BuffCut: Prioritized Buffered Streaming Graph Partitioning

Linus Baumgärtner
Heidelberg University, Germany

Adil Chhabra
Heidelberg University, Germany

Marcelo Fonseca Faraj
A+W Software, Fenwald, Germany

Christian Schulz
Heidelberg University, Germany

Abstract

Streaming graph partitioners enable resource-efficient and massively scalable partitioning, but one-pass assignment heuristics are highly sensitive to stream order and often yield substantially higher edge cuts than in-memory methods. We present BuffCut, a buffered streaming partitioner that narrows this quality gap, particularly when stream ordering is adversarial, by combining prioritized buffering with batch-wise multilevel assignment. BuffCut maintains a bounded priority buffer to delay poorly informed decisions and regulate the order in which nodes are considered for assignment. It incrementally constructs high-locality batches of configurable size by iteratively inserting the highest-priority nodes from the buffer into the batch, effectively recovering locality structure from the stream. Each batch is then assigned via a multilevel partitioning algorithm. Experiments on diverse real-world and synthetic graphs show that BuffCut consistently outperforms state-of-the-art buffered streaming methods. Compared to the strongest prioritized buffering baseline, BuffCut achieves 20.8% fewer edge cuts while running 2.9× faster and using 11.3× less memory. Against the next-best batched method, it reduces edge cut by 15.8% with only modest overheads of 1.8× runtime and 1.09× memory.

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

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

Digital Object Identifier 10.4230/LIPIcs.SEA.2026.5

Related Version *Full Version*: `https://arxiv.org/abs/2602.21248`

Supplementary Material *Software (Source Code)*: `https://github.com/KaHIP/HeiStream` archived at `swh:1:dir:e31d5515845ae387bac78c263c320f81f834095f`

Funding We acknowledge support by DFG grant SCHU 2567/5-1.

Acknowledgements Moreover, we would like to acknowledge Dagstuhl Seminar 23331 on Recent Trends in Graph Decomposition.

1 Introduction

Graphs are a central abstraction for modeling complex relationships in modern data-driven systems. Massive graphs with billions of nodes and edges are routinely used to represent social, biological, navigational, and technical networks, enabling tasks such as community detection, pathway analysis, and recommendation [1, 2, 10, 14, 19, 22, 28]. More recently, graphs have become a core data structure in emerging applications such as real-time analytics over social networks and financial transaction streams, and machine learning models based

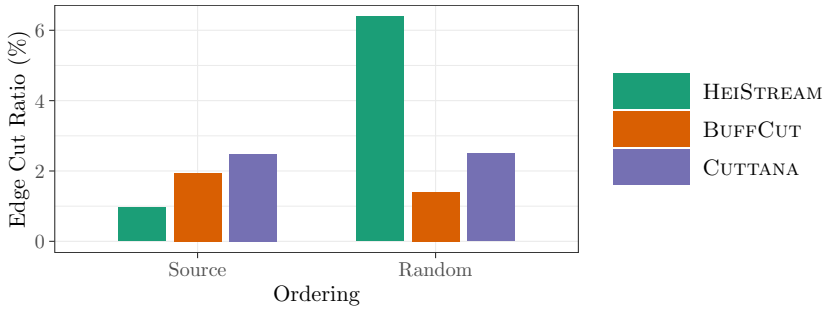


Figure 1 Edge cut on source and random ordering. The ratio of cut edges to total graph edges (%) on `uk-2007-05` web graph (k=16) comparing source ordering (original node sequence in source file) to random ordering (independent random permutations of node IDs resulting in adversarial stream order) for HeiStream, Cuttana and our proposed BuffCut.

on graph neural networks [25, 32]. These graphs see large numbers of edges and nodes added or removed frequently and have highly heterogeneous degree distributions, placing pressure on both memory and computation. Thus, graph processing systems must balance a trade-off between delivering high-quality analytics while operating under tight memory and runtime constraints, often in online or near-real-time settings.

To address these challenges, graph processing is typically performed in distributed environments. The input graph is divided into disjoint blocks that are stored and processed across multiple workers, which communicate along edges that cross block boundaries. In this setting, inter-block edges dominate communication and synchronization costs, while imbalanced partitions create stragglers that slow parallel execution. Graph partitioning is therefore a fundamental step in distributed graph processing: it aims to divide a graph into roughly equal-sized subgraphs while minimizing inter-block connectivity. However, graph partitioning is NP-hard [7, 21], and no constant-factor approximation exists for general graphs unless $P = NP$ [8], necessitating the use of heuristic algorithms in practice.

State-of-the-art in-memory partitioners such as METIS [26], KaHIP [36], and KaMinPar [23] achieve high-quality partitions but require memory proportional to the entire graph, making them impractical for billion-edge or rapidly evolving graphs. For modern applications that require the processing of complex graphs in real-time, streaming graph partitioners offer a resource-efficient alternative by processing nodes sequentially without storing the full graph in memory. However, this efficiency comes at the cost of reduced solution quality, as streaming methods must make placement decisions with limited structural information.

The need for high-quality resource-efficient streaming partitioners is highly evident in machine learning pipelines, notably in distributed training of graph neural networks (GNNs), widely used in recommendation systems, natural language processing, biological discovery, and fraud detection [17, 29, 32, 31, 39]. GNN training is computationally intensive, involving repeated communication of high-dimensional feature vectors and storage of large intermediate embeddings across layers. As a result, GNNs are typically trained in distributed environments, where partitioning quality affects memory consumption, communication volume, and training efficiency [33]. The computational requirements of GNN workloads indicate a clear motivation for improving streaming partitioners across quality and efficiency.

Classical one-pass streaming heuristics such as Fennel [38], while resource-efficient, lag in solution quality and are highly sensitive to stream order locality: when neighbors appear far apart in the stream, nodes are assigned with little structural context, resulting

in substantial quality degradation. Recent advances in buffered streaming attempt to bridge this gap. HeiStream [18] uses batch-wise multilevel partitioning to improve locality but remains vulnerable to adversarial input orders. For instance, on the `uk-2007` web graph (k=16), HeiStream's edge cut degrades from 31.5M in the graph's original node sequence, i.e. its source ordering, to 211.0M when the stream is randomized (Figure 1). Conversely, Cuttana [24] uses a prioritized buffer to handle poor orderings but relies on sequential assignments, missing the global refinement gains of batch-wise multilevel partitioning. While Cuttana improves the cut to 82.4M on the randomized instance, our proposed BuffCut achieves a significantly superior result of 46.7M, effectively recovering the structural information lost to randomization and approaching the quality achieved on the high-locality source ordering. In this paper, we contribute the following:

- We introduce BuffCut, a buffered streaming partitioner that delays premature block assignments using a bounded priority buffer and incrementally forms batches with high internal locality for multilevel partitioning. BuffCut combines high-quality batch-wise partitioning with prioritized buffering for stream-order robustness.
- We introduce the Hub-Aware Assigned Neighbors Ratio buffer score that refines degree-aware prioritization to systematically balance node degree and neighborhood information.
- We present a parallel implementation of BuffCut that accelerates end-to-end partitioning and restreaming passes for additional quality improvements.
- We share extensive experimental evaluation demonstrating that BuffCut outperforms existing streaming partitioners, achieving, in geometric means, 20.8% fewer edge cuts while running 2.9× faster and using 11.3× less memory than Cuttana, the strongest prioritized buffering baseline.

2 Preliminaries

2.1 Basic Concepts

Graphs and Graph Partitioning. We consider an undirected graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. Each node $v \in V$ has a weight $c(v) \in \mathbb{R}_{\geq 0}$, and each edge $e \in E$ has a weight $\omega(e) \in \mathbb{R}_{>0}$. Weight functions are extended additively to sets. We denote the neighborhood of v as $N(v) = \{u \in V \mid (u, v) \in E\}$, the degree as $d(v) = |N(v)|$, and the maximum degree in G as Δ. Given an integer $k \geq 1$, a *k-partition* of a graph $G = (V, E)$ divides the node set into k disjoint blocks $V_1, \ldots, V_k$ such that $\bigcup_{i=1}^{k} V_i = V$. The objective is to minimize the *edge cut* $\omega(E_{\text{cut}})$, where $E_{\text{cut}} := \{(u, v) \in E \mid u \in V_i, v \in V_j, i \neq j\}$, subject to the balance constraint $c(V_i) \leq L_{\max} := \lceil (1+\varepsilon)\frac{c(V)}{k} \rceil$ for a given imbalance parameter $\varepsilon \geq 0$. Minimizing the cut reduces inter-block communication, while the balance constraint ensures uniform load distribution in parallel systems.

