Improving Processing Order in Streaming Graph Partitioning

Linus Baumgärtner

October 10, 2025

3671380

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Referee:
Prof. Dr. Felix Joos

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Adil Chhabra, for his continuous support, valuable guidance, and many insightful discussions throughout the course of this thesis. His advice has been invaluable and greatly shaped the quality of this work.

I am also thankful to Prof. Christian Schulz, who first sparked my interest in algorithm engineering through his course and the competitive challenges he created for our class. Competing in those problems made problem solving genuinely addictive and left me with a lasting fascination for algorithms. His enthusiasm for the subject has also been a great source of inspiration, and I am grateful that he gave me the opportunity to pursue my thesis in this field under his supervision.

Beyond the academic support, I would like to thank my friends and my brother for their support and for the many moments of encouragement and good company throughout this journey. Above all, I owe special thanks to my family for their unconditional support, without which this work would not have been possible.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, October 10, 2025

Linus Baumgärtner

Zusammenfassung

Die Partitionierung von Graphen ist eine zentrale Technik für skalierbare Graphanalysen. Sie sorgt dafür, dass die Rechenlast gleichmäßig verteilt wird und die Kommunikation zwischen Maschinen möglichst gering bleibt. Klassische Offline-Methoden wie KaHIP oder METIS liefern sehr hohe Qualität, erfordern jedoch den gesamten Graphen im Arbeitsspeicher. Streaming-Ansätze umgehen dieses Problem, indem sie Knoten nacheinander einlesen und verarbeiten. Dadurch lassen sich auch sehr große Graphen auf kleineren Systemen bearbeiten, allerdings häufig mit Einbußen bei der Partitionierungsqualität.

Ein Mittelweg zwischen Offline und purem Streaming ist Buffered Streaming. Ein aktueller Vertreter davon ist HeiStream, welches Knoten in Batches sammelt und diese mithilfe eines Multilevel-Ansatzes partitioniert. Dies ist besonders effektiv, wenn die Eingabereihenfolge Lokalität aufweist – also aufeinanderfolgende Knoten im Stream auch im Graphen nahe beieinanderliegen. Fehlt diese Struktur, verschlechtert sich die Qualität jedoch deutlich. Cuttana, ein anderer Vertreter, setzt hier auf priorisiertes Buffering und erzielt robustere Ergebnisse, benötigt dafür aber erheblich mehr Speicher und Laufzeit.

In dieser Arbeit stellen wir BuffCut vor, ein Verfahren, das HeiStream um priorisiertes Buffering nach dem Vorbild von Cuttana erweitert, dabei aber deutlich effizienter bleibt. Kern ist eine Bucket-Queue mit einem neu entwickelten Buffer Score, ergänzt durch Ghost Neighbors für zusätzliche Lokalitätsinformation sowie eine parallelisierte Implementierung zur Laufzeitoptimierung.

Die Experimente zeigen: Bei zufällig permutierten Knotenreihenfolgen reduziert BuffCut den Schnitt im Mittel um etwa 16% gegenüber HeiStream, bei vergleichbarem Speicherbedarf und moderatem Laufzeitaufwand. Gegenüber Cuttana erzielt es eine Verbesserung um etwa 21% bei weniger als der Hälfte der an Laufzeit und Speicherbedarf. Mit nur einer zusätzlichen Restreaming-Runde lässt sich der Vorteil gegenüber Cuttana weiter vergrößern (38%) ohne dabei dessen Ressourcenverbrauch zu überschreiten. Bei günstigen Eingabereihenfolgen bleibt HeiStream zwar leicht überlegen, doch unser Ansatz erreicht stabile Ergebnisse und übertrifft Cuttana deutlich.

Insgesamt verbindet BuffCut die Stärken von HeiStream und Cuttana in einem einzigen Verfahren. Es erhöht die Robustheit bei ungünstigen Eingaben, ohne die Effizienz zu opfern, und setzt damit einen neuen Maßstab im Bereich des Buffered Streaming Graph Partitioning.

Abstract

Graph partitioning is a key technique for scalable graph analytics, as it balances workload across machines while minimizing communication. While classical offline methods such as KaHIP or METIS achieve very high quality, they require the full graph in memory. Streaming approaches, in contrast, process vertices sequentially with limited memory, enabling even very large graphs to be handled on modest machines, though often at the cost of quality.

Among streaming partitioners, HeiStream represents the state of the art in buffered streaming: instead of assigning each vertex immediately, it collects batches of nodes, builds a compact subgraph, and applies multilevel partitioning. This design is highly effective when the input exhibits locality—when consecutive vertices in the stream are also close in the graph—since batches then capture coherent neighborhoods. However, under stream inputs inheriting low locality, this advantage disappears and the quality of HeiStream drops considerably. An alternative approach, Cuttana, addresses this by prioritized buffering, yielding more robust results but at high runtime and memory cost.

To overcome these limitations, we develop BuffCut, a buffered streaming partitioner that extends HeiStream with prioritized buffering inspired by Cuttana. Vertices are ranked in a bucket-based priority queue using a novel buffer score, leading to more informed batch construction. Additional extensions include ghost neighbors for richer locality information and a parallelized implementation for improved runtime scalability.

Experiments show that BuffCut achieves the intended robustness: on randomly permutated orderings it reduces the geometric mean cut by about 16% compared to HeiStream while using comparable memory and only moderately more runtime. Compared to Cuttana, it achieves 21% better quality with less than half the resource requirements. Even a single restreaming pass amplifies these gains, yielding improvements over Cuttana by about 38%, while still remaining more memory and runtime efficient. On favorable orderings, HeiStream remains slightly ahead, but our method stays competitive and consistently outperforms Cuttana.

In summary, BuffCut combines the strengths of HeiStream and Cuttana in a single design. It improves robustness under poor orderings without sacrificing efficiency, thus advancing the state of the art in buffered streaming partitioning. At the same time, a clear gap remains between favorable and unfavorable orderings, reflecting the inherent limits of streaming where only part of the graph can be held in memory.

Contents

CHAPTER 1

Introduction

Graphs are a fundamental abstraction for modeling complex relationships in a wide range of domains, including social networks, citation networks, biological systems, and the web. As real-world graphs continue to grow in size, often comprising billions of nodes and edges, efficient storage and processing become critical. A key enabler for scalable graph computation is graph partitioning, which aims to divide the graph into k balanced blocks while minimizing the number of edges that span across blocks. This not only balances the computational load across machines but also reduces inter-machine communication — a dominant cost factor in distributed processing.

Since graph partitioning is NP-complete [20], heuristic approaches are used in practice. Existing algorithms fall into three main categories: offline shared-memory algorithms, distributed-memory parallel algorithms, and streaming algorithms. Offline tools such as KaHIP [36] and METIS [24] achieve high-quality partitions by repeatedly coarsening and refining the graph, but they require that the entire input fits into memory. Distributed approaches can scale to larger inputs but incur high resource costs and require sophisticated infrastructure. Streaming algorithms, on the other hand, process the input graph sequentially and assign nodes to partitions on the fly, using only limited memory. While appealing for scalability, one-pass streaming methods often suffer from poor partition quality.

To close this quality gap, the HeiStream [18] algorithm was introduced. Instead of assigning vertices one by one, HeiStream buffers a fixed-size batch of nodes together with their adjacency information before making assignment decisions. This allows the algorithm to capture more structural context while keeping memory usage bounded by the configurable batch size. For each batch, a compact model graph is constructed that includes both the buffered nodes and their already partitioned neighbors. A multilevel partitioning scheme is then applied to this model using a generalization of the Fennel [41] objective, leading to significantly improved partition quality. Experiments show that HeiStream can outperform classic streaming algorithms such as Fennel and LDG [39].

1.1 Motivation

Building on the strengths of HeiStream, we now turn to its limitations and the motivation for our work. A notable drawback of HeiStream, and of streaming partitioners more broadly, is their lack of *global information.* By design, one-pass (or buffered) streaming methods must make placement decisions using only the information available in the current window of the stream; they cannot exploit a holistic view of the graph as offline multilevel methods do. Consequently, each decision relies on partial, local context rather than global structure.

In HeiStream, nodes are processed in consecutive batches together with their currently assigned neighbors. The structural context available at each step therefore depends heavily on which nodes happen to be co-loaded. When the input ordering is favorable (e.g., nodes from the same community appear close together), HeiStream can leverage local neighborhoods to achieve excellent results. However, when the ordering is random or lacks locality, the model graph for a batch is fragmented, many neighborhood relations remain unseen within the batch, and the resulting decisions approximate random choices. This sensitivity to input order, rooted in the absence of global information, is a key bottleneck for robustness and broad applicability.

In response, recent work proposes streaming techniques that explicitly mitigate order dependence. Cuttana, for example, introduces prioritized buffering: a priority queue guided by a buffer score that quantifies how much is already known about a node's neighborhood within the stream. Rather than following the raw stream order, the algorithm evicts the most informative buffered node and assigns it using Fennel, thereby delaying premature placements and reducing the impact of poor locality. This strategy helps to maintain high partition quality even when the stream lacks structure.

Motivated by these insights, this thesis aims to extend HeiStream methodologically to better handle poorly ordered streams: we seek to preserve its efficiency and multilevel strengths while reducing order sensitivity by introducing principled buffering and prioritization mechanisms.

1.2 Our Contribution

The central contribution of this thesis is the design of BuffCut, a novel hybrid buffered streaming algorithm that unites the advantages of two state of the art methods. From Cuttana, BuffCut adopts the idea of a priority-based buffer that reorders nodes according to their neighborhood information, thereby reducing the sensitivity to poor input orderings. From HeiStream, it incorporates the use of multilevel partitioning on buffered batches, which leverages global structural information to produce high-quality partitions with controlled memory usage.

By carefully combining these complementary techniques, BuffCut achieves the best of both worlds: the robustness of priority-based streaming with the partitioning quality of

multilevel refinement. Nodes are streamed into a bucket-based priority queue, extracted in order of their buffer score, and accumulated into batches of configurable size. Once a batch is full, it is partitioned using a multilevel coarsening–refinement scheme. This pipeline allows BuffCut to adapt flexibly to the quality of the input ordering: when locality is high, it exploits it effectively in the multilevel partitioning; when locality is low, it compensates through informed buffering.

BuffCut can also be seen as an extension of HeiStream: by setting the buffer size to zero, it effectively reproduces the behavior of HeiStream, including its strong performance on favorable natural orderings. In its default configuration, however, the focus is on robustness rather than specializing for favorable orderings. In the most challenging scenario with randomly ordered graphs, BuffCut improves the geometric mean cut by about 16% compared to HeiStream and by about 21% compared to Cuttana, while keeping runtime and memory requirements moderate and far below those of Cuttana in competitive settings. On orderings containing high locality, BuffCut remains competitive, with HeiStream still holding a slight advantage, while still outperforming Cuttana by a clear margin.

Beyond this core pipeline, we introduce two methodological innovations. First, it features a novel buffer score that systematically balances vertex degree and assigned-neighbor ratio, leading to more consistent partition quality across diverse graphs and orderings. Second, it integrates the concept of *ghost neighbors*, which provide provisional locality information from unpartitioned vertices. While this comes at a cost in runtime and memory, it yields additional improvements in cut quality by enriching both buffer score and batch construction with structural hints that would otherwise be ignored.

Finally, to enhance practicality, BuffCut offers a parallelized implementation that decouples input, buffering, and partitioning into separate threads, thereby accelerating runtime with only a small memory overhead. Altogether, BuffCut advances the state of the art by outperforming HeiStream and Cuttana in input orderings with poor locality, while retaining competitive performance on inputs with high locality. [1]

1.3 Structure

The remainder of this thesis is organized as follows. After the introduction and the fundamental definitions in chapter 2, we discuss related work in chapter 3. Then, we present our main contribution in chapter 4, namely a new buffered streaming partitioning algorithm, BuffCut, that integrates priority-based buffering with multilevel refinement in order to reduce the sensitivity to the input ordering of the graph stream. Next, we provide comprehensive parameter studies and experimental evaluations with the state of the art algorithms in chapter 5. Finally, we discuss the results in chapter 6, conclude the thesis, and outline directions for future research.

[1] A C++ implementation of BuffCut is publicly available at `https://github.com/libaum/BuffCut`.

CHAPTER 2

Fundamentals

This chapter provides the theoretical background required for the development and analysis of our algorithm. We first recall basic graph-theoretic notation and the formal definition of the graph partitioning problem, before introducing streaming-based models and the notion of stream locality, which form the foundation of our contributions.

2.1 Basic Concepts

An undirected graph is defined as $G = (V, E)$, where $V = \{0, \ldots, n-1\}$ is the set of vertices (or nodes), and $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ is the set of undirected edges. We denote $n = |V|$ as the number of nodes and $m = |E|$ as the number of edges. Each vertex $v \in V$ can be assigned a non-negative node weight $c(v) \in \mathbb{R}_{\geq 0}$, and each edge $e = \{u, v\} \in E$ has an edge weight $\omega(e) \in \mathbb{R}_{>0}$. For a subset of vertices $V' \subseteq V$, we define $c(V') := \sum_{v \in V'} c(v)$ as its total weight. For a subset of edges $E' \subseteq E$, we define $\omega(E') := \sum_{e \in E'} \omega(e)$ as its total edge weight. The neighborhood $N(v)$ of a node v is the set of adjacent nodes: $N(v) = \{u \in V \mid \{u, v\} \in E\}$. A subgraph $S = (V', E')$ of G is defined by a subset of nodes $V' \subseteq V$ and a subset of edges $E' \subseteq E \cap (V' \times V')$. If $E' = E \cap (V' \times V')$, we call S an *induced subgraph*.

Graph Partitioning. The *Graph Partitioning Problem* seeks to divide the node set V into k disjoint blocks $V_1, \ldots, V_k$ such that $V = V_1 \cup \cdots \cup V_k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The partition must satisfy a balance constraint: for a given imbalance parameter $\epsilon \geq 0$, each block obeys

$$c(V_i) \leq L_{\max} := \left\lceil \frac{(1+\epsilon)\, c(V)}{k} \right\rceil .$$

Here $L_{\max}$ denotes the maximum allowed block weight.

Among all balanced partitions, the objective is to minimize the *edge cut*, defined as the total weight of edges crossing between blocks,

$$\text{cut}(V_1, \ldots, V_k) := \sum_{i<j} \omega(E_{ij}), \quad \text{where } E_{ij} := \{\{u, v\} \in E \mid u \in V_i, v \in V_j\}.$$

A helpful abstraction of a partition is the *quotient graph* $Q = (V_Q, E_Q)$, where each block V_i is represented as a single vertex in Q, and an edge connects V_i and V_j in Q if any edge exists between these blocks in G. The weight of a quotient node equals the total node weight in its block, and the weight of a quotient edge equals the total weight of the original edges crossing between the respective blocks.

2.2 Streaming Models and Locality

Classical offline partitioners assume full access to the graph. In contrast, streaming models restrict how vertices are processed. In the *one-pass streaming model*, vertices arrive sequentially together with their incident edges and must be assigned to blocks immediately upon arrival using only local information. This model keeps memory usage at $O(\text{polylog}(n))$ but suffers from limited global context, which often leads to high edge cuts, especially when the input order provides little locality.

Buffered Streaming Graph Partitioning. To overcome the shortcomings of one-pass streaming algorithms, the *buffered streaming model* extends the basic idea by postponing assignments. Instead of placing each vertex immediately, a bounded set of vertices together with their neighborhoods is first collected in memory. Once this set reaches capacity, its vertices are assigned simultaneously—often using more advanced methods such as multilevel partitioning. This trades moderate additional memory for substantially improved partitioning quality, since the algorithm can exploit more structural context before making decisions.

In this work, we distinguish between a *buffer* and a *batch*. The buffer is a priority queue that temporarily stores streamed vertices to delay their assignment and thus avoid premature decisions. Once the buffer reaches capacity, a batch of up to δ vertices is extracted and partitioned simultaneously using the multilevel scheme. We denote the buffer capacity by Λ and, following HeiStream, the batch size by δ. Together these parameters determine the additional memory footprint of the algorithm: beyond the $\Omega(n)$ space needed to store final vertex assignments, the buffer may hold up to Λ vertices with their incident edges, while the batch contains up to δ vertices at a time. In practice, both values are chosen according to the available memory, keeping overall space consumption linear in n with a moderate and controllable overhead.

Stream Order and Locality. A stream order is a permutation $S = (v_0, \ldots, v_{n-1})$ of V. The quality of a stream order can be assessed in terms of its *locality*, i.e., whether neighboring vertices in the graph also appear close to one another in the stream.

As a concrete locality measure, we adopt the *Neighbor to Neighbor Average ID Distance (AID)* proposed by Esfahani et al. [16]. For a vertex v with sorted neighbors $N_{v,1}, \ldots, N_{v,|N_v|}$, the measure is defined as

$$AID_v = \frac{1}{|N_v|} \sum_{i=2}^{|N_v|} \left| N_{v,i} - N_{v,i-1} \right|.$$

It captures how tightly the neighbors of a vertex are clustered in the vertex ID space: lower values indicate that neighbors are assigned consecutive IDs, which corresponds to higher spatial locality. As a graph-level locality value, we take the arithmetic mean of AID_v across all vertices with degree at least two:

$$AID(G) := \frac{1}{|V'|} \sum_{v \in V'} AID_v, \quad \text{where } V' = \{v \in V \mid |N_v| \geq 2\}.$$

This aggregate measure $AID(G)$ allows us to directly compare the locality of different stream orders. Intuitively, a low $AID(G)$ value indicates that neighbors tend to appear close together in the stream, which benefits streaming partitioners.

In practice, the stream order is not chosen by the partitioner itself but results from the way the graph is stored in the input file. The vertex order in the file directly determines the sequence in which nodes are streamed and thus implicitly encodes the available locality. In many datasets, graphs are stored in a manner that preserves relatively high locality—for example, by writing out clustered regions together or by following traversal orders—so that neighbors often appear close to one another in the stream. However, this is not guaranteed: depending on how the data was generated, processed, or stored, the resulting order may also scatter neighbors and lead to substantially lower locality.

