Buffered Streaming Edge Partitioning

Adil Chhabra
Heidelberg University, Germany

Marcelo Fonseca Faraj
Heidelberg University, Germany

Christian Schulz
Heidelberg University, Germany

Daniel Seemaier
Karlsruhe Institute of Technology, Germany

Abstract

Addressing the challenges of processing massive graphs, which are prevalent in diverse fields such as social, biological, and technical networks, we introduce HeiStreamE and FreightE, two innovative (buffered) streaming algorithms designed for efficient edge partitioning of large-scale graphs. HeiStreamE utilizes an adapted Split-and-Connect graph model and a Fennel-based multilevel partitioning scheme, while FreightE partitions a hypergraph representation of the input graph. Besides ensuring superior solution quality, these approaches also overcome the limitations of existing algorithms by maintaining linear dependency on the graph size in both time and memory complexity with no dependence on the number of blocks of partition. Our comprehensive experimental analysis demonstrates that HeiStreamE outperforms current streaming algorithms and the re-streaming algorithm 2PS in partitioning quality (replication factor), and is more memory-efficient for real-world networks where the number of edges is far greater than the number of vertices. Further, FreightE is shown to produce fast and efficient partitions, particularly for higher numbers of partition blocks.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases graph partitioning, edge partitioning, streaming, online, buffered partitioning

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.5


Supplementary Material Software (Source Code): https://github.com/KaHIP/HeiStream

Acknowledgements We acknowledge support by DFG grant SCHU 2567/5-1. This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 882500). Moreover, we would like to acknowledge Dagstuhl Seminar 23331 on Recent Trends in Graph Decomposition [30].

1 Introduction

Complex, large graphs, often composed of billions of entities, are employed across multiple fields to model social, biological, navigational, and technical networks. However, processing huge graphs requires extensive computational resources, necessitating the parallel computation of graphs on distributed systems. Large graphs are partitioned into sub-graphs distributed among \( k \) processing elements (PEs). PEs perform computations on a portion of the graph, and communicate with each other through message-passing. Graph partitioning models the distribution of graphs across PEs such that each PE receives approximately the same number of components (vertices) and communication between PEs (via edges between them) is minimized. Edge partitioning, which outperforms traditional vertex partitioning on real-world power-law graphs [22, 32, 34], partitions edges into \( k \) blocks such that vertex replication...
5:2 Buffered Streaming Edge Partitioning

is minimized, hence minimizing the communication needed to synchronize vertex copies. Graph vertex and edge partitioning are NP-hard [8, 21] and there can be no approximation algorithm with a constant ratio factor for general graphs unless \( P = NP \) [10]. Thus, heuristic algorithms are used in practice. Further, due to data proliferation, streaming algorithms are increasingly being used to partition huge graphs quickly with low computational resources [1, 3, 17, 18, 19, 24, 25, 35, 50, 51].

Streaming edge partitioning entails the sequential loading of edges for immediate assignment to blocks. One-pass streaming edge partitioners permanently assign edges to blocks during a single sequential pass over the graph’s data stream [41, 53]. Alternatively, buffered streaming algorithms receive and store a buffer of vertices along with their edges before making assignment decisions, thus providing information about future vertices [18, 35], and re-streaming algorithms gather information about the global graph structure [36, 37]. With a few exceptions [37, 53] most streaming edge partitioners have a high time complexity due to a linear dependency on the number of blocks \( k \). However, in recent years, high \( k \) values are frequently used in graph partitioning due to the increasing size of graphs, complexity of computations, and availability of processors. An existing re-streaming edge partitioner, 2PS-L [37], achieves a linear runtime independent of \( k \), but produces lower solution quality than state-of-the-art partitioners and has a linear memory dependence on \( k \). Thus, there remains potential to explore high-quality streaming edge partitioners without a runtime and memory dependency on \( k \).

Contribution. We propose \( \text{HeiStreamE} \), a buffered streaming algorithm for edge partitioning that leverages the performance efficacy of multilevel algorithms. By employing an adapted version of the SPlit-And-Connect (SPAC) model [33] and solving it with a Fennel-based multilevel scheme [18], our algorithm produces superior solution quality while maintaining time and memory complexities that are linearly dependent on the size of the graph and independent of \( k \). Our results establish the superiority of \( \text{HeiStreamE} \) over all current streaming algorithms, and even the re-streaming algorithm 2PS [36, 37], in replication factor. These outcomes highlight the considerable potential of our algorithm, positioning it as a promising tool for edge partitioning. We additionally provide an implementation of an efficient streaming edge partitioner, \( \text{FreightE} \), which uses streaming hypergraph partitioning [17] to partition edges on the fly. Our experiments demonstrate that \( \text{FreightE} \) is significantly faster than all competing algorithms, especially for high \( k \) values.

2 Preliminaries

2.1 Basic Concepts

(Hyper)Graphs. Let \( G = (V = \{0, \ldots , n - 1\}, E) \) be an undirected graph with no multiple or self-edges, such that \( n = |V| \), \( m = |E| \). Let \( c : V \rightarrow \mathbb{R}_{\geq 0} \) be a vertex-weight function, and let \( \omega : E \rightarrow \mathbb{R}_{\geq 0} \) be an edge-weight function. We generalize \( c \) and \( \omega \) functions to sets, such that \( c(V') = \sum_{v \in V'} c(v) \) and \( \omega(E') = \sum_{e \in E'} \omega(e) \). An edge \( e = (u, v) \) is said to be incident on vertices \( u \) and \( v \). Let \( N(v) = \{ u : (v, u) \in E \} \) denote the neighbors of \( v \). A graph \( S = (V', E') \) is said to be a subgraph of \( G = (V, E) \) if \( V' \subseteq V \) and \( E' \subseteq E \cap (V' \times V') \). If \( E' = E \cap (V' \times V') \), \( S \) is an induced subgraph. Let \( d(v) \) be the degree of vertex \( v \) and \( \Delta \) be the maximum degree of \( G \). Let \( H = (V, E) \) be an undirected hypergraph with \( n = |V| \) vertices, \( m = |E| \) hyperedges or nets. A net, unlike an edge of a graph, may consist of more than two vertices, and is defined as a subset of \( V \).
**Partitioning.** Given a number of blocks \( k \in \mathbb{N}_{\geq 1} \), and an undirected (hyper)graph with positive edge weights, the (hyper)graph partitioning problem pertains to the partitioning of a (hyper)graph into \( k \) smaller (hyper)graphs by assigning the vertices (vertex partitioning) or (hyper)edges (edge partitioning) of the graph to \( k \) mutually exclusive blocks, such that the blocks have roughly the same size and the particular objective function is minimized or maximized. More precisely, a \( k \)-vertex partition of a (hyper)graph partitions \( V \) into \( k \) blocks \( V_1, \ldots, V_k \) such that \( V_1 \cup \cdots \cup V_k = V \) and \( V_i \cap V_j = \emptyset \) for \( i \neq j \). The edge-cut (resp. cut net) of a \( k \)-partition consists of the total weight of the cut edges (resp. cut nets), i.e.,

\[
\Lambda(G) := \sum_{i=1}^{k} \omega\left(V_i \right)
\]

where \( \omega \) is the cardinality of its connectivity set, i.e., \( \omega(V) = |\Lambda(V)| \). The so-called connectivity metric (\( \lambda - 1 \)) is computed as \( \sum_{e \in E'} (\lambda(e) - 1) \omega(e) \), where \( E' \) is the cut-set.