Streaming Models. In the *node-streaming* model, nodes $v_1, \ldots, v_n$ and their incident edges arrive sequentially for processing. We distinguish three variations.

- **One-pass**: Nodes are assigned to a block immediately upon arrival based on a scoring function that favors blocks with existing neighbors while penalizing imbalance.
- **Buffered**: Nodes are held in sliding buffers of fixed capacity which expose partial structural information that allows for more informed assignments.
- **Restreaming**: The input is processed in multiple passes, allowing assignments to be refined based on global state from previous passes.

Stream Order Locality. The performance of streaming heuristics is highly sensitive to the stream order, defined as a permutation $S = (v_1, \ldots, v_n)$ of V. The *source ordering* is the order in which the graph is stored in the source file. This often exhibits high locality such as when nodes are generated through clustering or traversal-based schemes. We quantify stream locality using the *Neighbor to Neighbor Average ID Distance (AID)* [16]. For a node v with neighbors $N(v) = \{u_1, \ldots, u_{d(v)}\}$ sorted by their position in the stream, AID is defined as:

$$AID_v = \frac{1}{d(v)} \sum_{i=2}^{d(v)} |u_i - u_{i-1}| \tag{1}$$

Graph-level locality is the mean AID_v across all $v \in V$. Lower values indicate higher locality, where neighbors appear in close proximity within the stream.

2.2 Related Work

We refer readers to recent surveys for general graph partitioning [9, 11] and focus here on streaming partitioners. Early streaming approaches are based on one-pass algorithms like LDG [37] and Fennel [38] which greedily assign nodes upon arrival. Fennel assigns node v to the block V_i maximizing $g(v, V_i) = |N(v) \cap V_i| - f(|V_i|)$, where $f(|V_i|) = \alpha\gamma|V_i|^{\gamma-1}$ is an additive penalty to enforce balance. While computationally efficient, these methods make poor assignment decisions as they lack global context. Restreaming approaches [34] mitigate this by performing multiple passes to refine assignments using state from previous iterations.

Buffered streaming partitioners retain a subset of the graph in memory to exploit partial structural information. HeiStream [18] iteratively loads batches of nodes together with their incident edges and assigns blocks using a weighted variant of the Fennel objective within a multilevel partitioning scheme. While batch-wise assignment significantly improves partition quality, HeiStream remains vulnerable to adversarial or low-locality stream orders [24]. Cuttana [24] addresses order sensitivity through a two-phase prioritized buffering scheme. In phase 1, nodes are temporarily stored in a priority queue and ranked by the *Cuttana Buffer Score (CBS)* to avoid premature assignments:

$$CBS(v) = \frac{d(v)}{D_{\max}} + \theta \cdot \frac{\sum_{i=1}^{k} |N(v) \cap V_i|}{d(v)} \tag{2}$$

where $D_{\max}$ is a degree threshold. The first term favors high-degree nodes, while the second emphasizes the fraction of already assigned neighbors. When the buffer reaches capacity, the top node is evicted and assigned using a modified Fennel function. In Phase 2, Cuttana applies a refinement algorithm where nodes are grouped into sub-partitions, and coarse-grained trades between partitions are performed. While robust to adversarial ordering, Cuttana assigns nodes sequentially upon buffer eviction, failing to exploit the locality-capturing potential of multilevel batch partitioning.

3 BuffCut: Prioritized Buffered Streaming Partitioning

Streaming partitioners are attractive for large-scale and frequently updating graphs because they operate within strict memory constraints and process the input online. However, they often yield lower quality under unfavorable stream orderings as early assignments are made with limited structural context. We present BuffCut, a buffered streaming partitioner that achieves high quality and reduces order sensitivity, while remaining more resource-efficient than in-memory approaches, by combining (i) prioritized buffering for high-locality batch formation and (ii) batch-wise multilevel assignment on a compact model graph.

Algorithm 1 BuffCut (sequential, one-pass): core streaming loop with prioritized buffering and batch-wise multilevel assignment.

Input : Graph $G = (V, E)$; blocks k; buffer size $\mathcal{Q}_{\max}$; batch size δ; hub threshold $D_{\max}$; buffer score $s(\cdot)$
Output : Assignment block$(\cdot)$
init buffer $\mathcal{Q} \leftarrow \emptyset$, batch $\mathcal{B} \leftarrow \emptyset$, block$(v) \leftarrow \perp\ \forall v$;
foreach *streamed node* v *with neighbor list* $N(v)$ **do**
  if $d(v) > D_{\max}$ **then**
    block$(v) \leftarrow$ Fennel(v, k); `// assign hubs immediately`
    foreach $u \in N(v) \cap \mathcal{Q}$ **do**
      $\mathcal{Q}$.IncreaseKey$(u, s(u))$; `// update scores`
  else
    $\mathcal{Q}$.Insert$(v, s(v))$; `// buffer node`
  while $|\mathcal{Q}| = \mathcal{Q}_{\max}$ ***and*** $|\mathcal{B}| < \delta$ **do**
    $u \leftarrow \mathcal{Q}$.ExtractMax(); `// evict top node`
    add u to $\mathcal{B}$;
    foreach $w \in N(u) \cap \mathcal{Q}$ **do**
      $\mathcal{Q}$.IncreaseKey$(w, s(w))$;
  if $|\mathcal{B}| = \delta$ **then**
    `PartitionBatch`$(\mathcal{B}, \text{block}(\cdot), k)$; `// partition and clear batch`
while $\mathcal{Q} \neq \emptyset$ **do**
  add $\mathcal{Q}$.ExtractMax() to $\mathcal{B}$; `// flush`
  if $|\mathcal{B}| = \delta$ **then**
    `PartitionBatch`$(G, \mathcal{B}, \text{block}(\cdot), k)$;
if $|\mathcal{B}| > 0$ **then**
  `PartitionBatch`$(G, \mathcal{B}, \text{block}(\cdot), k)$; `// final`
Procedure `PartitionBatch`$(\mathcal{B}, \text{block}(\cdot), k)$
  $G_{\mathcal{B}} \leftarrow$ BuildBatchModel$(\mathcal{B}, \text{block}(\cdot), k)$; `// model graph`
  block$(\mathcal{B}) \leftarrow$ MLPartition$(G_{\mathcal{B}}, k)$; `// multilevel partition`
  $\mathcal{B} \leftarrow \emptyset$; `// clear`

3.1 Overall Algorithm

BuffCut processes the input graph G as a node stream under explicit, user-configurable memory bounds. It uses *buffering* to remain robust to stream order: rather than assigning each streamed node immediately, BuffCut delays placement until sufficient neighborhood evidence becomes available. Subsequently, several nodes are partitioned simultaneously using a multilevel partitioning scheme. The algorithm maintains two working sets with distinct purposes, terminology and separately configurable sizes: a bounded priority *buffer* $\mathcal{Q}$ of capacity $\mathcal{Q}_{\max}$ that stores deferred nodes and continuously reorders them by importance, and a *batch* $\mathcal{B}$ of target size δ whose nodes are assigned jointly in a multilevel partitioning scheme. Upon arrival of a node v, BuffCut treats high- and low-degree nodes differently. High-degree nodes ($d(v) > D_{\max}$) are assigned immediately using Fennel, which establishes stable anchors for subsequent decisions. All remaining nodes are inserted into $\mathcal{Q}$, implemented as a bucket-based max-priority queue keyed by a buffer score $s(v)$ that quantifies how

