Analysis on Graph Partitioning for Distributed Graph-centric Computing

Wei Zhang
Department of Computer Science
Texas Tech University
Lubbock, TX, 79409
Email: X-Spirit.zhang@ttu.edu

I. ABSTRACT

The amount of globally stored, electronic data is growing at an increasing rate. This growth is both in size and connectivity, where connectivity refers to the increasing presence of, and interest in, relationships between data. A typical example of such data is the social network graph created and stored by Twitter. Due to the growth of such data, graph-centric data processing has become a focus for data mining as well as the design of corresponding computing system, especially in recent decades, as the distributed computing system is widely used in various areas, the challenge of graph-centric computing has never been heeded yet more and more seriously considered. As the leading technology which can drastically affect the effectiveness and efficiency of graph-centric data processing, graph partitioning plays an important role in many aspects, e.g. load balancing in terms of communication overhead, computational complexity, storage capacity. In this work, we mainly review several graph partitioning approaches. Finally, by making comparisons and conclusion, we analyze the potential impact of each partitioning approach may have in the case of distributed computing.

Keywords: complex network; graph data management; graph; graph partitioning; distributed system

II. INTRODUCTION

Recently, graph-centric machine learning and data mining is increasingly arousing people’s attention as the graph computation exhibits the valuable capability of abstracting the complex network of real-world relationships into mathematical models and thereby simplifying the computation by various graph-oriented computational approaches[1].

However, unlike traditional distributed computing system in which data is processed in isolation, distributed network-centric computing system, in which complex network computations are carried out, requires each vertex or edge to be processed in the context of its neighborhood. Moreover each transformation depends on the result of distributed joins between vertices and edges. As a consequence, indexing and data layout are important steps in achieving an efficient distributed execution. Because different types of network tend to affect the computation on distributed system in various ways due to their structural difference[2]. For example, the social network, usually seen as a specific type of network which can be approximately categorized into scale-free network, basically follows the power-law degree distribution[3], which means a small portion of the nodes are extremely high degree node, and such structural feature in social network mainly affect the load balancing of a specific distributed computing system in terms of computational workload, storage capacity as well as communication overhead among different computing nodes[4]. Thus, distributed network-centric computing systems rely on graph partitioning minimize communication and storage overhead, and ensure balanced computation[2].

In addition, as for distributed computing, any complex network can be modeled by series of large distributed graphs which usually consist of terabytes of data, and the data are stored and managed on different storage with various I/O bandwidths. Therefore, performing computation on the large distribute graph data may become the bottleneck of the entire distributed network-centric computing system because of potential slow I/O bottleneck and the high communication overhead. In order to reduce the impact of slow I/O, we want to utilize the capability provided by the distributed system to access multiple storage systems in parallel, which may accelerate the data throughput of the overall system. A standard solution is to split the data across a large cluster of commodity machines. For graph data, the approach is known as balanced graph partitioning, where the goal is to minimize the number of cross partition edges, at the same time, trying to keep the number of edges in each partition evenly distributed. The interesting thing is that the graphs that we usually process are not random, where the edges demonstrate a great locality property. The locality is important, because we can potentially have locality-driven partitions that are significantly better than random cuts. A second important factor is the communication; it is true that the inter-machine communication is substantially more expensive than inter-processor communication. For large graphs, the data to be moved may beyond hundreds of gigabytes, which dramatically slows down the processing efficiency. Thus, it is critical to examine a good partitioning algorithm that favors the locality and benefits the intra communication, which may fit into certain scenario in which the data of large-scale network needs to be processed very quickly.

Outline: Section III gives a brief review to some im-
important concepts and definitions about graph partitioning. Section V reviews four graph partitioning algorithms which are of two categories - edge-cut and vertex-cut. Section V compares the similarity and difference of those algorithms, and address some basic principles for designing partitioning algorithms as well as applying partitioning algorithms to distributed computing system.

III. BACKGROUND

In terms of the object on which the graph cut is carried out, basically, the graph partitioning algorithms can be divided into two categories, and one of them is called edge-cut, with the other called vertex-cut.

The k-way edge-cut partitioning problem is defined as follows: Given a graph \( G = (V, E) \) with \( |V| = n \), partition \( V \) into \( k \) subsets, \( V_1, V_2, ..., V_k \) such that \( V_i \cap V_j = \emptyset \) for \( i \neq j \). Similarly, he \( k \)-way vertex-cut partitioning problem is defined as follows: Given a graph \( G = (V, E) \) with \( |E| = n \), partition \( E \) into \( k \) subsets, \( E_1, E_2, ..., E_k \) such that \( E_i \cap E_j = \emptyset \) for \( i \neq j \). Intuitively, the higher the modularity is, the denser is each partition and the smaller is the faction of edges running between partitions.

The graph partitioning problem is NP-complete, and there is no approximation algorithm with a constant ratio factor for general graphs.

Clearly, if would be of great help if we can find certain criteria for measuring the partitioning quality when we apply a specific graph partitioning algorithm to the graph data which will be stored in the distributed computing system. There are two criteria which can be applied, the expansion and modularity, which can be used to measure the cut-size ratio between partitions as well as the density of a partition respectively.