Similarly, a \( k \)-edge partition of a graph partitions the edge set \( E \) into \( k \) blocks \( E_1, \ldots, E_k \) such that \( E_1 \cup \cdots \cup E_k = E \) and \( E_i \cap E_j = \emptyset \) for \( i \neq j \). In edge partitioning, a common objective function is the minimization of the replication factor, which is defined as the number of replicated vertices divided by the total number of vertices in the graph. Formally, we define the set \( V(E_i) = \{ v \in V \mid \exists u \in V : (u, v) \in E_i \} \) for each partition \( E_i \) as the number of vertices in \( V \) that have at least one edge incident on them that was assigned to block \( E_i \). Taking the sum of \( |V(E_i)| \) over all \( k \) gives us the total number of vertex replicas generated by the partition. Replication factor is then defined as \( RF(E_1, E_2, \ldots, E_k) = \frac{1}{n} \sum_{i=1}^{k} |V(E_i)| \). Intuitively, a minimized replication factor suggests that vertices are replicated in minimum blocks. Minimum vertex replication, in turn, results in lower synchronization overhead in distributed graph processing due to reduced exchange of vertex state across blocks.

**Multilevel Scheme.** A successful heuristic for vertex partitioning is the multilevel [14] approach. It recursively computes a clustering and contracts it to coarsen the graph into smaller graphs that maintain the same basic structure as the input graph. An initial partitioning algorithm is applied to the smallest (coarsest) graph and then the contraction is undone. At each level, a local search method is used to improve the partitioning induced by the coarser level. Contracting a cluster of vertices \( C = \{u_1, \ldots, u_l\} \) involves replacing them with a new vertex \( v \) whose weight is the sum of the weights of the clustered vertices and is connected to all elements \( w \in \bigcup_{i=1}^{l} N(u_i) \), such that \( w \not\in C \), with edge weight \( \omega(\{v, w\}) = \sum_{i=1}^{l} \omega(\{u_i, w\}) \). This ensures that the transfer of partitions from a coarser to a finer level maintains the edge-cut. The uncontraction of a vertex undoes the contraction. Local search moves vertices between blocks to reduce the objective.

**SPAC Transformation.** The SPAC transformation [33] provides a means to employ a vertex partitioning tool on a transformed graph \( G' \), which is derived from the original graph \( G \), and subsequently apply the derived vertex partition to establish an edge partition for \( G \). The transformation assumes an undirected, unweighted graph \( G = (V, E) \) as input. The SPAC graph \( G' = (V', E') \) is then constructed in two phases: In the split phase, each vertex \( v \in V \) generates \( d(v) \) split vertices \( S_v := \{v'_1, \ldots, v'_d(v)\} \subseteq V' \). The connect phase introduces two kinds of edges in \( E' \), dominant edges and auxiliary edges. First, it assigns a dominant edge \( e' = (u'_i, v'_j) \) in \( G' \) for each edge \( e = (u, v) \) in \( G \). Dominant edges are created with...
Buffered Streaming. In the buffered streaming model, which is an extended version of the one-pass model, we load a δ-sized buffer or batch of input vertices along with their edges. We make block assignment decisions only after the entire batch has been loaded. In practice, the parameter δ can be chosen in accordance with memory available on the machine. In our contribution, we use a fixed δ throughout the algorithm. For a predefined batch size of δ, we load and repeatedly partition \( \lceil n/\delta \rceil \) batches.

2.2 Related Work

We refer the reader to recent surveys on (hyper)graph partitioning for relevant literature [11, 14, 47]. Here, we focus on the research on streaming vertex and edge partitioning. Most high-quality vertex partitioners for real-world graphs use a multilevel scheme, including KaHIP [44], METIS [28], Scotch [40], and (mt)-KaHyPar [23, 45]. Edge partitioning has been solved directly with multilevel hypergraph partitioners, including PaToH [13], hMETIS [29], KaHIP [46], Mondriaan [52], MLPart [2], Zoltan [15], SHP [27], UMPa [12], and kPaToH [13].

Streaming (Hyper)Graph Vertex Partitioning. Tsourakakis et al. [51] introduce Fennel, a one-pass partitioning heuristic adapted from the clustering objective modularity [9]. Fennel minimizes edge-cuts by placing vertices in partitions with more neighboring vertices. Fennel assigns a vertex \( v \) to the block \( V_i \) that maximizes the Fennel gain function \( |V_i \cap \Delta(v)| - f(|V_i|) \), where \( f(|V_i|) \) is a penalty function to respect a balancing threshold. The authors define the Fennel objective with \( f(|V_i|) = \alpha \gamma \cdot |V_i|^{\gamma - 1} \), in which \( \gamma \) is a free parameter and \( \alpha = n^{3\gamma / (\gamma - 1)} \). After parameter tuning, the authors define \( \gamma = \frac{3}{2} \) and \( \alpha = \sqrt{k_{n/\delta}} \). The time complexity of the algorithm depends on \( k \) and is given by \( O(nk + m) \). Stanton and Kliot [48] propose LDG, a greedy heuristic for streaming vertex partitioning. ReLDG and ReFennel are re-streaming versions of LDG and Fennel [39]. Prioritized re-streaming optimizes the ordering of the streaming process [3], Furaj and Schulz [18] propose HeiStream, which uses a generalized weighted version of the Fennel gain function in a buffered streaming approach. Eyubov et al. [17] introduce FREIGHT, a streaming hypergraph partitioner that adapts the Fennel objective function to partition vertices of a hypergraph on the fly.

Streaming Edge Partitioning. One-pass streaming edge partitioners include hashing-based partitioners like DBH [53], constrained partitioners like Grid and PDS [26], and HDRF, proposed by Petroni et al. [41]. HDRF exploits the skewed degree distribution of real-world graphs by prioritizing vertex replicas of high-degree vertices. HDRF outperforms DBH, Grid and PDS in solution quality, but has a longer runtime. Zhang et al. [54] introduced SNE, a streaming
version of the in-memory edge partitioner NE that utilizes sampling methods. SNE produces better solution quality than HDRF, but with increased memory consumption and runtime [37]. In contrast to one-pass streaming models, RBSEP uses a buffered approach to postpone assignment decisions for edges with limited neighborhood partitioning information available during streaming [49]. Additionally, Mayer et al. [35] introduce ADWISE, a window-based streaming edge partitioner, which uses a dynamic window size that adapts to runtime constraints.

Mayer et al. [36] subsequently propose 2PS-HDRF, a two-phase re-streaming algorithm for edge partitioning, using HDRF as the scoring function in its final partitioning step. The first phase uses a streaming clustering algorithm to gather information about the global graph structure; in the second phase, the graph is re-streamed and partitioned, using information obtained from clustering to make edge partitioning decisions. Mayer et al. [37] modify 2PS-HDRF to propose 2PS-L, which runs in time independent of \( k \). 2PS-L switches from HDRF to a new scoring function in the final partitioning step to remove its dependency on \( k \), and thus achieves a time complexity of \( \mathcal{O}(|E|) \). 2PS-L outperforms ADWISE; it is faster than HDRF and 2PS-HDRF, particularly at large \( k \) values, but has lower solution quality. 2PS-HDRF achieves 50% better solution quality than 2PS-L [37].

Sajjad et al. [43] propose HoVerCut, a platform for streaming edge partitioners, which can scale in multi-threaded and distributed systems by decoupling the state from the partitioner. Hoang et al. [24] propose CuSP, a distributed and parallel streaming framework to partition edges based on user-defined policies. CuSP is programmable and can express common streaming edge partitioning strategies from the literature.

### 3 Buffered Streaming Edge Partitioning

In this section, we present our algorithms, HeiStreamE and FreightE. First, we provide an overview of HeiStreamE’s iterative structure. Subsequently, we detail its input format and buffered graph model, and describe how it uses multilevel vertex partitioning to solve this model. Lastly, we discuss how FreightE builds a hypergraph representation to partition edges using a streaming hypergraph partitioner.

