Enhancing Balanced Graph Edge Partition with Effective Local Search

Zhenyu Guo,[1] **Mingyu Xiao,** [1*] **Yi Zhou,** [1] **Dongxiang Zhang,** [2] **Kian-Lee Tan** [3]
[1] University of Electronic Science and Technology of China
[2] Zhejiang University
[3] National University of Singapore
Harry.Guo@outlook.com, myxiao@gmail.com, zhouyi0922@gmail.com, zhangdongxiang@zju.edu.cn, tankl@comp.nus.edu.sg

Abstract

Graph partition is a key component to achieve workload balance and reduce job completion time in parallel graph processing systems. Among the various partition strategies, edge partition has demonstrated more promising performance in power-law graphs than vertex partition and thereby has been more widely adopted as the default partition strategy by existing graph systems. The graph edge partition problem, which is to split the edge set into multiple balanced parts to minimize the total number of copied vertices, has been widely studied from the view of optimization and algorithms. In this paper, we study local search algorithms for this problem to further improve the partition results from existing methods. More specifically, we propose two novel concepts, namely adjustable edges and blocks. Based on these, we develop a greedy heuristic as well as an improved search algorithm utilizing the property of the max-flow model. To evaluate the performance of our algorithms, we first provide adequate theoretical analysis in terms of the approximation quality. We significantly improve the previously known approximation ratio for this problem. Then we conduct extensive experiments on a large number of benchmark datasets and state-of-the-art edge partition strategies. The results show that our proposed local search framework can further improve the quality of graph partition by a wide margin.

Introduction

Graph partition plays a key role in performance improvement for massive graph processing systems. In a distributed graph system, such as Google Pregel (Malewicz et al. 2010), GraphX (Gonzalez et al. 2014), and GraphLab (Low et al. 2012), the original graph may be too large to fit in memory and has to be partitioned into multiple parts which are processed in parallel by multiple machines. The quality of graph partition is often measured by two important performance criteria. One is *workload balance* which expects the sizes of the partitioned parts to be as equal as possible. The goal is to reduce the overall job completion time (JCT) in parallel systems, where the bottleneck is caused by the slowest job. The other is *communication overhead* whose objective is to minimize the connection among the parts. It is challenging to compute the optimal partition as the problem has been proven to be NP-Hard (Goemans and Williamson 1995; Feder et al. 1999).

Vertex partition is a popular model in which the workload of each part is evaluated by its number of vertices and the communication overhead between two parts is evaluated by their connecting edges in the original graph. In the past decades, there have been significant efforts devoted to this problem, including both theoretical results (Andreev and Racke 2006; Feldmann 2013) and heuristic algorithms. However, the performance of vertex partition models may degrade for parallel algorithms to handle power-law graphs (Gonzalez et al. 2012). In fact, most natural graphs follow a skewed degree distribution similar to the power-law distribution (Faloutsos, Faloutsos, and Faloutsos 1999; Newman, Strogatz, and Watts 2001). We can observe that in real-world graphs, a small fraction of vertices may connect to a large part of the graph. For example, celebrities in a social network attract a huge number of followers. This property brings non-trivial challenges to vertex partitioners, including workload balance, partitioning, communication, storage, and computation (Gonzalez et al. 2012).

To address the issue of power-law distribution, edge partition was introduced to partition the graph based on edge sets (Gonzalez et al. 2012). A vertex is allowed to appear in multiple parts sharing this vertex. The workload of edge partition is evaluated by the number of edges in each part while the communication overhead is evaluated by the *replication factor*, which indicates the average time of each vertex appearing in all the parts. Edge partition is more efficient in power-law graphs, and several parallel graph processing systems, including PowerGraph (Gonzalez et al. 2012), Spark GraphX (Gonzalez et al. 2014) and Chaos (Roy et al. 2015), have adopted edge partition as the default partition strategy.

Contributions

In this paper, we focus on edge partition. Instead of proposing a new edge partition method, we adopt local search techniques to further improve the partition results from existing methods. We will prove several structural properties of edge partition and introduce two novel concepts named *adjustable edges* and *blocks*. Based on these concepts, we develop two types of effective local search algorithms, namely LS-G and LS-F, which are complementary to state-of-the-

*Corresponding author.

art approaches. From the initial partition solutions derived from existing partition methods, our local search algorithms can further improve the solution by neighborhood operators. LS-G is a fast greedy heuristic and LS-F leverages the max-flow model to yield partitions with higher quality. In theory, we present theoretical analysis in terms of the approximation quality and provide improved approximation ratios for this problem. In practice, our experiments are conducted on multiple large-scale benchmark datasets and results show that the solutions derived from state-of-the-art edge partition methods can be further improved by a wide margin.

Preliminaries

Let $G = (V, E)$ stand for an undirected graph with $n = |V|$ vertices and $m = |E|$ edges. The neighbor set of a vertex v is denoted by $N(v) = \{u \mid u, v \in E\}$ and the neighbor set of a vertex set S is denoted by $N(S)$. For a subgraph or an edge set G', we use $V(G')$ to denote the set of vertices appearing in G' and $E(G')$ to denote the set of edges appearing in G'. For an edge subset $E' \subseteq E$, we use $G[E']$ to denote the subgraph *induced* from the edge set E', i.e., the graph $(V(E'), E')$.