In terms of edge-cut, the expansion for two partitions is as follows:

\[
\text{expansion}_{\text{ecut}}(P_1, P_2) = \frac{\text{edge-cut}(P_1, P_2)}{\min\{|P_1|, |P_2|\}}
\]

where \( |P_1| \) and \( |P_2| \) denote the number of vertices in partition \( P_1 \) and \( P_2 \) respectively. Intuitively, the lower the expansion is, the less number of edges are separated between two partitions. So, in general, given an edge-cut \( E' \) which partitions \( G = (V, E) \) into partitions \( P_1, P_2, ..., P_m \), the edge-cut is thought to be good if

\[
\text{expansion}_{\text{ecut}}(E') = \max_{P_i \in P_1, P_2, ..., P_m} \text{expansion}_{\text{ecut}}(P_i, G - P_i)
\]

is small.

The modularity of an edge-cut is defined as:

\[
\text{modularity}_{\text{ecut}}(E') = \sum_{1 \leq i \leq m} \left( \frac{|E_{i,i}|}{|E|} - \left( \sum_{i \neq j} \frac{|E_{i,j}|}{|E|} \right)^2 \right)
\]

where \( E_{i,j} \subseteq E' \) is the set of edges in the cut which runs between two partitions. Intuitively, the higher the modularity is, the denser is each partition and the smaller is the faction of edges running between partitions.

Similarly, we may have the vertex-cut based expansion and modularity for measuring the partitioning quality for vertex-cut:

\[
\text{expansion}_{\text{vcut1}}(P_i, P_j) = \frac{\text{vertex-cut}(P_i, P_j)}{\min\{|P_i|, |P_j|\}}
\]

and

\[
\text{expansion}_{\text{vcut2}}(P_i, P_j) = \frac{\text{vertex-cut}(P_i, P_j)}{\min\{|P_i|, |P_j|, |E_{i,j}|\}}
\]

thus, given a vertex \( V' \) which partitions \( G = (V, E) \) into m partitions \( P_1, P_2, ..., P_m \), the vertex-cut is thought to be good if

\[
\max_{P_i \in P_1, P_2, ..., P_m} \text{expansion}_{\text{vcut}}(P_i, G - P_i)
\]

is small.

Likewise, the vertex-cut based modularity is defined as:

\[
\text{modularity}_{\text{vcut1}} = \sum_{1 \leq i \leq m} \left( \frac{|V_{i,i}|}{|V|} - \left( \sum_{i \neq j} \frac{|V_{i,j}|}{|V|} \right)^2 \right)
\]

and

\[
\text{modularity}_{\text{vcut2}} = \sum_{1 \leq i \leq m} \left( \frac{|E_{i,i}|}{|E|} - \left( \sum_{i \neq j} \frac{|V_{i,j}|}{|V|} \right)^2 \right)
\]

where \( |V_{i,j}| \) is the number of vertices in the graph that exist commonly between partitions \( P_i \) and \( P_j \) and \( |E_{i,i}| \) is the number of vertices in partition \( P_i \), \( |E_{i,j}| \) is the number of edges in partition \( P_i \).

IV. TECHNICAL APPROACH

In this section, we examine four different graph partitioning algorithms by two categories: edge-cut and vertex-cut. Section IV-A, IV-B and IV-C talk about three edge-cut graph partitioning algorithms while section IV-D talks about one vertex-cut partitioning algorithm.
A. Multilevel Partitioning Scheme

In [5], the author proposed the multilevel approach for graph partitioning with corresponding implementation called METIS. This multilevel graph partitioning approach contains three phases as shown in Figure 1:

1) Coarsening Phase: The graph $G_0$ is transformed into a sequence of smaller graphs $G_1, G_2, ..., G_m$ such that $|V_0| > |V_1| > |V_2| > ... > |V_m|$. The author proposed 4 matching strategies for coarsening phase:

Random Matching: By Random Matching, we randomly select one of its unmatched adjacent vertices. If such a vertex $v$ exists, we include the edge $(u, v)$ in the matching and mark vertices $u$ and $v$ as being matched. If there is no unmatched adjacent vertex $v$, then vertex $u$ remains unmatched in the random matching. The time complexity of this strategy is $O(|E|)$, however, there is no guarantee that the edge-cut is minimized.

Heavy Edge Matching: The vertices are visited in random order. If a certain vertex $u$ is not matched, match it with its unmatched adjacent vertex $v$, such that the weight of edge $(u, v)$ is maximum among all incident edges of $u$. (Edges with identical weights may be matched at the same time). If $v$ is found, include the edge $(u, v)$ into the matching, otherwise, $u$ remains unmatched. The time complexity of this strategy is $O(|E|)$. This strategy ensures that the edge-cut is minimized.

Light Edge Matching: The vertices are visited in random order. If a certain vertex $u$ is not matched, match it with its unmatched adjacent vertex $v$, such that the weight of edge $(u, v)$ is minimum among all incident edges of $u$. (Edges with identical weights may be matched at the same time). If $v$ is found, include the edge $(u, v)$ into the matching, otherwise, $u$ remains unmatched. The time complexity of this strategy is $O(|E|)$, however, this strategy can produce coarsened graph with higher average degree which may be suitable for certain partitioning algorithm such as Kernighan-Lin.