#### 3.1 Overall Algorithm

Our framework draws inspiration from Heistream [18]. We slide through the input graph \( G \) by iteratively performing the following series of operations until all the edges of \( G \) are assigned to blocks. First, we load a batch composed of \( \delta \) vertices and their associated neighborhood, thereby obtaining a subgraph \( G_b \) contained within the graph \( G \). This operation yields edges connecting vertices within the current batch, and edges connecting vertices in the current batch to vertices streamed in previous batches. Second, we build a model \( \beta_b \) corresponding to \( G_b \), where the edges of \( G_b \) are transformed into vertices. Additionally, we incorporate a representation of block assignments from previous batches into \( \beta_b \). Third, we partition

```plaintext
Algorithm 1 Overall Structure of HeiStreamE.
1: for \( b \in \{1, \ldots, \lceil \frac{n}{\delta} \rceil \} \) do
2: Load subgraph \( G_b \) from input graph \( G \)
3: Build model \( \beta_b \) from \( G_b \)
4: Run multilevel vertex partitioning on \( \beta_b \)
5: Permanently assign corresponding edges of \( G \) in \( G_b \) to blocks
```
Buffered Streaming Edge Partitioning

**Figure 1** Detailed structure of HeiStreamE. The algorithm starts by loading a batch graph $G_b$ consisting of vertices and their edges to the current batch and previous batches. Subsequently, it builds a meaningful model $\beta_b$ from the batch graph, transforming edges into vertices, and incorporating a synthetic representation of the assignments made in previous batches. This model is then partitioned using a multilevel algorithm. Lastly, the edges from the loaded batch, which correspond to vertices in the partitioned batch model, are permanently assigned to blocks. This process is repeated for subsequent batches until the entire graph has been partitioned.

$\beta_b$ using a multilevel vertex partitioning algorithm that has been shown to be effective in the context of buffered streaming [18]. We conclude by permanently assigning the edges in $G$ that correspond to vertices in our model $\beta_b$ to their respective blocks. Algorithm 1 summarizes the general structure of HeiStreamE, which is illustrated in Figure 1.

### 3.2 Input and Batch Format

HeiStreamE uses a vertex-centric input format and a buffered streaming approach, which refers to the sequential process of loading and handling the input graph in batches. Within each batch, it loads $\delta$ vertices one at a time along with their neighborhood, where $\delta$ is a parameter that defines the buffer size. Similar input formats are commonly used in streaming algorithms for vertex partitioning [3, 17, 18, 19, 25, 51] and are consistent with graph formats commonly found in publicly available real-world graph datasets, such as the METIS format. Each batch $b \in \{1, \ldots, \lceil n/\delta \rceil\}$ corresponds to a subgraph denoted as $G_b = (V_b, E_b)$ within the graph $G$. This subgraph is constructed as follows. Its vertex set $V_b$ includes the $\delta$ vertices from the current batch, labeled within the domain $[0, \delta - 1]$, as well as the $p$ vertices from past batches that share at least one edge with vertices in the current batch, labeled within the domain $[\delta, \delta + p - 1]$. Similarly, its edge set $E_b$ comprises of edges with one endpoint in the current batch and the other endpoint in either the current batch or previous batches, expressed as $E_b = \{(u, v) \in E \mid \text{Batch}(u) = b \land \text{Batch}(v) \leq b\}$. Edges with an endpoint in future batches are discarded, ensuring that each edge belongs exclusively to a unique batch graph $G_b$ across all batches.

### 3.3 Model Construction

Our model construction consists of two steps, detailed in this section. First, using the batch graph $G_b$, we create a corresponding Contracted Split-And-Connect (CSPAC) graph, denoted as $S^*_b$. In $S^*_b$, edges from $G_b$ are directly represented as vertices, while vertices from $G_b$ are indirectly represented as edges. Subsequently, we create a graph model $\beta_b$ based on $S^*_b$, which incorporates a representation of block assignments from prior batches.
Figure 2: Building SPAC and CSPAC Graphs: The SPAC graph, denoted as \( G' \), features \( d(v) \) split vertices for every vertex \( v \) in the original graph \( G \). These split vertices are represented in the same color as the vertex they originate from. Every (thick green) edge from \( G \) is directly converted into a distinct dominant (thick green) edge in \( G' \) that connects corresponding split vertices. The auxiliary (thin) edges in \( G' \), which create a path between split vertices, are depicted in the same color as the split vertices they link. The CSPAC graph \( G^* \) is formed by contracting the dominant edges of \( G' \). The vertices in \( G^* \) represent the edges in \( G \), while the edges in \( G^* \) mirror the auxiliary edges in \( G' \).

CSPAC Transformation. The CSPAC transformation is a faster and more condensed variant of the SPAC transformation conceived by Li et al. [33], as described in Section 2.1. Below we explain how the SPAC transformation evolves into the CSPAC transformation, and how this transformation is applied to \( G_b \) to yield the CSPAC graph \( S^*_b \).

The SPAC graph \( G' \) of a graph \( G \) consists of dominant edges, i.e., edges with a weight of infinity that have a one-to-one correspondence with edges of the original graph, and auxiliary edges, which define a path between vertices of the SPAC graph. The CSPAC transformation is derived from the SPAC graph by contracting the dominant edges into vertices. Due to construction, the dominant edges in \( G' \) do not share any endpoints; they effectively form a matching, ensuring a consistent contraction. Further, each endpoint of the auxiliary edges is incident to a single dominant edge. Thus, the contraction of all dominant edges produces a coarser graph, in which every vertex represents a unique edge in the original graph \( G \), and their connections correspond to the auxiliary edges in graph \( G' \). The CSPAC transformation is illustrated in Figure 2.

In HeiStreamE, we derive the CSPAC graph \( S^*_b \) directly from the batch graph \( G_b \), bypassing the construction of the intermediary SPAC graph. The procedure for this direct transformation is as follows. For each vertex \( u \in V_b \), each of its outgoing edges \( e = (u, v) \in E_b \) induces a vertex \( u^* \) in \( S^*_b \), if \( u < v \) to avoid redundancy. As each vertex is constructed, it is connected to at most two other vertices in \( S^*_b \) that are induced by other outgoing edges of \( u \) in \( G_b \) to form a path. This way, each undirected edge \( e \in E_b \), which would have induced a dominant edge in the SPAC graph, is represented as a vertex in \( S^*_b \). Further, each auxiliary edge of the SPAC graph is directly integrated into \( S^*_b \) to form paths between vertices in \( S^*_b \). The direct construction of the CSPAC graph maintains the same computational complexity as the SPAC construction alone, specifically \( O(|V_b| + |E_b|) \), where \( |V_b| \) and \( |E_b| \) are the number of vertices and edges of \( G_b \) respectively. In practice, building \( S^*_b \) directly from \( G_b \) offers a time-saving advantage compared to the alternative method of building the SPAC graph and then contracting the dominant edges.

In each batch, \( S^*_b \) has \( |E_b| \) vertices and \( 2|E_b| - |V_b| \) edges, i.e., it is linear in the size of the batch graph \( G_b \). In contrast to the SPAC graph, the CSPAC graph has fewer vertices and edges. Further, where a vertex partitioner might cut a dominant edge of the SPAC graph, every vertex partition of the CSPAC graph corresponds to a valid edge partition of the original graph. Thus, there is no need for a verification step to transform a vertex partition of \( S^*_b \) into an edge partition of \( G_b \).
Theorem 1 shows that when computing a vertex partition of $S^*_b$ to minimize the edge-cut, the corresponding edge partition of $G_b$ will also have a minimized number of vertex replicas. The SPAC approximation factor shown by Li et al. [33] is also directly valid for $S^*_b$.

\begin{theorem}
For any vertex partition $vp(S^*_b)$ of the CSPAC graph $S^*_b$ with edge-cut $cost(vp(S^*_b))$, there exists a corresponding edge partition $ep(G_b)$ of the batch graph $G_b$ with a number of vertex replicas $cost(ep(G_b))$, satisfying $cost(ep(G_b)) \leq cost(vp(S^*_b))$, which establishes a lower bound spanning the set of possible CSPAC graphs $S^*_b$ associated with $G_b$.
\end{theorem}

\begin{proof}
See Appendix B.
\end{proof}

**Integrating Connectivity Information.** Directly partitioning the CSPAC graph $S^*_b$ limits the partitioner’s view to the current batch, as $S^*_b$ does not take into account block assignments from previous batches. Specifically, when assigning a block to a vertex $u^* \in S^*_b$ induced by an edge $e = (u, v) \in E_b$, where $u$ is a vertex of the current batch and $v$ is a vertex of some previous batch, the partitioner might replicate $v$ into a new block in the absence of global information. To solve this, we extend the CSPAC graph $S^*_b$ with connectivity information derived from previous batch assignments to obtain the graph model $\beta_b$. We construct $\beta_b$ by augmenting $S^*_b$ with $k$ artificial vertices representing the $k$ partition blocks.

