallel and distributed algorithm for vertex-cut partitioning of large graphs. In a nutshell, Ja-be-Ja-vc is a local search algorithm that iteratively improves upon an initial random assignment of edges to partitions. We propose several heuristics for this optimization and study their impact on the final partitioning. Moreover, we employ simulated annealing technique to escape local optima. We evaluate our solution on various graphs and with variety of settings, and compare it against two state-of-the-art solutions. We show that Ja-be-Ja-vc outperforms the existing solutions in that it not only creates partitions of any requested size, but also requires a vertex-cut that is better than its counterparts and more than 70% better than random partitioning.

1 Introduction

A wide variety of real-world data can be naturally described as graphs. Take for instance communication networks, social networks, biological networks, etc. With the ever increasing size of such networks, it is crucial to exploit the natural connectedness of their data in order to store and process them efficiently. Hence, we are now observing an upsurge in the development of distributed and parallel graph processing tools and techniques. Since the size of the graphs can grow very large, sometimes we have to partition them into multiple smaller clusters that can be processed efficiently in parallel. Unlike the conventional parallel data processing, parallel graph processing requires each vertex or edge to be processed in the context of its neighborhood. Therefore, it is important to maintain the locality of information while partitioning the graph across multiple (virtual) machines. It is also important to produce equal size partitions that distribute the computational load evenly between clusters.

Graph partitioning is a well known NP-Complete problem in graph theory. In its classical form, graph partitioning usually refers to *edge-cut* partitioning, that

(a) (b)

Fig. 1. Partitioning a graph into three clusters, using (a) edge-cut partitioning and (b) vertex-cut partitioning

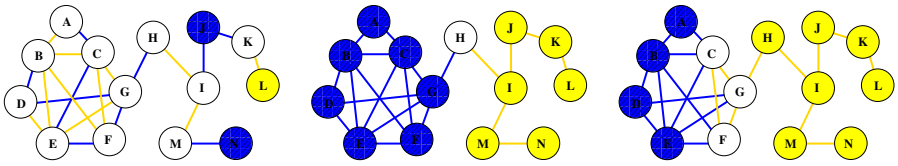
is, to divide vertices of a graph into disjoint clusters of nearly equal size, while the number of edges that span separated clusters is minimum. However, tools that utilize edge-cut partitioning do not achieve good performance on real-world graphs (which are mostly power-law graphs) [1–3], mainly due to unbalanced number of edges in each cluster. In contrast, both theory [4] and practice [5, 6] prove that power-law graphs can be efficiently processed in parallel if *vertex-cuts* are used.

A *vertex-cut* partitioning divides edges of a graph into equal size clusters. The vertices that hold the endpoints of an edge are also placed in the same cluster as the edge itself. However, the vertices are not unique across clusters and might have to be *replicated* (cut), due to the distribution of their edges across different clusters. A good vertex-cut is one that requires minimum number of replicas. Figure 1 illustrate the difference between these two types of partitioning.

While there exist numerous approximate solutions for edge-cut partitioning, very little work has investigated vertex-cut partitioning. Figure 2 shows a graph with three different vertex-cut partitionings. The graph edges are partitioned into two clusters. Two colors, yellow and blue, are representing these two partitions. Vertices that have edges of one color only, are also colored accordingly, and the vertices that have to be replicated are colored white. A very naïve solution is to randomly assign edges to partitions. As shown in Figure 2(a), nearly all the vertices have edges of different colors, thus, they have to be replicated in both partitions. Figure 2(b) illustrates what happens if we use an edge-cut partitioner, and then assign the cut edges to one of the partitions randomly. As shown, the vertex-cut improves significantly. However, the number of edges in the partitions is very unbalanced. What we desire is depicted in Figure 2(c), where the vertex-cut is kept as low as possible, while the size of the partitions, with respect to the number of edges, is balanced.

An alternative solution for vertex-cut partitioning of a graph G is to transform it to its corresponding line graph $L(G)$ (where $L(G)$ represents the adjacencies between edges of G) and then use an edge-cut partitioning algorithm. However, in most real-world graphs the number of edges are orders of magnitude higher than the number of vertices, thus, the line graph often has a significantly higher number of vertices. Consequently, the complexity of the partitioning could grow drastically. It is, therefore, necessary to devise algorithms that performs the vertex-cut partitioning on the original graph.

In this paper, we present a distributed vertex-cut partitioning algorithm, called Ja-be-Ja-vc, based on local search optimization, which is mainly inspired by our previous work for edge-cut partitioning [7]. The algorithm starts with a random assignment of edges to partitions. For simplicity we represent each



(a) A random vertex-cut. Edges are evenly distributed, but nearly all the vertices have to be replicated.

(b) Partitioning based on edge-cut and then assigning the cut edges randomly to one partition. Only one vertex is replicated, but edges are not evenly distributed.