Heavy Clique Matching: The motivation of this strategy is to compute a matching by collapsing vertices that have high edge density which can be represented as

$$D_G = \frac{2(w(u, v))}{w(u, v) \times (w(u, v) - 1)}$$

It is interesting to note that $D_G = 1$ if a clique, which is a fully connected sub-graph in undirected graph, is created by such matching, and since the algorithm always tries to find the sub-graph with the highest edge density, we can make it easier for partitioning algorithms to find good bisection.

2) Partitioning Phase: A 2-way partition $P_m$ of the graph $G_m = (V_m, E_m)$ is computed that partitions $V_m$ into two parts, each containing half the vertices of $G_0$. In this phase, there are 4 applicable algorithms which are:

Spectral Bisection: Suppose we represent the graph in an adjacency matrix $A$ and this algorithm first computes the Feidler vector $y$ corresponding to the second largest eigenvalue of the Laplacian matrix $Q = D - A$, where $D$ is a diagonal matrix within which the elements on the diagonal are the sum of all elements in the corresponding row. Let $r$ be the weighted median of the values of $y_i$, the we include all the vertices into one partition for all $y_i \leq r$ while other vertices being included into another partition.

Kernighan-Lin Algorithm: As for this algorithm, a certain initial bi-partition is given, then we repeatedly select a node in one partition that is going to be swapped with a node in another partition, and we mark these two node only if the reduction of edge-cut increases, and such reduction is called gain. The repeat terminates before the gain turns into negative.

Graph Growing Algorithm: Starting from a vertex, this algorithm grows a region around it in a breadth-first fashion until half of the vertices or half of the total vertex weight have been included.

Greedy Graph Growing Algorithm: Starting from a vertex, instead of growing a region around it in a strict BFS fashion, we sort the vertices in a graph’s frontier by the decreasing order of the gain, then we include the vertices with the largest gain into the partition at each iteration. It is noteworthy that at the end of each iteration, the gains of its adjacent vertices already in the frontier are updated.

3) Uncoarsening Phase: The partition $P_m$ of $G_m$ is projected back to $G_0$ by going through intermediate partitions $P_{m-1}, P_{m-2}, ..., P_1, P_0$. We can simply assign the vertices in a multinode $V_m$ in partition $P_m$ into $P_{m-1}$, however, in order to achieve better partitioning quality, we prefer to carry out some refinements:

Kernighan-Lin Refinement: The idea of Kernighan-Lin refinement is to use the projected partition of $G_{i+1}$ onto $G_i$ as the initial partition for the Kernighan-Lin algorithm.
Since we start with a good partition, the KL algorithm will converge after only few steps and no more positive gains can be acquired. So, we may prefer to carry out KL(1) which is only one single iteration of KL algorithm to decrease the edge-cut a little bit further to a certain level that is enough.

**Boundary Kernighan-Lin Refinement:** In both KL and KL(1) refinement algorithms, we have to insert the gains of all the vertices in the data structures. To reduce the waste of computation generated by swapping non-boundary nodes between two partitions, we initially insert the gains into data structures for only boundary vertices, and then we carry out KL algorithm. To further accelerate this process, we may use BKL repeatedly as long as the graph is small, and switch to BKL(1) whenever the graph is large enough.

4) **Experimental Results:** The author carried out detailed experiments for each phase. In general, the Heavy Edge Matching managed to generate good initial partition in most cases while spending least amount of running time, and Greedy Graph Growing Partitioning beat other 3 algorithms in terms of both edge-cut and running time while managed to generate good initial partition in experiments for each phase. In general, the BKL repeatedly as long as the graph is small, and switch to KL algorithm. To further accelerate this process, we may use BKL repeatedly as long as the graph is small, and switch to BKL(1) whenever the graph is large enough.

**B. Balanced Label Propagation**

While multilevel partitioning approach manages to generate partitions for large graphs in a reasonable running time, however, for large-scale distributed computing system, such algorithm fails to work in parallel, though its parallel variation, ParMETIS[12], was implemented. In addition, multilevel partitioning algorithm tends to ignore multiple underlying factors of the massive graph data, for example, the attribute of vertices may provide us some auxiliary information for graph partitioning to generate more refined partitioning result ready for massive graph data mining and machine learning system. Obviously, the multilevel partitioning algorithm doesn’t take such auxiliary information into account.

In [13], an efficient algorithm, balanced label propagation is introduced to precisely partition massive graphs for performing machine learning on the recommendation system in Facebook.

1) **Label Propagation:** Label propagation was first proposed as an efficient method for learning missing labels for graph data in a semi-supervised setting[14]. In a context very similar to graph partitioning, label propagation has been found to be a very efficient technique for network community detection[15]. The algorithm is conceptually simple in its operation. Initially, each vertex in the graph is assigned a unique numeric label. The label for each vertex is then replaced with the most frequent label amongst its neighbors; when several labels are equally frequent, the current label is kept if it is among the most frequent, while otherwise a new label is chosen at random from the most frequent. Vertices are repeatedly relabeled, with the algorithm terminating when the label for each vertex is (one of) the most frequent of the labels for the neighbors of the vertex. To avoid possible cycles and ensure termination, it is suggested that the vertex labels should be updated asynchronously and in random order. Network communities are then associated with sets of vertices bearing the same labels.