Each artificial vertex $i$ corresponds to an existing partition block $E_i$ in its current state, i.e., the weight of each artificial vertex $i$ is the weight of its corresponding block $E_i$ filled with edges that have been assigned to it from previous batches. An edge $e = (u, v) \in E_b$ of the current batch is represented in our model $\beta_b$ as a vertex that connects to an artificial vertex $i$ if $v$ is a vertex of a previous batch and an edge incident to $v$ has already been assigned to block $E_i$. However, in a streaming setting, there is limited knowledge of edge connectivity, i.e., it is not possible to directly determine which edges from previous batches are adjacent to edges from the current batch. To overcome this constraint, we maintain an array $B$ of size $n$ throughout the streaming process. This array records, for each vertex $u \in V$, block(s) assigned to edges incident on it. Then, in $\beta_b$, a vertex $u^* \in S^*_b$ induced by an edge $e = (u, v) \in E_b$ is connected to any artificial vertices $i$ representing blocks recorded in $B[v]$. An added benefit of this model construction is that we do not need to maintain a data structure of size $m$ in our algorithm.
We propose three configurations for $\beta_b$, which vary in how they use $B$, that we name, in decreasing level of exactness, maximal, $r$-Subset, and minimal. Each configuration has a runtime, memory and solution-quality trade-off. Let $u^*$ be a vertex in $S_b$ induced by an edge $e = (u,v) \in E_b$. In the maximal model $\beta_b$, $u^*$ is connected to all artificial vertices $i$ representing blocks recorded in $B[v]$. The memory required to store the array $B$ in the maximal setup is $O(nk)$. Theorem 2 demonstrates that computing a vertex partition of the maximal model $\beta_b$ to minimize edge-cut corresponds to an edge partition of $G_b$ that contributes a minimized number of new vertex replicas to the overall edge partition of the input graph $G$. In the $r$-Subset model, $u^*$ is connected to a sample of $r$ artificial vertices $i$ representing blocks recorded in $B[v]$, where $r$ is a parameter. Here, $B$ is identical as in the maximal setup, but the model $\beta_b$ is more concise, allowing for a faster partitioning phase. In the minimal model, $u^*$ is connected to a single artificial vertex $i$ representing the most recent block assigned to $v$ in a previous batch. We only store the latest assignment per vertex in $B$, thus the memory requirement is $O(n)$, and the model $\beta_b$ is also more concise than in the other two setups. We illustrate the various configurations in Figure 3.

**Theorem 2.** For any vertex partition $vp(\beta_b)$ of the maximal model $\beta_b$ with edge-cut cost($vp(\beta_b)$), there exists a corresponding edge partition $ep(G_b)$ of the batch graph $G_b$ that, when incorporated into the already partitioned section of the input graph $G$, introduces a number $\gamma$ of new vertex replicas, satisfying $\gamma \leq \text{cost}(vp(\beta_b))$.

Proof. See Appendix B.

### 3.4 Partitioning

In this section, we describe how we partition our model $\beta_b$. We employ a vertex partitioning algorithm on $\beta_b$, specifically an adapted version of the multilevel weighted Fennel algorithm utilized in HeiStream [18]. We describe this algorithm and then present a modification to the initial partitioning phase to enhance our runtime performance. Lastly, we discuss possible adaptations to the Fennel parameter $\alpha$.

**Multilevel Fennel.** Each per-batch graph model $\beta_b$ is partitioned using a multilevel partitioning scheme consisting of three successive phases, coarsening, initial partitioning, and uncoarsening.

In the coarsening phase, the algorithm computes a clustering and contracts the graph at each level until it is smaller than a specified threshold. These clusters are computed with label propagation while adhering to size constraints [38]. The clustering algorithm ignores artificial vertices and edges during the coarsening phase, to ensure that they are never contracted and that previous block assignment decisions are available at the coarsest level. For a graph with $n$ vertices and $m$ edges, a single round of size-constrained label propagation can be executed in $O(n + m)$ time. Initially, each vertex is placed in its own cluster, and in subsequent rounds vertices move to the cluster with the strongest connection, with a maximum of $L$ rounds, where $L$ is a tuning parameter. The coarsening phase ends as soon as the graph has fewer vertices than a threshold of $O(\max(|\beta_b|/k, k))$, where $|\beta_b|$ is the number of vertices of $\beta_b$. For large buffer sizes this threshold simplifies to $O(\beta_b|/k)$, while for small buffer sizes, it becomes $O(k)$.

Following the coarsening phase, an initial partitioning assigns all non-artificial vertices to blocks using the generalized Fennel algorithm [18] with a strict balancing constraint $L_{\text{max}}$. After initial partitioning, the current solution is transferred to the next finer level by mapping the block assignment of each coarse vertex to its constituent vertices at the
5:10 Buffered Streaming Edge Partitioning

finer level. Subsequently, a local search algorithm is applied at each level, which utilizes the size-constrained label propagation approach employed in the contraction phase but with a modified objective function. Specifically, when visiting a non-artificial vertex, we reassign it to a neighboring block to maximize the generalized Fennel gain function while strictly adhering to the balancing constraint $L_{\text{max}}$, considering only adjacent blocks in contrast to the initial partitioning which considers all blocks. This ensures that a single algorithm round remains linear in the current level’s size. Artificial vertices remain stationary during the process, but unlike during coarsening, they are not excluded from label propagation, as they contribute to the generalized Fennel gain function.

**Faster Initial Partitioning.** We adopt a modified Fennel function to enhance the initial partitioning step in HeiStreamE and demonstrate that this approach produces a better runtime without a decrease in solution quality.

For small buffer sizes, HeiStream has a linear dependency on $k$ for overall partitioning time, as during initial partitioning each vertex $u \in \beta_k$ is assigned to the block with the highest score among all $k$ blocks. To address this dependency, we adopt Eyubov et al.’s strategy [17] from streaming hypergraph partitioning to evaluate scores more efficiently, removing the runtime dependency on $k$. For the current vertex $u \in \beta_k$, we categorize $k$ blocks $V_i$ into two sets, $S_1$ and $S_2$, where $V_i \in S_1$ if a neighbor of $u$ was assigned to it, and $V_i \in S_2$ otherwise. This allows us to determine the blocks $V_{\text{max}}$ and $V'_{\text{max}}$ that respectively maximize Equation (1) and Equation (2). Then, the block that maximizes the generalized Fennel function is $\max(V_{\text{max}}, V'_{\text{max}})$ where

$$V_{\text{max}} = \arg\max_{i \in S_1} \left\{ \sum_{v \in V_i \cap N(u)} \omega(u, v) - c(u) f(c(V_i)) \right\} \tag{1}$$

$$V'_{\text{max}} = \arg\max_{i \in S_2} \left\{ -c(u) f(c(V_i)) \right\} \tag{2}$$

As $c(u)$ remains constant, determining the block $V_i \in S_2$ that maximizes Equation (2) is equivalent to finding the block $V_i$ minimizing $f(c(V_i)) = \alpha \ast \gamma \ast c(V_i)^{\gamma - 1}$, specifically, the $V_i \in S_2$ with the lowest block weight $c(V_i)$. In our scheme, $c(V_i) = c(E_i)$, that is, the number of edges assigned to block $E_i$ of the overall edge partition. This optimized process, facilitated by a priority queue, reduces the evaluation of blocks for each vertex to those assigned to its neighbors and the minimum weight block overall. It results in an optimal block $V_i$ for maximizing the generalized Fennel gain function in $O(d(u) + \log(k))$ time using a binary heap priority queue or $O(d(u))$ with a bucket priority queue, as suggested in [17], ultimately yielding an overall linear time complexity. With this enhanced approach, the runtime becomes independent of the parameter $k$.