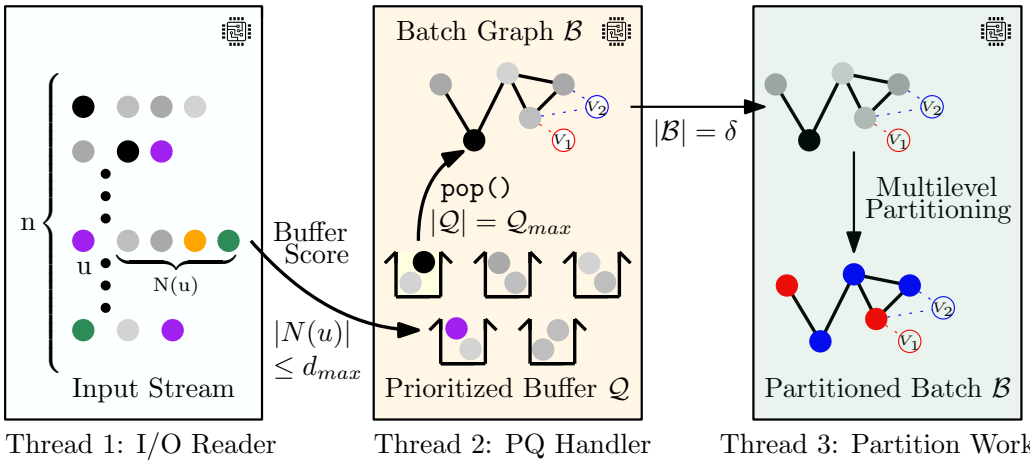


Figure 2 Overview of BuffCut and its parallel pipeline. Nodes arrive as a stream (left). Low-degree nodes are scored and inserted into a bounded prioritized buffer $\mathcal{Q}$, while hubs bypass the buffer. When $|\mathcal{Q}| = \mathcal{Q}_{\max}$, the highest-priority node is evicted to incrementally grow a batch $\mathcal{B}$ (middle); once $|\mathcal{B}| = \delta$, BuffCut constructs the batch model graph and partitions it via multilevel refinement, committing assignments (right). After committing, the batch $\mathcal{B}$ is cleared and construction resumes with the next evictions. The three stages are overlapped using an I/O reader, buffer handler, and partition worker.

informative v's current neighborhood is with respect to already assigned nodes (Section 3.3). Whenever the buffer reaches capacity ($|\mathcal{Q}| = \mathcal{Q}_{\max}$), BuffCut evicts the highest-priority buffered node into the active batch $\mathcal{B}$ until $|\mathcal{B}| = \delta$. Batch construction is incremental to allow frequent re-prioritization: as nodes are admitted to $\mathcal{B}$ (or assigned immediately as hubs), the buffer scores of their buffered neighbors are updated, so newly revealed neighborhood information immediately influences subsequent evictions and promotes high internal locality within $\mathcal{B}$ (Section 3.2). Once the batch is full, BuffCut constructs a compact batch model graph $G_{\mathcal{B}}$ and assigns all nodes in $\mathcal{B}$ via multilevel partitioning (Section 3.4). The batch is cleared and the process continues until all nodes are assigned. Algorithm 1 provides pseudocode for a single sequential pass and Figure 2 gives an overview. BuffCut supports restreaming for additional refinement and a parallel implementation for speed (Section 3.5).

3.2 Prioritized Buffered Streaming

We describe buffer and batch construction in BuffCut, highlighting two key improvements over Cuttana [24], which uses degree-aware buffering to reorder nodes for sequential assignment with subsequent refinement (Section 2.2). Firstly, BuffCut does not assign nodes immediately upon buffer eviction; instead, evicted nodes are accumulated into batches that are assigned jointly using multilevel partitioning, yielding higher-quality solutions. Moreover, BuffCut uses a more efficient priority queue implementation, which reduces overall buffer maintenance time.

As observed by Hajidehi et al. [24], low-degree nodes are highly prone to under-informed placement during streaming. BuffCut therefore assigns high-degree nodes with $d(v) > D_{\max}$ immediately using Fennel, while buffering nodes with $d(v) \leq D_{\max}$ in a priority queue $\mathcal{Q}$. Immediate assignment of high-degree hubs establishes stable anchors for neighboring nodes while preventing the buffer from being dominated by memory-intensive nodes that would trigger excessive score updates.

Algorithm 2 Bucket Priority Queue operations for efficient buffer score updates.

State: Array of dynamic arrays $\mathsf{Buckets}[0 \dots B-1]$;
Location map $\mathsf{L}[v] = (b, p)$ (bucket b, position p);
Top pointer $\rho = \max\{b : \mathsf{Buckets}[b] \neq \emptyset\}$;
Key: discretize score $s(v)$ as $\text{idx}(v) = \min\{\text{round}(s(v) \cdot \mathit{discFactor}),\ B-1\}$;

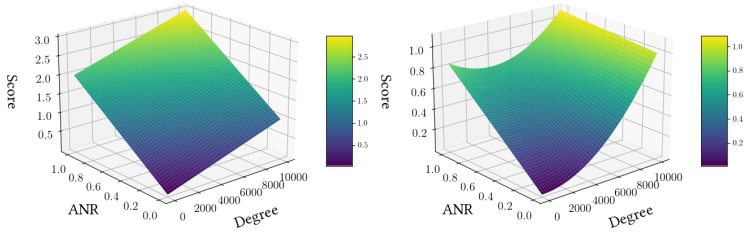
Procedure *Insert*$(v, s(v))$ `// O(1)`
- $b \leftarrow \text{idx}(v)$;
- $\mathsf{Buckets}[b].\textsc{Push}(v)$;
- $\mathsf{L}[v] \leftarrow (b,\ |\mathsf{Buckets}[b]| - 1)$;
- $\rho \leftarrow \max(\rho, b)$;

Procedure *IncreaseKey*$(v, s(v))$
- $(b, p) \leftarrow \mathsf{L}[v]$;
- $x \leftarrow \mathsf{Buckets}[b].\textsc{Pop}()$; `// pop O(1)`
- **if** $p < |\mathsf{Buckets}[b]|$ **then**
  - $\mathsf{Buckets}[b][p] \leftarrow x$; `// swap O(1)`
  - $\mathsf{L}[x] \leftarrow (b, p)$;
- $\textsc{Insert}(v, s(v))$; `// O(1)`

Procedure *ExtractMax*$() \to v$
- $v \leftarrow \mathsf{Buckets}[\rho].\textsc{Pop}()$; `// O(1)`
- **while** $\rho > 0$ ***and*** $\mathsf{Buckets}[\rho]$ *empty* **do**
  - $\rho \leftarrow \rho - 1$; `// rare worst-case O(B)`
- **return** v;

When the buffer $\mathcal{Q}$ reaches its maximum size $\mathcal{Q}_{\max}$, BuffCut extracts the top node from $\mathcal{Q}$ and appends it to the active batch $\mathcal{B}$ until $|\mathcal{B}| = \delta$ (Algorithm 1). Instead of evicting δ nodes from $\mathcal{Q}$ at once, batch construction is incremental to enable frequent updates to node order within the buffer. As soon as a node u is admitted to a batch, it is treated as assigned for the purpose of buffer score computation, though its block assignment is deferred until the batch model graph is partitioned. Consequently, the buffer scores of its buffered neighbors $v \in N(u) \cap \mathcal{Q}$ are updated immediately, allowing newly available neighborhood information to influence subsequent ordering decisions, with the same update applied when a hub is assigned immediately. This tight feedback loop promotes the grouping of structurally related nodes into the same batch, even when they appear far apart in the input stream.

Frequent buffer score updates are required to re-prioritize neighbors during batch construction. Over a streaming pass, scanning the neighborhoods of all nodes results in a total of $\mathcal{O}(m)$ updates. Specifically, each edge contributes to at most one such update event when one endpoint is admitted to a batch, while the other endpoint remains buffered. In Cuttana, the buffer is implemented using a red-black tree-based priority queue (`std::set`), imposing a logarithmic penalty of $\mathcal{O}(\log \mathcal{Q}_{\max})$ per update due to tree rebalancing. To eliminate this bottleneck, BuffCut implements $\mathcal{Q}$ as a bucket priority queue that exploits the bounded range and monotonicity of buffer scores to achieve amortized constant time updates. We first map continuous buffer scores to a finite set of priority levels. Let $s(v) \in [0, S_{\max}]$ denote the buffer score of node v. We discretize these scores into B integer buckets using a scaling factor $\mathit{discFactor}$, mapping each node to a bucket index $\text{idx}(v) = \min\{\text{round}(s(v) \cdot \mathit{discFactor}),\ B-1\}$, yielding an approximate priority ordering.



(a) CBS ($\theta = 2$). **(b)** HAA ($\theta = 0.75, \beta = 2$).

Figure 3 Heatmap visualizations of CBS and HAA as a function of degree (x-axis) and assigned-neighbor ratio (y-axis). Bright colors indicate high scores, which lead to earlier eviction from the buffer.

We store $\mathcal{Q}$ as an array of B dynamic arrays $\mathsf{Buckets}[0 \dots B-1]$ together with a location map $L[v] = (b, p)$ indicating the current bucket b and position p of each buffered node, and a top pointer ρ to the highest non-empty bucket (Algorithm 2).