Given a graph $G = (V, E)$, a *k-edge partition* divides E into k disjoint groups, denoted by $P = \{E_1, E_2, \cdots, E_k\}$, where $E_i \cap E_j = \emptyset, \forall i \neq j$ and $\bigcup_{1 \leq i \leq k} E_i = E$. A k-edge partition P is *α-balanced* if each part E_i in P satisfies:

$$|E_i| \leq \left\lceil \alpha \frac{|E|}{k} \right\rceil .$$

The *replication factor* of a k-edge partition P, denoted by $RF(P)$, is defined as follows:

$$RF(P) = \frac{1}{|V|} \sum_{i=1}^{k} |V(E_i)|.$$

Given a graph $G = (V, E)$ and two constants k and α, the EDGE PARTITION PROBLEM (EPP) is to find an α-balanced k-edge partition P such that the replication factor $RF(P)$ is minimized.

Adjustable Edges and Blocks

In this section, we first introduce some basic structural concepts that will be used in our local search strategies. As mentioned above, our algorithms are local-search algorithms based on a given k-edge partition. So next, we always assume that a k-edge partition $P = \{E_1, E_2, \cdots, E_k\}$ is given, which can be obtained by known algorithms or a random assignment. Based on a given k-edge partition, we will move some edges from one part to other parts to decrease the communication load (replication factor), and in the meanwhile keep each part under the workload balance constraint.

For the VERTEX PARTITION PROBLEM, a local-search strategy is easy to develop. We can move a vertex subset from one part to another part as long as the receiving part is still under the required workload balance and the number of crossing edges among the parts can be decreased. However, the local search strategy in EDGE PARTITION PROBLEM is more complicated because to decrease the replication factor, we are often required to move a subset of edges from one part to *multiple* different parts simultaneously. In this paper, we will present novel and effective strategies for edge movements. Before that, we first introduce the concept of *adjustable edge* which is important to understand our local search algorithms.

Definition 1 (Reachability). *Let $P = \{E_1, E_2, \cdots, E_k\}$ be a k-edge partition. A part E_i is* reachable *for an edge (u, v) if E_i contains both of the two endpoints of the edge, i.e., $u, v \in V(E_i)$.*

Definition 2 (Adjustable Edge). *Let $P = \{E_1, E_2, \cdots, E_k\}$ be a k-edge partition. An edge $(u, v) \in E_i$ is called an* adjustable edge *if there exists E_j such that $j \neq i$ and its two endpoints $u, v \in V(E_j)$.*

For an adjustable edge, we may be able to move it to another reachable part without increasing the replication factor. A simple approach is to treat the movement of the adjustable edge as a possible way to find better solution. . However, the replication factor can only be strictly decreased when the adjustable edge has a degree-1 endpoint in $G[E_i]$. In this case, the degree-1 endpoint will disappear from $G[E_i]$ after removing the edge, and thus reduce the total number of vertices $\sum_{i=1}^{k} |V(E_i)|$. Unfortunately, the number of degree-1 endpoints is limited, which restricts the optimization space for local search. Our strategy is to find further cases in which the replication factor can be strictly decreased by moving adjustable edges to other reachable parts. Based on the idea, we present another key structure called *block*.

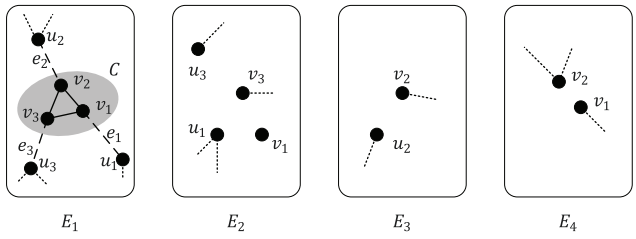
Definition 3 (Block). *Let $P = \{E_1, E_2, \cdots, E_k\}$ be a k-edge partition and E^* be the set of adjustable edges. Each connected component in the subgraph $G[E_i] - E^*$ is called a* block *of E_i. A block consists of a single vertex is called a* vertex block.

Our local search is designed to move a block from one part to another part. In detail, we first move all the adjustable edges incident on this block to other reachable parts. Note that this step will not increase the replication factor. After that, if we move a block C from part E_i to another part E_j, we can decrease the replication factor by

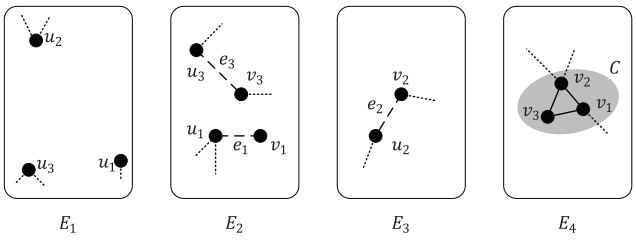
$$\frac{1}{|V|} |V(C) \cap V(E_j)|.$$

Fig. 1 shows an example of block movement that can reduce the replication factor. Consider a block C in part E_1 connecting 3 adjustable edges e_1, e_2, e_3. In the first step, we move e_1 and e_3 to part E_2, e_2 to part E_3, and the replication factor remains the same. In the next step, we move the whole block C from part E_1 to part E_4. Since $V(E_1) \cap V(E_4) = \{v_1, v_2\}$, we reduce 2 duplicate vertices.