Throughout this thesis we use the term *natural ordering* to denote the original order in which a graph is stored and read from file. As argued above, this ordering typically exhibits relatively high locality, although exceptions may occur. To study robustness against poor locality, we additionally evaluate *random orderings*, obtained by applying independent random permutations to the vertex set. To ensure reliable results, we generate three random permutations per graph and report geometric means of cut size, peak memory usage and runtime.

CHAPTER 3

Related Work

The balanced graph partitioning problem is NP-complete for most objective functions [20], and no constant-factor approximation algorithm exists for general graphs [11]. Consequently, practical approaches rely on heuristics that trade optimality for scalability. The problem has been studied extensively over several decades, making it one of the most explored topics in graph algorithms; see [6, 12, 38] for comprehensive surveys.

Early heuristic approaches include the local improvement methods of Kernighan and Lin [26] and Fiduccia and Mattheyses [19], which iteratively exchange vertices to reduce the edge cut while maintaining balance. Later extensions introduced k-way local search [35] and more sophisticated exchange strategies.

The most successful family of algorithms for graph partitioning are *multilevel methods*. Hendrickson and Leland [22] formulated the paradigm in the form that is still widely used today: the graph is recursively coarsened to a smaller instance, an initial partition is computed on the coarsest level, and the solution is then refined during uncoarsening. The first widely used implementation was METIS [24], which established the approach as a standard tool for large-scale partitioning. Together with KaHIP [36], these methods represent the state of the art in offline graph partitioning. Both frameworks also provide parallel implementations (ParMetis [25] and ParHIP [30]), which extend their applicability to distributed-memory environments. They typically achieve very high partitioning quality, but require random access to the full graph in memory. While billion-scale networks can in principle be handled on sufficiently large machines, such resources are not always available. This has motivated the development of *streaming partitioning*, which aim to produce competitive partitions with a much smaller memory footprint.

This idea was first explored by Stanton and Kliot [39], who proposed simple one-pass heuristics. The most basic is *hashing*, which assigns each vertex uniformly at random to one of the k blocks, yielding poor cut quality but minimal runtime. A more sophisticated method is the *Linear Deterministic Greedy* (LDG) heuristic, which places a vertex v into the block V_i maximizing $|N(v) \cap V_i| \cdot \lambda(i)$ with $\lambda(i) = 1 - |V_i|/L_{\max}$, where $L_{\max}$ is the maximum block size. This combines attraction to blocks with many neighbors with a

multiplicative penalty that discourages imbalance.

Later, Tsourakakis et al. proposed FENNEL [41], a one-pass partitioning heuristic that can be viewed as an adaptation of the widely-known clustering objective modularity [10]. Fennel reformulates the placement decision with an additive penalty: a vertex is assigned to the block maximizing $|N(v) \cap V_i| - f(|V_i|)$, where $f(|V_i|) = \alpha\gamma\,|V_i|^{\gamma-1}$. Here γ is a hyperparameter and α is coupled to it via $\alpha = m\,\frac{k^{\gamma-1}}{n^{\gamma}}$ to normalize the penalty; with the recommended setting $\gamma = \frac{3}{2}$ this simplifies to $\alpha = m\frac{\sqrt{k}}{n^{3/2}}$.

Extensions of the one-pass model aim to compensate for the lack of global context. Nishimura and Ugander [31] proposed *restreaming* algorithms (ReLDG, ReFennel) that make multiple passes over the input, allowing iterative improvements. Awadelkarim and Ugander [2] extended this line with *prioritized restreaming algorithms*, where vertex order is dynamically adjusted based on priority scores rather than fixed in advance. Patwary et al. developed WStream [33], which employs a sliding window of a few hundred vertices to gain lookahead, although it was evaluated only on relatively small graphs (up to 300 000 vertices). Jafari et al. [23] embed LDG as a fast heuristic in a shared-memory multilevel partitioner, improving runtime but still requiring the full graph in memory. Hence it remains an offline method that only borrows streaming ideas. Overall, the online variants explored so far yield only modest improvements over the original one-pass heuristics and remain well short of the quality achieved by classical offline multilevel partitioners.

A more powerful buffered approach is HeiStream [18], which processes the graph in fixed-size batches. For each batch, it constructs a *model subgraph* that contains all vertices of the batch together with k *quotient nodes* representing the already assigned blocks. Edges between batch vertices are included directly, while edges from a batch vertex to previously assigned vertices are aggregated into weighted edges towards the corresponding quotient node.

The resulting subgraph is partitioned using a multilevel scheme: vertices are iteratively contracted to produce a hierarchy of smaller graphs, an initial partition is computed on the coarsest level, and the solution is then successively refined during uncoarsening. To guide this process, HeiStream generalizes the Fennel objective to weighted graphs. Specifically, a vertex v of weight $c(v)$ is assigned to block V_i that maximizes

$$\sum_{u \in N(v) \cap V_i} \omega(v,u) - c(v)\, f(c(V_i)), \text{ where } f(c(V_i)) = \alpha\gamma c(V_i)^{\gamma-1}.$$

This preserves the additivity of the objective across contraction levels and enforces balance constraints during refinement. The block assignments obtained on the model subgraph are then mapped back to the global partition of the original graph.

By batching, HeiStream trades memory for quality: collecting each batch together with aggregated adjacency information increases the memory footprint but reveals more local structure, which typically reduces the edge cut compared to Fennel. Consequently, the batch size serves as a direct memory–quality knob. In practice, HeiStream achieves near-linear running time, $O(n + m)$, with only weak dependence on k.

A variant, HeiStreamE, extends the same buffered multilevel approach to the related edge-partitioning problem [13].

Although HeiStream generally achieves high-quality partitions, it remains highly sensitive to the input order, since unfavorable orderings can obscure neighborhood information and degrade quality. This issue directly connects to research on *node reordering* [1, 16, 17, 28, 32], where the goal is to arrange nodes to improve memory locality and overall runtime efficiency. For example, Barik et al. [5] show that locality- and community-based reorderings can yield runtime and cache-latency improvements of up to 4× and 2.6×, respectively. Although such studies aim to accelerate graph algorithms in general rather than streaming partitioners, their insights suggest that better-ordered input streams could also enhance partition quality. However, sophisticated reordering schemes come with significant overhead [4], which limits their practicality in real-time streaming scenarios.

Hajidehi et al. introduced Cuttana [21], which improves robustness to input orderings through a two-phase design. In Phase 1, a *prioritized buffer* is maintained: vertices are temporarily stored in a bounded priority queue and ranked by a buffer score designed to avoid premature assignments. The score for a vertex v_t is

$$s(v_t) = \frac{|N(v_t)|}{D_{\max}} + \theta \cdot \frac{\sum_i |N(v_t) \cap V_i|}{|N(v_t)|} \ .$$

The first term normalizes the degree to favor high-degree vertices, while the second emphasizes the fraction of already assigned neighbors; larger θ prioritizes vertices for which placement is better informed. Vertices with $|N(v_t)| \geq D_{\max}$ are not buffered and are assigned a partition immediately using the Fennel-score. $D_{\max}$ bounds per-vertex neighbor storage in the buffer and leverages the power-law structure of real graphs: most edges have at least one low-degree endpoint, so buffering low-degree vertices reduces premature assignments while early placement of hubs provides "anchors" that guide later decisions. When the buffer reaches capacity, the highest-scored vertex is evicted and assigned using a Fennel-inspired score *augmented for edge-balance*, i.e., it selects $\arg\max_i \left(|N(v_t) \cap V_i| - \delta\big(|V_i| + \mu \sum_{x \in V_i} |N(x)|\big)\right)$, where δ is the Fennel penalty and μ rescales edge counts; this reduces extreme edge-load skew while maintaining balance. Afterwards, the scores of buffered neighbors are updated to reflect the new assignment.

In Phase 2, Cuttana applies a scalable refinement: vertices are grouped into sub-partitions, and coarse-grained trades between partitions are performed to reduce cut edges while maintaining balance. A parallel implementation further reduces the overhead of buffering and refinement. In their paper, the authors demonstrate that Cuttana achieves better partition quality than HeiStream under unfavorable input orderings, where locality between consecutive vertices in the stream is low.

In summary, one-pass heuristics such as LDG and Fennel scale well but offer limited quality. HeiStream improves cuts by applying multilevel partitioning to batches, yet remains sensitive to stream order. Cuttana reduces this sensitivity via prioritized buffer-

ing and coarse refinement, but at notable runtime and memory cost. We combine the strengths of both: multilevel batching with a lightweight prioritized buffer, a hub-aware buffer score that balances degree and assigned neighbor signals, and a parallel implementation to deliver scalable, memory-efficient, and order-robust partitions.

CHAPTER 4

Improving Processing Order in Streaming Graph Partitioning

Buffered streaming partitioners represent a middle ground between lightweight one-pass heuristics and memory-intensive offline methods, as they use moderate amounts of memory to achieve significantly better partition quality than pure streaming approaches while remaining more scalable than offline algorithms. Among them, HeiStream [18] achieves high partition quality by applying multilevel partitioning to buffered batches, yet its effectiveness is highly dependent on the order in which vertices appear in the stream. Cuttana [21], by contrast, reduces this sensitivity through prioritized buffering, albeit at additional overhead. Our goal is to integrate the complementary strengths of both approaches into a single algorithm that is scalable, memory-efficient, and robust to unfavorable input orderings.

At a high level, our method delays assignments for vertices with little structural context and prioritizes those with stronger connections to already partitioned regions. Concretely, incoming vertices are inserted into a bucket-based priority queue, inspired by the buffering strategy of Cuttana, where vertices are temporarily stored and prioritized to avoid premature assignments. Hereby, each vertex is ranked by a buffer score that reflects how well its structural context is already known. When the queue reaches capacity, the highest-scored vertices are accumulated into batches, which are then partitioned using the multilevel coarsening–refinement scheme of HeiStream. By prioritizing vertices with many connections to already assigned regions, the buffering step yields batches whose subgraphs exhibit higher internal locality. This provides the multilevel partitioner with more coherent structural cues during contraction and refinement. Compared to HeiStream, which partitions consecutive stream segments and thus degrades under randomized orders, our method achieves greater robustness by constructing batches that preserve locality even when the stream offers little of it.

The remainder of this chapter is organized as follows. Section 4.1 introduces the algorithm at a high level and presents pseudocode that outlines the overall processing flow.

Section 4.2 then details its core components: Section 4.2.1 discusses several buffer score variants that were evaluated as potential scoring functions. Section 4.2.2 justifies our choice of a bucket-based priority queue, which reduces the runtime overhead of frequent score updates compared to the set-based structure used in CUTTANA. Section 4.2.3 explains how multilevel partitioning is integrated and how subgraph construction is adapted to the reordered batches. Section 4.2.4 introduces an optional mechanism for incorporating *ghost neighbors* to enrich locality information. Section 4.2.5 describes the parallelized variant and its runtime–memory trade-offs. Finally, Section 4.2.6 extends the design to multiple passes, combining buffering in the first pass with refinement in later ones.

4.1 Algorithm Overview

The overall procedure is illustrated in two complementary ways: Algorithm 1 summarizes the processing flow in pseudocode form, while Figure 1 provides a schematic visualization of the same steps. The pseudocode captures the details of the implementation, whereas the figure emphasizes the conceptual structure of the algorithm. The algorithm processes the input stream one vertex at a time while maintaining a bounded buffer in the form of a bucket-based priority queue denoted as Q. Each streamed vertex is either inserted into Q or directly partitioned, depending on its degree. The queue enforces an adaptive processing order: vertices with higher buffer scores, which indicate stronger structural context, are prioritized for early assignment, while less informative vertices remain buffered until more of their neighbors are known.

Handling high-degree vertices. Vertices whose degree exceeds the threshold $D_{\max}$ are immediately assigned using the FENNEL [41] heuristic, without entering the buffer. This prevents the queue from being dominated by hubs and establishes early "anchors" that guide subsequent assignments for their neighbors. After assignment, the scores of their buffered neighbors are updated to reflect the new structural information.

Buffering and eviction. For vertices whose degree does not exceed the threshold $D_{\max}$, a buffer score is computed. This score is discretized using the parameter $discFactor$, and the resulting bucket index determines the position of v in the bucket queue. If Q exceeds its maximum capacity Λ, the highest-ranked vertex is evicted and added to a temporary batch container. From that moment, the vertex is treated as *virtually partitioned*: although its actual assignment occurs only when the batch is processed, it is guaranteed to be placed before or together with its buffered neighbors. Therefore, their scores can already be updated immediately upon eviction. This ensures that the queue consistently reflects the most informative vertices, so that each batch contains the best candidates available at the time.

Algorithm 1 Pseudocode of the overall processing flow of BuffCut.

Data: Stream S, buffer size Λ, batch size Δ, discretization factor ϕ, maximum degree $D_{\max}$
Result: Partition assignments

```
Q ← BucketQueue(discFactor)
batch_nodes ← []
foreach v, N(v) in S do
    if |N(v)| > D_max then
        partitionSingleNode(v, N(v))
        Q.updateNeighborScores(v, N(v))
        continue
    score ← calcBufferScore(v, N(v))
    Q.insert(v, N(v), score)
    if |Q| ≥ Λ then
        (u, N(u)) ← Q.popTopNode()
        Q.updateNeighborScores(u, N(u))
        batch_nodes.append({u, N(u)})
        if |batch_nodes| = δ then
            partitionWithMLP(batch_nodes)
            batch_nodes ← []
while |Q| > 0 do
    batch_nodes ← extractTopNodes(min(δ, |Q|))
    partitionWithMLP(batch_nodes)
    batch_nodes ← []
Function extractTopNodes(num_nodes)
    batch_nodes ← []
    while |batch_nodes| < num_nodes do
        (v, N(v)) ← Q.popTopNode()
        batch_nodes.append(v, N(v))
        updateNeighborScores(Q, v, N(v))
    return batch_nodes
Function updateNeighborScores(Q, v, N(v)):
    foreach neighbor u in N(v) do
        if Q.contains(u) then
            new_score ← calcBufferScore(u, N(u))
            Q.updateScore(u, new_score)
```

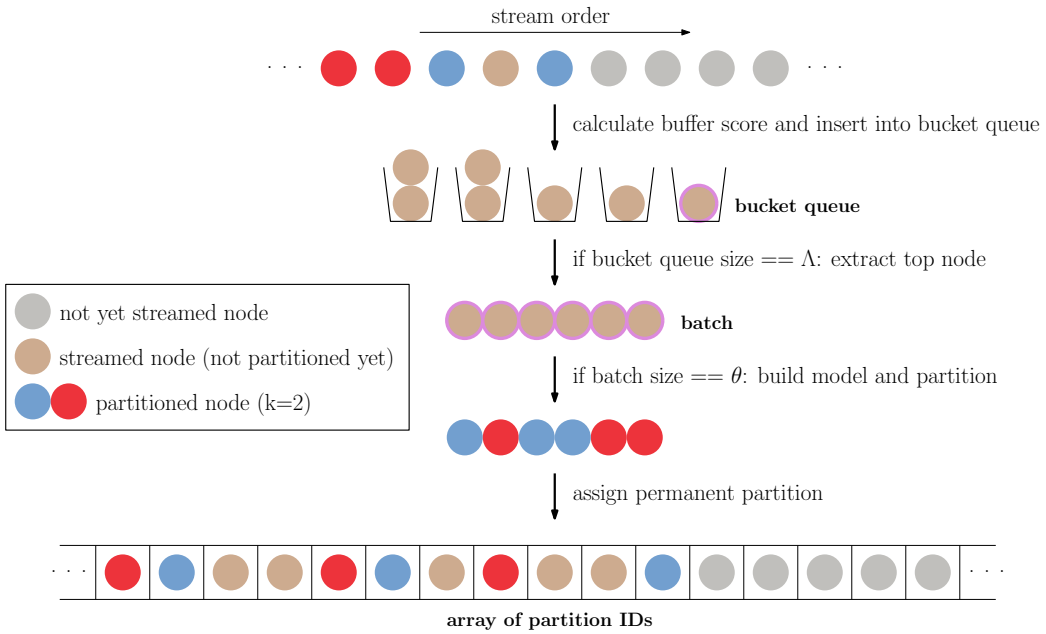


Figure 1: Schematic overview of the processing flow. Vertices arrive in stream order and are inserted into a bucket-based priority queue, according to their computed buffer score. High-degree vertices exceeding $D_{\max}$ are an exception: they are assigned immediately using the FENNEL score and never enter the queue (omitted here for clarity). Once the queue reaches capacity, the highest-ranked vertex (purple outline) is evicted into a batch container. When the batch size reaches θ, the corresponding subgraph is built and partitioned using multilevel partitioning, and the resulting assignments are made permanent. After the entire stream is processed, any remaining buffered vertices are partitioned in final batches.

Batch partitioning. Once the batch container accumulates δ vertices, it is handed off to the multilevel partitioning (MLP) procedure. This follows the HEISTREAM framework: the buffered vertices and their already partitioned neighbors form a compact subgraph, which is coarsened, initially partitioned at the coarsest level, and refined during uncoarsening. Results are mapped back to global IDs, the batch container is cleared, and the process continues. When the input stream ends, any vertices still in Q are partitioned in remaining batches using the same procedure.

4.2 Algorithmic Details

This section provides a closer look at the core components of our algorithm and their concrete realization. While the previous overview focused on the high-level workflow, we now turn to specific design choices and technical details that determine efficiency and quality in practice. We begin with buffer scores, which govern the prioritization of

vertices in the queue. Next, we discuss the bucket-based priority queue that replaces the set-based structure of CUTTANA, followed by the integration of multilevel partitioning as adopted from HEISTREAM. We then introduce the concept of ghost neighbors as an optional extension for increasing locality information. Next, we describe the parallelized version of the algorithm that improves runtime scalability and finally explain how we integrate restreaming.

4.2.1 Buffer Scores

A central component of our algorithm is the priority queue that organizes the buffered vertices. Its purpose is to mitigate the dependence on the arbitrary stream order by reordering vertices before they are processed batch-wise in the multilevel partitioning. This reordering is entirely determined by the buffer score: vertices are ranked by their score in the queue, and whenever eviction is required the vertex with the highest score is extracted and added to the batch. Consequently, the definition of the buffer score has a decisive impact on the composition of batches and the overall partitioning quality. In the following, we describe several scoring strategies that we have implemented and evaluated. Experimental evaluations of these scoring functions can be seen in Section 5.2.1, where we evaluate their impact on partition quality, memory usage and runtime.