During batch construction, buffer scores are monotonic non-decreasing, so all priority updates are `IncreaseKey` operations. We implement `IncreaseKey` in $\mathcal{O}(1)$ amortized time via a pop-and-swap operation followed by an `Insert`: to move a node v to a higher-index bucket, we swap v with the last element in its current bucket, update the location map L for the swapped element, and append v to the target bucket. `Insert` is an append plus constant-time updates to L and ρ, and thus also runs in $\mathcal{O}(1)$ amortized time with dynamic arrays. `ExtractMax` removes an element from $\mathsf{Buckets}[\rho]$ and then decreases ρ until the next non-empty bucket is found. This scan can traverse at most B buckets, and hence `ExtractMax` takes $\mathcal{O}(B)$ worst-case time. In our setting, however, this worst case is rare. Since buffer scores increase gradually, and B is fixed to a constant that is orders of magnitude smaller than m, buckets are dense and the top pointer typically only moves locally.

3.3 Buffer Scoring Functions

Our algorithm is parameterized by a buffer score $s(\cdot)$ that determines the eviction order from $\mathcal{Q}$. We define four buffer scores that quantify how informative a node's neighborhood is and thus determine its priority in the buffer, shaping batch composition, locality, and ultimately partition quality. Section 4.1 shows our experimental evaluation of these scores (including CUTTANA's). As validated in our experiments, our proposed Hub-Aware Assigned Neighbors Ratio improves on CUTTANA's buffer score. Let v_t be the node streamed at time t, with neighborhood $N(v_t)$ and degree $d(v_t) = |N(v_t)|$. Let $V_1, \dots, V_k$ be the current blocks and $\text{block}(u)$ the assigned block of any already placed node u. For degree normalization, we define $\hat{d}(v_t) = d(v_t)/D_{\max}$, where $D_{\max}$ is the threshold for buffering (Section 3.2). For all scores, larger values indicate higher buffer priority.

1. **Assigned Neighbors Ratio (ANR).** ANR measures the fraction of a node's neighbors that have already been assigned to some partition:

$$\text{ANR}(v_t) = \frac{\sum_{i=1}^{k} |N(v_t) \cap V_i|}{d(v_t)} \tag{3}$$

2. **Hub-Aware Assigned Neighbors Ratio (HAA).** Building on the *Cuttana Buffer Score* (Equation 2), we introduce a refined parametric score in which the influence of ANR decreases with node degree, prioritizing hubs by degree and low-degree nodes by ANR:

$$\text{HAA}(v_t) = \hat{d}(v_t)^{\beta} + \theta \cdot \left(1 - \hat{d}(v_t)\right) \cdot \text{ANR}(v_t) \tag{4}$$

where $\beta \geq 1$ controls the shape of the degree contribution and θ balances the relative influence of neighborhood information.

3. **Neighborhood Seen Score (NSS).** NSS factors in neighbors currently present in the buffer $\mathcal{Q}$. For a weighting parameter $\eta \in [0, 1]$:

$$\mathrm{NSS}_\eta(v_t) = \frac{\sum_{i=1}^{k} |N(v_t) \cap V_i| + \eta\, |N(v_t) \cap \mathcal{Q}|}{d(v_t)} \tag{5}$$

4. **Community-Majority Score (CMS).** CMS prioritizes nodes by the largest number of assigned neighbors in the same partition to reflect underlying community structure:

$$\mathrm{CMS}(v_t) = \max_{p \in \{1,\ldots,k\}} \frac{|\{u \in N(v_t) : \mathrm{block}(u) = p\}|}{d(v_t)} \tag{6}$$

ANR (Eq. 3) measures available neighborhood assignment information: nodes with many already-assigned neighbors have more placement evidence and are evicted earlier. However, since ANR is normalized by $d(v_t)$, it systematically favors low-degree nodes and delays hubs, even though hubs serve as structural anchors for subsequent placements. The *Cuttana Buffer Score* (CBS, Eq. 2) addresses this by adding an explicit degree term, but combines it linearly with ANR, thus prioritizing a node only once both degree and ANR are large. HAA (Eq. 4) decouples these effects by decreasing the weight of ANR with normalized degree: the degree term $\hat{d}(v_t)^\beta$ prioritizes high-degree nodes as anchors irrespective of available neighborhood information, while the ANR term is emphasized for low-degree nodes via $(1-\hat{d}(v_t)) \cdot \mathrm{ANR}(v_t)$. Thus, HAA promotes two desirable classes of early evictions: (i) low-degree nodes once their neighborhood evidence becomes reliable, and (ii) hubs even when their assigned-neighbor ratio is still small. Figure 3 visualizes this separation: HAA assigns high priority to both "low-degree/high-ANR" and "high-degree/low-ANR" regions, whereas CBS concentrates priority near the diagonal where both signals are high.

3.4 Batch-Wise Multilevel Partitioning

Batched nodes in $\mathcal{B}$ and their incident edges define a *batch model graph* that is partitioned with the multilevel scheme used in HeiStream [18], summarized here. The model subgraph is augmented with k auxiliary nodes a_i, each representing a block i of the partition. For a node v and block i, we add an auxiliary edge (v, a_i) if v has at least one neighbor assigned to block i; its weight equals the number of such neighbors. Multilevel partitioning proceeds on the model graph: nodes are iteratively contracted to produce a hierarchy of smaller graphs, an initial partition is computed on the coarsest level using a weighted variant of the Fennel objective, and the solution is successively refined during uncoarsening [12, 18]. After multilevel partitioning, block assignments on the model subgraph are mapped back to the global partition. In HeiStream, this mapping is trivial because nodes are streamed in input order, so local IDs are offsets from the batch's first global ID. In BuffCut, nodes are buffered and processed out of order, requiring explicit local–to-global mappings.

3.5 Parallelization and Restreaming

Parallelization. We implement a parallel version of BuffCut that overlaps I/O, buffer management, and partitioning in three concurrent threads using bounded lock-free queues. Thread 1 (*I/O Reader*) parses the stream and pushes `ParsedLine` objects into `input_queue`. Thread 2 (*Priority Queue Handler*) pops from `input_queue`, computes buffer scores, maintains the priority buffer, and emits single-node or batch `PartitionTasks` into `partition_task_queue`. Thread 3 (*Partitioning Worker*) executes tasks (immediate assignment or batch multilevel partitioning) and commits the resulting assignments. Because

partitioning tasks overlap with continued buffering, the parallel schedule can yield slightly different batch composition than the sequential run; to keep scoring consistent, nodes are treated as *assigned* for buffer scoring as soon as their task is enqueued.

Restreaming. To improve solution quality, BuffCut supports restreaming with a configurable number of passes. The first pass operates as described above; subsequent passes refine the existing partition without buffering or prioritization. Since all nodes are assigned after the first pass, neighborhood-based ordering is no longer meaningful; nodes are processed sequentially and repartitioned using batch-wise multilevel refinement.

4 Experimental Analysis

We evaluate BuffCut on a diverse set of real-world and synthetic graphs and answer the following:

- **RQ1:** How effective are prioritized buffering and batch-wise partitioning in improving partition quality?
- **RQ2:** What is the impact of restreaming and parallelization on solution quality, runtime and memory consumption?
- **RQ3:** How does BuffCut compare to state-of-the-art buffered streaming partitioners in partition quality, memory and runtime?

We first describe our baselines, setup, instances, and evaluation methodology, then address our research questions through parameter studies on a Tuning Set and comparative analysis against state-of-the-art partitioners on a larger Test Set.