**The Parameter $\alpha$.** The authors of Fennel [51], define the parameter $\alpha = \sqrt{k \frac{m}{m^*}}$ for vertex partitioning of an input graph $G = (V, E)$ with $n$ vertices and $m$ edges. However this choice of $\alpha$ is not directly applicable for edge partitioning in our CSPAC-based model. If we built a single model $\beta_k$ for the whole graph $G$ at once, it would have $n^* = m$ vertices and a number $m^*$ of edges equal to the number of auxiliary edges in $\beta_k$. While we know $n^*$ immediately, we cannot directly obtain $m^*$ without visiting all vertices of the whole graph. Thus, we need to estimate $m^*$. 


We thus have three distinct approaches for determining the Fennel parameter $\alpha$. In the static $\alpha$ method, we keep $\alpha$ constant across all batches, setting it to $\alpha = \sqrt{\frac{m_{approx}}{n^*}}$, where $n^* = m/2$, $m_{approx} = y + n^*$, and $y$ is a tuning parameter. For the batch $\alpha$ approach, we update $\alpha$ for each batch, calculating it as $\alpha = \sqrt{\frac{m}{n_s}}$, where $n_s$ and $m_s$ are the number of vertices and edges of each CSPAC graph $S^*_b$ respectively. The dynamic $\alpha$ method also updates $\alpha$ per batch. It begins with the static $\alpha$ setting and refines $m_{approx}$ in each batch by computing the number of auxiliary edges, determined through counting vertices of degree less than or equal to 2.

3.5 FreightE

In addition to HeiStreamE, we present FreightE, a fast streaming edge partitioner that uses streaming hypergraph partitioning to assign blocks to edges on the fly. In general, a hypergraph vertex partitioner can partition edges of an input graph $G$ by first transforming it into its dual hypergraph representation $H$, where each edge of $G$ is a hypervertex, and each vertex of $G$ induces a hyperedge spanning its incident edges. Then, a hypergraph vertex partitioner that assigns the hypervertices of $H$ into $k$ blocks, while optimizing for the connectivity metric, directly provides an edge partition of $G$. The intuition behind this approach is that a hypergraph vertex partitioner that optimizes for the connectivity metric directly optimizes the replication factor of the underlying edge partition [14]. In FreightE, we perform the transformation of $G$ into $H$ on the fly as follows. At a given step, we read a vertex of the input graph (METIS format) along with its neighborhood. Each undirected edge in the neighborhood is treated as a unique hypervertex of the hypergraph, and permanently assigned to a block using the FREIGHT streaming hypergraph partitioner [17]. This process is repeated until all vertices along with their neighborhoods are visited, at which point each edge of the input graph is assigned to a block. Following the FREIGHT partitioner, the overall runtime of FreightE is $O(n + m)$, and memory complexity is $O(m + k)$. In comparison to HeiStreamE, FreightE is faster, as it does not require the construction of an equivalent CSPAC graph, and requires less memory unless $m \gg n$.

4 Experimental Evaluation

Setup. We implemented HeiStreamE and FreightE inside the KaHIP framework (using C++) and compiled it using gcc 9.3 with full optimization enabled (-O3 flag). Except for the largest graph instance gsh-2015, all experiments were performed on a single core of a machine consisting of a sixteen-core Intel Xeon Silver 4216 processor running at 2.1 GHz, 100 GB of main memory, 16 MB of L2-Cache, and 22 MB of L3-Cache running Ubuntu 20.04.1. To facilitate algorithms that required greater memory, gsh-2015 was run on a single core of an alternate machine, consisting of a 64-core AMD EPYC 7702P Processor containing 1 TB of main memory.

Baselines. We compare HeiStreamE and FreightE against the state-of-the-art streaming algorithm HDRF, as well as the re-streaming algorithms 2PS-HDRF [36] and 2PS-L [37], which require three passes over the input graph. We exclude the following algorithms: SNE [54], as it fails to execute for $k > 127$ and is outperformed by 2PS-HDRF [36]; DBH [53], as it ignores past partition assignments and thus has poor solution quality; ADWISE and RBSEP as they have limited global information during streaming, and 2PS-L outperforms them [37].

For comparison with competitors, we obtained implementations of 2PS-HDRF and 2PS-L from their official repository, which also provides an implementation of HDRF. We configure all competitor algorithms with the optimal settings provided by the authors. 2PS-HDRF.
2PS-L and HDRF require a vertex-to-partition table of size $O(n \times k)$, which stores the blocks that each vertex was replicated on. To optimize memory usage, these partitioners can be built with the number of partitions at compile time to only allocate the required amount of memory. Thus, we re-compile them with the CMake flag for the number of blocks for each $k$ value in our experiments. We set the HDRF scoring function parameter $\lambda = 1.1$, similar to the authors of 2PS [36]. The provided codes for 2PS-HDRF, 2PS-L, and HDRF in their official repositories set a hard-coded soft limit on the number of partitions to 256, which we override to test the algorithms with larger $k$ values.

All competitors read a binary edge list format with 32-bit vertex IDs. This allows for faster IO during program execution. Additionally, all competitor programs offer a converter which can load a graph in the standard edge list format, convert it into the binary edge list format, and write it to memory before proceeding. For a fair comparison with HeiStreamE and FreightE, which are capable of reading both text-based (METIS) and binary (adjacency) graphs, we perform experiments with binary graph formats only. Further, we exclude the IO time since the objective of the experiments is to measure the performance of the partitioners and not IO efficiency. Similarly, we exclude the time it takes to convert the graphs into the binary format for all partitioners.

**Instances.** Our graph instances for experiments are shown in Appendix Table 2 and are sourced from Ref. [4, 5, 6, 7, 20, 31, 42]. All instances evaluated have been used for benchmarking in previous works on graph partitioning. From these graph instances, we construct three disjoint sets: a tuning set for parameter study experiments, a test set for comparison against state-of-the-art and a set of huge graphs, for which in-memory partitioners ran out of memory on our machine. We set the number of blocks to $k = \{2^1, 2^2, \ldots, 2^{14}\}$ for all experiments except those on huge graphs, for which $k$ values are shown in Table 1. We allow an imbalance of $\varepsilon = 3\%$ for all partitioners. While streaming, we use the natural order of the vertices in these graphs.

**Methodology.** We measure running time, replication factor and memory consumption, i.e., the maximum resident set size for the executed process. When averaging over all instances, we use the geometric mean to give every instance the same influence on the final score. Further, we average all results of each algorithm grouped by $k$, to explore performance with increasing $k$ values. Let the runtime, replication factor or memory consumption be denoted by the score $\sigma_A$ for some $k$-partition generated by an algorithm $A$. We express this score relative to others using the following tools: improvement over an algorithm $B$, computed as a percentage $(\frac{\sigma_A}{\sigma_B} - 1) \times 100\%$ and relative value over an algorithm $B$, computed as $\frac{\sigma_A}{\sigma_B}$. Additionally, we present performance profiles by Dolan and Moré [16] to benchmark our algorithms. These profiles relate the running time (resp. solution quality, memory) of the slower (resp. worse) algorithms to the fastest (resp. best) one on a per-instance basis, rather than grouped by $k$. Their $x$-axis shows a factor $\tau$ while their $y$-axis shows the percentage of instances for which an algorithm has up to $\tau$ times the running time (resp. solution quality, memory) of the fastest (resp. best) algorithm.