**Partition constraints:** The goal of our balanced label propagation algorithm is to take a graph $G = (V, E)$ and produce a partition $\{V_1, ..., V_n\}$ of $V$, subject to explicitly defined size constraints in the form of lower bounds $S_i$ and upper bounds $T_i$, such that $S_i \leq |V_i| \leq T_i$, for all $i$. These constraints can easily take the form of balanced constraints, targeting exact balance operating with leniency, $S_i = \lceil (1 - f)|V| \rceil$ and $T_i = \lceil (1 + f)|V| \rceil$, for some fraction $f > 0$.

To initialize the algorithm, we begin by randomly assigning nodes to shards, in proportions that are feasible with respect to these sizing constraints.

**Constrained Relocation:** It is ideal to maintain the specified balance of nodes across shards between iterations. The key challenge of sharding is, however, that some shards will be more popular than others. In fact, under ordinary label propagation without any balance constraints, labelling all nodes with the same single label is a trivial equilibrium. Because we won’t be able to move all nodes, our greedy approach is to synchronously move those nodes that stand to increase their colocation count (the number of graph neighbors they are colocated with) the most. Thus, this greedy relocation strategy is formalized as a maximization problem subject to the above constraints.

Let $u_{ij}(k)$ be the change in utility (colocation count) from moving the $k$th node from shard $i$ to $j$. Let $f_{ij}(x)$ be the relocation utility function between shard $i$ and $j$, the total utility gained from moving the leading $x$ nodes from $i$ to $j$. Observe that because $u_{ij}(k) \geq 0$ and $u_{ij}(k) \geq u_{ij}(k + 1)$ for all $k$ (since they are ordered), all $f_{ij}(x)$ are increasing and concave. The goal can then be formulated as a concave utility maximization problem with linear constraints.

Then the constrained relocation problem is to maximize:

$$\max_x \sum_{i,j} f_{ij}(x_{ij}) \text{ s.t.}$$

$$S_i - |V_i| \leq \sum_{j\neq i} (x_{ij} - x_{ji}) \leq T_i - |V_i|, \forall i$$

$$0 \leq x_{ij} \leq P_{ij}, \forall i, j.$$  \hspace{1cm} (10)

Here $P_{ij}$ is the number of nodes that desire to move from shard $i$ to $j$, and $f_{ij}(x)$ is the relocation utility function between shard $i$ and $j$, both derivable from the graph and the partition.

The steps to carry out a size-constrained relocation are as follows:

1 Determine where every node would prefer to move, and how much each node would gain from its preferred relocation.
2. Sort the node gains for each shard pair and construct the Constrained Relocation linear program.
3. Solve the linear program, which determines how many nodes should be moved, in order, between each shard pair.
4. Move these nodes. This constitutes one iteration.
5. Keep iterating step 1 to step 4 until reach the constrain values.

2) Geographic Initialization: It is found that the quality of the final sharding and the number of iterations can be improved by using node data to complete the initialization. For instance, on the web one might use domain, or in a computer network one might use IP-address. In the case of the Facebook social network, we find that geographical information gives us good initial conditions. Therefore, in the initialization part, an initial graph partitioning based on a geographic partitioning is implemented. Instead of cutting up the geographic space to form shards of exactly equal population, in this paper, average degree are considered as a means of distributing computational load. The cost can be calculated by the following function:

\[ Cost(c) = n_c(1 + \lambda d_c) \]  

where \( n_c \) is every city \( c \) with population \( n_c \) and \( d_c \) is the average degree, \( \lambda \) is a weight parameter determined by the proportion of node attribute data to edge attribute data to be sharded. In this paper’s implementation \( \lambda \) is set to be equal to \( 1/\bar{d}_{avg} \).

Balloon partitioning algorithm: This partitioning algorithm presented here form the shards centered at the cities with the highest population in the data set. Starting from the city with highest population not yet assigned to a shard, and grows circular "balloons" around these cities, which is indicated in Figure 2.

At first, the city \( C \) with the largest unassigned cost are selected as the center-point of a new shard. Then all cities with a non-zero remaining cost are sorted according to their geodesic distance from \( C \).

Since edges in the social network are overwhelmingly internal to countries, a negative distance reward is introduced to all cities in the same country as \( C \), thus the balloon algorithm manages to finish assigning each country completely before moving on to another country.

Starting with \( C \), the algorithm progresses down the sorted list of cities, and while there is capacity, it assigns each city to the shard currently being constructed. When the algorithm changes to assign another country, a new negative distance reward is given to cities in that country. If there is not enough capacity in the current shard to accommodate an entire city, a fractional assignment is noted, and the cost is subtracted from the remaining cost of that city. Eventually, the result is \( n \) shards, centered over population centers, each containing all nodes within a certain radius of the central city, adaptively configured such that each shard is of equal burden under the cost model.