Assigned Neighbors Ratio (ANR). The most basic scoring function considers the fraction of neighbors that are already assigned to some partition. For a vertex v_t with degree $d(v_t)$, this is defined as

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

This score increases as more of a vertex's neighborhood becomes placed. It therefore favors low-degree vertices, which can quickly accumulate a high ratio once a few neighbors are assigned. In contrast, high-degree vertices require a large portion of their neighbors to be placed before their score rises substantially. This behavior may be beneficial (by postponing hubs until more context is available), but can also prevent them from serving as early anchors in the partitioning process.

CUTTANA 's Buffer Score (CBS). The buffer score used in CUTTANA combines degree-based information with the fraction of neighbors that are already assigned. Formally, for a vertex v_t with neighborhood $N(v_t)$ and degree $d(v_t) = |N(v_t)|$, the score is defined as

$$\text{CBS}(v_t) \;=\; \frac{d(v_t)}{D_{\max}} \;+\; \theta \cdot \frac{\sum_{i=1}^{k} |N(v_t) \cap V_i|}{d(v_t)}.$$

Here, $D_{\max}$ denotes a degree cap applied to the buffer, and θ is a hyperparameter that balances the influence of the assigned-neighbor term. Following the original implementation, θ is set to 2.

The rationale behind this design is to avoid premature assignments, especially of low-degree vertices, which would otherwise be placed early with little context. By integrating the normalized degree term $\frac{d(v_t)}{D_{\max}}$, high-degree "hub" vertices are given higher priority, as they provide stronger guidance for the assignment of their many neighbors once placed. At the same time, the assigned-neighbor ratio rewards vertices that already have a substantial fraction of their neighborhood placed, ensuring that local connectivity is exploited. The parameter θ controls the trade-off: larger values emphasize neighborhood conformity, whereas smaller values increase the relative importance of degree. Overall, this score has been shown to yield good performance in practice by balancing hub-driven anchoring and neighborhood-based consistency.

Hub-Aware Assigned Neighbors Ratio (HAA). Motivated by the idea in CBS to combine assigned-neighbor ratio (ANR) with a degree term, we introduce a new parametric buffer score that refines this balance. The guiding principle is that the importance of ANR should decrease with vertex degree: for hubs, the neighborhood is almost never sufficiently placed to provide a reliable ANR signal during streaming, so they should be prioritized mainly by degree. Conversely, low-degree vertices can be placed consistently with their neighborhood once only a few neighbors are assigned, so their score should be dominated by ANR. This design yields a more systematic trade-off between the two factors and allows us to control the relative influence through a simple parameterization.

Formally, let

$$\rho(v_t) = \frac{d(v_t)}{D_{\max}}$$

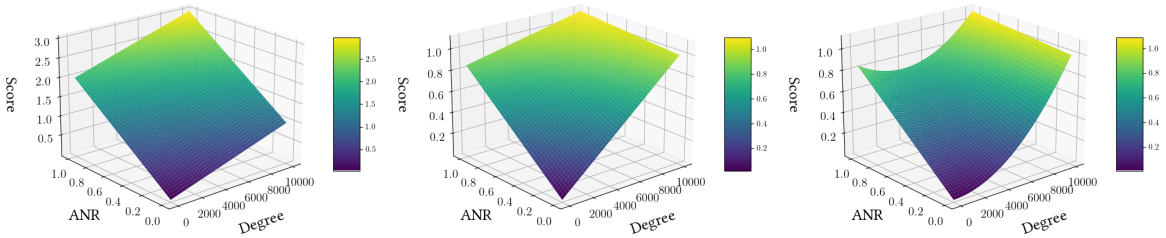
be the *degree ratio* of a vertex. The score is then defined as

$$\mathrm{HAA}(v_t) \;=\; \rho(v_t)^{\beta} \;+\; \theta \cdot \big(1 - \rho(v_t)\big) \cdot \mathrm{ANR}(v_t),$$

where $\beta \geq 1$ controls the shape of the degree contribution and θ balances the role of ANR. Larger values of β reduce the relative influence of degree for low-degree vertices, so that their score depends more strongly on ANR, while hubs with large degree are prioritized almost independently of ANR.

Compared to CBS, HAA changes the balance between degree and ANR more systematically. In CBS, ANR remains influential even for high-degree vertices, with degree added only linearly. HAA instead suppresses the ANR component as degree grows: hubs are scored almost purely by degree and can act as anchors early without occupying buffer space for long, while low-degree vertices are governed almost entirely by ANR. As illustrated in Figure 2a–2c, this yields markedly different score surfaces: ANR dominates only at the low-degree end, whereas hubs are prioritized almost independently of it.

The effect of β and θ is studied in detail in Section 5.2.1, where we evaluate different parameter choices and show that, after tuning, HAA achieves the best overall performance among the tested buffer scores.



(a) CBS ($\theta = 2$) **(b)** HAA ($\theta = 0.75$, $\beta = 1$) **(c)** HAA ($\theta = 0.75$, $\beta = 2$)

Figure 2: 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. For CBS (a), the assigned-neighbor ratio dominates the score, while degree contributes only as an additive linear term. In contrast, HAA (b,c) assign higher values to high-degree vertices even when their ANR is low. The variant shown in (b) has a linear dependence on the degree, whereas the quadratic variant (c) reduces the relative influence of degree for low-degree vertices, making their scores depend more strongly on ANR.

Neighborhood Seen Score (NSS). This score extends the idea of ANR by also considering neighbors that are currently buffered, in addition to those already assigned. Formally, for $\eta \in [0, 1]$ we define

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

where $V_1, \ldots, V_k$ are the current partitions, Q is the buffer, and η controls the weight of buffered neighbors. Setting $\eta = 0$ reduces to ANR, while $\eta = 1$ counts buffered and assigned neighbors equally.

The intuition is that vertices with many "visible" neighbors (already assigned or present in the buffer) might be better candidates for early processing. In practice, however, buffered neighbors provide only weak evidence: their presence in the buffer does not imply that they will be assigned in the same batch. As a result, the information added by the buffer term is often noisy and of limited value. Our experiments showed that neither the unweighted variant ($\eta = 1$) nor weighted versions with reduced buffer influence ($\eta < 1$) yielded improvements compared to plain ANR.

NSS extended with Degree. We also tested a degree-extended variant that augments NSS with an additional degree term, analogous to CBS. However, this variant suffered from the same limitation as NSS: buffered neighbors provide little reliable information, and the degree contribution did not compensate for this. Accordingly, this variant did not achieve improvements over ANR either.

Community-Majority Score (CMS). This score prioritizes vertices according to the dominant partition among their already assigned neighbors:

$$\text{CMS}(v_t) = \max_{p \in \{1,\dots,k\}} \frac{|\{u \in N(v_t) : \text{block}(u) = p\}|}{d(v_t)}.$$

The intuition is that assigning a vertex to the same partition as most of its placed neighbors may strengthen local community structure. In practice, however, this heuristic performed poorly: it overemphasizes local majority effects, leading to premature and unbalanced assignments, and consistently yielded lower partitioning quality than simpler scores such as ANR.

.

4.2.2 Bucket Priority Queue

In our setting, buffer scores are recalculated frequently: whenever a vertex is extracted from the PQ, the scores of all its buffered neighbors are updated. Since these scores only increase over time, *increase-key* operations dominate the workload.

CUTTANA uses a `std::set`-based priority queue, which maintains a strict total order of all elements. While extraction of the top element is $O(1)$, insertions and deletions are $O(\log n)$. An increase-key operation is implemented as a deletion followed by an insertion, and thus also costs $O(\log n)$. With the high frequency of score updates in our algorithm, these $\log n$ operations accumulate and become a runtime bottleneck.

Because of this, we opted for a more efficient structure: a bucket priority queue that discretizes buffer scores into $discFactor$ integer buckets. This coarser ordering allows all relevant operations—insert, delete, and increase-key—to be performed in $O(1)$ amortized time. In practice, we found that partition quality remains essentially unchanged compared to the set-based PQ used in CUTTANA, while runtime decreases substantially.

Discretization. In our implementation, our default buffer score corresponds to the HAA score (see Section 4.2.1). This score takes values in the range $[0, \max\{1, \theta\}]$, where θ is the weight parameter for the assigned-neighbor ratio component. To enable efficient priority handling, we discretize $s(v)$ into $discFactor$ integer buckets via

$$\text{idx}(v) = \lfloor s(v) \cdot discFactor \rfloor.$$

Vertices are placed in bucket $\text{idx}(v)$, and a *top pointer* keeps track of the highest non-empty bucket, ensuring that re-bucketing and extraction remain efficient.

In practice, we observed consistent trends as $discFactor$ increases. Partition quality improves steadily, with most of the benefit already achieved at values around $1\,000$. Runtime, on the other hand, remains low up to this point but grows substantially for larger values (up to an order of magnitude at $100\,000$). Memory usage decreases with increasing $discFactor$, which we attribute to more balanced bucket utilization that avoids large

underutilized allocations when only few buckets are used. However, the reduction in memory does not justify the dramatic runtime overhead of very large $discFactor$. We therefore fix $discFactor = 1\,000$ as the default, since it offers near-optimal cut quality with good runtime performance and acceptable memory consumption.

Degree Threshold $D_{\max}$. Following CUTTANA, we treat very high-degree vertices as *hubs* that are placed immediately without buffering: if $d(v) \geq D_{\max}$, the vertex bypasses the buffer and is assigned in one pass using the Fennel score. The main motivation is efficiency: hubs contribute disproportionately to resource usage in several ways. They occupy substantial buffer memory, they enlarge the induced subgraphs passed to multilevel refinement, and they trigger numerous score updates because of their many neighbors. Capping their eligibility therefore avoids excessive overhead while still preserving the essential effect of buffering for the vast majority of vertices. Assigning hubs early is thus primarily a memory– and update-efficiency trade-off, with the additional side effect that they can act as stable anchors for their neighborhoods.

Our experiments indicate that $D_{\max} = 10\,000$ strikes a good balance: it keeps essentially all low- and medium-degree vertices in the buffer, achieves consistent quality improvements over smaller thresholds, and keeps overhead within budget. On very large graphs, higher thresholds could be considered if resources allow, but in practice it often seems to be more beneficial to increase the overall buffer size Λ rather than further raising $D_{\max}$. We therefore adopt $D_{\max} = 10\,000$ as our default.

4.2.3 Integration of Multilevel Partitioning

HEISTREAM combines streaming partitioning with a multilevel refinement step applied to buffered batches, resulting in higher partition quality compared to purely local heuristics such as FENNEL. We retain this approach and apply MLP to batches of vertices extracted from our bucket priority queue. To achieve this, we maintain a vector for the current batch. Whenever eviction is required, the vertex with the highest buffer score is extracted from the queue and appended to this vector along with its adjacency information. Once the vector reaches the specified batch size δ, we assemble the subgraph induced by these vertices and apply the unmodified MLP pipeline from HEISTREAM (coarsening, initial partitioning, refinement). After the MLP completes, assignments are written back to global IDs and current batch vector is cleared for the next iteration.

In HEISTREAM, vertices are processed strictly in stream order, meaning the global vertex IDs are contiguous and directly aligned with their stream positions. This makes it trivial to (i) determine whether a vertex belongs to the current batch, and (ii) construct the mapping between global and local node IDs in the subgraph: local IDs are simply offsets from the batch's first global ID. In our case, the priority queue changes the processing order, breaking this direct correspondence. This introduces two challenges: first, we can no longer derive batch membership from stream position, and second, explicit mappings are required to translate between global and local IDs.

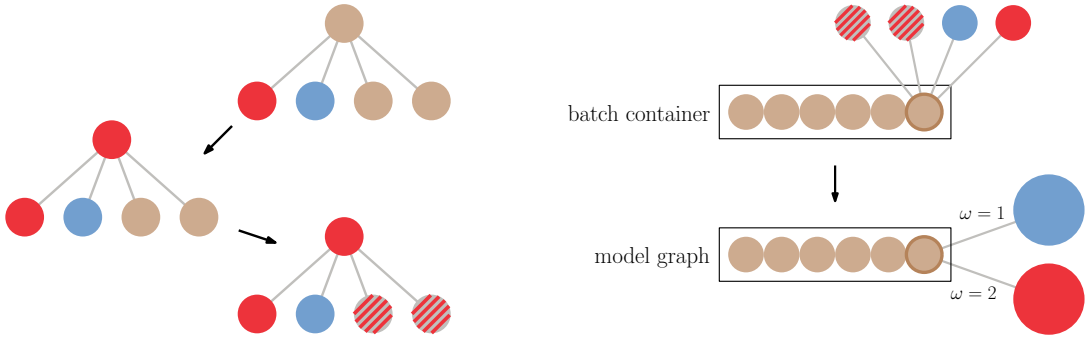
To address the first challenge, we reuse the already existing array that stores the final partition ID of each vertex. This array has size n and is initialized at the beginning of the algorithm to store the final partition ID of each vertex. Since it is guaranteed to be available throughout, it provides a natural choice for marking intermediate states as well. Concretely, when a vertex is inserted into the current batch, we temporarily store the batch ID in this array at its index. During subgraph construction, membership can then be tested in $O(1)$ time by checking whether the stored value matches the active batch ID. This marking also serves a second purpose: vertices can already be treated as virtually partitioned for the computation of buffer scores, even though their actual assignment is only determined after the MLP step. Once the batch has been processed, the temporary IDs are simply overwritten with the final partition IDs, so that consistency is maintained without additional data structures.

The second challenge, global–local translation, is solved by constructing explicit mapping arrays during subgraph assembly. The global-to-local map has size n and is used only temporarily to assign consecutive local IDs within the subgraph, while the local-to-global map has size δ and is retained until the MLP results are written back.

4.2.4 Ghost Neighbors

In HeiStream, unpartitioned neighbors are ignored when constructing the subgraph for multilevel partitioning. This neglects potentially valuable locality information: even though such neighbors are not yet placed, parts of their neighborhoods may already contain permanently assigned vertices, which indicates a structural tendency that could be exploited.

To capture this signal, we introduce the concept of *ghost neighbors*. Whenever a vertex is permanently assigned to a partition, each of its unpartitioned neighbors temporarily inherits the same partition ID. If a vertex has multiple assigned neighbors, its temporary assignment is always updated to the partition of the most recently assigned neighbor. In this way, unpartitioned vertices carry additional context that can strengthen locality information during both buffer scoring and multilevel partitioning. This information is stored in the same data structure that holds final partition IDs: for k partitions, IDs in $[0, k-1]$ denote permanent assignments, while IDs in $[k, 2k-1]$ represent temporary ones. This encoding enables efficient distinction without additional structures.



(a) Ghost partition assignment. **(b)** Ghost neighbors in the model graph.

Figure 3: Illustration of ghost neighbors. Colors: blue/red = permanently assigned partitions, beige = unassigned, striped = ghost-assigned. (a) Whenever a vertex is permanently assigned a partition, its unpartitioned neighbors inherit the same temporary ghost partition. (b) In the model graph, batch vertices are connected to artificial quotient nodes (large blue/red), which represent already assigned partitions by contracting all edges to vertices in that partition. Edges to permanent neighbors contribute with full weight ($\omega = 1$), while ghost neighbors contribute with reduced weight (for simplicity $\omega_{ghost} = 0.5$), here resulting in total edge weights of $\omega = 1$ (blue) and $\omega = 2$ (red).

Ghost neighbors affect three parts of the algorithm: the direct partitioning using the FENNEL score for high degree nodes exceeding $D_{\max}$, the buffer score computation and subgraph construction. Since their information is only provisional, their contribution should be discounted compared to permanently assigned neighbors. We therefore introduce a weighting parameter $w_{\text{ghost}} \in [0, 1]$ that regulates the relative impact of ghost neighbors. Low values treat them as weak evidence, while higher values increase their influence, at the risk of overemphasizing predictions that may later be contradicted. This same parameter is consistently applied in both buffer scoring and subgraph construction, ensuring a unified control over the role of ghost information. Figure 3 illustrates the mechanism, showing how ghost partitions are assigned to neighbors and how they later contribute in the model graph construction.

Buffer Score Adaptation. The idea behind integrating ghost neighbors into the buffer score is conceptually simple: not only permanently assigned neighbors contribute to the assigned-neighbor ratio component, but also ghost neighbors, scaled by w_{ghost}. Formally, ghost neighbors extend the hub-aware assigned neighbors ratio (HAA) score (see Section 4.2.1) by enlarging the set of considered neighbors:

$$\text{HAA}(v_t) \;=\; \rho(v_t)^{\beta} \;+\; \theta \cdot \big(1 - \rho(v_t)\big) \cdot \frac{\sum_{i=1}^{k} |N(v_t) \cap V_i| + w_{\text{ghost}} \cdot \sum_{i=1}^{k} |N(v_t) \cap G_i|}{d(v_t)}$$

where $V_1, \ldots, V_k$ denote the sets of permanently assigned vertices in each partition, and $G_1, \ldots, G_k$ the sets of ghost-assigned vertices. The parameter $w_{\text{ghost}} \in [0, 1]$ regulates the additional contribution of ghost neighbors. Setting $w_{\text{ghost}} = 0$ reduces the score to plain HAA, while larger values incorporate more information from ghost neighbors.

Fennel adaptation. For high-degree vertices whose degree exceeds $D_{\max}$ and which therefore skip the buffer, partitioning is performed directly using the Fennel score. To incorporate ghost neighbors consistently, we extend the original objective by adding their discounted contribution. Formally, a vertex v is assigned to the block i that maximizes $|N(v) \cap V_i| + w_{\text{ghost}} \cdot |N(v) \cap G_i| - f(|V_i|)$, where V_i denotes the set of permanently assigned vertices in partition i, G_i the set of ghost-assigned vertices, and $f(|V_i|) = \alpha\gamma|V_i|^{\gamma-1}$ the penalty term with $\alpha = m\frac{\sqrt{k}}{n^{3/2}}$ for $\gamma = \frac{3}{2}$.