Setup and Reproducibility. All experiments were run on a dedicated machine with an `AMD EPYC 9754` CPU (128 cores, 256 threads, base frequency 2.25GHz), 755 GiB of DDR5 main memory, and an L2/L3 cache of 128 MiB / 256 MiB. The system also features an 894 GB NVMe solid-state drive and runs Ubuntu 22.04.4 LTS with Linux kernel version 5.15.0-140. We implemented BuffCut in `C++` within HeiStream[1] and compiled it with full optimization enabled (`-O3`). BuffCut uses $\mathit{discFactor} = 1\,000$, and $D_{\max} = 10\,000$ for all experiments. For comparison against the state-of-the-art, we obtained official implementations of HeiStream [18] and Cuttana [24]. Here, HeiStream refers to the legacy implementation accompanying in its original paper, prior to the integration of BuffCut in it. HeiStream requires a batch-size parameter δ (default 32 684); unless mentioned otherwise, we use $\delta = 1\,048\,576$ to ensure a fair comparison in terms of memory consumption. For Cuttana, we use the parallel implementation provided by the authors. To ensure that reported memory usage reflects algorithmic behavior rather than differences in file handling, we replace Cuttana's default memory-mapped I/O with standard streaming access (`ifstream`), consistent with HeiStream and our implementation. Cuttana is evaluated using the parameters recommended in the publication: $D_{\max} = 1\,000$, a queue size of 10^6 and a subpartition ratio $k'/k = 4\,096$ (referred to as Cuttana4k). We additionally evaluate a reduced configuration with $k'/k = 16$ (referred to as Cuttana16), which substantially lowers memory usage and runtime. All experiments are run using GNU Parallel, with up to five concurrent instances by default and twelve for the KONECT ordering experiments.

[1] The implementation is available as open source at `https://github.com/KaHIP/HeiStream`.

Table 1 Graph Instances. We use the Tuning Set for parameter studies and the Test Set for state-of-the-art comparisons, using three random stream orderings for each instance. Graphs with † are also tested under KONECT's first-appearance ordering [27] to match the setup in [24].

Exploration & Tuning Set				Test Set			
Graph	n	m	Type	Graph	n	m	Type
coPapersDBLP	540 486	15 245 729	Cit.	†orkut	3 072 411	117 185 082	Soc.
soc-lastfm	1 191 805	4 519 330	Soc.	arabic-2005	22 744 080	553 903 073	Web
in-2004	1 382 908	13 591 473	Web	nlpkkt240	27 933 600	373 239 376	Mat.
Flan_1565	1 564 794	57 920 625	Mesh	it-2004	41 291 594	1 027 474 947	Web
G3_Circuit	1 585 478	3 037 674	Circ.	†twitter-2010	41 652 230	1 202 513 046	Soc.
soc-flixster	2 523 386	7 918 801	Soc.	sk-2005	50 636 154	1 810 063 330	Web
Bump_2911	2 852 430	62 409 240	Mesh	com-Friendster	65 608 366	1 806 067 135	Soc.
FullChip	2 986 999	11 817 567	Circ.	rgg26	67 108 864	574 553 645	Gen.
cit-Patents	3 774 768	16 518 947	Cit.	rhg1B	100 000 000	1 000 913 106	Gen.
com-LJ	3 997 962	34 681 189	Soc.	rhg2B	100 000 000	1 999 544 833	Gen.
Ljournal-2008	5 363 186	49 514 271	Soc.	†uk-2007-05	105 896 555	3 301 876 564	Web
italy-osm	6686493	7 013 978	Road	webbase-2001	118 142 155	854 809 761	Web
great-britain-osm	7 733 822	8 156 517	Road				
uk-2002	18 520 486	261 787 258	Web				

Datasets. We evaluate on a diverse collection of real-world and synthetic benchmark graphs drawn from established repositories [3, 4, 5, 6, 20, 30, 35]. All instances have been used in prior work on graph partitioning and streaming algorithms [12, 13, 18]. We convert all graphs to the METIS format by removing self-loops and parallel edges, ignoring directions, and assigning unit weights to nodes and edges. Since Cuttana additionally requires node degrees to be explicitly provided, we convert compatible variants for it. Our evaluation uses two instance groups: a *Tuning Set* for parameter studies and a larger *Test Set* for scalability and comparisons. Table 1 summarizes all instances.

Experiments on the Tuning and Test Sets use *random stream orders*. For each graph, we generate three independent random permutations of node IDs and report geometric means. To characterize stream locality, we measure their AID (Equation 1). Under source node orderings, the geometric mean AID is 37 794 for the Tuning Set and 51 104 for the Test Set. Under random orderings, these values increase to 219 685 and 2 581 859, respectively, confirming that random orderings induce substantially lower locality and provide a challenging, order-agnostic evaluation setting. Graphs marked with † in Table 1 are additionally evaluated under the node order of these graphs as found in the KONECT repository [27] (henceforth the *KONECT ordering*) used by Hajidehi et al. [24]. Concretely, when extracting these graphs from their source, KONECT renumbers nodes in *first-appearance order* while scanning the edge list, which substantially reduces locality compared to the source ordering. We report results for this ordering to enable a direct comparison to the Cuttana publication.

Methodology. Unless stated otherwise, we enforce an imbalance of $\varepsilon = 3\%$. Tuning experiments are conducted with $k = 32$, and test experiments with $k \in \{4, 8, 16, 32, 64, 128, 256\}$. We evaluate solution quality using the *edge cut ratio* $\omega(E_{\text{cut}})/\omega(E)$, alongside end-to-end runtime and peak memory (maximum resident set size). To quantify how buffering impacts within-batch locality, we define the *Internal Edge Ratio* (IER). For a batch $B \subseteq V$, let

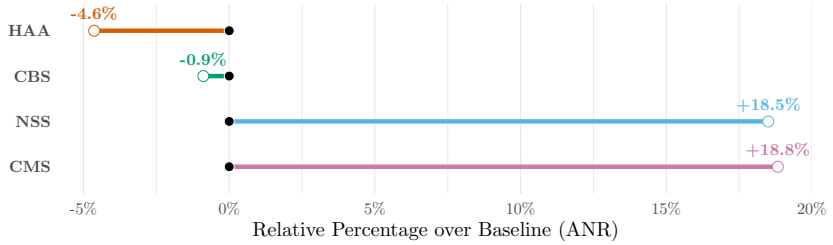
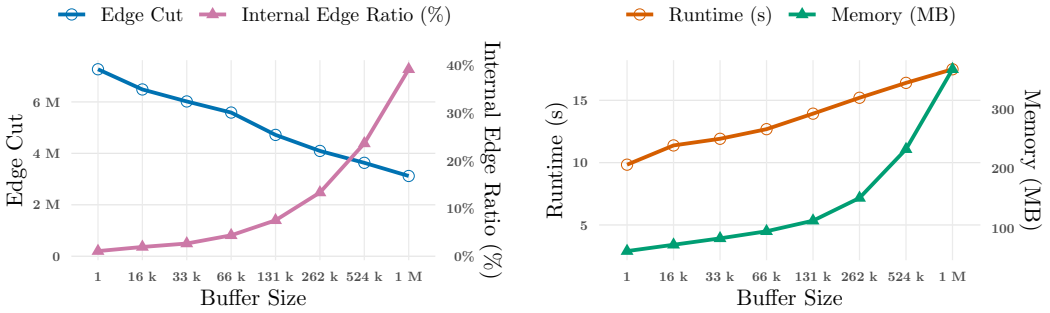


Figure 4 Impact of buffer score on cut quality (Tuning Set, random order, $k = 32$). Geometric means of edge cut relative to ANR (lower is better).



(a) Edge Cut and Internal Edge Ratio.

(b) Runtime and Memory.

Figure 5 Effect of buffer size $\mathcal{Q}_{\max}$ (Tuning Set, random order, $k = 32$). Larger buffers increase within-batch locality (IER) and reduce cut at increased memory cost.

$E(B) \coloneqq \{(u, v) \in E : u, v \in B\}$ be the set of edges induced by B. We define the ratio as:

$$\mathrm{IER}(B) \coloneqq \frac{2\,\omega(E(B))}{\sum_{v \in B} d_\omega(v)} \tag{7}$$

where $d_\omega(v)$ is the weighted degree of v. IER represents the fraction of incident edge weight contained entirely within B; we report the mean IER across all batches. For aggregate comparisons, we use geometric means and *performance profiles* [15], which shows the fraction of instances where an algorithm's performance is within a factor τ of the best observed result.