![Figure 3: Output of the greedy geographic initialization algorithm for the 750,000 known cities, obtaining 234 shards of equal cost](image)

This assignment algorithm operates only on the set of cities and not on the social graph itself, which means it can be possible to run this complete assignment on a single machine in a single processor thread in seconds, with no actual graph analysis.

It should be noted that the geographic sharding performed here is ultimately static, and as new users register to join the site, these users can be assigned in accordance with the map from cities to shards. We note that as the geographic distribution of Facebook users slowly changes over time, resharding may be useful.

Oversharding: For the purposes of load-balancing a service that handles real-time requests, it is important to mitigate potentially problematic peak loads that result from assigning geographically concentrated regions to the same shard.

A three day window of user traffic is shown in Figure 4, where we see that local geographic regions experience much more volatile peak loads than the full site on average.

The solution to this problem was to create 3 times more shards than there are machines, and then sort shards by the longitude of their most populated city. The shards are then distributed cyclically across the n machines, e.g., shard 1, \((n + 1)\), and \((2n + 1)\) in longitudinal order are assigned to the same machine.
3) **Experimental Results**: As shown in the Figure 5, the sharding with geographic initialization algorithm + label propagation as the best performance that the fraction of local edges is the highest among the 3 sharings.

In this paper, they consider the performance of cutting the graph into 20, 40, 60, 80, and 100 symmetric shards.

![Figure 4: A three day window of user traffic in New Zealand, Italy, and Chile](Image)

![Figure 5: The fraction of local edges for each shard](Image)

In the Figure 5, we can see that when partitioning the LiveJournal graph into 20 parts, 63% of edges are local to a single partition. Impressively, when the algorithm is scaled up to 100 shards, fully 51% of edges are local to a single partition.

Overall, we observe that the fraction of edges that are local is nearly unchanged when increasing the number of shards from 40 to 100. We interpret this performance to be a consequence of the greedy nature of the algorithm, as the algorithm principally exploits local graph relocations that are significantly below the scale of any of the shard sizes, while global improvements are less possible.

The sharding algorithm developed here is also evaluated in a realtime distributed graph computation service: Facebook’s 'People You May Know' (PYMK) service for suggesting friend recommendations.

![Figure 6: Iterating the balanced label propagation algorithm with 78 shards, from a geographic and random initialization](Image)

![Figure 7: The fraction of local edges for each shard](Image)

![Figure 8: Number of non-local machines queried per request during friend-of-friend calculations in PYMK](Image)

In Figure 8 we can see that the following 3 shardings three implemented:

1. Baseline sharding: assigning users by the modulus sharding, 'user ID % 78'.
2. Geographic sharding: assigning users using 234 geographic shards, with no label propagation.
3. Propagated sharding: one step of balanced label propagation after geographic initialization.

The median number of non-local machines queried for the baseline, geographic, and once-propagated shardings were 59, 12, and 9 machines, respectively. Notice that under the old naive system, the most common occurrence was
that friend lists had to be aggregated from all 77 non-local machines, while the modes for the new shardings are 6 non-local machines for the geographic sharding and 3 non-local machines for the once-propagated sharding.

![Figure 9: Query time and network traffic for the three different shardings applied to the PYMK service][13]

As shown in the Figure 9, the time-averages of the average machine query times for the baseline, geographic, and once-propagated shardings were 109ms, 68ms, and 55ms, respectively. The time-averages of the maximum machine query times were 122ms, 106ms, and 100ms, respectively. The time-averages of the maximum machine traffic levels only 37.1% of the baseline system machines.

The geographic system, which operated on hardware identical to the baseline system, featured an average query time only 62.3% of the baseline system. The total improvement when comparing to the propagated system was an average query time only 50.5% of the baseline system.

The geographic system, which operated on hardware identical to the baseline system, featured an average query time only 62.3% of the baseline system. The total improvement when comparing to the propagated system was an average query time only 50.5% of the baseline system.

The time-averages of the average machine traffic for the baseline, geographic, and once-propagated shardings were 35.3 MB/s, 21.8 MB/s, and 13.1 MB/s, respectively.

Thus, the machines in the propagated system saw network traffic levels only 37.1% of the baseline system machines.

C. Streaming Graph Partitioning

In [IV-A] and [IV-B] we’ve reviewed two different edge-cut graph partitioning algorithms, however, both of them requires multiple steps to generate the final result, which are too computationally intensive to be applied in a realtime graph computing system.

In [16], the author studies the 1-pass graph partitioning and proposed a set of heuristics to partition the large graph dataset in a streaming fashion.

1) Heuristics: 10 heuristics were proposed in [16]. Instead of repeating all heuristics one by one, we review pros and cons on those heuristics.