Subgraph Construction and Implementation. Ghost neighbors also influence the multilevel partitioning step. When assembling the batch subgraph, edges to already assigned neighbors are aggregated into the quotient nodes that represent connections to the external graph. To incorporate ghost information consistently, we use an integer-based weighting scheme that links buffer scoring and subgraph construction. Specifically, edges to permanently assigned neighbors contribute with a configurable weight parameter $w_{\text{non-ghost}} \in \mathbb{N}$, while edges to ghost neighbors contribute with weight 1. Hence, the relative impact of ghost edges is implicitly determined by the ratio $1/w_{\text{non-ghost}}$, ensuring a unified and efficient scaling without floating-point arithmetic. This ratio is applied consistently in both buffer scoring and subgraph construction, so that ghost contributions are always derived directly from the chosen non-ghost weight. We evaluate this mechanism in Section 5.2.6, where we systematically vary $w_{\text{non-ghost}}$ to study its effect on partition quality and identify a suitable default value.

4.2.5 Parallelization

To improve runtime performance and scalability, we implemented a parallelized version of our buffered streaming graph partitioning algorithm. This design follows a multi-threaded streaming architecture, where input processing, buffer management, and partitioning are decoupled into three dedicated threads that operate concurrently. Thread 1 (*IO Reader*) is responsible for reading the input graph line-by-line, parsing each node together with its adjacency list, and inserting the resulting $ParsedLine$ objects into a bounded $input\_queue$. Thread 2 (*Priority Queue Handler*) consumes entries from the $input\_queue$, computes the buffer score for each node, and maintains the priority buffer. Depending on node degree and buffer state, it generates either single-node or batch partitioning tasks, which are encapsulated as $PartitionTask$ objects and pushed into the $partition\_task\_queue$. Finally, Thread 3 (*Partitioning Worker*) processes these tasks, applying either direct single-node assignments or multilevel batch partitioning.

Initially, communication between threads used mutex-protected queues with condition variables. To reduce contention and improve throughput, these were replaced by a lock-free queue implementation (`moodycamel::ReaderWriterQueue`), which supports single-producer/single-consumer access without locking overhead. This change removed most synchronization bottlenecks and increased concurrency, leading to a measurable speedup in the partitioning pipeline.

Compared to the sequential version, the parallelized design requires slightly more memory, as multiple stages of the pipeline operate in parallel and hold intermediate data simultaneously. This overhead is modest and outweighed by the substantial runtime improvements, representing an acceptable trade-off between memory usage and processing speed.

4.2.6 Restreaming

In addition to the standard one-pass setting, our implementation also supports restreaming. The number of passes can be specified by a parameter. In the first pass, the algorithm behaves exactly as described in the previous sections: vertices are inserted into the bucket priority queue, scores are updated, and batches are constructed and partitioned using the multilevel scheme.

In subsequent passes, the priority queue and buffer score lose their purpose. With all vertices already assigned to partitions from the previous pass, prioritization based on neighbor information is no longer meaningful. Consequently, in later passes we omit the buffering step entirely and revert to plain HeiStream-style batching. Vertices are simply collected sequentially into batches of the specified size and directly partitioned using the multilevel procedure, with the partition from the previous pass serving as the initial solution.

As in HeiStream, restreaming tends to increase memory usage. The reason is that in later passes every vertex already has a partition assignment, so all of its neighbors contribute to the partition context. This results in larger model graphs during subgraph construction and therefore higher memory consumption. High-degree vertices whose degree exceeds the threshold $D_{\max}$ are by default excluded from the refinement batches, consistent with the handling in the first streaming pass. For these vertices, the partition assignment obtained in the previous pass is kept unchanged. This avoids batches being dominated by extremely large neighborhoods that would otherwise inflate the subgraph size significantly. An optional parameter lets you include those vertices in the refinement, trading off the gains from revisiting low-degree nodes—which benefit most from extra passes—against the overhead of processing hubs.

Overall, restreaming integrates naturally with our algorithm: the first pass leverages buffering and prioritization to reduce premature assignments, while subsequent passes focus purely on refinement in the sense of HeiStream. This combination allows us to capture both the robustness benefits of buffering and the quality improvements of iterative refinement.

CHAPTER 5

Experimental Evaluation

The purpose of this chapter is to validate the design choices of our algorithm BuffCut and to assess its performance in practice. We first conduct controlled experiments to study the influence of central hyperparameters such as buffer score, buffer size Λ, and batch size δ, as well as the trade-offs introduced by parallelization. These studies provide insights into how different components of the algorithm affect partition quality, runtime, and memory usage.

Subsequently, we benchmark BuffCut against state of the art streaming partitioners, namely HeiStream and Cuttana, under both natural and randomly permuted graph orderings. Natural orderings, as they occur in the original graph files, often exhibit a relatively high degree of locality: vertices that are close in the graph are frequently stored consecutively due to dataset construction or storage conventions. To disentangle algorithmic performance from such favorable input bias, we additionally evaluate under random orderings, which deliberately disrupt this structure. Our primary focus is on the random case, since HeiStream already performs very strongly on natural orderings, whereas its effectiveness degrades under weaker locality conditions. The experiments demonstrate that our method can alleviate this weakness and improve stability across different orderings.

The chapter is structured as follows. Section 5.1 introduces the experimental setup, including hardware environments, datasets, and evaluation metrics. Section 5.2 presents parameter studies that isolate the effect of individual hyperparameters. Finally, Section 5.3 reports results of the comparative evaluation against existing algorithms.

5.1 Experimental Setup

Hardware and Software Environment. All experiments were conducted on a dedicated machine equipped with an AMD EPYC 9754 CPU (128 cores, 256 threads, base frequency 2.25 GHz), 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. To ensure that measured memory consumption reflects only the algorithmic behavior, all input graphs are streamed directly from disk rather than being preloaded into main memory.

Our implementation builds upon the KaHIP graph partitioning framework, integrating and extending the HeiStream algorithm. The code base is written in C++ and compiled with g++ 9.4.0 using the `-O3` optimization flag.

All experiments are executed using `GNU Parallel` [40]. Unless stated otherwise, we run 16 concurrent instances to reduce overall test time. For the state of the art comparisons on the *Test Set*, we restrict this to 5 concurrent instances since the extremely large graphs would otherwise exceed the available main memory when too many runs are executed in parallel.

We implemented two variants of our algorithm: a sequential version and a parallelized version with three dedicated threads for I/O reading, priority queue management, and partitioning (see Section 4.2.5). By default, all experiments use the parallelized version, since it is more efficient. The sequential variant is only included in the dedicated evaluation of parallelization effects (see Section 5.2.5).

Datasets. We evaluate our algorithm on a diverse set of real-world and synthetic benchmark graphs drawn from well-established benchmark collections [3, 7, 8, 9, 14, 29, 34, 37]. To structure the evaluation, we group these graphs into two datasets. The *Exploration & Tuning Set* contains medium-to-large instances that are small enough to allow systematic parameter studies and sensitivity analyses. The *Test Set* consists of much larger graphs, which serve to evaluate scalability and to benchmark against state of the art partitioners in practically relevant scenarios. Together, these graphs cover a wide range of sizes, densities, and structural characteristics. Table 1 lists all graphs used in our experiments, including their number of vertices n, edges m, and type. Detailed sources per category are given below.

All *social networks* (`orkut`, `twitter-2010`, `com-Friendster`, `com-LJ`, `Ljournal-2008`, `soc-flixster`, `soc-lastfm`) were obtained from SNAP [29]; `in-2004`, `uk-2007-05` and `coPapersDBLP` stem from the 10th DIMACS Implementation Challenge benchmark set [3]. *Meshes, circuits, and matrices* (`Flan_1565`, `Bump_2911`, `FullChip`, `G3_Circuit`, `nlpkkt240`) are taken from the SuiteSparse Matrix Collection [14]. While `it-2004`, `arabic-2005`, `sk-2005`, `webbase-2001`, and `uk-2002` originate from the Laboratory for Web Algorithmics (LAW) [7, 9, 8]. `rgg26`, `rhg1B`, `rhg2B` are *synthetic graphs* of type *Random Geometric Graph* and were generated with the KaGen framework [37]. The remaining graphs (`cit-Patents`, `italy-osm`, `great-britain-osm`) are obtained from the Network Repository [34]. All instances are preprocessed into the METIS [24] graph file format: we remove self-loops and parallel edges, ignore directions, and assign unit weights to all vertices and edges.

For each graph we consider two vertex orderings. Every instance is available both in

Graph	n	m	Type	Graph	n	m	Type
Exploration & Tuning Set				Test Set			
coPapersDBLP	540 486	15 245 729	Citation	orkut	3 072 411	117 185 082	Social
soc-lastfm	1 191 805	4 519 330	Social	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	Matrix
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	Circuit	twitter-2010	41 652 230	1 202 513 046	Social
soc-flixster	2 523 386	7 918 801	Social	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	Social
FullChip	2 986 999	11 817 567	Circuit	rgg26	67 108 864	574 553 645	Gen
cit-Patents	3 774 768	16 518 947	Citation	rhg1B	100 000 000	1 000 913 106	Gen
com-LJ	3 997 962	34 681 189	Social	rhg2B	100 000 000	1 999 544 833	Gen
Ljournal-2008	5 363 186	49 514 271	Social	uk-2007-05	105 896 555	3 301 876 564	Web
italy-osm	6 686 493	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				

Table 1: Benchmark graphs used in our experiments. The *Exploration & Tuning Set* consists of medium-to-large graphs that are still small enough to allow systematic parameter studies. The *Test Set* contains much larger graphs, designed to evaluate scalability and to compare against state of the art partitioners.

its *natural ordering*—the order in which vertices appear in the original input file—and in a *random ordering*, obtained by applying a uniform random permutation to the vertex IDs. To avoid biases from individual permutations, we generate three such permutations per graph and report results as the geometric mean across them.

In most graphs, the natural ordering exhibits relatively high locality, since graphs are often stored such that structurally related vertices appear consecutively. To quantify this effect, we use the *Neighbor to Neighbor Average ID Distance (AID)* [16] introduced in Section 2.2. For each set of graphs, we aggregate per-graph $AID(G)$ values by reporting their geometric mean. Note that lower AID values indicate higher locality, as neighbors tend to appear closer together in the vertex ordering.

For the *Exploration & Tuning Set* the geometric mean AID is 37 794 under the natural ordering and 219 685 under random permutations. The effect is even stronger on the *Test Set*: the geometric mean AID is 51 104 for natural orderings but rises to 2 581 859 for random orderings. In other words, natural orderings exhibit substantially higher locality than the random permutations used in our experiments. This gap tends to grow with graph size and helps explain why HeiStream performs well on naturally ordered inputs, while its quality degrades significantly under graphs that exhibit low stream locality.

We focus on the random case, as it robustly tests performance independent of input order, while the natural ordering serves as a reference to show how much structural context can be exploited when it is present.

Experimental Metrics. We enforce a maximum imbalance of $\epsilon = 3\%$, i.e., each block is limited to $L_{\max} = \lceil \frac{1.03\,n}{k} \rceil$ vertices. All experiments are performed for $k \in \{4, 8, 16, 32, 64, 128, 256\}$ (i.e., $k = 2^2, \ldots, 2^8$), covering the range of partition counts commonly used in practice.

Partitioning quality is defined in terms of the *edge cut*, i.e. the number of edges crossing between partitions. Throughout this work, however, we report the normalized *cut edges ratio* (in %), obtained by dividing the edge cut by the total number of edges m in the graph. This relative measure provides more interpretable results and allows for easier comparison across graphs of different sizes. In addition, we record the *runtime*, measured as the total time from start to completion of the partitioning process, and the *peak memory usage*, reported as resident set size in *MB* or *GB*. While cut quality remains the central quality criterion, it is always interpreted in conjunction with runtime and memory consumption, since the overarching objective is to achieve high-quality partitions with as little resource usage as possible. Since memory consumption can be explicitly controlled through algorithmic parameters such as buffer size, we ensure that comparisons with other algorithms are conducted under comparable memory budgets.

Performance Profiles. For aggregated comparisons across datasets and k values, we employ *performance profiles* [15]. Given a set of algorithms $\mathcal{A}$ and problem instances $\mathcal{P}$, a performance profile plots, for each factor $\tau \geq 1$, the fraction of problems on which an algorithm performs within a factor τ of the best known result. This approach allows a concise visual summary of trade-offs between different algorithms across multiple metrics.

Notation. Throughout this chapter we use the suffix "k" to denote a binary kilo—meaning $X\text{k} := X \cdot 1024$, so that, for example, $64\text{k} = 65{,}536$—and the suffix "M" to denote a decimal million, $1\text{M} := 1{,}000{,}000$.

5.2 Parameter Studies

In the following, we examine how our algorithm behaves under variations of its key hyperparameters and highlight the trade-offs they induce. We focus on five key factors: the buffer score, the buffer size Λ, the batch size δ used by the multilevel partitioner, the effect of parallelization, and the use of ghost neighbors. These choices govern the trade-off between partition quality, runtime, and memory consumption.

All parameter studies in Section 5.2 are carried out on the *Exploration & Tuning Set* listed in Table 1, a representative collection of 14 graphs with diverse sizes and structures. Unless stated otherwise, each experiment is evaluated on both the natural vertex ordering and on randomized orderings (three independent permutations); results for randomized runs are reported as geometric means to reduce variance. The only exception is one experiment studying ghost neighbors (see Section 5.2.6), which was executed on the larger *Test Set* to validate behavior at scale.

To ensure comparability, we follow a controlled approach: at any point, exactly one parameter is varied while all others remain fixed to default values. Unless noted otherwise, the default configuration is set to buffer score = HAA ($\beta = 2$, $\theta = 0.75$), buffer size $\Lambda = 128\text{k} = 131\,072$, batch size $\delta = 16\text{k} = 16\,384$, $discFactor = 1\,000$, and $D_{\max} = 10\,000$. These defaults were selected based on exploratory experiments and yield robust baseline performance across a wide range of graph instances. Some earlier experiments used slightly different HAA parameters (e.g., $\beta = 1.5$), which causes small numeric variations between tables but does not affect the conclusions.

5.2.1 Buffer Scores

Since the buffer score governs the eviction policy of the priority queue, it effectively determines the entire behavior of the buffering mechanism. The choice of score decides which vertices leave the buffer early and thus directly shapes the batches passed to the multilevel partitioner. Hence, the definition of the buffer score has a decisive influence on both partitioning quality and robustness to input orderings.

In the following, we build upon the scoring functions introduced in Section 4.2.1 and evaluate their performance experimentally. Since our algorithm is designed primarily for scenarios with weak locality (as in our random orderings), we focus our analysis mainly on this case. For completeness, we also report results on natural orderings, which confirm that the relative behavior of the scores is consistent there as well. We first provide a comparative overview of all tested scores, including weaker variants, to obtain a complete picture. Based on this comparison, we then focus on the most promising score—our Hub-Aware Assigned Neighbors Ratio (HAA)—and study its parameter sensitivity in detail.

Ordering	Score	Runtime (s)	Memory (MB)	Cut Edges (%)
Random	ANR	**7.30**	121.0	24.13
	CBS	7.61	117.1	24.67
	HAA	7.32	120.9	**24.02**
	CMS	8.43	**112.1**	30.92
	NSS	12.28	114.5	29.30
Natural	ANR	**6.20**	215.6	9.05
	CBS	6.36	208.8	8.98
	HAA	**6.20**	211.7	**8.97**
	CMS	6.86	239.7	11.38
	NSS	20.30	**152.2**	12.66

Table 2: Geometric mean results for different buffer scores under random and natural orderings. For HAA we use $(\beta = 2, \theta = 0.75)$ and for CBS the standard $\theta = 2$. Best result per metric and ordering in bold.

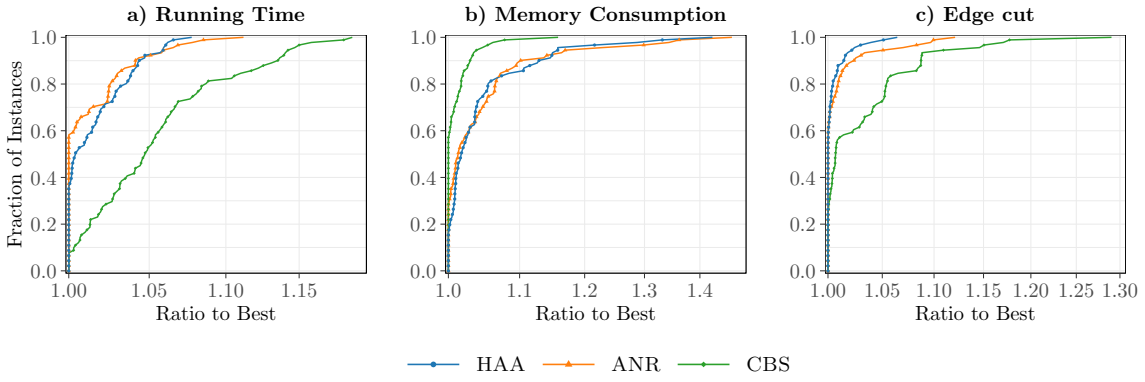


Figure 4: Performance profiles comparing the strongest buffer score competitors: HAA, ANR, and CBS. For HAA we use $(\beta = 2, \theta = 0.75)$ and for CBS $\theta = 2$.

Comparison of candidate scores. Table 2 reports geometric mean results across the *Exploration & Tuning Set* for all tested buffer scores. Overall, on random orderings the differences in cut ratios are not dramatic, yet several tendencies stand out clearly.

First, two variants can be dismissed right away: CMS and NSS perform significantly worse in terms of cut quality and are therefore not considered further. In addition, NSS exhibits substantially higher runtime due to the lack of optimizations for score updates in our prototype implementation. Since neither variant improved partition quality, further engineering effort did not appear justified.

Among the competitive designs, HAA achieves the lowest cut ratio overall. ANR already performs strongly, but HAA still improves by about 0.5% compared to ANR and by roughly 2.5% relative to CBS. On natural orderings, the three competitive scores (ANR, CBS, HAA) perform almost identically, with HAA again yielding the lowest cut. The difference to CBS is marginal, confirming that all competitive scores perform nearly identically when locality is strong.