Different ways to move adjustable edges and blocks will generate different algorithms. We need to consider how and when to move adjustable edges and blocks under the balance constraint. In this paper, we will give two ideas to do these adjustable operations, one is based on a greedy idea and one is based on the max flow technique.



(a) Before moving block C



(b) After moving block C

Figure 1: An example of block movement, where the black part containing three vertices v_1, v_2 and v_3 is the block C, solid edges mean edges in the block, dashed edges mean adjustable edges, and dotted edges mean other edges.

A Fast Algorithm: LS-G

In this section, we introduce a simple and fast local-search algorithm. We first introduce the most important ingredient, the sub-algorithm for the adjustment operation for a single block, and then we present the whole algorithm.

Greedy Adjustments. Let C be a block in part E_i. We use $A(C)$ to denote the set of adjustable edges incident on C in the subgraph $G[E_i]$, i.e., $A(C) = \{(a,b) | (a,b) \in E_i$ is an adjustable edge $\wedge\ |\{a,b\} \cap V(C)| = 1\}$. For a block C in E_i, the algorithm first checks whether C can be moved to another part $E_j \neq E_i$ to decrease the replication factor. After successfully moving C, then we consider the adjustable edges in $A(C)$ in random order. For each adjustable edge $e \in A(C)$, we check whether it can be moved to a reachable part different from E_i. If any step of the algorithm cannot be executed, then we undo all the moving operations in the algorithm. The algorithm to check whether a block C in E_i can be moved to another part is denoted by RA(C) and its pseudocode is shown in Algorithm 1.

In an adjustment operation, we move all adjustable edges in $A(C)$ to other parts and then move the whole block to another part to decrease the replication factor. For the special case that the block is a vertex block, the second step of moving the block can be omitted since the vertex will automatically disappear in E_i after removing all adjustable edges incident on it. In our implementation, we first move the whole block C (the edges in C) to another part E_j and then consider moving the adjustable edges in $A(C)$. It may be more effective to check the feasibility: when C contains several edges, it may be hard to find a part E_j to "receive" all the edges together under the balance constraints, while the edges in $A(C)$ can be moved to different parts separately.

Algorithm 1 RA(C)

1: **if** there are some $E_j \neq E_i$ such that $|E_j| \leq \lceil \alpha \frac{|E|}{k} \rceil - |V(C)|$ and $V(E_j) \cap V(C) \neq \emptyset$ **then**
2: Let $E_{j'}$ be a part satisfying the condition such that $|V(E_{j'}) \cap V(C)|$ is maximized
3: $E_i \leftarrow E_i \setminus E(C)$ and $E_{j'} \leftarrow E_{j'} \cap E(C)$
4: **else**
5: goto Step 15
6: **end if**
7: **for** each adjustable edge $e \in A(C)$ **do**
8: **if** there is a reachable part $E_j \neq E_i$ such that $|E_j| < \lceil \alpha \frac{|E|}{k} \rceil$ **then**
9: $E_i \leftarrow E_i \setminus \{e\}$ and $E_j \leftarrow E_j \cap \{e\}$
10: **else**
11: goto Step 15
12: **end if**
13: **end for**
14: **return** yes
15: Undo all the moving operations and **return** no

Algorithm 2 LS-G

Require: a graph $G = (V, E)$ with an α-balanced k-edge partition $P = \{E_1, \ldots, E_k\}$ of the edge set E.
Ensure: an α-balanced k-edge partition.
1: all parts of P are marked 'b'
2: **while** there is a part E_i marked 'b' **do**
3: mark E_i with 'w'
4: compute and order all blocks $C_{i1}, C_{i2}, \ldots, C_{il_i}$ in E_i
5: **for** $j = 1$ **to** l_i **do**
6: call the algorithm RA(C_{ij})
7: for any changed part during RA(C_{ij}), mark it 'b'
8: **end for**
9: **end while**
10: **return** $P = \{E_1, \ldots, E_k\}$

The Algorithm LS-G. The whole algorithm, named LS-G, is shown in Algorithm 2. It scans the parts from an initial solution of any existing edge partition method. For each part E_i, we identify the adjustable edges as well as the blocks. Then, we apply RA(C) to deal with each block as the adjustment operation. The blocks within a part are processed in non-decreasing order of their sizes. Note that edge movement will not trigger block reconstructions based on the following lemma.

Lemma 1. *Given a k-edge partition $P = \{E_1, E_2, \cdots, E_k\}$ of the edge set of graph $G = (V, E)$. Let C be a block in part $E_i \in P$ and $e \in E_i$ be an edge not in C. Let $P' = \{E'_1, E'_2, \cdots, E'_k\}$ be the new edge partition obtained by moving e to another part $E_j \neq E_i$, where $E'_l = E_l$ for $l \in \{1, 2, \ldots, k\} \setminus \{i, j\}$. Then C is still a block in E'_i of the new partition P'.*

Proof. Since we only move an edge in $E_i \setminus E(C)$ to another part E_j, we know that C is still a connected subgraph in $G[E_i]$. For any edge (a, b) in $E(C) \cap E'_i \subseteq E(C) \cap E_i$, it is an adjustable edge in P before moving e and there is

a part $E_{i_0} \neq E_i$ such that $a, b \in V(E_{i_0})$. Note that after moving e to E_j, no matter if $E_{i_0} = E_j$ or not, no vertex in $V(E_{i_0})$ will be removed, i.e., $V(E'_{i_0}) \subseteq V(E_{i_0})$. So part E_{i_0} is still a reachable part for (a, b) and then (a, b) is still an adjustable edge. Thus, C is still a block in E'_i of the new partition P'. □

Lemma 1 implies that after moving C together with $A(C)$ from a part E_i to other parts, any other block in E_i is still a block in the new edge partition. We do not need to compute new blocks in E_i after the operators.