The notation $P^t$ refers to the set of the partition at time $t$. Each individual partition is referred to by its index $P^t(i)$ so $\bigcup_{t=1}^{\infty} P_t(i)$ is equal to all of the vertices placed so far. Let $v$ denote the vertex that arrives at time $t$ in the stream, $\Gamma(v)$ refers to the set of vertices that $v$ neighbours and $|S|$ refers to the number of elements in a set $S$. $C$ is the capacity constraint on each partition. Each of the heuristics gives an algorithm for selecting the index $ind$ of the partition where $v$ is assigned. The first seven heuristics do not use a buffer, while the last three do.

1) Balanced - Assign $v$ to a partition of minimal size, breaking ties randomly: $ind = \arg \min_{i \in [k]} |P^t(i)|$

2) Chunking - Divide the stream into chunks of size $C$ and fill the partitions completely in order: $ind = \left\lfloor t/C \right\rfloor$

3) Hashing - Given a hash function $H(v) = (v \mod k) + 1$, assign $v$ to $ind = H(v)$.

4) (Weighted) Deterministic Greedy - Assign $v$ to the partition where it has the most edges. Weight this by a penalty function based on the capacity of the partition, penalizing larger partitions. Break ties using Balanced.

$$ind = \arg \max_{i \in [k]} |P^t(i) \cap \Gamma(v)| w(t,i)$$  (12)

where $w(t,i)$ is a weighted penalty function. $w(t,i) = 1$ for unweighted greedy $w(t,i) = 1 - |P^t(i)|/|P^t(i)|$ for linear weighted $w(t,i) = 1 - \exp(|P^t(i)| - C)$ for exponentially weighted

5) (Weighted) Randomized Greedy - Assign $v$ according to the distribution defined by

$$Pr(i) = |P^t(i) \cap \Gamma(v)| w(t,i) / Z$$  (13)

where $Z$ is the normalizing constant and $w(t,i)$ is the above 3 penalty functions.

6) (Weighted) Triangles - Assign $v$ according to

$$\arg \max_{i \in [k]} \left\{ \frac{|E(P^t(i) \cap \Gamma(v), P^t(i) \cap \Gamma(v))|}{2} w(t,i) \right\}$$  (14)

where $w(t,i)$ is the above 3 penalty functions and $E(S,T)$ is the set of edges between the nodes in $S$ and $T$.

7) Balance Big - Given a way of differentiating high and low degree nodes, if $v$ is high-degree, use Balanced. If it is low-degree, use Deterministic Greedy. The following heuristics all use a buffer.

8) Prefer Big - Maintain a buffer of size $C$. Assign all high degree nodes with Balanced, and then stream in more nodes. If the buffer is entirely low degree nodes, then use Deterministic Greedy to clear the buffer.

9) Avoid Big - Maintain a buffer of size $C$ and a threshold on large nodes. Greedily assign all small nodes in the buffer. When the buffer is entirely large nodes, use Deterministic Greedy to clear the buffer.

10) Greedy EvoCut - Use EvoCut on the buffer to find small Nibbles with good conductance. Select a partition for each Nibble using Deterministic Greedy.

Overall, each of these heuristics has a different motivation with some arguably more natural than others.

Balanced and Chunking are simple ways of load balancing while ignoring the graph structure.

Hashing is currently used by many real systems. The benefit of Hashing is that every vertex can be quickly found, from any machine in the cluster, without the need to maintain a distributed mapping table. If the IDs of the nodes are
consecutive, the hash function \( H(v) = (v \mod k) + 1 \) makes Balanced and Hashing equivalent. More generally, a pseudorandom hash function should be used, making Hashing equivalent to a random cut.

The greedy approach is standard, although the weighted penalty is inspired by analysis of other online algorithms. The randomized versions of these algorithms were explored because adding randomness can often be shown to theoretically improve the worst-case performance.

The (Weighted) Triangles heuristic exploits work showing that social networks have high clustering coefficients by finding triangles completed triangles among the vertices neighbours in a partition and overweighting their importance.

Heuristics Balance Big, Prefer Big and Avoid Big assume we have a way to differentiate high and low degree nodes. This assumption is based on the fact that many graphs have power law degree distributions. These three heuristics propose different treatments for the small number of high degree nodes and the large number of low degree nodes.

Balance Big uses the high degree nodes as seeds for the partitions to "attract" the low degree nodes. The buffered version, Prefer Big, allows the algorithm more choice in finding these seeds.

Avoid Big explores the idea that the high degree nodes form the expander portion of the graph, so perhaps the low degrees nodes can be partitioned after the high degree nodes have been removed.

Avoid Big explores the idea that the high degree nodes form the expander portion of the graph, so perhaps the low degree nodes can be partitioned after the high degree nodes have been removed.

The final heuristic, Greedy EvoCut, uses EvoCut, a local partitioning algorithm, on the buffer. This algorithm has very good theoretical guarantees with regards to the found cuts, and the amount of work spent to find them, but the guarantees do not apply to the way we use it.

2) **Stream Orders:** The performance of a heuristic highly depends on the streaming order. In [16], the following three stream orderings were considered - Random, BFS and DFS. Each of these stream orderings has a different justification.