4.1 Prioritized Buffering and Batch-Wise Partitioning

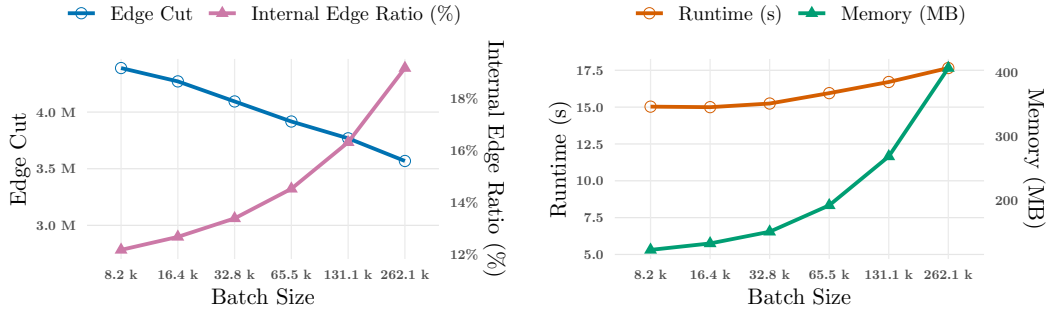
We assess the buffer scoring functions from Section 3.3. To address **RQ1**, we evaluate the effect of buffer size $\mathcal{Q}_{\max}$, i.e., how many nodes are stored for prioritized ordering, and batch size δ, i.e., how many nodes are partitioned together in the multilevel scheme.

Buffer Scoring Functions. We evaluate buffer scoring functions with respect to partition quality, fixing $\mathcal{Q}_{\max} = 262\,144$ and $\delta = 32\,768$ (Figure 4). Our proposed *Hub-Aware Assigned Neighbors Ratio* (HAA, with $\beta = 2, \theta = 0.75$) consistently yields the highest partition quality, reducing edge cuts by 4.6% compared to the *Assigned Neighbors Ratio* (ANR) baseline.

While the *Cuttana Buffer Score* (CBS) also improves upon ANR by 0.9%, it is consistently outperformed by HAA. Although both CBS and HAA incorporate degree information, HAA performs better because it more effectively balances degree-based prioritization with neighborhood information, as discussed in Section 3.3. NSS and CMS perform poorly, increasing edge

Table 2 Parallelization and restreaming trade-offs (Tuning Set, random order, $k = 32$). Geometric means of edge cut ratio, runtime, and peak memory. Parallelization reduces runtime with negligible impact on cut; additional restreaming passes improve cut at higher runtime.

	Configuration	Edge cut (%)	Runtime (s)	Memory (MB)
Par	Sequential	20.29	14.65	191.58
	Parallel	20.48	**7.85**	218.76
Restream	1 stream	20.29	**15.60**	191.81
	2 streams	17.33	22.54	192.02
	3 streams	16.71	29.50	192.17
	4 streams	16.43	36.60	192.08
	5 streams	**16.25**	43.72	191.96



(a) Edge Cut and Internal Edge Ratio.

(b) Runtime and Memory.

Figure 6 Effect of batch size δ (Tuning Set, random order, $k = 32$). Larger batches improve cut but increase memory due to larger induced model graphs.

cuts by $> 18\%$ relative to ANR, indicating that incorporating buffered neighbors provides little reliable information for prioritization, and emphasizing local majority effects leads to premature assignments. We utilize HAA as the default scoring function for BuffCut.

Buffer Size ($\mathcal{Q}_{\max}$). We study the effect of buffer size $\mathcal{Q}_{\max}$ on quality and computational overhead, fixing batch size at $\delta = 32\,768$ (Figure 5). Increasing $\mathcal{Q}_{\max}$ improves solution quality and within-batch locality, as measured by the Internal Edge Ratio (IER), by allowing the algorithm to defer assignments until more neighborhood information is available.

Disabling the priority buffer ($\mathcal{Q}_{\max} = 1$) results in the highest edge cut and negligible within-batch locality. Even modest buffer sizes provide significant gains: a buffer of 32 768 nodes reduces edge cut by 17.2% compared to the no-buffer baseline with minimal overhead. As $\mathcal{Q}_{\max}$ increases to 262 144, the IER grows from 1% to 13.4%, yielding a 43.7% reduction in edge cut. At the maximum tested buffer size of $2^{20} \approx 10^6$ nodes, BuffCut captures 39.2% of edges within batches, reducing the edge cut by 57.1% compared to using no buffer ($\mathcal{Q}_{\max} = 1$). Runtime increases moderately (1.8× from $\mathcal{Q}_{\max} = 1$ to 10^6), with a sharper increase in memory consumption (Figure 5b). These results demonstrate that BuffCut effectively trades memory for partition quality. Moderate buffer sizes ($\mathcal{Q}_{\max} \approx 262\,144$) represent a "sweet spot", capturing significant locality and reducing edge cut by over 40% while maintaining low memory and runtime overhead. This highlights the effectiveness of prioritized buffering in mitigating the effects of low-locality stream orders.

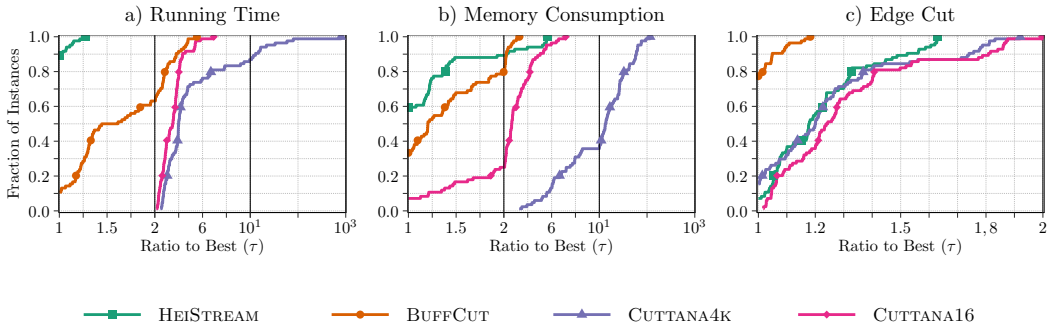


Figure 7 Performance profiles on the Test Set (random order, $k \in \{4, \ldots, 256\}$). Fraction of instances within factor τ of the best for edge cut, runtime, and peak memory.

Batch Size (δ). We analyze the sensitivity of solution quality and computational overhead to the batch size δ, with $\mathcal{Q}_{\max}$ fixed at 262 144 (Figure 6). Increasing δ consistently improves partition quality, as larger subgraphs provide the multilevel scheme with richer structural context. Specifically, expanding the batch size from 8 192 to 262 144 yields a 18.7% reduction in edge cut, driven by a steady increase in IER from 12% to nearly 20%, while runtime increases moderately (1.2×) and the memory footprint grows near-linearly (3.3×). Overall, batch size presents a trade-off between improved partition quality and increased memory and runtime overhead, with moderate batch sizes offering a favorable balance.

Takeaway. Our proposed HAA score outperforms all evaluated buffer scores, including Cuttana's. Prioritized buffering and batch-wise partitioning improve solution quality, up to 57.1% and 20% on average, respectively, for the largest buffer and batch size tested, with computational overhead controlled via the configurable buffer and batch size.

4.2 Parallelization and Restreaming

To answer **RQ2**, we compare the parallel and restreaming implementations of BuffCut to its default sequential, single stream implementation (Table 2), setting $\mathcal{Q}_{max} = 262\,144$ and $\delta = 65\,536$. Parallel BuffCut yields a substantial 1.87× speedup over the sequential implementation with the same solution quality, at the cost of a modest 14.2% increase in memory consumption. Our results show that restreaming with five passes improves partitioning quality up to 19.9%, but is 2.8× slower than single-stream BuffCut. Using two streams provides the best trade-off between solution quality and runtime, reducing the edge cut by 14.6%, while being only 1.44× slower than using one stream. Restreaming does not incur proportional slowdowns because subsequent passes omit buffering.

4.3 Comparison to State-of-the-Art

To answer **RQ3**, we compare parallelized BuffCut against all baselines on the Test Set under *random* node orderings. In addition, for graphs marked with † in Table 1, we report results under KONECT's reordering [27] to match Cuttana's experimental setup [24].