An Algorithm Based on Max Flow: LS-F

In this section, we introduce an algorithm with a more sophisticated technique for adjustment operation based on blocks. The aforementioned algorithm RA(C) is fast. However, as it heuristically moves blocks one at a time, it may fail to move certain blocks (due to the constraints). To search for more domains, we suggest the following adjustment algorithm based on the max-flow model. This algorithm considers several different blocks together in each iteration to find better movements for adjustable edges. To ensure that we can move several blocks simultaneously, we need the following definitions.

Independent Block Set. The strategy of our algorithm recommends us to seek for some blocks which do not affect each other when moving together. Based on this motivation, we present important structures about blocks called *independent block set* as our adjustment operation. We will move all blocks in an independent block set simultaneously.

Definition 4 (Independent Block Set)**.** *Let P be a k-edge partition of a graph G. Two blocks C_i and C_j in P are called* independent *if they are from the same part of P or the shortest distance between $V(C_i)$ and $V(C_j)$ in G is at least two. A set of blocks $\mathcal{C} = \{C_1, C_2, \ldots, C_l\}$ is* independent *if any pair of blocks in it are independent.*

In the above definition, we set the distance between two blocks is at least two. So no two blocks in an independent set intersect, and the adjustable edges incident on two blocks in an independent set are different. Thus, for a set of independent blocks, we can move all the adjustable edges incident on all blocks in the set to other parts simultaneously without increasing the replication factor. After this, we may reduce the replication factor by moving these blocks.

Adjustments Based on Max Flow. Given an independent block set $\mathcal{C}$, we consider whether we can move together all adjustable edges in $\cup_{C\in\mathcal{C}} A(C)$ from its own part to other reachable parts under the balance constraints. Since we consider moving all adjustable edges incident on a set of blocks together, we may be able to reach more search domains and find better results. This is the advantage of this method, compared with the previous greedy method which only considers one block each time.

We use a max flow model to solve the problem of moving adjustable edges incident on a set of blocks together under the balance constraints. We construct a directed graph $H = (V_H, A_H)$ and reduce our problem to the problem of finding a maximum flow in H from the source v_{source} to the sink v_{sink}. The graph $H = (V_H, A_H)$ is constructed as follows, where $V_H = V_{edge} \cup V_{part} \cup \{v_{source}, v_{sink}\}$.

- Introduce two vertices, the source v_{source} and the sink v_{sink}.
- For each adjustable edge $e \in \bigcup_{C\in\mathcal{C}} A(C)$, introduce a vertex v_e with an arc $\overrightarrow{v_{source}v_e}$ from the source v_{source} to v_e of capacity $c(v_{source}v_e) = 1$. The set of vertices corresponding to edges in $\bigcup_{C\in\mathcal{C}} A(C)$ is denoted by V_{edge}.
- For each part E_i of P, introduce a vertex v_{E_i} with an arc $\overrightarrow{v_{E_i}v_{sink}}$ from v_{E_i} to the sink v_{sink} of capacity $c(v_{E_i}v_{sink}) = \Delta_i$, where Δ_i is the remaining capacity for part E_i to reach the bound of the balance constraint.
- For each vertex $v_e \in V_{edge}$, add an arc $\overrightarrow{v_e v_{E_i}}$ from v_e to v_{E_i} of capacity $c(v_e v_{E_i}) = 1$ for each reachable part E_i of e except the original part containing e.

In the above model, we have not given the precise definition of Δ_i but it does not cause trouble in understanding the model. In fact, we will let $\Delta_i = \lceil \alpha \frac{|E|}{k} \rceil - |E_i^*|$. Here we use E_i^* instead of E_i because we still need to save some space for moving blocks. Assume that two blocks C_1 and $C_2 \in \mathcal{C}$ will be moved to E_i, then we will let $E_i^* = E_i \cup C_1 \cup C_2$. However, we no longer know which block will be moved to which part. To fix this and simplify the algorithm, in our algorithm, we will first determine the 'destination part' for each block before moving the adjustable edges. We will select the destination part as the part after moving the block to where the replication factor is minimized.

Let f be a maximum flow in H, which can be computed by standard max-flow algorithms. For a block $C \in \mathcal{C}$, if for any edge $e \in A(C)$ it holds that $f(\overrightarrow{v_{source}v_e}) = 1$, i.e., there is an individual flow going through the arc $\overrightarrow{v_{source}v_e}$ in f, then we let the indication function $I(C) = 1$; otherwise let $I(C) = 0$. Let $\mathcal{C}'$ be the set of blocks with $I(C) = 1$.