(c) A good vertex-cut partitioning. Three vertices are replicated and the number of edges in the two partitions are balanced.

Fig. 2. Vertex-cut partitioning into two clusters. The color of each edge/vertex represents the partition it belongs to. The white vertices belong to both partitions.

partition with a distinct *color*. Over time, vertices exchange information (about the color of their edges) with each other and try to locally reduce the vertex-cut, by negotiating over the assignment of their edges to partitions. Every vertex attempts to assign all its edges to the same partition (same color), because this means the vertex does not have to be cut. If this case is not possible due to the contention between neighboring vertices, then the vertex tries to have the minimum number of distinct assignments. Two vertices will decide to exchange the colors of their candidate edges, if the vertex-cut can be reduced. Otherwise, the edge colors are preserved.

The aforementioned heuristic is likely to get stuck in local optima due to initial random partitioning and the nature of the problem, which is NP-Complete. Thus, we employ the well-known simulated annealing technique [8] to escape local optima and find a better vertex-cut. Note, Ja-be-Ja-vc will always maintain the initial distribution of partition sizes. That is, if the initialization is uniformly random, the partition sizes are expected to be balanced. If we require to have partitions of a different distribution, e.g., one partition twice as big as the others, then we only need to change the initialization step to produce the required distribution.

We observe through experiments, that Ja-be-Ja-vc produces quality partitions on several large graphs and scales well with varying number of partitions. We also study the trade-off between the vertex-cut and the computation cost, in terms of the number of iterations to compute the partitioning. Finally, we compare Ja-be-Ja-vc to the state-of-the-art vertex-cut partitioner [9], as well as the state-of-the-art edge-cut partitioner [7], and study their existing the trade-offs. We show that Ja-be-Ja-vc consistently outperforms [7] and [9] on producing requested size partitions, while not sacrificing the vertex-cut. Even for varying number of partitions, Ja-be-Ja-vc always reduces the vertex-cut to lower than 30% and down to 10% for some graphs.

2 Problem statement

We are given an undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. A k-way balanced *vertex-cut* partitioning divides the

set of edges E into k subsets of equal size, where k is an input parameter. Each partition also has a subset of vertices that hold at least one of the edges in that partition. However, vertices are not unique across partitions, that is, some vertices may appear in more than one partition, due to the distribution of their edges across several partitions. A good edge partitioning strives to minimize the number of vertices that belong to more than one partition.

A k-way balanced vertex-cut partitioning can be given with the help of a partition function $\pi : E \rightarrow \{1, \ldots, k\}$ that assigns a *color* to each edge. Hence, $\pi(e)$, or π_e for short, refers to the color of edge e. Edges with the same color form a partition. We denote the set of edges that are connected to vertex p by E_p. Accordingly, $E_p(c)$ indicates the subset of edges of p that have color c:

$$E_p(c) = \{e \in E_p : \pi_e = c\} \tag{1}$$

We refer to $|E_p(c)|$ as the *cardinality* of color c at vertex p. Then, the *energy* of a vertex p is shown with $\gamma(p, \pi)$ and it is defined as the number of different colors that has a cardinality greater than zero.

$$\gamma(p, \pi) = \sum_{|E_p(c)|>0} 1, \forall\ c \in \{1, \ldots, k\} \tag{2}$$

In other words, the energy of a vertex p for a partition function π is the number of different colors that are assigned to the edges of p, which is equivalent to the number of required replicas (vertex-cut) for p. The energy of the graph is sum of the energy of all its vertices:

$$\Gamma(G, \pi) = \sum_{p \in V} \gamma(p, \pi) \tag{3}$$

Now we can formulate an optimization problem as follows: find the optimal partitioning π^* such that:

$$\pi^* = \arg\min_{\pi} \Gamma(G, \pi) \tag{4}$$

$$s.t.\ \ |E(c_1)| = |E(c_2)|, \forall\ c_1, c_2 \in \{1, \ldots, k\} \tag{5}$$

where $|E(c)|$ is the number of edges with color c. Note, in all practical cases the second condition is relaxed, such that it requires partitions of **approximately** equal size. This is important, because the number of edges of the graph is not necessarily a multiple of k. Therefore, throughout this paper, we address the relaxed version of the problem.