To further assess robustness, Figure 4 shows performance profiles of the three competitive scores. Here, the advantage of HAA becomes even clearer: while absolute differences are small, HAA consistently avoids unfavorable cases. It never exceeds a factor of about 1.05 over the best solution on any instance, whereas ANR can deviate by up to 1.1 and CBS by as much as 1.25. In terms of runtime and memory, both HAA and ANR behave almost identically, while CBS uses slightly less memory but suffers from higher runtime. Overall, although the absolute differences between the competitive scores are not large, HAA achieves the lowest average cut and delivers the most consistent performance across instances. We therefore adopt HAA with $\beta = 2$ and $\theta = 0.75$ as the default buffer score in the following experiments.

Parameter sensitivity of HAA. Having identified HAA as the most promising buffer score, we now analyze its sensitivity to the parameters β (degree exponent) and θ (ANR

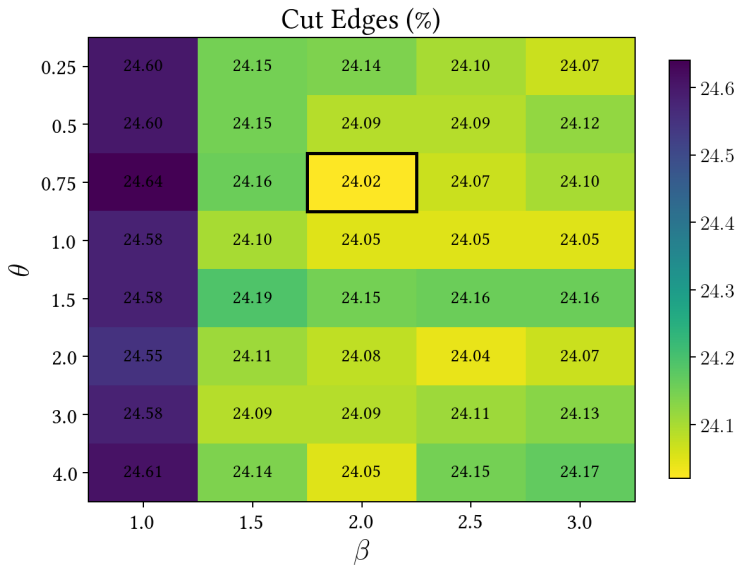


Figure 5: Geometric mean cut edges ratio (in %) for HAA as a function of its parameters β (degree exponent) and θ (weight of assigned neighbors ratio). Lighter colors indicate lower cut ratios.

weight). Figure 5 shows a heatmap of the geometric mean cut ratio across parameter pairs, where lighter colors indicate lower cuts and thus better partitioning quality.

The best performance is obtained for $\beta = 2$ and $\theta = 0.75$, which we therefore select as the default configuration. More generally, values of β around 2–2.5 perform best across a range of θ. By contrast, $\beta = 1$, corresponding to a purely linear degree contribution, is clearly inferior for all θ, confirming that a stronger-than-linear weighting of degree is essential.

For the θ parameter, the effect is more subtle. Across the tested range, the differences remain very small: the worst configuration reaches 24.19%, compared to 24.02% at the optimum. This indicates that HAA is robust with respect to θ: while values between 0.5–1.0 and also $\theta = 2$ (in combination with $\beta = 2.5$) yield slightly stronger results, no setting performs significantly worse once $\beta > 1$. In other words, θ fine-tunes the balance but does not fundamentally alter the effectiveness of the score.

It is worth noting that the tuning set consists of medium-sized graphs, which contain comparatively few extreme high-degree vertices (with $D_{\max} = 10\,000$ as normalization). This may partly explain why θ shows only marginal influence here: the degree term is already dominated by the stronger-than-linear exponent β. On larger graphs with more extreme degree distributions, the role of θ could become more pronounced, but testing this would require substantially more expensive experiments beyond the scope of the tuning set.

Overall, the heatmap confirms that HAA does not require fine-tuned parameters to

perform well. The sweet spot lies around $\beta = 2$ and $\theta = 0.75$, but many alternatives remain competitive as long as $\beta > 1$.

5.2.2 Buffer Size

The buffer plays a central role in our algorithm since it temporarily stores vertices before they are assigned to a batch. Increasing the buffer size Λ allows the algorithm to postpone more decisions and to exploit a larger amount of neighborhood information, which in principle should reduce premature assignments and thus improve partition quality. At the same time, a larger buffer inevitably increases memory usage, as more vertices and their adjacency information must be maintained, and also affects runtime, since more updates of buffer scores are triggered.

Table 3 summarizes the effect of increasing Λ for natural and random orderings. Figure 6 shows performance profiles for random orderings; the natural ordering profiles are similar and omitted for brevity. As expected, runtime and memory usage grow almost linearly with Λ, since larger queues store more vertices and trigger more score updates of their neighbors. However, Table 3 also shows that memory consumption remains relatively stable across the three smallest configurations and is even slightly higher for the smallest buffer ($\Lambda = 64$k). This effect can be explained by the fact that memory usage is initially dominated by the subgraph construction and multilevel partitioning phase, which require additional auxiliary data structures, whereas the buffer itself only stores vertex IDs and their adjacency lists. The actual contribution of the buffer to total memory

Ordering	Λ	Runtime (s)	Memory (MB)	Cut Edges (%)
Natural	32k	**5.40**	236.2	9.56
	64k	5.74	**219.2**	8.91
	128k	6.14	229.5	8.08
	256k	6.54	271.5	7.76
	512k	6.83	354.2	**6.90**
	1024k	7.29	486.3	6.92
Random	32k	**6.87**	159.0	30.21
	64k	7.71	**143.8**	28.33
	128k	8.43	150.9	24.30
	256k	9.07	185.8	20.03
	512k	9.55	275.7	16.94
	1024k	10.18	418.2	**14.29**

Table 3: Geometric mean results for varying buffer sizes Λ under natural and random orderings. The batch size is fixed to $\delta = 32$k in all configurations. Best result per metric and ordering in bold.

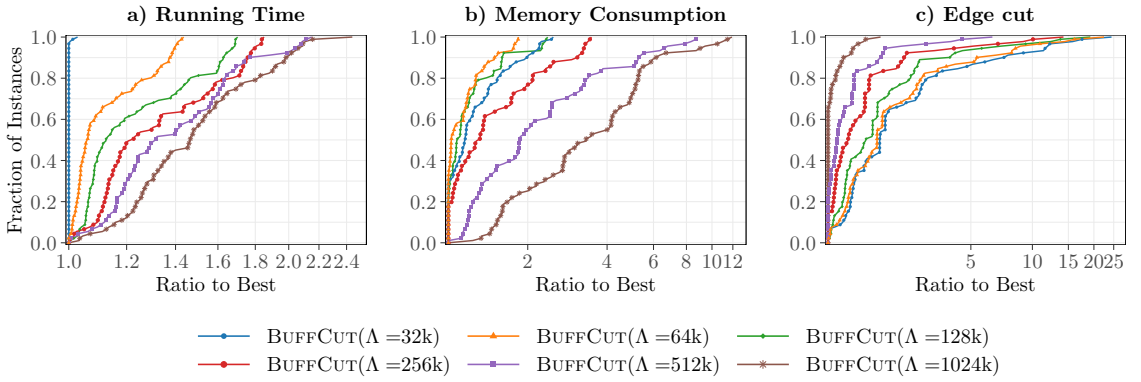


Figure 6: Performance profiles for varying buffer sizes Λ under random orderings. The batch size is fixed to $\delta = 32$k in all configurations.

usage becomes visible only once Λ exceeds roughly 256k, after which memory grows more clearly with buffer size.

Regarding partition quality, the results differ significantly between the two ordering scenarios. For random orderings, the geometric mean edge cut decreases from 28.3% at $\Lambda = 64$k to 14.3% at $\Lambda = 1024$k (Table 3), corresponding to a reduction of nearly 50%. Performance profiles (Figure 6) confirm this trend: the largest configuration ($\Lambda = 1$M) achieves the best cut on more than 60% of the instances, consistently outperforming smaller buffers. In extreme cases, the smallest buffers even yield edge cuts up to $25\times$ worse than the best instance.

This comes at the cost of memory usage, which is more than $4\times$ higher than the smallest configuration for roughly 45% of the instances. Runtime differences are less pronounced, with only moderate slowdowns for larger buffers. Overall, these results demonstrate that additional buffering is highly beneficial in random orderings, where little or no locality is present in the input stream.

For natural orderings, edge cuts are already much lower at small buffer sizes (8.9% at $\Lambda = 64$k compared to 28.3% in the random case), highlighting the strong inherent locality of these orderings. Improvements with increasing Λ are therefore modest: the cut ratio decreases slightly to 6.9% at $\Lambda = 512$k, but no further gain is observed at $\Lambda = 1024$k. This plateau suggests that even small buffers capture sufficient structural information when locality is present, while very large buffers may even dilute this structure by mixing vertices from different graph regions. This behavior also suggests that there are notions of locality that are not fully captured by the buffer score, indicating potential areas for future improvements.

In summary, buffer size Λ presents a clear trade-off: larger values improve partition quality at the cost of increased memory consumption and, to a lesser extent, runtime overhead. For practical deployments, Λ should therefore be chosen according to the available memory budget and the expected degree of locality in the input stream. At the same time,

the effect of buffering cannot be seen in isolation, as it also interacts with the batch size δ, which we analyze in the following section.

5.2.3 Batch Size

The batch size δ parameter originates from HeiStream and controls how many vertices are collected from the queue before forming a batch. Once the specified number of vertices has been extracted, they are combined into a subgraph that includes the batch vertices, the edges among them, and their connections to already partitioned neighbors. This subgraph is then partitioned using a multilevel pipeline consisting of coarsening, initial partitioning, and refinement during uncoarsening. Intuitively, larger batches provide more structural information about the graph and may therefore lead to improved partition quality. At the same time, increasing δ is expected to raise memory consumption and computational overhead, since more vertices and adjacency data have to be stored and processed simultaneously.

Table 4 and Figure 7 summarize the effect of varying δ with a fixed buffer size of $\Lambda = 1\text{M}$. As expected, memory consumption grows monotonically with larger batches: from 379 MB at $\delta = 4\text{k}$ to over 1.1 GB at $\delta = 512\text{k}$ for natural orderings, and from 348 MB to 1056 MB in the random case. This trend reflects the increasing size of the induced subgraphs that must be stored and processed in the multilevel step.

Runtime exhibits a U-shaped behavior. Very small batches (e.g., $\delta = 4\text{k}$) are inefficient because the construction of many small subgraphs introduces overhead, resulting in runtimes up to $3.5\times$ slower than the best configuration. For medium batch sizes (around 64k), runtimes are lowest (7.42s for natural, 10.32s for random), while very large batches again increase runtime due to the higher cost of processing large subgraphs (9.26s and 13.31s at $\delta = 512\text{k}$).

Partition quality steadily improves with larger δ. For natural orderings, the cut

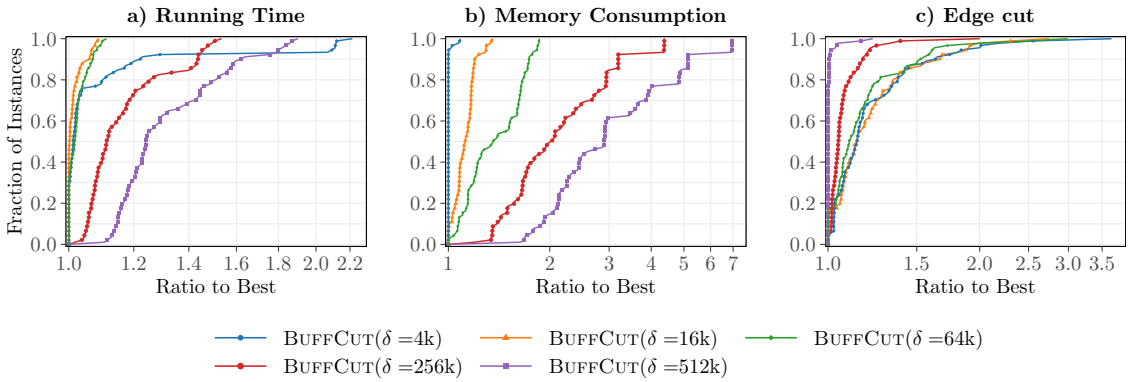


Figure 7: Performance profiles for varying batch sizes δ under random orderings. The buffer size is fixed to $\Lambda = 1\text{M}$ in all configurations.

Ordering	δ	Runtime (s)	Memory (MB)	Cut Edges (%)
Natural	4k	8.26	**379.4**	7.26
	8k	7.71	392.6	7.21
	16k	7.45	431.5	6.93
	32k	**7.29**	486.3	6.92
	64k	7.42	559.2	6.73
	128k	7.74	643.1	6.66
	256k	8.38	798.2	6.38
	512k	9.26	1115.2	**5.99**
Random	4k	11.02	**348.5**	14.55
	8k	10.42	355.4	14.66
	16k	10.22	389.9	14.50
	32k	**10.18**	418.2	14.29
	64k	10.32	478.8	14.03
	128k	10.78	570.1	13.40
	256k	11.76	739.0	12.79
	512k	13.31	1056.0	**11.97**

Table 4: Geometric mean results for varying batch sizes δ under natural and random orderings. The buffer size is fixed to $\Lambda = 1$M in all configurations. Best result per metric and ordering in bold.

decreases from 7.26% at $\delta = 4k$ to 5.99% at $\delta = 512$k. For random orderings, the cut improves from 14.55% to 11.97% over the same range, corresponding to a relative reduction of about 18%. This confirms that larger batches provide more structural information to the multilevel pipeline and thus enable better decisions. At the same time, the relative gain diminishes at very large batch sizes compared to the steep rise in memory consumption.

Overall, larger batches consistently improve partition quality but come at a clear cost in memory and runtime. In practice, δ should therefore be chosen not in isolation, but in relation to the buffer size Λ, since the ratio Λ/δ ultimately determines how much structural context is available to each batch. We analyze this interaction in the following subsection.

5.2.4 Buffer–Batch Trade-off

Our experiments show that buffer size Λ and batch size δ should not be tuned in isolation, but jointly. While our results do not indicate a single universally optimal setting, they suggest that maintaining a ratio of Λ/δ between 8 and 16 provides a good balance between partition quality, runtime, and memory usage across both natural and random orderings.

The rationale behind this ratio is that the buffer should always contain more vertices than the batch, providing a pool from which vertices with higher locality can be se-

lected. At the same time, the batch should not be too large, since subgraph construction and multilevel partitioning require considerably more memory than buffering alone. In this sense, especially for randomly ordered graphs, memory is often used more effectively in buffering, because even moderate buffers already provide structural context that would otherwise be missing.

Batch sizes below 8k are not advisable, since the repeated construction of many small subgraphs incurs high runtime overhead without notable quality gains. Conversely, for very large graphs, δ should be increased beyond the minimum to ensure that the cost of subgraph construction remains manageable.

In summary, we recommend maintaining a ratio of Λ/δ between 8 and 16, while adjusting both parameters within the limits of the available memory budget. This ensures a balanced trade-off between partition quality and resource usage.

5.2.5 Evaluation of Parallelization

Parallel execution is a central feature of BuffCut, as it aims to accelerate the streaming pipeline by overlapping buffer handling, score updates, and multilevel partitioning. To assess its impact, we compare the sequential baseline with the parallel version on the *Exploration & Tuning Set*. Table 5 reports geometric mean results, while Figure 8 provides performance profiles for the random-ordering scenario.

The results show a substantial runtime improvement: the parallel version reduces runtime by roughly 37–39% (e.g., from 10.0s to 6.2s under natural ordering and from 12.0s to 7.3s under random ordering). The performance profiles (Figure 8) confirm that in more than 90% of instances the parallel algorithm is at least $1.5\times$ faster than the sequential version.

This speed-up comes at the cost of higher peak memory usage. For natural orderings, memory increases by about 51% ($141 \rightarrow 214$ MB), while for random orderings the overhead is smaller at around 25% ($96 \rightarrow 120$ MB). We attribute this difference to the structure of the batch-induced subgraphs: under natural orderings, batches often contain many tightly connected vertices, which increases the number of intra-batch edges and thus the

Ordering	Mode	Runtime (s)	Memory (MB)	Cut Edges (%)
Natural	`Sequential`	10.01	**141.2**	8.98
	`Parallel`	**6.22**	213.5	**8.97**
Random	`Sequential`	12.04	**96.4**	**24.16**
	`Parallel`	**7.29**	120.0	24.23

Table 5: Geometric mean results comparing the sequential and parallel implementations of BuffCut under natural and random orderings. Best per metric and ordering in bold.

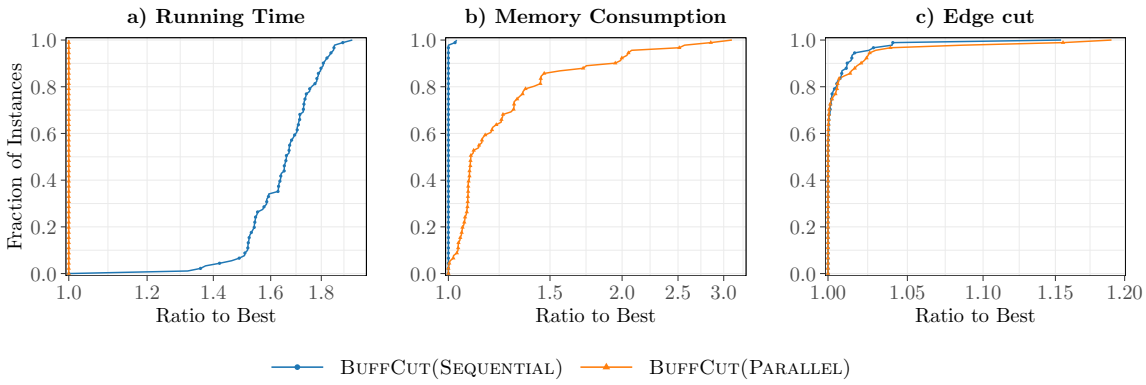


Figure 8: Performance profiles comparing the sequential and parallel implementations of BuffCut under random ordeings.

size of temporary data structures in the parallel pipeline. In contrast, random orderings yield smaller and sparser subgraphs, so the additional memory overhead of parallelization remains more moderate. The performance profiles further show that only about 10% of the random instances exceed a $2\times$ increase in memory, while the majority of cases remain below this threshold.