We claim that based on the flow f we can move all adjustable edges in $A(C)$ for all blocks in $\mathcal{C}'$ to other parts without breaking the balance constraints. For each block $C \in \mathcal{C}'$, the algorithm moves each adjustable edge $e \in A(C)$ to part E_j for $f(\overrightarrow{v_e v_{E_i}}) = 1$, i.e., there is an individual flow going through the arc $\overrightarrow{v_e v_{E_i}}$ in f. In fact, for a vertex $v_{E_i} \in V_{part}$, the flow on the arc $\overrightarrow{v_{E_i}v_{sink}}$ is at most Δ_i, thus the number of adjustable edges moving to E_i will not break the balance constraint even no edge moves out from E_i and all edges in $E_i^* \setminus E_i$ move to E_i.

We will use MF($\mathcal{C}$) to denote the above algorithm based on the maximum flow in H. Note that MF($\mathcal{C}$) only moves edges in $A(C)$ for blocks C with $I(C) = 1$ and keeps unchanged for blocks C with $I(C) = 0$. In fact, we will also undo the moving of blocks C with $I(C) = 0$.

Algorithm LS-F. The algorithm is to iteratively deal with an independent block set by calling MF($\mathcal{C}$). Two concerns are mainly resolved: the generation of independent sets of blocks and the algorithm termination condition. We do not need to find a maximum independent set of blocks because it is hard to compute and not very helpful in our heuristic algorithm. In our algorithm, we find an independent set of

blocks containing at most one block from each part by a greedy method. We first pick an arbitrary block in the first part, and then iteratively try to pick a block from the next part that is independent with all picked blocks. Note that there may be many different independent block sets and it is time-consuming to consider all of them. In fact, it will be a rare case that we can not find an independent block set of size at least two after a large number of iterations. So we set the stop condition of our algorithm as a running time bound or a maximum number of rounds.

Approximation Ratio

We first provide theoretical analysis for EDGE PARTITION PROBLEM in terms of the approximation quality. The previous known approximation ratio for this problem is $O(d_{max}\sqrt{\log k \log n})$, which was first proved on graphs with some restrictions in (Bourse, Lelarge, and Vojnovic 2014) and then extended to general graph in (Li et al. 2017). In this section, we will show an approximation ratio of $\min\{k, \widetilde{d}\}$, where $\widetilde{d}$ is the average degree of the graph. Note that $\widetilde{d} \leq d_{max}$. The new result significantly improves the previous approximation ratio. We will also consider the lower bounds on the approximation ratio of our algorithms.

Lemma 2. *Any feasible edge partition P of a graph $G = (V, E)$ is an approximation solution with ratio at most $\min\{k, \widetilde{d}\}$.*

Proof. Each vertex can be copied at most k times in any feasible edge partition since there are only k parts. So it is trivially to get the approximation ratio of k.

Each edge appears only once in a feasible edge partition and then it can contribute at most 2 vertices. So it always holds that $\sum_{i=1}^{k} |V(E_i)| \leq 2|E|$. Thus,

$$RF(P) \leq \frac{2|E|}{|V|} = \widetilde{d}.$$

On the other hand, the optimal replication factor can be 1 (when no vertex is copied).

Thus we get a bound $\min\{k, \widetilde{d}\}$ for the ratio. □

The result in the above lemma does not rely on any algorithm. Any feasible solution will hold the approximation ratio. Next, we consider the approximation ratio related to our algorithms. Our algorithms will move adjustable edges and blocks to decrease the replication factor. We show that when the edge partition does not have any adjustable edges the approximation ratio k in Lemma 2 can be improved.

Lemma 3. *For a feasible edge partition P of a graph $G = (V, E)$, if P does not have any adjustable edges, then it is an approximation solution with ratio $\leq \min\left\{\widetilde{d}, t(\frac{|I_t|}{|V|} + 1)\right\}$, where $t = \frac{k+1}{2}$, $V_t = \{v | v \in V \wedge |N(v)| > t\}$ and I_t is a maximum independent set in the induced graph $G[V_t]$.*

Proof. By Lemma 2, we know that the approximation ratio is not greater than $\widetilde{d}$. We only need to consider $t(\frac{|I_t|}{|V|} + 1)$.

Let $P = \{E_1, E_2, \ldots, E_k\}$ be a feasible edge partition having no adjustable edges. For a vertex v, we use p_v to denote the number of parts in P containing v. Let $V_> = \{v \in V | p_v > t\}$ and $V_\leq = V \setminus V_>$.

We can see that $V_>$ is an independent set, otherwise, there are two vertices $u, v \in V_>$ such that $p_u + p_v > 2t = k + 1$, which implies that u and v will appear in at least two same parts and then edge (u, v) would be adjustable. Since $V_>$ is an independent set, we have that $|V_>| \leq |I_t|$. So it holds that

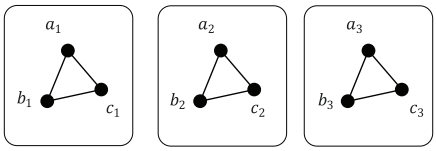
$$\begin{aligned} RF(P) &= \frac{1}{|V|}\left(\sum_{v \in V_>} p_v + \sum_{v \in V_\leq} p_v\right) \\ &\leq \frac{1}{|V|}\left(k|V_>| + \frac{k+1}{2}|V_\leq|\right) \\ &< \frac{1}{|V|}\left((k+1)|I_t| + \frac{k+1}{2}(|V| - |I_t|)\right) \\ &= t\left(\frac{|I_t|}{|V|} + 1\right). \end{aligned}$$

□

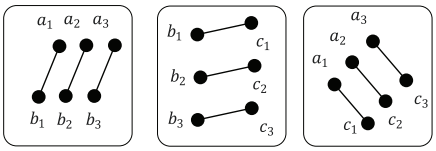
Note that when the graph is sparse, the number of vertices in V_t may not be large, and then the maximum independent set in $G[V_t]$ will be small. For this case, the ratio $t(\frac{|I_t|}{|V|} + 1)$ will be strictly smaller than k.