3 Solution

In order to partition the edges of a given graph, we use an approach inspired by Ja-be-Ja [7], our previous work on edge-cut partitioning. Our algorithm, called Ja-be-Ja-vc, is vertex-centric and fully distributed, and no central point with

Algorithm 1 Optimization sketch

```
1: procedure RUN()
2:     if self.isInternal() is not TRUE then
3:         selfEdge ← self.selectEdge()                          ▷ Select an edge (Algorithm 2)
4:         candidates ← self.selectCandidates()  ▷ Select a list of candidate vertices (Algorithm 3)
5:         for all partner in candidates do      ▷ Look for a swap partner among the candidates
6:             if partner is not internal then
7:                 if policy is DominantColor then
8:                     selfColor ← self.getDominantColor()
9:                     partnerColor ← partner.getDominantColor()
10:                    if selfColor ≠ partnerColor and partner.hasEdge(selfColor) then
11:                        partnerEdge ← partner.selectEdge(selfColor)
12:                        swapColor(selfEdge, partnerEdge)
13:                        break
14:                    end if
15:                else                                  ▷ If the policy is based on Edge Utility
16:                    partnerEdge ← partner.selectEdge()
17:                    if swapUtility(selfEdge, partnerEedge) > 0 then
18:                        swapColor(selfEdge, partnerEedge)
19:                        break
20:                    end if
21:                end if
22:            end if
23:        end for
24:    end if
25: end procedure
```

a global knowledge is required. Vertices of the graph execute the algorithm independently and iteratively. In this algorithm, initially a random color is assigned to each edge of the graph. This is equivalent to a random assignment of edges to partitions. Then we allow vertices to exchange the color of their edges, provided that the exchange leads to a better cut of the graph. In the initialization step we can control the required partition size distribution. If edges are initially assigned to partitions uniformly at random, then size of the partitions is expected to be equal. We could also use any other distribution for initial edge assignment, and Ja-be-Ja-vc guarantees to preserve this distribution during the course of optimization.

The optimization step is illustrated in Algorithm 1. In each iteration, first a vertex checks whether or not it is *internal*. An internal vertex is a vertex that is surrounded with the edges of the same color (i.e., $\gamma(p, \pi) = 1$). If the vertex is internal, it does not need to perform any optimization and waits for its turn in the next round. Otherwise, the vertex proceeds with the following three steps: (i) *edge selection*, (ii) *partner selection*, and (iii) *swap*. Each of these steps could be realized by means of various policies. Here we explain a few possible policies for these steps separately.

3.1 Edge Selection

In this step a vertex selects one of its edges for color exchange. We consider two policies for edge selection: (i) *random* and (ii) *greedy*. In the random policy, a vertex chooses one edge of its edges randomly. Although random selection is very straight forward, it will not lead our local search in the right direction. Consider, for example, a vertex with a majority of edges with `blue` color and just very few

Algorithm 2 Edge Selection

```
1: procedure selectEdge(color)
2:     if color is null then
3:         color ← self.getColorWithMinCardinality()
4:     end if
5:     edges ← self.getEdges(color)
6:     return edges.getRandomElement()
7: end procedure
```

Algorithm 3 Partner Selection

```
1: procedure selectCandidates()
2:     candidates ← self.getNeighbours().getSubset()          ▷ a subset of direct neighbors
3:     candidates.add(getRandomVertices())          ▷ a subset of random vertices from the graph
4:     return candidates
5: end procedure
```

Algorithm 4 Calculate Swap Utility

```
1: procedure swapUtility(edge1, edge2)
2:     c1 ← edge1.getColor()
3:     c2 ← edge2.getColor()
4:     u1c1 ← getEdgeValue(edge1.src, edge1.dest, c1);          ▷ utility of edge1 before swap
5:     u2c2 ← getEdgeValue(edge2.src, edge2.dest, c2);          ▷ utility of edge2 before swap
6:     u1c2 ← getEdgeValue(edge1.src, edge1.dest, c2);          ▷ utility of edge1 after swap
7:     u2c1 ← getEdgeValue(edge2.src, edge2.dest, c1);          ▷ utility of edge2 after swap
8:     return ((u1c2 + u2c1) × T_r) − (u1c1 + u2c2)
9: end procedure
```

`red` edges. If this vertex selects a random edge for a color exchange, it is more likely that the vertex selects a `blue` edge (because it has a majority of `blue` edges), and such selection is not in the interest of the vertex. Whereas, if the vertex selects an edge with `red` color, it will have a higher chance of unifying the color of its edges. With a greedy policy, a vertex selects one of its edges, e.g., e, which has a color with minimum cardinality:

$$e \in E_p(c^*), \quad c^* = arg\min_c |E_p(c)|$$

Since random policy is ineffective in our optimization process, we only consider the greedy edge selection in our experiments. Algorithm 2 describes how an edge is selected with this policy.

3.2 Partner Selection

In this step a vertex selects a subset of other vertices from the graph as candidates for a color exchange. A vertex considers two sets of vertices for partner selection: (i) *direct neighbors*, and (ii) *random* vertices. Direct neighbors of a vertex p are a set of vertices that are directly connected to p. Every vertex has knowledge about its directly connected neighbors. Since some vertices may have a very large degree, this local search could become exhaustive for such vertices if they have to check each and every of their neighbors. Hence, vertices only consider a fixed-size random subset of their direct neighbors.