Partition quality remains essentially unchanged: sequential and parallel versions yield virtually identical edge cuts, with only marginal differences observable in either direction. For natural orderings, the parallel version is in fact slightly better on average, whereas for random orderings it is minimally worse. These deviations are negligible and can be attributed to the small amount of non-determinism introduced by parallel execution. Although the algorithm is deterministic in its main design, concurrent scheduling can subtly affect the processing order of certain vertices, in particular those that exceed $D_{\max}$, which may lead to minor variation in the final assignment.

Overall, the parallel version of BuffCut provides consistent runtime improvements at the cost of moderately higher memory usage, while partition quality remains effectively unchanged. The trade-off is especially favorable under random orderings, which are the main focus of our evaluation: here, runtime decreases by nearly 40% while memory grows by only about 25%. Given this clear benefit in this setting, we adopt the parallel implementation as the default configuration in all subsequent experiments, unless explicitly stated otherwise.

5.2.6 Impact of Ghost Neighbors

Ghost neighbors enrich the algorithm with additional structural context. They affect three parts of the pipeline: (i) buffer scoring, where the assigned-neighbor ratio is extended by ghost neighbors, (ii) subgraph construction, where edges to ghosts are added as quotient edges with reduced weight, and (iii) the partitioning of high-degree vertices, where ghost

neighbors contribute to the Fennel score alongside permanently assigned neighbors (see Section 4.2.4). Their influence is controlled by the *ghost weight* w_{ghost}, which specifies how strongly ghost edges contribute relative to ordinary edges. A setting of $w_{\text{ghost}} = 1$ makes ghost edges equivalent to regular edges, while $w_{\text{ghost}} = 0$ disables them entirely. In our experiments we test intermediate fractions (e.g., $1/2$, $1/5$, $1/10$), which arise naturally from scaling ghost edges against the weight of ordinary ones.

We proceed in two steps. First, we analyze the effect of varying w_{ghost} on the *Exploration & Tuning Set*, which consists of medium-sized graphs and allows us to study a broad range of values. Based on these results we select a reasonable default configuration. Second, we validate this choice on the *Test Set* of large graphs, where buffer-based streaming is particularly relevant in practice. This two-stage evaluation clarifies both the sensitivity of the mechanism and its practical impact on large-scale graphs.

Exploration & Tuning set. In the experiments on the *Exploration & Tuning Set* we varied $w_{\text{ghost}} \in \{0, 0.1, 0.2, 0.25, 0.33, 0.5, 1.0\}$ to assess its effect on cut quality, runtime, and memory. Figure 9 (relative edge cut) and Table 6 (geometric means) reveal three consistent effects. First, enabling ghost neighbors ($w_{\text{ghost}} > 0$) almost always improves quality compared to the baseline $w_{\text{ghost}} = 0$. Second, the optimal setting differs by ordering: random streams benefit from stronger ghost influence (best cut at $w_{\text{ghost}} = 0.5$ with 21.78%), while natural streams perform best with weaker contributions (minimum at $w_{\text{ghost}} = 0.1$ with 7.33%). Third, the overall gain from ghost information is much larger on natural orderings: compared to the best configuration, disabling ghosts increases cut by about 11% on random orderings, but by more than 23% on natural orderings.

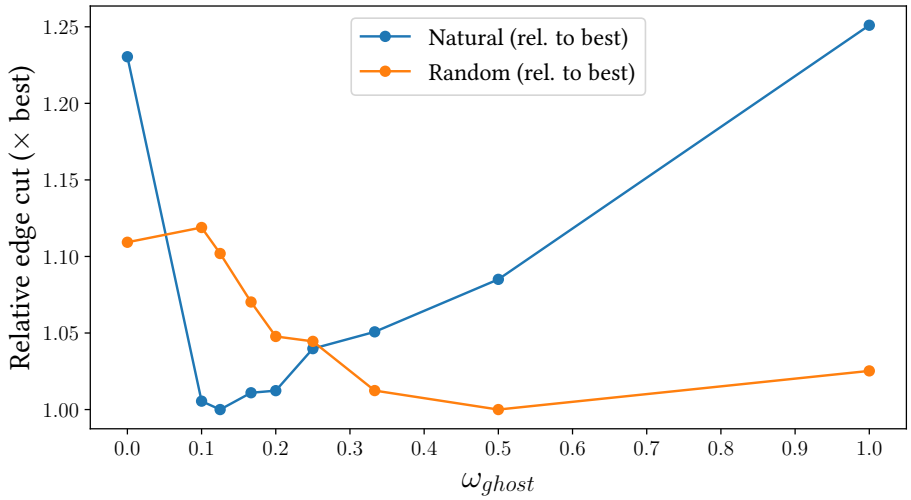


Figure 9: Relative edge cut of BuffCut under natural and random orderings on the *Exploration & Tuning Set* for different ghost weights w_{ghost}, normalized per ordering to the best configuration (lower is better). A ghost weight of 0 disables the mechanism entirely (baseline).

Ordering	ω_{ghost}	Runtime (s)	Memory (MB)	Cut Edges (%)
Natural	1	8.49	270.3	9.12
	0.5	6.80	275.4	7.91
	0.33	6.81	268.6	7.66
	0.25	6.87	268.8	7.58
	0.2	6.85	264.2	7.38
	0.1	6.87	254.9	**7.33**
	0	**6.18**	**213.0**	8.97
Random	1	10.70	178.3	22.33
	0.5	9.10	200.1	**21.78**
	0.33	9.04	202.9	22.05
	0.25	9.03	203.5	22.75
	0.2	9.03	202.9	22.82
	0.1	9.11	210.4	24.37
	0	**7.36**	**119.8**	24.16

Table 6: Geometric mean results for varying ghost weights w_{ghost} under natural and random orderings on the *Exploration & Tuning Set*. A ghost weight of 0 disables the mechanism entirely (baseline). Best result per metric and ordering in bold.

These patterns are consistent with structural differences. For random orderings, batches contain fewer intra-batch ties, so stronger ghost weighting provides valuable external hints and improves partitioning quality. For natural orderings, locality is already strong; large ghost weights blur this signal by pulling towards external assignments, while small weights preserve in-batch cohesion and still offer a light bias. The larger relative gain on natural streams indicates that ghost assignments there tend to align more coherently with the true partition structure, making their contribution particularly valuable.

Runtime varies only modestly once $w_{\text{ghost}} > 0$. The largest overhead appears at $w_{\text{ghost}} = 1$, which is not a competitive setting in terms of quality and can be regarded as an outlier. Excluding this case, runtimes on natural orderings rise from 7.36 s at $w_{\text{ghost}} = 0$ to about 9.0–9.1 s for $w_{\text{ghost}} \in [0.1, 0.5]$, i.e. an increase of roughly 20–25%. For random orderings, the overhead is smaller: from 6.18 s at 0 to around 6.8–6.9 s across the same range, which corresponds to about 10% more runtime. Overall, the variation is moderate, and the relative cost is more pronounced on natural graphs.

Memory shows a more pronounced overhead than runtime. For natural orderings, enabling ghosts increases usage from 120 MB at $w_{\text{ghost}} = 0$ to about 200–210 MB across the tested range, which corresponds to roughly 70–75% more memory. For random orderings, the effect is milder: memory grows from 213 MB at baseline to 255–275 MB with ghost weighting, an overhead of about 20–30%. The stronger increase on natural streams likely stems from the fact that more vertices are exploited as ghost neighbors,

so that additional quotient edges are inserted during subgraph construction. This higher overhead is consistent with the larger impact that ghost information has on partition quality in the natural case.

To avoid per-ordering tuning and keep experiments consistent, we adopt $w_{\text{ghost}} = 0.2$ as a global default. This value provides a reasonable compromise: close to optimal for natural streams, competitive for random ones, and consistently superior to disabling ghost neighbors altogether.

Test set. To validate these findings on larger graphs, we repeat the comparison on the *Test Set*, restricting attention to the baseline without ghost neighbors ($w_{\text{ghost}} = 0$) and the chosen default ($w_{\text{ghost}} = 0.2$). Here we also increase buffer size to $\Lambda = 1\text{M}$ and batch size to $\delta = 64\text{k}$ to account for the larger input.

Table 7 shows a clear contrast between orderings. On *natural* orderings, ghost neighbors provide a substantial improvement: the edge cut drops from 5.63% to 4.46%, a relative reduction of about 21%, while runtime rises moderately by $\approx 15\%$ (315 s $\rightarrow$ 363 s) and memory by $\approx 10\%$ (3479 MB $\rightarrow$ 3831 MB). Performance profiles (Figure 10) confirm this result: in roughly 70% of the instances the ghost-enabled variant achieves the lowest cut. Overheads remain moderate, with memory below $1.2\times$ the baseline for more than 85% of cases, and runtime below $1.5\times$ for about 90%. Only a handful of very dense graphs show higher slowdowns (up to $3.5\times$), likely due to larger subgraphs that inflate local partitioning costs.

On *random* orderings, by contrast, ghost neighbors no longer provide a benefit. The geometric mean edge cut remains essentially unchanged (17.72% vs. 17.69%), while runtime grows by around 20% (338 s $\rightarrow$ 407 s) and memory by around 30% (1.66 GB $\rightarrow$ 2.17 GB). Performance profiles (not shown) confirm the absence of quality gains, so we omit them for brevity. This reinforces the tendency already observed on the *Exploration & Tuning Set*: ghost neighbors are most helpful when locality is strong (natural orderings), whereas in random streams their additional context does not translate into quality improvements. A plausible explanation is that on large, randomly permuted graphs the pool of potential

Ordering	Ghost Neighbors	Runtime (s)	Memory (GB)	Cut Edges (%)
Natural	Disabled	**315.11**	**3.48**	5.63
	Enabled	362.76	3.83	**4.46**
Random	Disabled	**337.72**	**1.66**	17.72
	Enabled	407.34	2.17	**17.69**

Table 7: Geometric mean results under natural and random orderings on the *Test Set* with ghost neighbors either disabled or enabled with the default weight $w_{\text{ghost}} = 0.2$. All configurations use buffer size $\Lambda = 1\text{M}$ and batch size $\delta = 64\text{k}$. Best result per metric and ordering in bold.

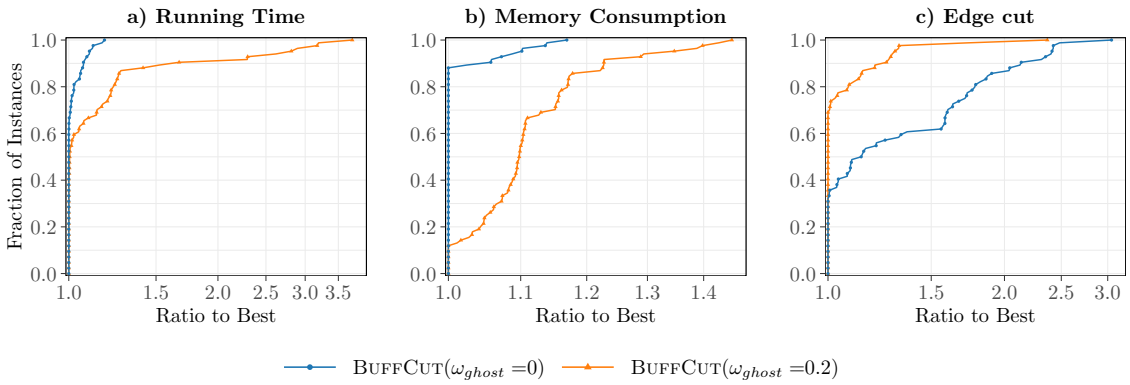


Figure 10: Performance profiles on the *Test Set* under natural orderings comparing BuffCut with ghost neighbors disabled ($w_{\text{ghost}} = 0$) and enabled at the default weight ($w_{\text{ghost}} = 0.2$). All configurations use buffer size $\Lambda = 1\text{M}$ and batch size $\delta = 64\text{k}$.

ghost neighbors grows rapidly, while many of these edges connect to weakly structured or noisy regions. As a result, the provisional signals carried by ghost assignments become diluted and add little reliable locality information for placement decisions, yet they still incur significant memory and runtime overhead.

Conclusion. Overall, ghost neighbors are a promising mechanism, but their benefit depends strongly on the input ordering. On natural orderings, where locality is already pronounced, they yield clear quality improvements at moderate overhead. On random orderings, however, the expected gains do not materialize on large graphs, as the additional ghost context appears too noisy to provide value. Thus, while we adopt $w_{\text{ghost}} = 0.2$ as a consistent global default, our results indicate that ghost neighbors are most useful in settings with strong structural locality. Developing adaptive strategies that can selectively exploit ghost information in less structured streams remains an interesting direction for future work.

5.3 Comparison with State of the Art Algorithms

In this section, we compare our algorithm BuffCut against the two most competitive buffered streaming partitioners, HeiStream and Cuttana. Earlier one-pass heuristics such as LDG or Fennel are not considered here, since they are consistently outperformed by HeiStream and thus not representative state of the art baselines. Our focus is therefore on buffered approaches.

In addition to the experiments presented here, we also reproduce the evaluation setup from the original Cuttana paper (Appendix A), which allows a direct comparison under

the authors' settings. Moreover, since our evaluation revealed a strong dependence of CUTTANA 's resource consumption on the number of partitions, we provide a dedicated analysis of its memory and runtime behavior in the appendix as well.

5.3.1 Baselines and Configurations

Before turning to the results, we summarize the configurations used for all algorithms. For BUFFCUT, we employ the parameter setting in Table 8. This configuration reflects the outcome of our parameter studies, but in the state of the art comparison the decisive parameters—buffer size Λ and batch size δ–are chosen such that memory usage remains both comparable across algorithms and reasonable for the large graphs that are contained in the benchmark. The remaining values follow our standard defaults.

Parameter	Value
Buffer score	HAA ($\theta = 0.75,\ \beta = 2$)
Buffer size Λ	1M
Batch size δ	64k
$discFactor$	1 000
$D_{\max}$	10 000
Parallelization	3 threads
Ghost neighbors	Disabled

Table 8: Configuration of BUFFCUT used in the state of the art comparisons.

CUTTANA. We obtained the code from CUTTANA's official repository and evaluate it in its parallel implementation. This matches the setup in the original publication and ensures a fair comparison with our algorithm, which is also evaluated in parallel. Moreover, we replaced memory-mapped I/O with standard streaming access, consistent with HEISTREAM and BUFFCUT, so that memory usage reflects algorithmic behavior rather than differences in file handling.

Implementation Issues. During our evaluation we identified two problems in the public code. First, the implementation crashes on graphs containing isolated vertices, which we fixed by simply excluding zero-degree vertices from the buffer. Second, the maximum buffer score used for early eviction decisions is stored as an `int` rather than a `double`, causing precision loss. This can lead to many vertices being prematurely assigned instead of buffered, especially when the graph contains more vertices than the buffer can hold. We tested CUTTANA both with and without this fix, but, in line with the developers' decision not to integrate it (to preserve their published results), we treat the official unmodified version as the baseline in our main comparisons.

Parameterization. According to its original publication [21], CUTTANA is evaluated with $D_{\max} = 1\,000$, a maximum queue size of 1M, and a subpartition parameter defined by the ratio $\frac{k'}{k} = 4096$, where k is the number of partitions and k' the total number of subpartitions. The only exception reported in the paper is for the `twitter` instance, where the authors set $D_{\max} = 100$ and $\frac{k'}{k} = 256$. The subpartition parameter plays a central role in CUTTANA, since refinement is performed between subpartitions; it therefore directly affects both partition quality and resource consumption. In our experiments, we adopt the recommended settings from the publication, but apply them uniformly across all graphs rather than making dataset-specific adjustments. Concretely, we use $D_{\max} = 1\,000$, a maximum queue size of 1M, and $\frac{k'}{k} = 4096$ for all instances.

This default choice makes CUTTANA highly resource-intensive, since the total number of subpartitions grows linearly with k and directly drives both memory consumption and runtime. For large k (e.g., 128 or 256), these requirements escalate sharply and render the algorithm impractical in real-world settings. A detailed breakdown of this effect, including the underlying data structures, is given in the Appendix in Section A.3.

To provide a fair comparison with HEISTREAM and BUFFCUT, we also evaluate CUTTANA with a reduced setting of $\frac{k'}{k} = 16$. This significantly reduces memory and runtime while still capturing the algorithm's intended behavior. We therefore report both configurations in our evaluation: the default $\frac{k'}{k} = 4096$ for completeness, and the reduced $\frac{k'}{k} = 16$ as the practical baseline.

HEISTREAM We also identified a bug in the publicly available HEISTREAM code: quotient edges to already partitioned neighbors were not aggregated correctly, leading to redundant entries and increased runtime and memory usage. We fixed this issue locally, and the patch has since been integrated into the official repository. All results for HEISTREAM are based on the corrected version.

Naming Conventions. Throughout this section, we use a consistent notation for algorithm configurations. For HEISTREAM, the value in parentheses refers to the batch size δ (e.g., HEISTREAM(512k) = 512k vertices). For BUFFCUT, the parentheses list first the batch size δ, then the buffer size Λ (e.g., BUFFCUT(64k, 1M) = batch size 64k, buffer size 1M). For CUTTANA, the number in parentheses denotes the subpartition ratio $\frac{k'}{k}$ (e.g., CUTTANA(16) = $\frac{k'}{k} = 16$). When relevant, we append the suffix `-Bugfix` to indicate the corrected implementation with the buffer-score fix applied. We follow the unit conventions introduced in Section 5.1 (suffix "k" binary, "M" decimal).

5.3.2 Experiments on Naturally Ordered Graphs

Figure 11 and Table 9 summarize the results on naturally ordered graphs. As expected, HEISTREAM achieves the best performance in this setting, benefiting from the strong locality of the input. Our algorithm, BUFFCUT, achieves results close to HEISTREAM and in

many cases nearly matches its quality, while CUTTANA remains behind and, depending on its configuration, incurs substantial runtime and memory overheads.