Test Set Evaluation. We evaluate parallelized BuffCut ($\mathcal{Q}_{max} = 1\,048\,576$, $\delta = 65\,536$) against HeiStream and parallelized Cuttana on the Test Set under random stream orders (Figure 7). BuffCut demonstrates clear dominance in solution quality, achieving the best

Table 3 Quality–runtime comparison with $k = 8$ and 5% imbalance. Edge cut ratio (%) and running time (s); lower is better. Results are reported for original source ordering and their KONECT-ordering counterparts [27]. Restreaming variants (one additional pass) are separated by a thick rule. Best values per column are highlighted.

	twitter		uk-2007-05		orkut		uk-2002	
Algorithm	Source	KONECT	Source	KONECT	Source	KONECT	Source	KONECT
Edge cut ratio (%)								
HeiStream	50.74	52.57	3.07	6.29	42.85	42.15	1.62	11.16
Cuttana	50.99	43.24	22.31	1.37	34.04	34.59	21.70	3.24
BuffCut	49.32	44.79	1.45	1.30	27.95	27.92	1.33	2.74
HeiStream-Re	47.98	50.67	**0.34**	**0.72**	36.36	36.29	**0.84**	2.34
BuffCut-Re	**46.23**	**41.92**	1.27	1.04	**25.31**	**26.08**	1.07	**1.26**
Running time (s)								
HeiStream	**157.84**	**136.74**	**477.45**	**365.39**	**14.95**	**12.98**	**44.17**	**36.60**
Cuttana	670.71	507.60	898.98	767.74	251.99	259.68	253.59	256.31
BuffCut	317.49	340.92	741.63	550.30	46.15	45.96	62.03	55.93
HeiStream-Re	407.55	376.63	1019.65	846.11	39.77	35.60	90.47	80.20
BuffCut-Re	517.06	504.45	1310.19	1060.54	68.67	67.50	120.23	98.09

edge cut on roughly 80% of all instances. While HeiStream is the most resource-efficient as it lacks prioritized reordering, achieving the best runtime and memory on 90% and 60% of instances, respectively, BuffCut provides a significantly more favorable quality-efficiency trade-off. Compared to HeiStream, BuffCut reduces the edge cut by 15.8% on average with modest overheads of 1.8× in runtime and 1.09× in memory. BuffCut consistently outperforms Cuttana across all metrics. Both Cuttana variants are at least 2× slower on all instances; Cuttana4k is up to 1000× slower than the best baseline, while BuffCut is at worst 6× slower. Memory usage shows the same pattern: Cuttana4k always uses at least 3× more memory than the best baseline and requires up to 100× more memory in the worst case. Cuttana16 reduces this but still exceeds BuffCut on most instances. Overall, BuffCut is the most memory-efficient in about 35% of cases and always within approximately 3× of the best baseline. Comparing geometric means across all instances and k values, BuffCut achieves 20.8% and 26.4% fewer edge cuts than Cuttana4k and Cuttana16, respectively, while running 2.9× faster with 11.3× less memory than Cuttana4k, and 1.9× faster with 2× less memory than Cuttana16.

KONECT Ordering. To match Hajidehi et al.'s [24] experimental evaluation, we present results on the graphs found in the KONECT repository [27], marked with † in Table 1, on both their source ordering and KONECT ordering. We run BuffCut with $\mathcal{Q}_{\max} = 2{,}097{,}152$ and $\delta = 262{,}144$. We configure Cuttana and HeiStream with the parameters set by Hajidehi et al. [24] and use an imbalance of $\varepsilon = 5\%$ to match their experimental setup. Although [24] reports Cuttana to be more efficient than HeiStream, our results show that Cuttana's gains in quality come at a substantial computational cost. On the KONECT orderings, Cuttana improves the edge cut by between 17.7% and 78.2% relative to HeiStream, but is between 2.1× and 20.0× slower (Table 3). Moreover, Cuttana exhibits significantly higher memory consumption as k increases, using, for example, up to 260× more memory than HeiStream at $k = 256$ (Figure 8). HeiStream's weaker performance

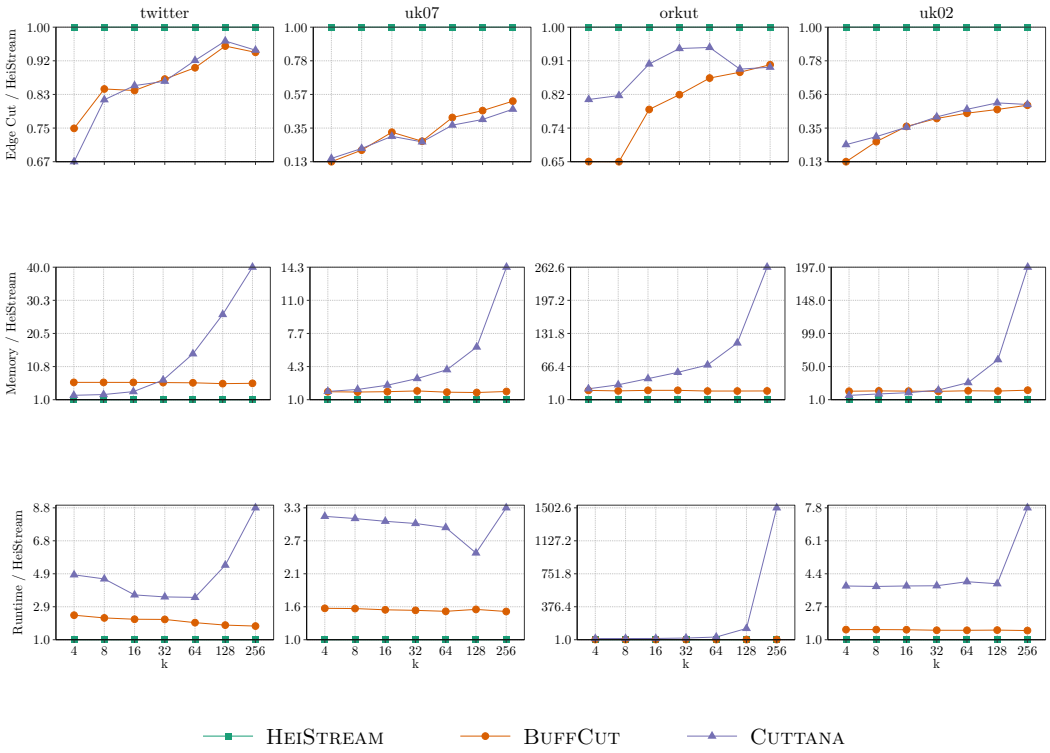


Figure 8 Baseline comparisons under KONECT orderings (matching [24]). Normalized edge cut, runtime, and peak memory versus k relative to HeiStream.

in these experiments is largely attributable to stream ordering rather than inherent graph structure. The KONECT repository induces a first-appearance renumbering of node IDs that substantially reduces stream locality. When evaluated on the same graphs in their source orderings, HeiStream outperforms Cuttana in three out of four instances. In contrast, BuffCut is explicitly designed to be robust to adverse stream orderings, outperforming both Cuttana and HeiStream on all instances, except `twitter` under KONECT ordering. On KONECT ordering, it improves upon Cuttana's solution quality by up to 20.0% while using substantially fewer resources. BuffCut is faster across all four instances, with memory consumption remaining effectively constant as k increases, while Cuttana's resource consumption escalates sharply for $k > 32$ (Figure 8). With one additional streaming pass, BuffCut achieves lower edge cut than Cuttana across all KONECT instances and is faster in three of four instances.

Takeaways. BuffCut achieves the best solution quality, 15.8% and 20.8% better than HeiStream and Cuttana respectively, on the low-locality, randomly ordered Test Set, while being 2.9× faster and using 11.3× less memory than Cuttana, with modest overheads of 1.8× runtime and 1.09× memory over HeiStream. On KONECT-ordered instances, BuffCut achieves comparable or better edge cut (up to 20.0%), while being faster on all instances and significantly more memory-efficient with increasing k.

5 Conclusion

We present BuffCut, a buffered streaming graph partitioner for computing high-quality balanced k-way partitions under strict memory constraints on adversarial stream ordering. Our approach uses prioritized batching: a bounded priority buffer actively regulates the order in which nodes are committed, delaying poorly informed decisions and incrementally forming batches with high internal locality, which are then partitioned using a multilevel scheme. Additionally, we introduce a refined buffer score that balances degree and available neighborhood information, and present a parallel implementation of BuffCut that achieves 1.8× faster runtime. Experiments on real-world and synthetic graphs under random node orderings demonstrate that, in geometric means, BuffCut achieves 20.8% fewer edge cuts while being 2.9× faster and using 11.3× less memory than the state-of-the-art prioritized buffering baseline, Cuttana. It produces 15.8% better solution quality than the next-best buffered streaming partitioner, HeiStream, with modest overheads of 1.8× runtime and 1.09× memory. Compared to Cuttana on the KONECT-ordered instances evaluated by its authors, BuffCut is faster, uses less memory with increasing k, and achieves up to 20% improvement in solution quality. In future work, we aim to extend prioritized buffered streaming to dynamic graphs and distributed-memory settings. Our implementation of BuffCut is integrated into HeiStream and is available as open source at `https://github.com/KaHIP/HeiStream`.