Vertices choose their partners for color exchange first from their direct neighboring. To increase the chance of finding a swap partner, vertices also consider a few random vertices from the graph. This random subset of vertices could be acquired through a peer sampling service [10–13] or random walk [14] that is continuously running on all the graph vertices. In our previous work [7] we have extensively discussed these two vertex selection policies (i.e., direct and random) and how they can be realized. Moreover, we showed that the best outcome is achieved while the *hybrid* of direct and random neighbors are taken into account. We, therefore, use the hybrid policy, where as shown in Algorithm 1 (Line 5) and Algorithm 3 vertices first check a subset of their direct neighbours, and then if they do not succeed, they check some random vertices.

3.3 Swap Heuristics

To make a decision for color exchange we consider two heuristics: (i) *dominant color* and (ii) *edge utility*.

Dominant Color (DC). We define the *dominant color* of a vertex p as the color with the maximum cardinality at p. That is:

$$c_p^* = arg\max_c |E_p(c)|$$

With this heuristic, a vertex p looks for a partner which (i) is not internal, and (ii) has an edge with vertex p's dominant color. If vertex p finds such a vertex, it exchanges with that vertex one of its non-dominant colors for the dominant color. In other words, every vertex tries to unify the color of its edges, in order to reduce its energy. Since the global energy of the graph is sum of all vertices' energy, this optimization has the potential to lead us toward a globally optimal state. Although condition (i) prevents disturbing those vertices that are already stabilized, vertices are still acting very greedily and do not consider the benefits of the other endpoint of the edge that they are negotiating over. Consequently, this policy could end up in contention between neighboring vertices, and the color of some edges might fluctuate. In Section 4, we will study the evolution of the partitioning under such policy.

Edge Utility (EU). An alternative policy is to assign a *utility* value to every edge based on the cardinality of the colors at its endpoints. The main idea of this heuristic is to check that whether exchanging the color of two edges decreses the energy of the their connected vertices or not. If it does, two edges swap their colors, otherwise they keep them. To every edge e_{pq} (with two endpoints p and q) we assign a value v, with respect to color c, that indicates the relative number of neighboring edges of e with color c. That is:

$$v(e,\ c) = \begin{cases} \frac{|E_p(c)|-1}{|E_p|} + \frac{|E_q(c)|-1}{|E_q|} & if\ c = \pi_e \\ \frac{|E_p(c)|}{|E_p|} + \frac{|E_q(c)|}{|E_q|} & otherwise \end{cases}$$

Graph Name	$\|V\|$	$\|E\|$	power-law	Avg. Clustering Coeff.	Diameter	Source
Data	2851	15093	no	0.486	79	Walshaw Archive [15]
4elt	15606	45878	no	0.408	102	Walshaw Archive [15]
Astroph	17903	196972	yes	0.6306	14	Stanford Snap Datasets [16]
Email-Enron	36692	367662	yes	0.4970	11	Stanford Snap Datasets [16]

Table 1. Datasets used in the experiments

Note, in the first case, $E_p(c)$ and $E_q(c)$ include edge e, and that is why we need to decrement them by one. Next, the objective is to maximize the overall value of edges during the color exchange process. More precisely, vertex p exchanges the color of its edge e_{pq} with the color of another edge $e'_{p'q'}$, if and only if:

$$\upsilon(e, c') + \upsilon(e', c) > \upsilon(e, c) + \upsilon(e', c')$$

where $c = \pi_e$ and $c' = \pi'_e$. Hence, the swap utility is calculated as follows:

$$utility = (\upsilon(e, c') + \upsilon(e', c)) - (\upsilon(e, c) + \upsilon(e', c'))$$

Simulated Annealing. Since our swap policy is based on local optimization with limited information at each vertex and no central coordinator, it is prone to getting stuck in local optima. Therefore, we need to employ some techniques to help it get out of local optima and move towards better configurations over time. To achieve this goal, we use simulated annealing [8] with constant cool down rate. Two parameters of the simulated annealing are the *initial temperature* T_0 and the *cool down rate* δ. The *temperature* at round r is then calculated as follows:

$$T_r = max(1,\ T_0 - r \cdot \delta)$$

Finally, as shown in Algorithm 4 we bias the utility computation with the value of temperature at each round, as follows:

$$utility = ((\upsilon(e, c') + \upsilon(e', c)) \times T_r) - (\upsilon(e, c) + \upsilon(e', c'))$$

4 Experiments

In this section, we first introduce the datasets and metrics that we used for evaluating our solution. Then, we study the impact of our simulated annealing parameters on the partitioning quality. Next, we show how different policies, introduced in Section 3.3, perform. We also measure the performance of these policies in scale, and compare them to two state of the art solutions.