The random ordering is a standard assumption when theoretically analysing streaming algorithms. While we generate these orderings by selecting a random permutation of the vertices, one could view this as a special case of a generic ordering that does not respect connectivity of the graph. The benefit of a random ordering is that it avoids adversarial bad orderings. The downside is that it does not preserve any locality in the edges so we expect it to do poorly for statistical reasons like the Birthday paradox. Via the Birthday paradox, we can argue that for sparse graphs, we expect to go through \( O(n) \) of the vertices before we find a first edge.

Both BFS and DFS are natural ways of linearizing graphs and are highly simplified models of a web crawler. In practice, web crawlers are a combination of local search approaches - they follow links, but fully explore domains and sub-domains before moving on. This is breadth-first search between domains, and depth-first search within. The main benefit of both orderings is that they guarantee that the partitioner sees edges in the stream immediately. Additionally, they maintain some locality. Each has their drawbacks, but it should be noted that BFS is a subroutine that is often used in partitioning algorithms to find a good cut, particularly for rounding fractional solutions to LPs.

3) **Experimental Results:** The paper used several sources to collect multiple datasets for the experiments. From the SNAP archive, we used soc-Slashdot0811, wiki-Vote and web-NotreDame. From the Graph Partitioning Archive: 3elt, 4elt, and vibrobox. Besides, Astrophysics collaborations (astroph), C. Elegans Neural Network (celegans), and the Marvel Comics social network. The paper used two large social networks (LiveJournal and Twitter) to evaluate the heuristics in a real system.

![Figure 10: PL1000 results. The top line is the cost of a random cut and the bottom line is METIS. The best heuristic is Linear Deterministic Greedy[16].](image)

Figure 10 depicts the performance on the PowerLaw Clustered graph of size 1,000 with 4 partitions. This is one of the synthetic graphs where the model is intended to capture power law graphs observed in nature. The lower bound provided by METIS is 58.9% of the edges cut, while the upper bound for 4 partitions is 75%. The first heuristic, Avoid Big, is worse than a random cut. Linear Deterministic Greedy and Balance Big both perform very well for all 3 stream orderings. These each had a best average performance of 61.7% and 63.2% of the edges cut respectively, corresponding to 82% and 73% of the possible gain in performance. This gain was calculated as the fraction of edges cut by the random minus the fraction cut by the heuristic, divided by the fraction cut by a random cut minus the fraction cut by METIS, i.e., \( \frac{\text{random-heuristic}}{\text{random-METIS}} \).

As shown in Figure 11, the 50,000 node PowerLaw Clustered graph, but all graphs have similar characteristics. The heuristics performance closely tracks that of METIS.
D. Structural Balanced Vertex-cut based Graph Partitioning

Previously, we’ve reviewed three edge-cut based algorithms. However, there is another type of algorithm, the vertex-cut, by which the graph is divided into several components with equal number of edges and minimum number of disjoint vertices.

1) Importance of vertex-cut: Apparently, the only difference between vertex-cut and edge-cut is the object which spans across partitions. However, when it comes to distributed graph computing system, the difference becomes prominent.

2) Structural Balanced Vertex-cut: In [11], the author proposed a Structural Balanced Vertex-cut approach, shorten as SBV-cut.

As shown in Figure 13, the basic steps are as follows:


2. Compute min-max ratio by the equation below:

\[
\text{minmax}(v) = \frac{\max\{\text{asd}(S,v), \text{asd}(v,T)\}}{\min\{\text{asd}(S,v), \text{asd}(v,T)\}}
\]

where \(\text{asd}(S,v)\) stands for the average shortest-path distance from the source vertices \(S\) to \(v\) and \(\text{asd}(v,T)\) stands for the average shortest-path distance from \(v\) to the sink vertices \(T\).

3. Compute the transition matrix \(M\) according to the equation below:

\[
m_{ij} \times \left( \sum_{v_h \in \text{neighbors}(v)} \frac{\text{minmax}(v)}{\text{minmax}(v_h)} \right) = 1.0
\]

where \(m_{ij}\) is the entry of transition matrix \(M\).

4. Compute balance score through Eigen-decomposition of \(M\), in which we find the eigenvector corresponding to eigenvalue of 1 such that \(\sum x_i = 1.0\), and \(x_i\) is the balance score of vertex \(v_i\). The the dominant node in this iteration is the one with the highest balance score.

5. Iteratively perform a vertex-cut by the dominant node which is associated with highest \(x_i\) among all remaining vertices, see Figure 15. In this step, all the nodes that can reach to the dominant node are included in one partition, and those who are reachable from the dominant node are included in another.

6. Repeat above steps for the biggest partition until \(k\) partitions are found.
3) **Two Variations:** Since during the above steps, the algorithm aggressively include those nodes which can reach to or be reachable from the dominant node into partitioning result, the author name it an *optimistic SBV-cut* strategy.

However, by *optimistic SBV-cut*, the partitioning result highly depends on the initial choice of the dominant node. To reduce such impact, the author proposed another two variations, which are *pessimistic SBV-cut* and *balance-recomputed SBV-cut*.

In *pessimistic SBV-cut*, instead of including the vertices on all paths of the selected balance vertices into partitioning result, pessimistic SBV-Cut include only those vertices that are not on any remaining source-to-sink paths into partitioning result. , see Figure 16.