The condition in Lemma 3 is not easy to achieve. But at least Lemma 3 implies that the solution quality may be better when there are fewer adjustable edges.

On the other hand, we show that the approximation ratio $\widetilde{d}$ cannot be improved even when there are no adjustable edges. We give an example of the 1-balanced k-partition problem ($k = p(p-1)/2$ for some integer p), where the ratio $\widetilde{d}$ holds. The graph contains k independent cliques, each of which has exactly k edges. In the optimal solution, each connected component is partitioned to one part and then each vertex appears in exactly one part. We can also construct a solution, where each part takes exactly one edge from each connected component. Then each vertex appears in exactly $(p-1)/2 = \widetilde{d}$ parts. Furthermore, in this partition, there is no adjustable edge. Please see Figure 2 for an illustration of $k = 3$.

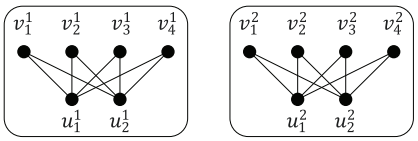


(a) An optimal partition

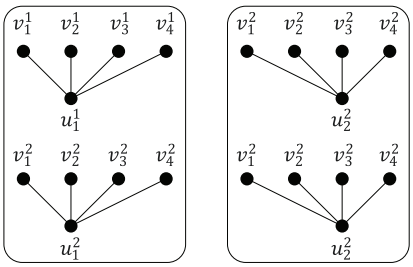


(b) A partition without adjustable edges

Figure 2: An example to achieve the ratio $\widetilde{d}$.



(a) An optimal partition



(b) A partition without adjustable edges

Figure 3: An example to achieve the ratio $O(k)$.

We then show the approximation ratio $O(k)$ is also tight when no adjustable edges exist. Note, although the approximation ratio $t(\frac{|I_t|}{|V|}+1)$ we proved above is slightly better than k, this ratio still belongs to $O(k)$ since $|I_t|$ may be as large as $|V|$. We still use an example of the 1-balanced k-partition to illustrate this tight result. The graph consists of k independent complete bipartite subgraphs $K^1_{k^2,k}, K^2_{k^2,k}, \ldots, K^k_{k^2,k}$, where each subgraph $K^i_{k^2,k}$ has exactly k^2 vertices on one side denoted by $U^i = \{u^i_1, u^i_2, \ldots, u^i_{k^2}\}$ and k vertices on the other side denoted by $V^i = \{v^i_1, v^i_2, \ldots, v^i_k\}$. Similar to the previous case, in the optimal solution, each connected component is partitioned into one part and then each vertex appears in exactly one part. We can also construct a solution without any adjustable edges: for each subgraph $K^i_{k^2,k}$, we pick all incident edges of v^i_j to part j. Then each vertex in V^i appears in k parts while each vertex in U^i appears in 1 part. Totally, the replication factor of this partition is $\frac{k^2+1}{k+1} = O(k)$. Please see Fig. 3 for an illustration of $k = 2$.

Computational Experiments

We evaluate the performance of our proposed algorithms LS-G and LS-F. Our objective is to minimize the replication factor (RF), which will be regarded as the quality measure.

Comparing Algorithms. We consider initial edge partitions generated by the following four algorithms: METIS (Karypis and Kumar 1998), NE (Zhang et al. 2017), SHEEP (Margo and Seltzer 2015) and SPAC (Li et al. 2017). The recent dSPAC (Schlag et al. 2019) is only a parallelized version of SPAC, and it generates the same edge partition as SPAC. We will show the results obtained by our LS-G and LS-F with initial edge partitions generated by them. The algorithm after running our local search algorithm A on the initial partition generated by B is denoted by B+A, where A can be LS-G or LS-F, and B can be METIS, NE, SHEEP, or SPAC.

Environment. Our algorithms are implemented in C++ and compiled with g++ version 5.4.0 with -O3 option.[1] These experiments are carried out under Ubuntu 16.04.3 LTS, using an Intel Core i5-7200U CPU at 2.50GHZ and 8GB RAM. For algorithms involving randomness, we run them for 10 times and report the average RF.

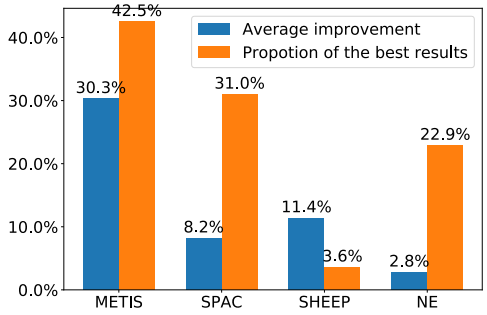


Figure 4: The average improvement for LS-F and the proportion of best results among the four initial partitions.