To evaluate Ja-be-Ja-vc we used four graphs of different nature and size for evaluating our ideas. These graphs and some of their properties are listed in Table 1. Note, graphs Astroph and Email-Enron have power-law degree distribution. We measure the following metrics to evaluate the quality of the partitioning:

- *Vertex-cut*: this metric counts the number of times that graph vertices has to be cut. That is, a vertex with one cut has replicas in two partitions, and a vertex with two cuts is replicated over three partitions. This is an important metric, because when a graph vertices are scattered over several partitions, every computation that involves a modification to a vertex, should be propagated to all the other replicas of that vertex, for the sake of consistency. Therefore, vertex-cut directly affects the required communication cost of the partitioned graph.

– *Normalized vertex-cut*: this metric calculates the vertex-cut of the final partitioning relative to the random partitioning, thus, it shows to what extent the algorithm can reduce the vertex-cut.
– *Standard deviation of partition sizes*: this metric measures the Standard Deviation (STD) of normalized size of the partitions. First, we measure the size of the partitions (in terms of the number of edges) relative to the average (expected) size. Then, we calculate how much the normalized size deviates from the perfect normalized size, i.e., 1.

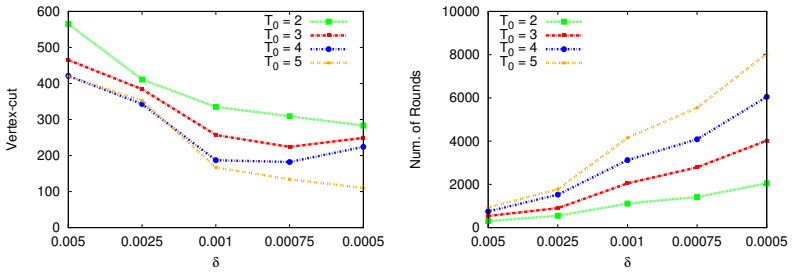
4.1 Experimental setting

We conducted several experiments to tune the two parameters of the simulated annealing, namely T_0 and δ. For these experiments we selected the Data graph (Table 1) and $k = 2$. As shown in Figure 3(a), the vertex-cut decreases when T_0 increases. However, Figure 3(b) illustrates that this improvement is achieved in a higher number of rounds, that is, a bigger T_0 delays the convergence time. Similarly, a smaller δ results in a better vertex-cut, at the cost of more rounds. In other words, T_0 and δ are parameters of a trade-off between vertex-cut and the convergence time and can be tuned based on the priorities of the applications. Moreover, we found out that for a larger k, it is better to choose a smaller δ, because when the number of partitions increases, the solution space expands and it is more likely for the algorithm to get stuck in local optima. Unless otherwise mentioned, in the rest of our experiments $T_0 = 2$ and we use $delta = 0.0005$ for $k = 32$ and $k = 64$, and $\delta = 0.001$ for other values of k. Moreover, each node selects 4 candidates in a round, including three random nodes among its neighbors (line 3 in Algorithm 3) and one random node from the whole graph, provided by the peer sampling service (line 4 in Algorithm 3).

4.2 Performance

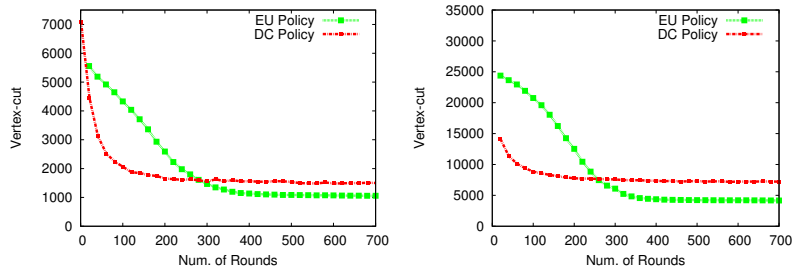
We observe the evolution of vertex-cut over time on different graphs, with two different swap policies: (i) DC, i.e., dominant color, and (ii) EU, i.e., edge utility. For this experiment $k = 4$, and the results are depicted in Figure 4. As shown, the main gain with the DC policy is acquired in the very beginning, when the vertex-cut drops sharply. After the first few iterations, the vertex-cut does not change considerably. In contrast, the EU policy results in a lower vertex-cut, but in a larger number of iterations. It is important to note, the convergence time is independent of the graph size and is mainly determined by the parameters of the simulated annealing. The algorithm converges soon after the temperature reaches value 1.