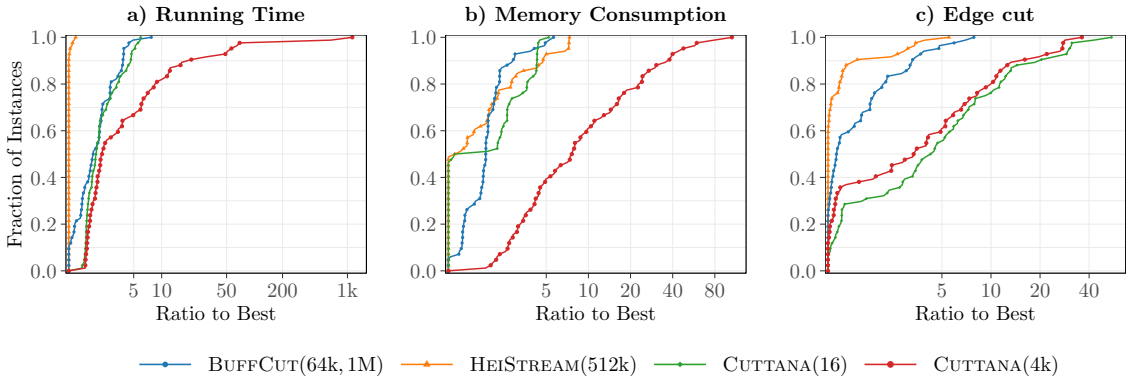
Edge Cut. Compared to CUTTANA, BUFFCUT achieves a clear improvement on natural orderings. Both configurations (16 and 4k) of the official version of CUTTANA produce cuts substantially higher (15.42% and 12.50% vs. 5.48%, Table 9), and even after applying the bugfix CUTTANA(16)-`Bugfix` still trails behind at 6.71%. CUTTANA(4k)-`Bugfix` improves upon BUFFCUT, however this comes at the cost of drastically increased resource consumption (see memory and runtime consumption discussions below). By contrast, BUFFCUT consistently remains close to HEISTREAM, which achieves the best cuts overall as expected (4.24% geometric mean). Thus, while HEISTREAM retains a small advantage thanks to its strong exploitation of input locality, BUFFCUT clearly outperforms CUTTANA and establishes itself as the better alternative for streams containing higher locality.

These aggregate findings are confirmed by the performance profiles in Figure 11a. HEISTREAM achieves the best cuts on roughly 60% of the instances, while our BUFFCUT and CUTTANA(4k) each reach about 20%. The overall picture, however, is very different: our method remains relatively close to HEISTREAM across most graphs, whereas CUTTANA quickly drops off and produces much weaker partitions, with both CUTTANA(16) and CUTTANA(4k) being up to 35× worse than the best results. The difference between CUTTANA(16) and CUTTANA(4k) is negligible, indicating that the buffer-score bug dominates the quality degradation. Figure 11b illustrates that the bugfix substantially improves CUTTANA 's performance, yet it continues to fall short of BUFFCUT across most graphs.

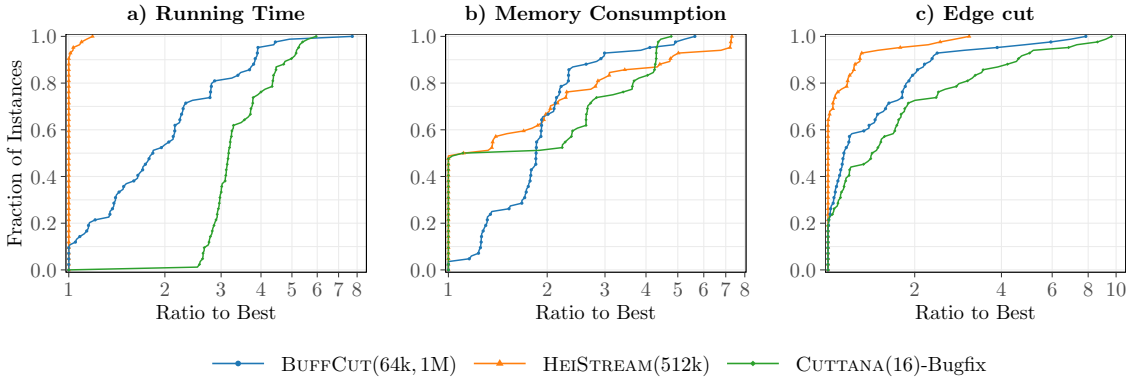
Runtime. HEISTREAM is by far the fastest method, requiring less than half the runtime of both our algorithm and CUTTANA(16), which perform on a similar level (Table 9). By contrast, CUTTANA(4k) is entirely impractical: as visible in Subfigure 11a, its runtime exceeds that of the best algorithm by factors of up to 1 000 on some datasets, making it unsuitable for any realistic setting. After applying the bugfix, CUTTANA becomes even slower, which is consistent with the fact that premature assignments are avoided and more processing steps are carried out. As a result, CUTTANA(4k)-`Bugfix` falls further behind our method in runtime efficiency.

Algorithm	Runtime (s)	Memory (GB)	Cut Edges (%)
HEISTREAM(512k)	**169.27**	**2.92**	**4.24**
BUFFCUT(64k, 1M)	326.06	3.30	5.48
CUTTANA(16)	378.37	3.19	15.42
CUTTANA(4k)	715.89	15.29	12.50
CUTTANA(16)-`Bugfix`	577.10	3.15	6.71
CUTTANA(4k)-`Bugfix`	964.10	14.38	5.07

Table 9: Geometric mean metrics of BUFFCUT, HEISTREAM and CUTTANA on the *Test Set* under natural orderings. Best result per metric and ordering in bold.



(a) Comparison of BuffCut, HeiStream and the official version of Cuttana, including Cuttana(16) and Cuttana(4k).



(b) Comparison of BuffCut, HeiStream and the bugfixed version of Cuttana.

Figure 11: Performance profile comparison on the *Test Set* under natural orderings. Both subfigures compare BuffCut, HeiStream, and Cuttana; subfigure (a) compares against the official version of Cuttana, while subfigure (b) against the bugfixed version.

Memory Consumption. All three methods — HeiStream, BuffCut, and Cuttana(16) — use comparable amounts of memory (around 3 GB on average, see Table 9). By contrast, Cuttana(4k) consumes drastically more memory: around five times the average and, as visible in Subfigure 11a, up to 100× higher on individual instances. The bugfix has little effect on memory usage, leaving Cuttana 's overall footprint essentially unchanged. Taken together, these results underline that while HeiStream dominates and our method provides competitive quality at moderate cost, Cuttana remains not scalable under either configuration.

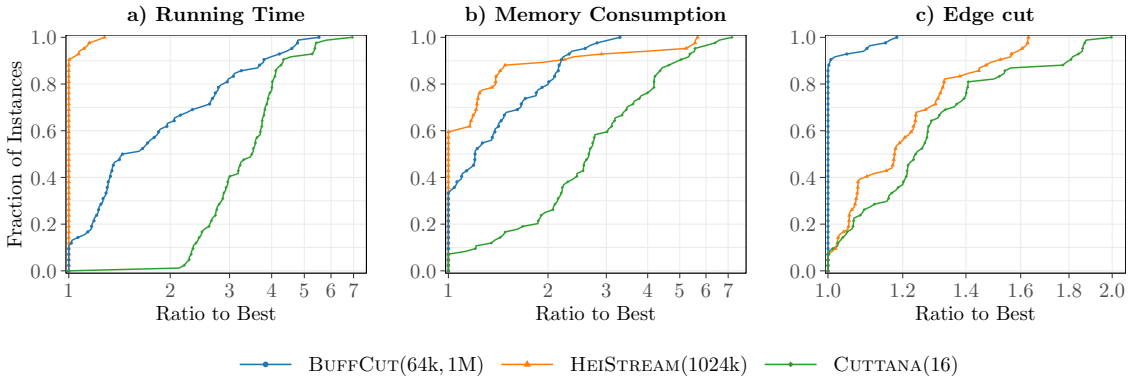
5.3.3 Experiments on Randomly Ordered Graphs

Random orderings represent the most challenging scenario, since input locality is destroyed and the quality of HeiStream degrades noticeably. This is precisely the setting our method is designed for: to maintain strong partition quality while keeping runtime and memory overhead moderate. In the following, we discuss results for partition quality, runtime, and memory consumption, referring to both the original and bugfixed versions of Cuttana. Aggregate statistics are reported in Table 10, while performance profiles are shown in Figure 12.

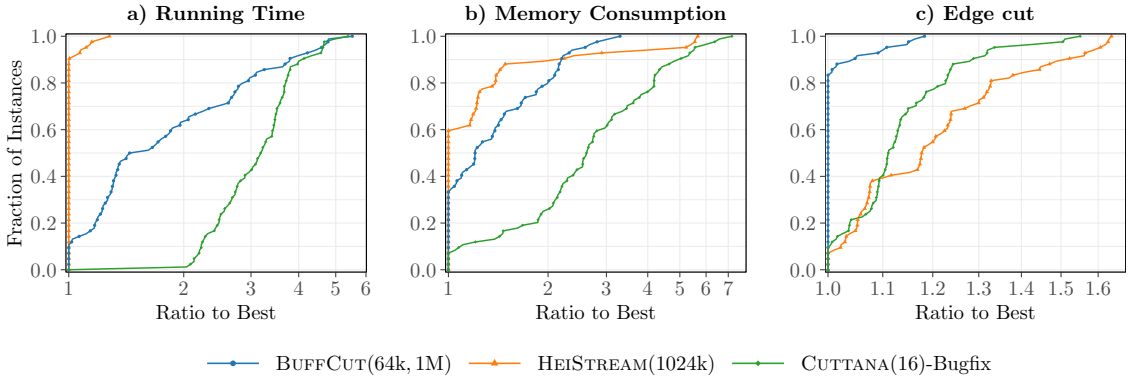
Edge Cut. In the performance profile comparison with HeiStream and the official version of Cuttana (Subfigure 12a), our algorithm dominates with nearly 85% of the instances yielding the best cuts. HeiStream and Cuttana(16) each achieve the best result in only about less than 10% of the cases, with HeiStream slightly ahead: its worst cuts are at most 1.6× worse than the optimum, whereas Cuttana(16) deteriorates up to 2×. Table 10 confirms this picture: our algorithm achieves the lowest geometric mean cut (17.76%), an 15.79% improvement over HeiStream (21.09%) and a 20.86% improvement over Cuttana(16) (22.44%). After applying the buffer-score fix (Subfigure 12b), Cuttana improves slightly to 19.90%, occasionally surpassing HeiStream, but still remains well behind our algorithm, which continues to dominate with about 80% of the best results. In comparison to the natural-ordering case, the impact of this bugfix is less pronounced here, since the absence of input locality limits the effect of buffering decisions in general.

Algorithm	Runtime (s)	Memory (GB)	Cut Edges (%)
HeiStream(1024k)	**195.70**	**1.51**	21.09
BuffCut(64k, 1M)	348.83	1.65	**17.76**
Cuttana(16)	647.81	3.19	22.44
Cuttana(4k)	1001.50	18.65	21.46
Cuttana(16)-Bugfix	600.27	3.14	19.90
Cuttana(4k)-Bugfix	918.94	17.69	18.96
HeiStream(512k)(2×)	**480.91**	3.14	16.21
HeiStream(512k)(3×)	773.20	3.17	15.35
BuffCut(64k, 1M)(2×)	594.32	**2.04**	13.88
BuffCut(64k, 1M)(3×)	863.11	2.14	**13.23**

Table 10: Geometric mean metrics of BuffCut, HeiStream, and Cuttana on the *Test Set* under random orderings. The upper block reports the main comparison. The lower block lists additional experiments with restreaming, where $(2\times)$ and $(3\times)$ denote two and three total passes, respectively. Best result per metric and block in bold.



(a) Comparison of BuffCut, HeiStream and the official version of Cuttana.



(b) Comparison of BuffCut, HeiStream and the bugfixed version of Cuttana.

Figure 12: Performance profile comparison on the *Test Set* under random orderings. Both subfigures compare BuffCut, HeiStream, and Cuttana; subfigure (a) compares against the official version of Cuttana, while subfigure (b) against the bugfixed version.

Runtime. As can be seen in Table 10, HeiStream exhibits the lowest average runtime (195 s). This observation is reinforced by Figure 12, which shows HeiStream to be the fastest solver on approximately 90% of the instances. Our algorithm requires about 78% more runtime (348 seconds) but achieves a better cut quality, which makes the trade-off acceptable in practice. Cuttana(16) is slower still, averaging 647 seconds, while Cuttana(4k) exceeds 1 000 seconds on average, which confirms its impracticality for bigger graphs. The performance profiles confirm this trend, with Cuttana(16) requiring more than double the runtime of the fastest method on almost all instances. After applying the buffer-score fix, Cuttana's runtime decreases slightly, though without changing the relative ranking.

Memory Consumption. HeiStream and our algorithm use comparable amounts of memory with average values around 1.5–1.6 GB (Table 10). By contrast, Cuttana(16) consumes roughly twice the memory on average. This overhead is reflected in the performance profiles (Figure 12): on more than 70% of the instances Cuttana(16) requires over twice the memory of the competing methods. The figure further shows that HeiStream exhibits higher peaks on a small fraction of instances, whereas our method varies less and remains closer to its typical memory usage. As expected, the buffer-score fix has virtually no effect on memory consumption, with Cuttana(16) and Cuttana(16)-`Bugfix` showing nearly identical behavior. Finally, Cuttana(4k) requires about an order of magnitude more memory on average, making any comparison with HeiStream or our method meaningless in terms of resource efficiency. For this reason, the configuration is not included in the performance profiles.

Overall, the results on random orderings confirm the robustness of our approach: BuffCut achieves the best partition quality, improving the geometric mean cut by 15.79% compared to HeiStream, while requiring about 78% more runtime. Memory consumption of both methods is comparable, while Cuttana trails behind in all three metrics and does not reach the effectiveness of either HeiStream or our algorithm.

Additional experiment: Restreaming. Since both HeiStream and our algorithm require substantially fewer resources than Cuttana in its Subp16 configuration—less than half the memory and runtime in most cases—it is natural to ask how much additional quality can be gained if we allow them to use comparable resources. To this end, we evaluated a single round of restreaming under random orderings. As shown in Table 10, the improvements are significant: for HeiStream the geometric mean cut decreases from 21.09% to 16.21%, corresponding to a relative improvement of about 23%. Our algorithm shows a similar gain, reducing its cut from 17.76% to 13.88%, which amounts to a relative improvement of 22%. Importantly, these results are achieved while still consuming less memory and runtime than Cuttana(16), and far below the demands of Cuttana(4k). Even compared to the bugfixed version of Cuttana (21.50%), HeiStream with one restreaming pass already produces clearly better cuts, and our algorithm further improves upon this by a substantial margin. This experiment highlights that even modest restreaming can significantly boost partition quality, while both HeiStream and our algorithm remain considerably more resource-efficient than Cuttana despite the additional pass.

CHAPTER 6

Discussion

The central goal of this thesis was to develop a new buffered streaming partitioner that reduces the sensitivity to unfavorable input orderings while preserving the efficiency that makes HeiStream attractive. To this end, we introduced BuffCut, which extends HeiStream by integrating prioritized buffering into the multilevel batching pipeline. In its default configuration, BuffCut mitigates the negative impact of poor locality in the input stream, while disabling the buffering step essentially reproduces the original HeiStream behavior.

Beyond this integration, we contributed two further extensions that strengthen the algorithm. First, we designed a new buffer score, that more effectively balance degree effects with neighborhood conformity. These scores deliver noticeable improvements over the score originally used in Cuttana, particularly on random inputs. Second, we introduced the concept of *ghost neighbors*, which temporarily propagate partial partition information from already assigned vertices to their unassigned neighbors. This mechanism enriches both buffer scoring and multilevel partitioning with predictive locality signals. On graphs with natural orderings and strong inherent locality, we observed a clear benefit in cut quality, including for large instances. The effect on randomly permuted inputs, however, is less consistent: while small benchmark graphs show improvements, larger ones did not exhibit measurable gains so far. In terms of resources, ghost neighbors introduce only a moderate increase in memory and runtime, which we consider acceptable given the potential quality improvements.

The experimental results show that BuffCut achieves the intended improvements. Compared to HeiStream, it consistently yields better partition quality under random orderings, which represent the more challenging scenario. On average, it reduces edge cuts by about 15.79% while keeping runtime and memory overheads within a comparable range. At the same time, it preserves competitive performance on favorable orderings, where HeiStream remains very strong. In this sense, BuffCut, in its default setting, acts as a robust ”all-rounder”: it does not specialize for one specific ordering, but instead provides stable performance across both structured and unstructured inputs.

When compared to CUTTANA, BUFFCUT shows clear advantages in both quality and efficiency. On input streams with weak locality (random orderings), it improves partition quality by about 21% while requiring only a fraction of the runtime and memory cost. With one additional restreaming round, partition quality improves to about 38% better than CUTTANA, while runtime and memory consumption remain slightly lower, making this a fairer like-for-like comparison.

The picture is equally favorable on stream inputs containing high locality. Here, the official CUTTANA implementation produces cuts above 15%, whereas BUFFCUT achieves about 5%, i.e., more than a threefold improvement. Even when applying the bugfix we identified for CUTTANA, its cut ratio decreases to 6.7%, but our method still delivers consistently better results at significantly lower runtime. In other words, BUFFCUT outperforms CUTTANA not only on adversarial streams but also when input orderings are already favorable, combining robustness with efficiency.

Naturally, limitations remain. While BUFFCUT narrows the gap between natural and random orderings, it does not close it entirely: partition quality under randomized streams remains lower, as expected, since no buffering scheme can fully compensate for the lack of locality. This reflects a fundamental constraint of streaming partitioning, where only limited structural information is available at each step. Nevertheless, by improving robustness without introducing prohibitive resource overheads, BUFFCUT makes a clear step forward over existing buffered approaches.

6.1 Conclusion

This thesis presented BUFFCUT, a new buffered streaming partitioner that integrates prioritized buffering with multilevel batch partitioning. The algorithm reduces the sensitivity of HEISTREAM to unfavorable orderings and consistently outperforms CUTTANA in terms of efficiency, while maintaining competitive partition quality. Key contributions include the design of improved buffer scores, the introduction of ghost neighbors, the use of a bucket-based priority queue, and a parallelized implementation. Across a diverse benchmark set, BUFFCUT achieves robust performance on both natural and random orderings, clearly improving upon the state of the art in buffered streaming partitioning.