In *balance-recomputed SBV-cut*, the balance scores of the remaining vertices are re-computed at each iteration, thus, the impact of the initial choice of dominant node is minimized.

4) **Experimental Results:** The author of this paper also carried out the comparison between *optimistic SBV-cut*, *pessimistic SBV-cut* and *balance-recomputed SBV-cut*.

Overall, the optimistic strategy generates the partition with lowest quality, while spending least running time. And re-compute strategy generates the partition with highest quality, but spend the most running time, as show in Figure 17.

However, the optimistic algorithm never fails. But the pessimistic one and recomputed one seem to be pretty hesitated to reach a final solution, as show in Figure 18.

And the author of this paper also compare this algorithm with the multilevel approach we’ve talked in IV-A, which is a typical edge-cut algorithm. To make sure SBV-cut is comparable to the multilevel approach, the author just convert the vertex-cut into an edge-cut by simply inserting some edges between those disjoint vertices in a vertex-cut, see Figure 19.

As shown in Figure 20, the multilevel approach keeps almost constant running time as the size of graph increases, however, SBV looks quite time consuming, especially for large-scale network partitioning, and the most amount time was spend in shortest path finding process as well as dominant node finding process.
Impressively, as shown in Figure 21, in most cases, the quality of SBV-cut exceeds what in multilevel approach which is a typical edge-cut partitioning algorithm.

Another example can be seen in the streaming graph partitioning, where 10 different heuristics are provided for us to carry out some trade-offs between partitioning quality as well as the running time.

4) **Think out of the box.**

When considering graph partitioning problem, what we usually consider is to partition a graph into several components that holds the same number of vertices while minimizing the edge-cut. However, vertex-cut provides us a new perspective to look at the partitioning problems which successfully minimize the impact of power-law degree distribution to the data sharding problem in distributed computing system.

### B. Tao of Distributed Network-centric Computing

And our original motivation is to study the network data partitioning for distributed computing system. So, when we try to apply graph partitioning algorithms to our distributed system, there are still some principles we have to bear in mind:

1) **Performance**
   
   This is the reason why we prefer a 1-pass streaming partitioning algorithm.

2) **Minimize Inter-communication Across Machines**
   
   This means that we have to minimize the cut-size and boost the data locality.

3) **Keep Load Balanced**
   
   In order to fully utilize the storage resources and avoid bottleneck effect that may potentially come from a certain machine, we have to take load-balancing into account. And in terms of this, vertex-cut wins, especially when we are facing with the network with power-law degree distribution.

4) **Be Deterministic**

   The system has to be simple, which favors the deterministic, which means, it is better for a partitioning algorithm to generate the result partitions where the data can be easily allocated. So, that’s why hashing, deterministic greedy, and shardmap is what we prefer.

### C. Comparisons

By understanding these principles, we take a simple review on those four algorithms we’ve discussed in previous section.

Overall, to address a graph partitioning problem, we may have three basic steps - Preprocessing, Partitioning and Refinement. Since we have reviewed three edge-cut graph partitioning algorithms which are comparable to each other following these steps, we now examine the similarities and the differences between these 3 edge-cut algorithms.

1) **Preprocessing:**

   In both [5] and [13], the function of preprocessing step are quite similar in terms of the actual effect yet slight different in terms of the mechanism. The similarity
exists in the fact that both preprocessing steps aims to boost the performance of the actual partitioning phase and improve the quality of partitioning result, and both of them succeed in generating rough partitions. However, in [5], the proposed coarsening phase create coarser graph only with the dependence on the characteristics of the entire graph structure, while in [13], the rough partitions were created in the preprocessing step with the utilization of the geographic information of all the users in the social network, which is represented as vertices in the graph data. Nevertheless, it’s interesting to note that, as a 1-pass graph partitioning approach which is capable of generating acceptable result, streaming graph partitioning may also be used as a preprocessing step for other high-quality partitioning algorithms.

2) **Partitioning:**
In both [5] and [13], the actual partitioning steps were carried out to generate better partitioning result. However, in [5], the actual partitioning step is performed by running a real graph partitioning algorithm, which means the number of partitions are already specified, while in [13], the partitioning step is done by running a community detection algorithm under the constraints of partition size.

3) **Refinement:**
In both [5] and [13], the refinement steps are quite similar. As the former one swaps nodes for edge-cut reduction, the latter one tries to relocate vertices to improve the data locality. [5] uses Boundary Kernighan-Lin algorithm which is basically a node-swapping technique between partitions to further reduce the edge-cut, while in [13], the vertex data is moved between partitions to increase the colocation count which is the number of graph neighbors the vertices are co-located with.

**VI. Future Work**

Graph partitioning problem is a well-studied research area with a huge amount of work. However, to apply them on the distributed graph computing system for better performance and reliability is what needs to be studied, testified and practiced. In the future, we would like to study more about the application of graph partitioning to distributed computing system, examine the impact of different network data to the partitioning result in terms of the distributed computing performance boost. Also, to design a partitioning algorithm which is able to run in a streaming style with high partitioning quality ready for graph data mining and machine learning is something even more exciting in the further study.
References