**Parameter Tuning.** We tuned the parameters used by HeiStreamE, namely the selection of the batch graph model $\beta_b$ mode and the choice of Fennel $\alpha$, through experiments run on the Tuning Set (see Appendix Table 2). Due to space constraints we only give a short summary. Our results support the use of the minimal mode for $\beta_b$ and batch $\alpha$ for Fennel’s $\alpha$ parameter for subsequent experiments. Other parameters for the multilevel partitioning
Figure 4 Comparison of HeiStreamE and FreightE with 2PS-HDRF, 2PS-L and HDRF on the Test Set in Appendix Table 2 using performance profiles. Let $A$ be the set of all algorithms, $I$ the set of instances, and $q_A(I)$ the quality of algorithm $A \in A$ on instance $I \in I$. For each algorithm $A$, we plot the fraction of instances $\frac{|I_A(\tau)|}{|I|}$ (y-axis) where $I_A(\tau) := \{I \in I | q_A \leq \tau \cdot \min_{A' \in A} q_{A'}(I)\}$ and $\tau$ is on the x-axis. Includes all $k$ values. Note the logarithmic scale in the final third of the plots. Memory consumption is measured as the maximum resident set size of the program execution. 2PS-HDRF and 2PS-L have the same memory consumption.

Comparison with State-of-the-Art

We now provide experiments in which we compare HeiStreamE and FreightE against the current state-of-the-art algorithms for (re)streaming edge partitioning, namely, HDRF, 2PS-HDRF and 2PS-L. These experiments were performed on the Test Set and the Huge Set of graphs in Appendix Table 2. Figure 4 gives performance profiles for the Test Set and Table 1 gives detailed per instance results for instances of the Huge Set. We distinguish between buffer sizes by defining, for example, HeiStreamE(32x) as HeiStreamE with a buffer size of $32 \times 1024 = 32 \times x$ vertices, where $x = 1024$.

Replication Factor. HeiStreamE(32x) produces better solution quality than state-of-the-art (re)streaming edge partitioners for all $k$ values. The results from our Test Set demonstrate that HeiStreamE(32x) achieves an average improvement in solution quality of 7.56% (or 13.69% when using HeiStreamE(256x)) compared to 2PS-HDRF, which produces the next best solution quality. As displayed in Figure 4a, HeiStreamE(32x) produces better solution quality than 2PS-HDRF in approximately 80% of all Test Set instances. The largest improvement in replication factor that we observed for HeiStreamE(32x) over 2PS-HDRF is of $\approx 53\%$ on graph circuit5M for $k \geq 8192$. Further, HeiStreamE(32x) achieves an average improvement in solution quality of 51.84% percent over 2PS-L, and an average improvement of 202.86% over HDRF, the only other on-the-fly streaming algorithm presented here. These results from the Test Set are reflected in our experiments on huge graphs, shown in Table 1: HeiStreamE(32x) and HeiStreamE(256x) produce the best solution quality in most instances, providing the best solution quality for all $k$ values for five of the six huge graphs. While FreightE produces
lower solution quality than 2PS-L and 2PS-HDRF on most Test Set instances, for $k \geq 512$, FreightE produces 9.88% higher solution quality than 2PS-L on average. FreightE produces 87.28% better solution quality than HDRF on average across all Test Set instances and $k$ values. The same trends for FreightE are reflected in our results on huge graphs.

**Runtime.** HeiStreamE(32x) is on average slower than 2PS-HDRF and HDRF for $k \leq 256$; however, since its runtime is not linearly dependent on $k$, HeiStreamE is substantially faster than 2PS-HDRF and HDRF for higher $k$ values (Figure 5b). On the Test Set, HeiStreamE(32x) is on average $6.7 \times$ faster than 2PS-HDRF and $8.7 \times$ faster than HDRF for $k \geq 512$. Compared to 2PS-L, whose time complexity is also independent of $k$, HeiStreamE(32x) is on average slower across all instances. Similarly, in our experiments on huge graphs, HeiStreamE(32x) and HeiStreamE(256x) are faster than 2PS-HDRF and HDRF for high $k$ values, but slower than 2PS-L. On the Test Set, FreightE is the fastest algorithm among competitors; it is $1.3 \times$ faster than 2PS-L which is the next fastest algorithm. Additionally, it is $9.9 \times$ faster than HDRF and $8 \times$ faster than 2PS-HDRF on average across all Test Set instances. In our experiments on huge graphs, FreightE is faster than HDRF and 2PS-HDRF on average across all instances, and the fastest algorithm for $k \geq 2048$.

**Memory Consumption.** Since HeiStreamE(32x) uses a buffered streaming approach, it consumes on average more memory than 2PS-HDRF, 2PS-L and HDRF for $k \leq 256$ on the Test Set; however, since its memory consumption is not asymptotically dependent on $m$ or $k$, HeiStreamE(32x) consumes significantly less memory than 2PS-HDRF, 2PS-L, and HDRF for higher $k$ values. On average, for $k \geq 512$, 2PS-HDRF and 2PS-L use $3.0 \times$ more memory than HeiStreamE(32x), and HDRF uses $2.7 \times$ more memory than HeiStreamE(32x). Notably, while HeiStreamE(32x) consumes $1.46 \times$ more memory on average than 2PS-HDRF and 2PS-L for $k \leq 256$ on the Test Set, in three out of six of the huge graphs, it is more memory efficient than them across all $k$ values. On the Test Set, FreightE consumes significantly less memory than all competitors as shown in Figure 4c. It consumes $4.9 \times$ less memory than HDRF and $7.2 \times$ less memory than 2PS-L and 2PS-HDRF on average across all instances and all $k$ values. Further, on average, FreightE consumes $16 \times$ less memory than 2PS-L and 2PS-HDRF for $k \geq 512$ (Figure 5c). Since FreightE’s memory consumption is linearly dependent on $m$, it is less memory efficient on the huge graphs, which, like other real-world graphs, tend to have many more edges than vertices. FreightE uses more memory than competitors for $k \leq 256$ in five out of six of the huge graphs, but is more memory efficient at
high $k$ due to its memory being asymptotically independent of $k$. In our experiments on huge graphs, 2PS-HDRF, 2PS-L and HDRF exceed the available memory on the machine at $k \geq 1096$ for uk-2007-05 and webbase-2001, at $k \geq 8192$ for com-friendster. At $k \geq 16384$ they exceed the available memory for all graphs, while HeiStreamE and FreightE's memory consumption is predictable and consistent across $k$.

Table 1: Results of experiments on the Huge Set in Appendix Table 2. Here, we compare HeiStreamE(32x), i.e., HeiStreamE with a buffer size of 32 - 1024 (resp. HeiStreamE(256x)), and FreightE with state-of-the-art (re)streaming edge partitioners on huge graph instances, displaying the Replication Factor $RF$, Running Time $RT$ [s] and Memory Consumption Mem [GB]. The best solution for each instance is framed.

The missing values indicate when an algorithm exceeded the memory of the machine for all graphs except gsh-2015, where the missing value indicates a timeout, i.e., the algorithm exceeded a runtime limit of 24 hours.