It is interesting to note, for the first two graphs, the DC policy can produce a better vertex-cut in a short time, but in the long run, the EU policy outperforms it. For the other graphs in Figures 4(d) and 4(c), the EU policy is always performing better. This is due to the different structural properties of these graphs. More precisely, Astroph and Email-Enron are power-law graphs (Figures 4(d) and 4(c)), that is the the degree distribution of graph vertices resembles a power-law distribution. More structural properties of these graphs are listed in Table 1.

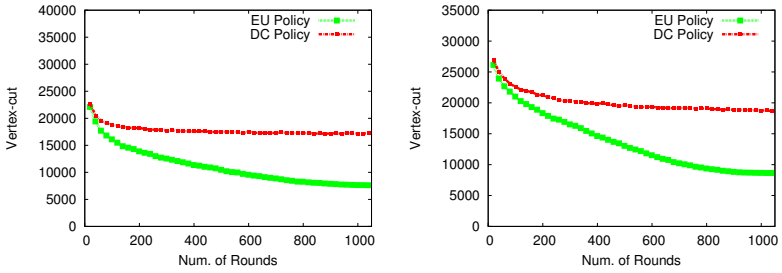


(a) Vertex-cut (b) Rounds to Converge

Fig. 3. Tuning SA parameters on Data graph (K=2)



(a) Data graph ($\delta = 0.0025$) (b) 4elt graph ($\delta = 0.0025$)



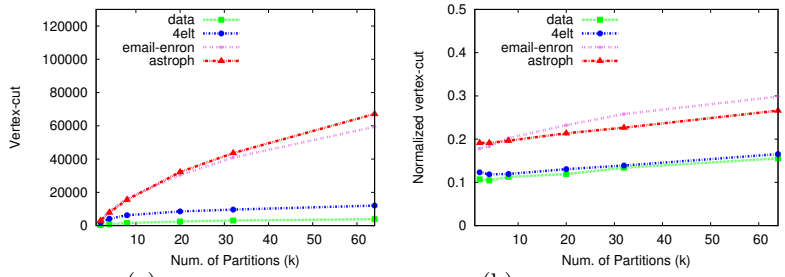
(c) Email graph ($\delta = 0.001$) (d) Astroph graph ($\delta = 0.001$)

Fig. 4. Evolution of vertex-cut with different swap policies (K=4).

We also measure the vertex-cut for various number of partitions. For this experiment, we only use the EU policy. Figure 5(a) depicts how the vertex-cut changes for various number of partitions. To better understand this result, we also report the vertex-cut of JA-BE-JA-VC relative to that of a random partitioning in Figure 5(b). As shown, JA-BE-JA-VC reduces the vertex-cut to nearly 10-15% for Data and 4elt graphs, and to 20-30% for our power-law graphs.

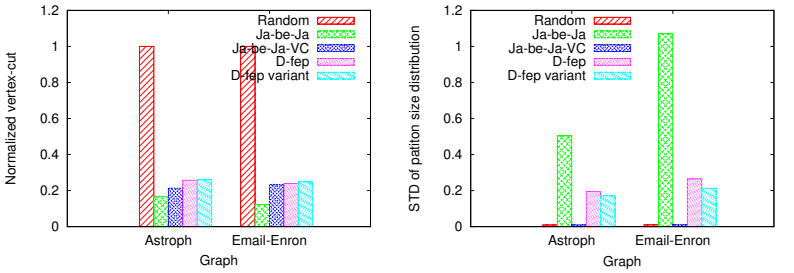
4.3 Comparisons to the state of the art

In this section we show that our solution outperforms two state-of-the-art solutions in that it produces partitions with equal size, while requiring a very low vertex-cut. First, we use an edge-cut partitioner, Ja-be-Ja [7] to partition the graph. Then, the cut edges are randomly assigned to one of the partitions, where their endpoints belong to. This is similar to the example in Figure 2(b). We also compare JA-BE-JA-VC to the state-of-the-art vertex-cut partitioner by Alessio et



(a) Vertex-cuts. (b) Normalized vertex-cut

Fig. 5. The improvements for different number of partitions.



(a) Normalized vertex-cuts. (b) Standard deviation of partition sizes

Fig. 6. Comparisons (k=20)

al. [9], which includes two policies of its own, namely D-fep, and D-fep Variant. This experiment is performed on Astroph and Email-Enron graphs with $k = 20$. To make the comparisons more meaningful, we report the normalized vertex-cut, that is the vertex-cut relative to that of a random partitioning. As shown in Figure 6(a), Ja-be-Ja produces the minimum vertex-cut. However, Figure 6(b) shows that the partition sizes are very unbalanced. The number of cut vertices in D- fep and its variant is more than that of Ja-be-Ja, but their partition sizes are much more balanced. Ja-be-Ja-vc has a better vertex cut than D-fep, while the partition sizes are nearly equal.