Datasets. We use real datasets from the Network Data Repository online (Rossi and Ahmed 2015), which is a well-known network repository containing a large number of networks in several different domains. We select 1872 graphs from that repository after discarding datasets that are not in any graph format or of small sizes (less than 1,000 edges) since it is less interesting to partition small graphs in a distribution system. We ensure that these selected graphs can cover both a wide range of size levels (from thousand edges to more than 17 million edges) and various domains (including 19 different domains: from real-life social graphs to manually generated graphs). The detailed information about these 1872 selected graphs can be found in our GitHub repository[1], which also contains our source code and some detailed experimental results.

Average Evaluations. First, we present the average results on the 1872 datasets with the default setting $k = 64$ and $\alpha = 1.1$. Overall, we get an average improvement of 12.07% for LS-G and 13.20% for LS-F. Although LS-F get more average improvements, LS-G uses less average running time. We show the performance of LS-F with different initial partitions in Fig. 4, where also give the proportion of best results among all results with four initial partitions. The detailed performance of LS-G and the running time are omitted here due to the limited space. The improvements on NE are not significant. However, for most instances, the performance of METIS+LS-G and METIS+LS-F is much better than that of NE+LS-G and NE+LS-F. For LS-F, about 42.51% best results are obtained by using initial partitions of METIS.

Detailed Comparisons. To give clear comparisons, we select four instances from the 1872 instances as examples to illustrate the details. The four instances are selected from different domains with different size levels: coauthors-dblp $(540486, 15245729)$ from "collaboration networks", grid-yeast $(6008, 156945)$ from "biological networks", Texas84 $(36364, 1590651)$ from "Facebook networks", and lastfm

[1]Our code is put in https://github.com/fafafafafafa7/LS_Algorithm

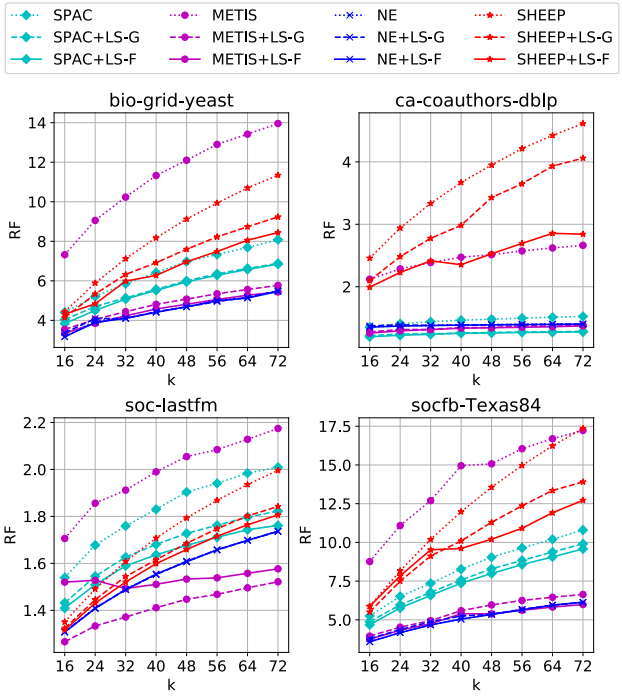


Figure 5: The results with different values of k.

	grid-yeast	coauthors-dblp	lastfm	Texas84
METIS	76,321	870,709	1,172,995	587,398
SHEEP	40,107	567,944	226,445	477,879
SPAC	10,997	139,119	395,783	64,399
NE	642	1534	238	127

Table 1: The number of blocks.

$(1191805, 4519330)$ from "social networks". The two numbers in the brackets are the numbers of vertices and edges.

Most previous algorithms, say METIS, NE, SHEEP, and SPAC, fixed the balance value $\alpha = 1.1$. We also take this setting and show the results under different values of k in Fig. 5. As k grows, LS-G and LS-F can consistently and effectively enhance the results produced by initial algorithms.

To make a full understanding of our local search methods, we also do break-down analysis by visualizing the effect of our local search methods. We show in Table 1 the number of blocks in the initial partitions generated by different edge partitioners. We can see that METIS generates much more blocks than the other three algorithms, which as a consequence enlarges the search space of METIS. The number of blocks generated by NE is small. So for initial partitions generated by NE, our local search algorithms can only improve a small part, as shown in Fig. 4.

Fig. 6 shows the number of blocks of different sizes before and after applying LS-G and LS-F on METIS. More than 95% blocks are of size at most 20. Most of the blocks with medium or large sizes have been removed by our algorithms. On these instances, LS-F reduces more blocks and generates a better result than LS-G. We can also see that the number of size-1 blocks drops sharply. The number of adjustable edges