References

1 C. J. Alpert and A. B. Kahng. Recent Directions in Netlist Partitioning: A Survey. *Integration, the VLSI Journal*, 19(1-2):1–81, 1995. doi:10.1016/0167-9260(95)00008-4.

2 C. J. Alpert, A. B. Kahng, and S. Z. Yao. Spectral Partitioning With Multiple Eigenvectors. *Discrete Appl. Math.*, 90(1):3–26, 1999. doi:10.1016/S0166-218X(98)00083-3.

3 David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014. doi:10.1007/978-1-4614-6170-8_23.

4 Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. BUbiNG: Massive Crawling for the Masses. In Chin-Wan Chung, Andrei Z. Broder, Kyuseok Shim, and Torsten Suel, editors, *23rd International World Wide Web Conference, WWW '14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 227–228, 2014. doi:10.1145/2567948.2577304.

5 Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered Label Propagation: A Multiresolution Coordinate-free Ordering For Compressing Social Networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 587–596, 2011. doi:10.1145/1963405.1963488.

6 Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. doi:10.1145/988672.988752.

7 Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced Graph Edge Partition. In Sofus A. Macskassy, Claudia Perlich, Jure Leskovec, Wei Wang, and Rayid Ghani, editors, *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 1456–1465, 2014. doi:10.1145/2623330.2623660.

8 Thang Nguyen Bui and Curt Jones. Finding Good Approximate Vertex and Edge Partitions Is Np-Hard. *Information Processing Letters*, 42(3):153–159, 1992. doi:10.1016/0020-0190(92)90140-Q.

9 Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. Springer, 2016. doi:10.1007/978-3-319-49487-6_4.

10 Ü. V. Çatalyürek and C. Aykanat. Decomposing Irregularly Sparse Matrices For Parallel Matrix-Vector Multiplication. In *Proc. of the 3rd Intl. Workshop on Parallel Algorithms for Irregularly Structured Problems*, volume 1117, pages 75–86, 1996. doi:10.1007/BFB0030098.

11 Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More Recent Advances in (Hyper)Graph Partitioning. *ACM Comput. Surv.*, 55(12):253:1–253:38, 2023. doi:10.1145/3571808.

12 Adil Chhabra, Marcelo Fonseca Faraj, Christian Schulz, and Daniel Seemaier. Buffered Streaming Edge Partitioning. In Leo Liberti, editor, *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*, volume 301 of *LIPIcs*, pages 5:1–5:21, 2024. doi:10.4230/LIPIcs.SEA.2024.5.

13 Adil Chhabra, Florian Kurpicz, Christian Schulz, Dominik Schweisgut, and Daniel Seemaier. *Partitioning Trillion Edge Graphs on Edge Devices*, pages 74–89. SIAM, 2025. doi:10.1137/1.9781611978759.6.

14 D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *Proc. of the 10th Intl. Sym. on Experimental Algorithms*, volume 6630 of *LCNS*, pages 376–387, 2011. doi:10.1007/978-3-642-20662-7_32.

15 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking Optimization Software With Performance Profiles. *Math. Program.*, 91(2):201–213, 2002. doi:10.1007/S101070100263.

16 Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. Locality Analysis of Graph Reordering Algorithms. In *IEEE International Symposium on Workload Characterization, IISWC 2021, Storrs, CT, USA, November 7-9, 2021*, pages 101–112, 2021. doi:10.1109/IISWC53511.2021.00020.

17 Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. Graph Neural Networks for Social Recommendation. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia, editors, *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 417–426, 2019. doi:10.1145/3308558.3313488.

18 Marcelo Fonseca Faraj and Christian Schulz. Buffered Streaming Graph Partitioning. *ACM J. Exp. Algorithmics*, 27:1.10:1–1.10:26, 2022. doi:10.1145/3546911.

19 J. Fietz, M. Krause, C. Schulz, P. Sanders, and V. Heuveline. Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. In *Proc. of Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 818–829, 2012. doi:10.1007/978-3-642-32820-6_81.

20 Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free Massively Distributed Graph Generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019. doi:10.1016/J.JPDC.2019.03.011.

21 Michael R. Garey, David S. Johnson, and Larry Stockmeyer. Some Simplified Np-Complete Problems. In *Proc. of the 6th ACM Sym. on Theory of Computing*, (STOC), pages 47–63, 1974. doi:10.1145/800119.803884.

22 A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

23 Lars Gottesbüren, Tobias Heuer, Peter Sanders, Christian Schulz, and Daniel Seemaier. Deep Multilevel Graph Partitioning. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, Lisbon, Portugal (Virtual Conference), September 6-8, 2021*, LIPIcs, pages 48:1–48:17, 2021. doi:10.4230/LIPIcs.ESA.2021.48.

24 Milad Rezaei Hajidehi, Sraavan Sridhar, and Margo I. Seltzer. CUTTANA: Scalable Graph Partitioning for Faster Distributed Graph Databases and Analytics. *Proceedings of the VLDB Endowment*, 18(1):14–27, 2024. URL: https://www.vldb.org/pvldb/vol18/p14-hajidehi.pdf.

25 Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving Graph Processing at Scale. In Peter Boncz and Josep Lluís Larriba-Pey, editors, *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*, page 5, 2016. doi:10.1145/2960414.2960419.

26 George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme For Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998. doi:10.1137/S1064827595287997.

27 Jérôme Kunegis. KONECT: the Koblenz Network Collection. In Leslie Carr, Alberto H. F. Laender, Bernadette Farias Lóscio, Irwin King, Marcus Fontoura, Denny Vrandecic, Lora Aroyo, José Palazzo M. de Oliveira, Fernanda Lima, and Erik Wilde, editors, *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*, pages 1343–1350, 2013. doi:10.1145/2487788.2488173.

28 U. Lauther. An Extremely Fast, Exact Algorithm For Finding Shortest Paths in Static Networks With Geographical Background. In *Proc. of the Münster GI-Days*, 2004.

29 Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved Code Summarization via a Graph Neural Network. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 184–195, 2020. doi:10.1145/3387904.3389268.

30 Jure Leskovec. Stanford Network Analysis Package (Snap), 2013.

31 Shuangli Li, Jingbo Zhou, Tong Xu, Liang Huang, Fan Wang, Haoyi Xiong, Weili Huang, Dejing Dou, and Hui Xiong. Structure-aware Interactive Graph Neural Networks for the Prediction of Protein-ligand Binding Affinity. In Feida Zhu, Beng Chin Ooi, and Chunyan Miao, editors, *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, pages 975–985, 2021. doi:10.1145/3447548.3467311.

32 Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. Pick and Choose: A Gnn-based Imbalanced Learning Approach for Fraud Detection. In Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia, editors, *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, pages 3168–3177, 2021. doi:10.1145/3442381.3449989.

33 Nikolai Merkel, Daniel Stoll, Ruben Mayer, and Hans-Arno Jacobsen. An Experimental Comparison of Partitioning Strategies for Distributed Graph Neural Network Training. In Alkis Simitsis, Bettina Kemme, Anna Queralt, Oscar Romero, and Petar Jovanovic, editors, *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*, pages 171–184, 2025. doi:10.48786/EDBT.2025.14.

34 Joel Nishimura and Johan Ugander. Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing. In Inderjit S. Dhillon, Yehuda Koren, Rayid Ghani, Ted E. Senator, Paul Bradley, Rajesh Parekh, Jingrui He, Robert L. Grossman, and Ramasamy Uthurusamy, editors, *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, pages 1106–1114, 2013. doi:10.1145/2487575.2487696.

35 Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 4292–4293, 2015. doi:10.1609/AAAI.V29I1.9277.

36 Peter Sanders and Christian Schulz. Engineering Multilevel Graph Partitioning Algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480, 2011. doi:10.1007/978-3-642-23719-5_40.

37 Isabelle Stanton and Gabriel Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In Qiang Yang, Deepak Agarwal, and Jian Pei, editors, *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, pages 1222–1230, 2012. doi:10.1145/2339530.2339722.

38 Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: Streaming Graph Partitioning For Massive Scale Graphs. In Ben Carterette, Fernando Diaz, Carlos Castillo, and Donald Metzler, editors, *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 333–342, 2014. doi:10.1145/2556195.2556213.

39 Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-scale Recommender Systems. In Yike Guo and Faisal Farooq, editors, *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 974–983, 2018. doi:10.1145/3219819.3219890.