As explained in Section 4.1, the convergence time of Ja-be-Ja-vc is independent of the graph size and is mainly affected by the parameters of the simulated annealing process. While this is true for Ja-be-Ja, [9] shows that both D-fep and its variant converge in only very few rounds and produce very good vertex-cuts for graphs Astroph and Email-Enron. However, as depicted in Figure 6(b) these algorithms do not maintain the balance of the partition sizes. In fact, without proper coordination, the standard deviation of the partition size distribution could grow to prohibitively large levels. Ja-be-Ja-vc, however, maintains the initial distribution of edge colors, and can even be used to produce partitions of any desired size distribution, with a better vertex-cut. This comes, however, at the cost of longer running time.

5 Related Work

In this section we study some of the existing work on both edge-cut and vertex-cut partitioning.

5.1 Edge-cut partitioning

A significant number of algorithms exist for edge-cut partitioning [17–23]. These algorithms can be classified into two main categories: (i) centralized algorithms, which assume cheap random access to the entire graph, and (ii) distributed algorithms.

A common approach in the centralized edge-cut partitioning is to use Multilevel Graph Partitioning (MGP) [19]. METIS [20] is a well-known algorithm based on MGP that combines several heuristics during its coarsening, partitioning, and un-coarsening phases to improve the cut size. KAFFPA [23] is another MGP algorithm that uses local improvement algorithms based on flows and localized searches. There exist also other works that combined different metaheuristics with MPG, e.g., Soper et al. [24] and Chardaire et al. [25] used Genetic Algorithm (GA) with MPG, and Benlic et al. [26] utilized Tabu search.

Parallelization is a technique that is used by some systems to speedup the partitioning process. For example, PARMETIS [21] is the parallel version of METIS, KAFFPAE [27] is a parallelized version of its ancestor KAFFPA [23], and [28] is a parallel graph partitioning technique based on parallel GA [29].

Although the above algorithms are fast and produce good min-cuts, they require access to the entire graph at all times, which is not feasible for large graphs. Ja-be-Ja [7] is a recent algorithm, which is fully distributed and uses local search and simulated annealing techniques [8] for graph partitioning. In this algorithm each vertex is processed independently, and only the direct neighbors of the vertex, and a small subset of random vertices in the graph need to be known locally. DIDIC [30] and CDC [31] are two other distributed algorithms for graph partitioning, which eliminate global operations for assigning vertices to partitions. However, they may produce partitions of drastically different sizes.

5.2 Vertex-cut partitioning

While there exist numerous solutions for edge-cut partitioning, very little effort has been made for vertex-cut partitioning. SBV-Cut [32] is one of the recent work for vertex-cut partitioning. The authors proposed a solution to identify a set of balanced vertices that can be used to bisect a directed graph. The graph can then be further partitioned by a recursive application of structurally-balanced cuts to obtain a hierarchical partitioning of the graph.

PowerGraph [5] is a distributed graph processing framework that uses vertex-cuts to evenly assign edges of a graph to multiple machines, such that the number of machines spanned by each vertex is small. PowerGraph reduces the communication overhead and imposes a balanced computation load on the machines. GraphX [6] is another graph processing system on Spark [33, 34] that uses a vertex-cut partitioning to improve its performance.

ETSCH [9] is also a graph processing framework that uses a distributed vertex-cut partitioning algorithm, called DFEP [9]. DFEP works based on a market model, where the partitions are buyers of vertices with their funding. Initially, all partitions are given the same amount of funding. The algorithm, then, proceeds in rounds, such that in each round, a partition p tries to buy

edges that are neighbors of the already taken edges by p, and an edge will be sold to the highest offer. There exists a coordinator in the system that monitors the size of each partition and sends additional units of funding to the partitions, inversely proportional to the size of each partition.

6 Conclusions

We presented Ja-be-Ja-vc, a distributed and parallel algorithm for vertex-cut partitioning. Ja-be-Ja-vc partitions edges of a graph into a given number of clusters with any desired size distribution, while the number of vertices that have to be replicated across clusters is low. In particular, it can create balanced partitions while reducing the vertex-cut. Ja-be-Ja-vc is a local search algorithm that iteratively improves upon an initial random assignment of edges to partitions. It also utilizes simulated annealing to prevent getting stuck in local optima. We compared Ja-be-Ja-vc with two state-of-the-art systems, and showed that Ja-be-Ja-vc not only guarantees to keep the size of the partitions balanced, but also outperforms its counterparts with respect to vertex-cut.