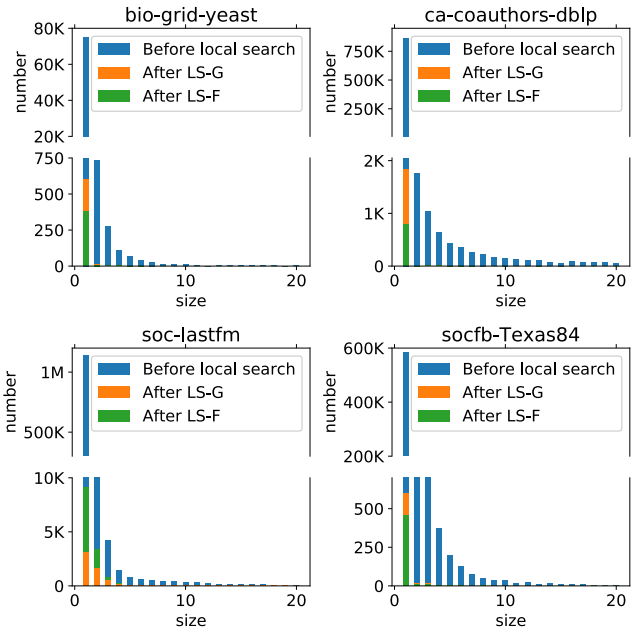


Figure 6: The number of blocks before and after applying LS-G and LS-F on METIS.

incident on size-1 blocks may be small and then it may be easy to be reduced by our algorithms.

Further Applications and Discussion

There are two frequently used computation tasks in graph-parallel computation: PageRank (Brin and Page 1998) and triangle counting (Zhang et al. 2017; Xie et al. 2014). We run these two tasks on the above four selected instances. Compared with initial edge partitions obtained by METIS, SHEEP, SPAC, and NE, our local search algorithms improve the running time for most cases (except NE on grid-yeast) with an average speedup of 9.23% for LS-G (resp., 10.60% for LG-F) in task PageRank; and with an average speedup of 8.00% for LS-G (resp., 7.92% for LG-F) in task triangle counting. Note that although our algorithms always get improved edge partitions, the running time of some concrete computation tasks under better partitions may not always be improved. The reason should be that different computation tasks within some components may become worse even the partition is improved.

GraphX is a well known graph-parallel computation system (Gonzalez et al. 2014). It has some built-in edge partition algorithms, which cannot be exported from the system, and then we are unable to apply our local search algorithms on them directly. Compared with the results obtained by the built-in algorithms of GraphX, our local search algorithms (with METIS, SHEEP, SPAC, and NE) can get an average speedup of 30.82% (the running time of computation tasks after giving the partition). Further applications of our algorithms in graph analytic systems are worthy of deep study. Anyway, this paper gives some structural properties and efficient algorithms together with theoretically proved approximation ratios for an important optimization problem.

Acknowledgements

The work is supported by the National Natural Science Foundation of China, under grants 61972070 and 61802049, Singapore Ministry of Education, under grant MOE2017-T2-1-141, and Sub Project of Independent Scientific Research Project, under grant ZZKY-ZX-03-02-04.

References

Andreev, K.; and Racke, H. 2006. Balanced graph partitioning. *Theory of Computing Systems* 39(6): 929–939.

Bourse, F.; Lelarge, M.; and Vojnovic, M. 2014. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 1456–1465. ACM.

Brin, S.; and Page, L. 1998. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems* 30(1): 107 – 117. ISSN 0169-7552. Proceedings of the Seventh International World Wide Web Conference.

Faloutsos, M.; Faloutsos, P.; and Faloutsos, C. 1999. On power-law relationships of the internet topology. In *ACM SIGCOMM computer communication review*, volume 29, 251–262. ACM.

Feder, T.; Hell, P.; Klein, S.; and Motwani, R. 1999. Complexity of graph partition problems. In *STOC*, 464–472. Citeseer.

Feldmann, A. E. 2013. Fast balanced partitioning is hard even on grids and trees. *Theoretical Computer Science* 485: 61–68.

Goemans, M. X.; and Williamson, D. P. 1995. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM)* 42(6): 1115–1145.

Gonzalez, J. E.; Low, Y.; Gu, H.; Bickson, D.; and Guestrin, C. 2012. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, 2.

Gonzalez, J. E.; Xin, R. S.; Dave, A.; Crankshaw, D.; Franklin, M. J.; and Stoica, I. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *OSDI*, volume 14, 599–613.

Karypis, G.; and Kumar, V. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20(1): 359–392.

Li, L.; Geda, R.; Hayes, A. B.; Chen, Y.; Chaudhari, P.; Zhang, E. Z.; and Szegedy, M. 2017. A simple yet effective balanced edge partition model for parallel computing. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1(1): 14.

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

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

Margo, D.; and Seltzer, M. 2015. A scalable distributed graph partitioner. *Proceedings of the VLDB Endowment* 8(12): 1478–1489.

Newman, M. E.; Strogatz, S. H.; and Watts, D. J. 2001. Random graphs with arbitrary degree distributions and their applications. *Physical review E* 64(2): 026118–1–026118–17.

Rossi, R. A.; and Ahmed, N. K. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In (Rossi and Ahmed 2015), 4292–4293. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9553.

Roy, A.; Bindschaedler, L.; Malicevic, J.; and Zwaenepoel, W. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 410–424. ACM.

Schlag, S.; Schulz, C.; Seemaier, D.; and Strash, D. 2019. *Scalable Edge Partitioning*, 211–225. doi:10.1137/1.9781611975499.17. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611975499.17.

Xie, C.; Yan, L.; Li, W.-J.; and Zhang, Z. 2014. Distributed power-law graph computing: Theoretical and empirical analysis. In *Advances in Neural Information Processing Systems*, 1673–1681.

Zhang, C.; Wei, F.; Liu, Q.; Tang, Z. G.; and Li, Z. 2017. Graph edge partitioning via neighborhood heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 605–614. ACM.