<table>
<thead>
<tr>
<th>$G$</th>
<th>$k$</th>
<th>HeiStreamE(32x)</th>
<th>HeiStreamE(256x)</th>
<th>FreightE</th>
<th>2PS-HDRF</th>
<th>2PS-L</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16</td>
<td>1.33</td>
<td>1.30</td>
<td>1.25</td>
<td>1.09</td>
<td>1.09</td>
</tr>
<tr>
<td>32</td>
<td>105</td>
<td>1.35</td>
<td>1.31</td>
<td>1.24</td>
<td>1.09</td>
<td>1.09</td>
</tr>
<tr>
<td>128</td>
<td>107</td>
<td>1.05</td>
<td>1.02</td>
<td>1.08</td>
<td>1.05</td>
<td>1.05</td>
</tr>
<tr>
<td>512</td>
<td>110</td>
<td>1.05</td>
<td>1.03</td>
<td>1.08</td>
<td>1.05</td>
<td>1.05</td>
</tr>
<tr>
<td>2048</td>
<td>118</td>
<td>1.04</td>
<td>1.03</td>
<td>1.08</td>
<td>1.05</td>
<td>1.05</td>
</tr>
</tbody>
</table>

The missing values indicate when an algorithm exceeded the memory of the machine for all graphs except gsh-2015, where the missing value indicates a timeout, i.e., the algorithm exceeded a runtime limit of 24 hours.

The best solution for each instance is framed.

Table 1: Results of experiments on the Huge Set in Appendix Table 2. Here, we compare HeiStreamE(32x), i.e., HeiStreamE with a buffer size of 32 - 1024 (resp. HeiStreamE(256x)), and FreightE with state-of-the-art (re)streaming edge partitioners on huge graph instances, displaying the Replication Factor $RF$, Running Time $RT$ [s] and Memory Consumption Mem [GB]. The best solution for each instance is framed.

The missing values indicate when an algorithm exceeded the memory of the machine for all graphs except gsh-2015, where the missing value indicates a timeout, i.e., the algorithm exceeded a runtime limit of 24 hours.

The best solution for each instance is framed.

Table 1: Results of experiments on the Huge Set in Appendix Table 2. Here, we compare HeiStreamE(32x), i.e., HeiStreamE with a buffer size of 32 - 1024 (resp. HeiStreamE(256x)), and FreightE with state-of-the-art (re)streaming edge partitioners on huge graph instances, displaying the Replication Factor $RF$, Running Time $RT$ [s] and Memory Consumption Mem [GB]. The best solution for each instance is framed.

The missing values indicate when an algorithm exceeded the memory of the machine for all graphs except gsh-2015, where the missing value indicates a timeout, i.e., the algorithm exceeded a runtime limit of 24 hours.

The best solution for each instance is framed.

Table 1: Results of experiments on the Huge Set in Appendix Table 2. Here, we compare HeiStreamE(32x), i.e., HeiStreamE with a buffer size of 32 - 1024 (resp. HeiStreamE(256x)), and FreightE with state-of-the-art (re)streaming edge partitioners on huge graph instances, displaying the Replication Factor $RF$, Running Time $RT$ [s] and Memory Consumption Mem [GB]. The best solution for each instance is framed.

The missing values indicate when an algorithm exceeded the memory of the machine for all graphs except gsh-2015, where the missing value indicates a timeout, i.e., the algorithm exceeded a runtime limit of 24 hours.

The best solution for each instance is framed.

Table 1: Results of experiments on the Huge Set in Appendix Table 2. Here, we compare HeiStreamE(32x), i.e., HeiStreamE with a buffer size of 32 - 1024 (resp. HeiStreamE(256x)), and FreightE with state-of-the-art (re)streaming edge partitioners on huge graph instances, displaying the Replication Factor $RF$, Running Time $RT$ [s] and Memory Consumption Mem [GB]. The best solution for each instance is framed.

The missing values indicate when an algorithm exceeded the memory of the machine for all graphs except gsh-2015, where the missing value indicates a timeout, i.e., the algorithm exceeded a runtime limit of 24 hours.

The best solution for each instance is framed.
5 Conclusion

In this work, we propose HeiStreamE, a buffered streaming edge partitioner that achieves state-of-the-art solution quality, and FreightE, a highly efficient streaming edge partitioner that uses streaming hypergraph partitioning to assign blocks to edges on the fly. HeiStreamE processes the input graph in batches, constructs a novel graph transformation on the per-batch graph model, extends it with global partitioning information, and partitions it with a multilevel scheme. Aside from hashing-based streaming partitioners, which have poor solution quality, HeiStreamE and FreightE are the only known streaming edge partitioners whose runtime and memory consumption are both linear and asymptotically independent of the number of blocks of partition $k$. Our experiments demonstrate that HeiStreamE and FreightE consistently outperform all existing (re)streaming edge partitioners with regard to vertex replication metrics and runtime respectively. Additionally, HeiStreamE uses less memory than high-quality (re)streaming edge partitioners at $k \geq 256$ or when the graph has far more edges than vertices, as is the case for most real-world networks. Our findings underscore HeiStreamE as a highly memory-efficient and effective solution for streaming edge partitioning of large-scale graphs.

References


Buffered Streaming Edge Partitioning


A Instance Properties

Table 2 Graphs used for experiments. The roadNet graphs, wiki graphs, web-Google, web-NotreDame, and all social, co-purchasing, and autonomous systems graphs were obtained from the publicly available SNAP dataset [31]. We also used graphs from the 10th DIMACS Implementation Challenge, namely eu-2005, in-2004 and uk-2007-05 [4]. Any remaining graphs are available on the network repository website [42] or on the Laboratory for Web Algorithmics website [5, 6, 7]. For our experiments, we converted these graphs to a vertex-stream format (METIS) while removing parallel edges, self-loops, and directions, and assigning unitary weight to all vertices and edges. If a graph was unavailable in an equivalent edge list format, we converted the METIS format into the edge list format while preserving edge order for fairness during streaming.