References

1. Abou-Rjeili, A., Karypis, G.: Multilevel algorithms for partitioning power-law graphs. In: Proc. of IPDPS'06, IEEE (2006) 10–pp
2. Lang, K.: Finding good nearly balanced cuts in power law graphs. Preprint (2004)
3. Leskovec, J., Lang, K., Dasgupta, A., Mahoney, M.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics **6**(1) (2009) 29–123
4. Albert, R., Jeong, H., Barabási, A.: Error and attack tolerance of complex networks. Nature **406**(6794) (2000) 378–382
5. Gonzalez, J., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Proc. of OSDI'12. (2012) 17–30
6. Xin, R., Gonzalez, J., Franklin, M., Stoica, I.: Graphx: A resilient distributed graph system on spark. In: Proc. of GRADES'13, ACM (2013) 1–6
7. Rahimian, F., Payberah, A., Girdzijauskas, S., Jelasity, M., Haridi, S.: Ja-Be-Ja: A distributed algorithm for balanced graph partitioning. In: Proc. of SASO'13, IEEE (2013)
8. Talbi, E.: Metaheuristics: from design to implementation. Volume 74. John Wiley & Sons (2009)
9. Guerrieri, A. Montresor, A.: Distributed edge partitioning for graph processing. In: Under submission. (2013)
10. Voulgaris, S., Gavidia, D., Van Steen, M.: Cyclon: Inexpensive membership management for unstructured p2p overlays. Journal of Network and Systems Management **13**(2) (2005) 197–217
11. Jelasity, M., Montresor, A.: Epidemic-style proactive aggregation in large overlay networks. In: Proc. of ICDCS'04, IEEE (2004) 102–109
12. Payberah, A., Dowling, J., Haridi, S.: Gozar: Nat-friendly peer sampling with one-hop distributed nat traversal. In: Proc. of DAIS'11, Springer (2011) 1–14
13. Dowling, J., Payberah, A.: Shuffling with a croupier: Nat-aware peer-sampling. In: Proc. of ICDCS'12, IEEE (2012) 102–111

14. Massoulié, L., Le Merrer, E., Kermarrec, A., Ganesh, A.: Peer counting and sampling in overlay networks: random walk methods. In: Proc. of PODC'06, ACM (2006) 123–132
15. Leskovec, J.: The graph partitioning archive. http://staffweb. cms. gre. ac. uk/~wc06/partition (2012)
16. Leskovec, J.: Stanford large network dataset collection. URL http://snap. stanford. edu/data/index. html (2011)
17. Baños, R., Gil, C., Ortega, J., Montoya, F.: Multilevel heuristic algorithm for graph partitioning. In: Applications of Evolutionary Computing. Springer (2003) 143–153
18. Bui, T., Moon, B.: Genetic algorithm and graph partitioning. Transactions on Computers **45**(7) (1996) 841–855
19. Hendrickson, B., Leland, R.: A multi-level algorithm for partitioning graphs. SC **95** (1995) 28
20. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. Journal on Scientific Computing **20**(1) (1998) 359–392
21. Karypis, G., Kumar, V.: Parallel multilevel series k-way partitioning scheme for irregular graphs. Siam Review **41**(2) (1999) 278–300
22. Walshaw, C., Cross, M.: Mesh partitioning: a multilevel balancing and refinement algorithm. Journal on Scientific Computing **22**(1) (2000) 63–80
23. Sanders, P., Schulz, C.: Engineering multilevel graph partitioning algorithms. In: Algorithms. Springer (2011) 469–480
24. Soper, A., Walshaw, C., Cross, M.: A combined evolutionary search and multilevel optimisation approach to graph-partitioning. Journal of Global Optimization **29**(2) (2004) 225–241
25. Chardaire, P., Barake, M., McKeown, G.: A probe-based heuristic for graph partitioning. Transactions on Computers **56**(12) (2007) 1707–1720
26. Benlic, U., Hao, J.: An effective multilevel tabu search approach for balanced graph partitioning. Computers & Operations Research **38**(7) (2011) 1066–1075
27. Sanders, P., Schulz, C.: Distributed evolutionary graph partitioning. arXiv preprint arXiv:1110.0477 (2011)
28. Talbi, E., Bessiere, P.: A parallel genetic algorithm for the graph partitioning problem. In: Proceedings of the 5th international conference on Supercomputing, ACM (1991) 312–320
29. Luque, G., Alba, E.: Parallel Genetic Algorithms: Theory and Real World Applications. Volume 367. Springer (2011)
30. Gehweiler, J., Meyerhenke, H.: A distributed diffusive heuristic for clustering a virtual p2p supercomputer. In: Proc. of IPDPSW'10, IEEE (2010) 1–8
31. Ramaswamy, L., Gedik, B., Liu, L.: A distributed approach to node clustering in decentralized peer-to-peer networks. Transactions on Parallel and Distributed Systems **16**(9) (2005) 814–829
32. Kim, M., Candan, K.: SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. Data & Knowledge Engineering **72** (2012) 285–303
33. Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Proc. of HotCloud'10, USENIX (2010) 10–10
34. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proc. of NSDI'12, USENIX (2012) 2–2