<table>
<thead>
<tr>
<th>Graph</th>
<th>n</th>
<th>m</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>icircuit</td>
<td>105 670</td>
<td>203 734</td>
<td>Circuit</td>
</tr>
<tr>
<td>coAuthorsCiteseer</td>
<td>227 320</td>
<td>814 134</td>
<td>Citations</td>
</tr>
<tr>
<td>coAuthorsDBLP</td>
<td>299 067</td>
<td>977 676</td>
<td>Citations</td>
</tr>
<tr>
<td>com-DBLP</td>
<td>317 080</td>
<td>1 049 866</td>
<td>Social</td>
</tr>
<tr>
<td>roadNet-PA</td>
<td>1 088 092</td>
<td>1 541 898</td>
<td>Roads</td>
</tr>
<tr>
<td>web-Google</td>
<td>356 648</td>
<td>2 093 324</td>
<td>Web</td>
</tr>
<tr>
<td>com-Orkut</td>
<td>403 394</td>
<td>2 443 408</td>
<td>Co-Purch.</td>
</tr>
<tr>
<td>com-Youtube</td>
<td>1 134 890</td>
<td>2 987 624</td>
<td>Social</td>
</tr>
<tr>
<td>Amazon-2008</td>
<td>735 332</td>
<td>3 523 472</td>
<td>Similarity</td>
</tr>
<tr>
<td>soc-lastfm</td>
<td>1 191 805</td>
<td>4 519 330</td>
<td>Social</td>
</tr>
<tr>
<td>as-Skitter</td>
<td>554 930</td>
<td>5 797 633</td>
<td>Aut. Syst.</td>
</tr>
<tr>
<td>italy-osm</td>
<td>6 686 493</td>
<td>7 013 978</td>
<td>Roads</td>
</tr>
<tr>
<td>in-2004</td>
<td>1 382 908</td>
<td>13 591 473</td>
<td>Web</td>
</tr>
<tr>
<td>ML_Laplace</td>
<td>377 002</td>
<td>13 650 485</td>
<td>Mesh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph</th>
<th>n</th>
<th>m</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dubcova1</td>
<td>16 129</td>
<td>118 440</td>
<td>Circuit</td>
</tr>
<tr>
<td>DBLP-2010</td>
<td>300 647</td>
<td>807 700</td>
<td>Citations</td>
</tr>
<tr>
<td>com-Amazon</td>
<td>334 863</td>
<td>925 872</td>
<td>Social</td>
</tr>
<tr>
<td>web-NotreDame</td>
<td>325 729</td>
<td>1 090 108</td>
<td>Web</td>
</tr>
<tr>
<td>citationCiteseer</td>
<td>268 495</td>
<td>1 156 647</td>
<td>Citations</td>
</tr>
<tr>
<td>wiki-Talk</td>
<td>232 314</td>
<td>1 458 806</td>
<td>Web</td>
</tr>
<tr>
<td>roadNet-TX</td>
<td>1 379 917</td>
<td>1 921 660</td>
<td>Roads</td>
</tr>
<tr>
<td>amazon0312</td>
<td>400 727</td>
<td>2 349 267</td>
<td>Co-Purch.</td>
</tr>
<tr>
<td>amazon0505</td>
<td>410 236</td>
<td>2 439 437</td>
<td>Co-Purch.</td>
</tr>
<tr>
<td>roadNet-CA</td>
<td>1 965 206</td>
<td>2 766 607</td>
<td>Roads</td>
</tr>
<tr>
<td>G1_circuit</td>
<td>1 585 478</td>
<td>3 037 674</td>
<td>Circuit</td>
</tr>
<tr>
<td>soc-Finster</td>
<td>2 523 386</td>
<td>7 918 801</td>
<td>Social</td>
</tr>
<tr>
<td>great-britain-osm</td>
<td>7 733 822</td>
<td>8 156 517</td>
<td>Roads</td>
</tr>
<tr>
<td>FullChip</td>
<td>2 986 999</td>
<td>11 817 567</td>
<td>Circuit</td>
</tr>
<tr>
<td>cdPapersDBLP</td>
<td>540 486</td>
<td>15 245 729</td>
<td>Citations</td>
</tr>
<tr>
<td>cdPapersCiteseer</td>
<td>434 102</td>
<td>16 036 720</td>
<td>Citations</td>
</tr>
<tr>
<td>eu-2005</td>
<td>862 664</td>
<td>16 138 468</td>
<td>Web</td>
</tr>
<tr>
<td>soc-psokec</td>
<td>1 632 803</td>
<td>22 301 964</td>
<td>Social</td>
</tr>
<tr>
<td>wikis-topcats</td>
<td>1 791 489</td>
<td>25 444 207</td>
<td>Social</td>
</tr>
<tr>
<td>circuit5M</td>
<td>5 558 311</td>
<td>26 983 926</td>
<td>Circuit</td>
</tr>
<tr>
<td>com-LJ</td>
<td>3 997 962</td>
<td>34 681 189</td>
<td>Social</td>
</tr>
<tr>
<td>soc-LiveJournal</td>
<td>1 486 609</td>
<td>2 851 237</td>
<td>Social</td>
</tr>
<tr>
<td>Ljournal-2008</td>
<td>5 363 186</td>
<td>49 514 271</td>
<td>Social</td>
</tr>
<tr>
<td>ca-hollywood-2009</td>
<td>1 069 126</td>
<td>56 396 653</td>
<td>Roads</td>
</tr>
<tr>
<td>Flan_1565</td>
<td>1 564 794</td>
<td>57 920 625</td>
<td>Mesh</td>
</tr>
<tr>
<td>Bump_2011</td>
<td>2 852 430</td>
<td>62 499 240</td>
<td>Mesh</td>
</tr>
<tr>
<td>com-Ebay</td>
<td>3 072 441</td>
<td>117 185 083</td>
<td>Social</td>
</tr>
<tr>
<td>HV15R</td>
<td>2 017 169</td>
<td>162 357 569</td>
<td>Mesh</td>
</tr>
</tbody>
</table>

B Omitted Proofs

Theorem. For any vertex partition vp(Sep) of the CSPAC graph Sep with edge-cut cost(vp(Sep)), there exists a corresponding edge partition ep(Gb) of the batch graph Gb with a number of vertex replicas cost(ep(Gb)), satisfying cost(ep(Gb)) ≤ cost(vp(Sep)), which establishes a lower bound spanning the set of possible CSPAC graphs Sep associated with Gb.
Proof. Our proof can be delineated through three consecutive claims. (i) The existence of a singular edge partition $ep(G_b)$ that corresponds to a specified vertex partition $vp(S_b^*)$. (ii) The general validity of the inequality $cost(ep(G_b)) \leq cost(vp(S_b^*))$. (iii) The validity of the equality $cost(ep(G_b)) = cost(vp(S_b^*))$ for a specific CSPAC graph $S_b^*$.

Claim (i) trivially holds, as $ep(G_b)$ stems directly from $vp(S_b^*)$ by virtue of the one-to-one correspondence between edges in $G_b$ and vertices in $S_b^*$. For the proof of assertions (ii) and (iii), consider the following. (a) There can be no replicas of vertices with degree lower than 2. (b) Vertices in $G_b$ with a degree lower than 2 are not represented by any edges in $S_b^*$. (c) Vertices $u$ in $G_b$ that possess a degree $d(u) \geq 2$ are represented in $S_b^*$ through unique, edge-disjoint paths, each connecting the $d(u)$ vertices in $S_b^*$ corresponding to the edges incident to $u$ in $G_b$.

To prove (ii), we show that the number of replicas of any vertex $u$ in $ep(G_b)$ does not exceed the number of cut edges directly induced by $u$ in $vp(S_b^*)$. From (a) and (b), it trivially holds for vertices with a degree lower than 2. Assuming there are $x > 0$ replicas of $u$ in $ep(G_b)$, it implies that edges incident on $u$ are distributed across $x + 1$ nonempty blocks. According to (c), $u$ is uniquely represented by an edge-disjoint path connecting the vertices in $S_b^*$ that correspond to the edges of $u$ in $G_b$. When the vertices of a connected (sub)graph are partitioned into $x + 1$ blocks, these blocks themselves are interconnected. Therefore, the edge-cut directly attributable to vertex $u$ is at least $x$, which completes the proof for (ii).

To prove (iii), we show how to build a valid CSPAC graph $S_b^*$, such that $cost(ep(G_b)) = cost(vp(S_b^*))$. For vertices of $G_b$ with a degree lower than 2, the equality trivially holds based on (a) and (b). For a vertex $u$ in $ep(G_b)$ with $x > 0$ replicas (edges distributed across $x + 1$ nonempty blocks), we create $x + 1$ independent paths connecting the vertices in $S_b^*$ that represent the edges incident to $u$ in $G_b$. Vertices in each path are then assigned to a common block. As these paths are between vertices of the same block, they have no cut edges. Subsequently, these paths are interlinked to form a unified path. This introduces exactly $x$ cut edges directly associated with vertex $u$, thereby concluding the proof for condition (iii) and the overall proof.

Theorem. For any vertex partition $vp(\beta_b)$ of the maximal model $\beta_b$ with edge-cut cost $\gamma \leq cost(vp(\beta_b))$, there exists a corresponding edge partition $ep(G_b)$ of the batch graph $G_b$ that, when incorporated into the already partitioned section of the input graph $G$, introduces a number $\gamma$ of new vertex replicas, satisfying $\gamma \leq cost(vp(\beta_b))$.

Proof. The only disparity between the CSPAC graph $S_b^*$ and the maximal model $\beta_b$ is the presence of artificial vertices and edges in $\beta_b$. If there are no artificial edges in $\beta_b$, it implies that none of the edges in the batch graph $G_b$ are adjacent to edges in $G$ that have previously been assigned to blocks. In this scenario, Theorem 1 provides sufficient grounds to establish the claim. If $\beta_b$ contains artificial edges, each artificial edge signifies a unique adjacency in $G$ of an edge in the batch graph $G_b$ to edges already assigned to blocks in previous batches. In this scenario, we complete the proof of the claim by noting that the number of cut artificial edges in $vp(\beta_b)$ cannot be less than the number of new replicas introduced exclusively for vertices contained in previous batches.