6.2 Future Work

While this thesis improves the robustness and efficiency of buffered streaming partitioning, several avenues remain open for further research.

A first direction concerns the design of buffer scores. Although our hub-aware variant provided the best overall results, the margins over simpler scores such as the assigned-neighbor ratio were modest. This suggests that the current scoring schemes are still suboptimal. For example, on inputs with strong locality we observed that increasing the

buffer size did not always lead to monotonic improvements in cut quality, indicating that the score definition leaves room for refinement. Future work could therefore explore alternative designs or entirely new ideas for scoring, with the goal of unlocking further gains in cut quality and robustness.

Another natural direction is to further develop the idea of ghost neighbors. While they provided consistent improvements on inputs with locality, their benefit largely disappeared on large random graphs. This indicates that the mechanism itself is promising, but its current definition may be too simplistic to remain effective on graphs with very low locality. One direction would be to refine how ghost affiliations are assigned, for example by replacing the ”last assigned neighbor” rule with more robust metrics, potentially incorporating confidence measures or majority signals across multiple neighbors. A second direction would be to exploit ghost information more broadly within the pipeline, in particular during the initial partitioning of contracted subgraphs. Here, contracted nodes consist of multiple original vertices, some of which may carry ghost affiliations; aggregating these signals could provide more informative guidance for Fennel's initial placements and thereby improve the quality of early partitioning decisions.

Finally, the ideas developed here may generalize beyond vertex partitioning. In particular, buffering with priority mechanisms, combined with existing partitioning or refinement strategies, could also be explored in the context of *edge partitioning* or *graph clustering*. Applying such concepts in these domains might similarly help to reduce sensitivity to input orderings and improve robustness, much as we observed in the vertex partitioning setting.

APPENDIX A

Reevaluating CUTTANA

The authors of CUTTANA [21] report that the algorithm not only achieves lower edge cuts than HEISTREAM, but also requires less memory and runtime. This positions CUTTANA as a superior alternative to HEISTREAM.

In our experiments, however, we observed a different picture: while CUTTANA indeed yields slightly better edge cut results than HEISTREAM under streams with low locality, this improvement comes at the cost of substantially higher memory consumption and runtime under the configuration recommended in the original publication.

To clarify this discrepancy, we reproduced the experiments from the CUTTANA paper on the same benchmark graphs (orkut, uk-2002, uk-2007-05 and twitter). This allows us, first, to evaluate our algorithm BUFFCUT in the same setting and against the same baselines, and second, to complement the analysis with memory and runtime measurements that were not included in the original publication but proved decisive in practice.

In addition, we extend the evaluation beyond the scope of the original publication. The CUTTANA paper reports results only up to $k = 64$, without analyzing scalability at higher partition counts. In practice, however, larger values such as $k = 128$ or $k = 256$ are common in distributed systems. In this regime we observe a dramatic growth in both memory usage and runtime, in some cases exceeding 100 GB of memory and rendering the algorithm infeasible. Since this effect is not covered in the original paper, we explicitly reproduce and analyze it in Section A.3, including a breakdown of the underlying data structures that drive this resource growth.

A.1 Experimental Setup

We reproduced the experiments on the benchmark graphs `orkut`, `uk-2002`, `uk-2007-05`, and `twitter`, obtained from the `Konect` network repository [27]. The repository is currently offline. The graph properties are summarized in Table 1.

GRAPH	n	m	TYPE
orkut	3 072 411	117 185 082	Social
uk-2002	18 520 486	261 787 258	Web
twitter-2010	41 652 230	1 202 513 046	Social
uk-2007-05	105 896 555	3 301 876 564	Web

Table 1: Benchmark graphs used for reproducing the evaluation of CUTTANA. The set corresponds to the instances used in the original CUTTANA publication, allowing a direct comparison of results.

For twitter, our dataset differs from the instance reported in the CUTTANA paper: the number of vertices matches, but our version contains fewer edges. The difference corresponds closely to the amount of duplicate and self-loop edges removed during our preprocessing, suggesting that these edges were still included in the dataset used in [21]. When we normalize our edge-cut results by the larger edge count reported there, we obtain ratios that match the published values. Nevertheless, throughout this work we report edge-cut ratios with respect to our preprocessed graph (undirected, loop-free, deduplicated), while noting that the relative behavior remains the same.

As specified in the original publication, we executed HEISTREAM using its default configuration. For consistency, we did not incorporate the bug fix described in Section 5.3.1, which improves runtime and memory usage, since it was not part of the implementation at the time. For CUTTANA we used the authors' reported default settings (queue size 1M, $D_{\max} = 1\,000$ and $\frac{k'}{k} = 4096$; for twitter: $D_{\max} = 100$, $\frac{k'}{k}$ =256). By default, CUTTANA employs memory mapping, but we adapted it to use streaming instead, consistent with HEISTREAM and BUFFCUT, to avoid inflated memory usage and ensure fair comparison of runtime and memory consumption. BUFFCUT was run with the configuration described in Section 5.3.1, with batch size δ =256k and buffer size Λ =2M. Partition counts were chosen identically to the original evaluation, i.e., $k \in \{4, 8, 16, 32, 64\}$.

A.2 Reproduced Results

Figure 13 shows the reproduced results for $k \in \{4, 8, 16, 32, 64\}$ on all four benchmark graphs. With the exception of twitter (see discussion in Section A.1), our edge-cut ratios for both CUTTANA and HEISTREAM closely match the values reported in the original paper, confirming the correctness of our setup. In addition to edge-cut, we also report peak memory usage and runtime, metrics not included in [21] but highly relevant in practice.

Edge Cut. BUFFCUT matches or surpasses CUTTANA on most graphs, while only in few cases performing slightly worse. As in the original publication, CUTTANA consistently outperforms HEISTREAM in terms of cut quality.

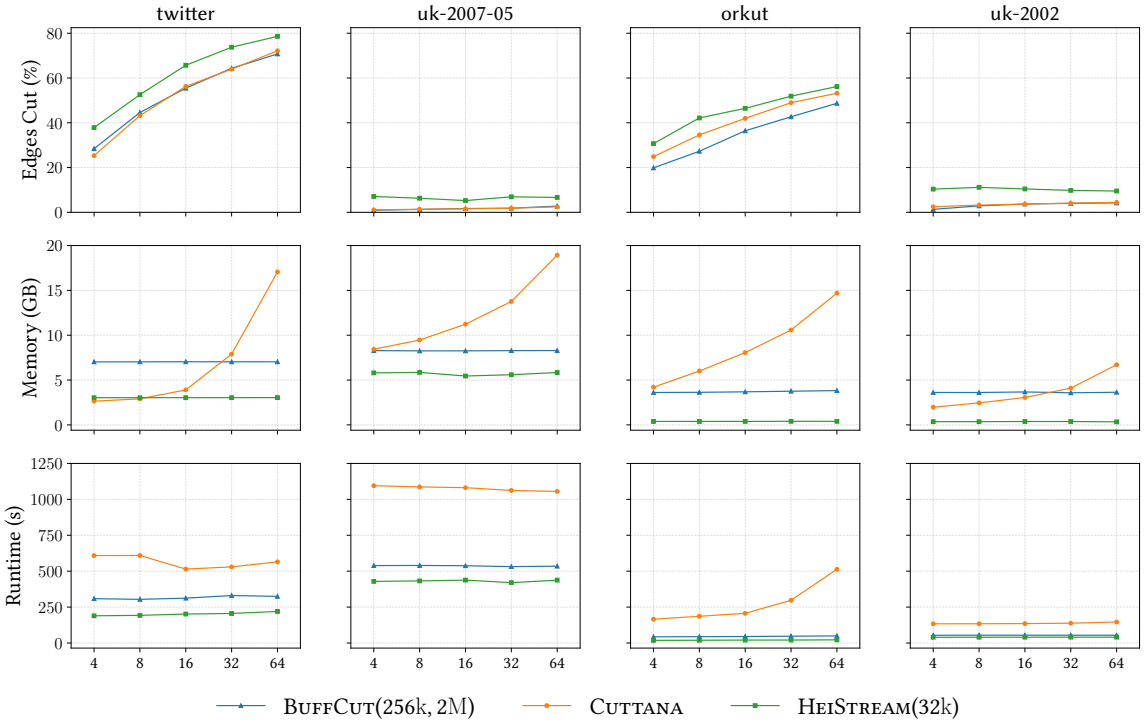


Figure 13: Reproduced results for Cuttana, HeiStream, and BuffCut. Each column corresponds to one graph (`twitter`, `uk-2007-05`, `orkut`, `uk-2002`), and rows show (top to bottom) edge cut ratio, peak memory usage, and runtime. Curves report values for partition counts $k = 4$ to 64.

Memory Usage. Across all graphs, we observe that HeiStream consistently requires the least memory, with the exception of `twitter` at small partition counts ($k = 4, 8$), where Cuttana 's footprint is marginally lower. BuffCut generally consumes less memory than Cuttana as well, apart from these small-k cases on `twitter` (and to a lesser extent $k = 16$ on `uk-2002`). Beyond these instances, Cuttana incurs substantially higher memory usage than both baselines. In particular, as k increases, Cuttana 's memory requirements rise steeply, while HeiStream and BuffCut remain stable. This growth becomes even more critical at larger k values (128 and beyond), and we analyze it in detail in Section A.3.

When comparing with the original publication, our findings deviate from the memory results reported in [21]. The paper presents values only for `uk-2007-05` and `twitter` (see Figure 14), where HeiStream is shown as more memory-intensive than Cuttana. We were not able to reproduce this behavior for either graph: in all our experiments, HeiStream consistently required less memory than Cuttana. Since the paper does not specify the partition count k used for these measurements, the reported values cannot be directly traced back to a comparable configuration. *Note:* The absolute values for

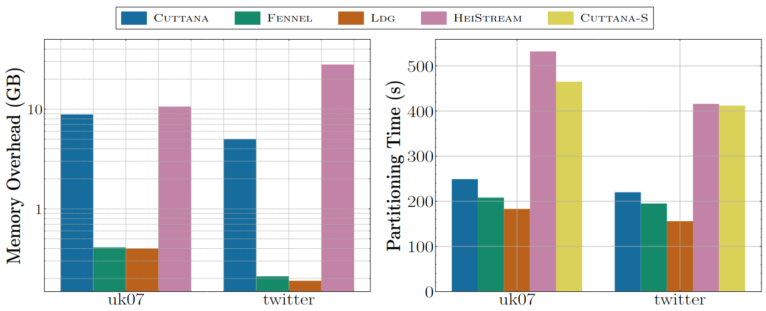


Figure 14: Memory and runtime results as reported in the original CUTTANA paper [21]. Our reproduction (see Figure 13) does not confirm these trends: in particular, we observe consistently lower memory and runtime for HEISTREAM compared to CUTTANA.

CUTTANA also differ from those in [21], as their implementation relies on memory mapping by default, whereas we switched to standard streaming for consistency across algorithms.

Runtime. In our experiments the picture is clear: HEISTREAM is consistently the fastest, BUFFCUT comes second, and CUTTANA is always the slowest across all graphs and partition counts. On average, CUTTANA requires more than twice the runtime of HEISTREAM, while BUFFCUT remains closer to HEISTREAM in efficiency.

The results reported in [21] (Figure 14) show the opposite relation, with HEISTREAM appearing slower than CUTTANA. We could not reproduce this behavior on any dataset. Note, that the absolute runtime of CUTTANA can differ from the values reported in [21] because we use streaming input instead of memory mapping, which leads to slower execution. However, this does not account for the reversed ordering of algorithms observed in their results.

Summary. Overall, BUFFCUT achieves cut quality comparable to or better than CUTTANA, while requiring significantly less memory and runtime. This makes it clearly superior in terms of efficiency–quality balance. By contrast, HEISTREAM consistently delivers the weakest cuts but is by far the most resource-efficient.

Finally, it should be noted that the comparison between CUTTANA and HEISTREAM in [21] is not entirely fair: while CUTTANA is allowed to consume vastly more memory, HEISTREAM was only tested in its default configuration. Granting HEISTREAM a larger buffer budget would likely have improved its cut quality, making the reported advantage of CUTTANA over HEISTREAM less pronounced than suggested in the original paper.

A.3 Scalability Limits of CUTTANA

In Section A.2 we already observed that memory usage of CUTTANA increase with the number of partitions k. The original publication reports results only up to $k = 64$ and does not discuss this effect, yet in our experiments we found it to be a decisive factor: memory requirements grow substantially with k on all graphs, and runtime also begins to rise noticeably at higher partition counts. Because values such as $k = 128$ or $k = 256$ are standard in distributed environments, this trend has strong practical implications. We therefore provide here a dedicated analysis, including an explanation of the underlying implementation choices that cause this steep growth.

To illustrate this effect more concretely, Table 2 reports runtime and memory usage for varying partition counts k of CUTTANA on the `twitter` graph. The same trend is visible across all benchmark graphs, with the growth becoming particularly pronounced once k exceeds 64. Here, we use `twitter` as a representative example.

The numbers highlight the severity of the effect: memory usage rises steadily from 2.6 GB at $k = 4$ to 17 GB at $k = 64$, and then almost triples again to 49 GB at $k = 256$. Runtime shows a similar pattern: it remains relatively flat up to $k = 64$, but then increases sharply, by roughly 50% at $k = 128$ and by more than a factor of two at $k = 256$ compared to $k = 4$. This pattern repeats with varying intensity on the other graphs, underlining that resource consumption grows disproportionately with larger partition counts.

Analysis. A central factor behind the steep growth in memory and runtime is the subpartition ratio $\frac{k'}{k}$, which is set to 4096 by default in [21]. This parameter directly determines the total number of subpartitions as $k' = k \cdot (k'/k)$. For instance, at $k = 256$ the default setting yields $k' = 256 \cdot 4096 \approx 10^6$ subpartitions. Each of these subpartitions allocates its own data structures, so the algorithm already incurs gigabytes of memory overhead before any edges are processed.

During the streaming phase, every subpartition instantiates a hash map with hundreds of preallocated buckets, so memory usage grows linearly with k'. Adjacency information

k	Runtime (s)	Memory (GB)
4	608.55	2.65
8	609.30	2.90
16	514.70	3.90
32	529.80	7.89
64	564.96	17.06
128	933.10	32.28
256	1648.35	48.98

Table 2: Runtime and memory usage of CUTTANA for various k on the `twitter` dataset.

is also stored redundantly in both streaming queues and subpartition maps, leading to additional temporary spikes. In the subsequent refinement phase, the implementation allocates dense $k' \times k$ matrices and a full $k \times k$ grid of segment trees. Together with permanent per-subpartition bookkeeping arrays, this produces a memory footprint that scales as $\Theta(k \cdot k')$.

Runtime increases for similar structural reasons. More subpartitions lead to more updates per vertex, each costing $\mathcal{O}(\log k)$ or $\mathcal{O}(\log(k'/k))$. Refinement is explicitly quadratic: it constructs $k \times k$ segment trees and scans partition pairs in every round. Each subpartition move triggers updates across all partitions, causing per-move costs of $\Theta(k \log(k'/k))$. As k and $\frac{k'}{k}$ grow, both initialization and refinement become increasingly expensive.

Our experiments confirm this effect: reducing the subpartition ratio to a moderate value (e.g., $\frac{k'}{k} = 16$ as used in Section 5.3.2 and 5.3.3) drastically lowers memory and runtime costs, but at the expense of weaker refinement and hence higher edge cuts. This illustrates the fundamental trade-off: the aggressive subpartitioning recommended in the original paper drives cut quality, but also renders the algorithm impractical for larger k.

In summary, the resource growth of CUTTANA stems from aggressive preallocation and dense structures in both phases of the algorithm. This explains the infeasible behavior observed in our experiments for $k \geq 128$ and highlights a fundamental scalability limitation that was not addressed in the original publication.

Bibliography

[1] Junya Arai et al. "Rabbit Order: Just-in-time Parallel Reordering for Fast Graph Analysis". In: *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. IEEE Computer Society, 2016, pp. 22–31. DOI: 10.1109/IPDPS.2016.110.

[2] Amel Awadelkarim and Johan Ugander. "Prioritized Restreaming Algorithms for Balanced Graph Partitioning". In: *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*. Ed. by Rajesh Gupta et al. ACM, 2020, pp. 1877–1887. DOI: 10.1145/3394486.3403239.

[3] David A. Bader et al. "Benchmarking for Graph Clustering and Partitioning". In: *Encyclopedia of Social Network Analysis and Mining*. Ed. by Reda Alhajj and Jon Rokne. New York, NY: Springer New York, 2014, pp. 73–82. DOI: 10.1007/978-1-4614-6170-8_23.

[4] Vignesh Balaji and Brandon Lucia. "When Is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering across Applications and Input Graphs". In: *2018 IEEE International Symposium on Workload Characterization, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018*. IEEE Computer Society, 2018, pp. 203–214. DOI: 10.1109/IISWC.2018.8573478.

[5] Reet Barik et al. "Vertex Reordering for Real-World Graphs and Applications: An Empirical Evaluation". In: *IEEE International Symposium on Workload Characterization, IISWC 2020, Beijing, China, October 27-30, 2020*. IEEE, 2020, pp. 240–251. DOI: 10.1109/IISWC50251.2020.00031.

[6] Charles-Edmond Bichot and Patrick Siarry. *Graph Partitioning*. John Wiley & Sons, Ltd, 2013. DOI: 10.1002/9781118601181.ch1.

[7] Paolo Boldi and Sebastiano Vigna. "The Webgraph Framework I: Compression Techniques". In: *Proceedings of the 13th International Conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004*. Ed. by Stuart I. Feldman et al. ACM, 2004, pp. 595–602. DOI: 10.1145/988672.988752.

[8] Paolo Boldi et al. "BUbiNG: Massive Crawling for the Masses". In: *ACM Trans. Web* 12.2 (2018), 12:1–12:26. DOI: 10.1145/3160017.

[9] Paolo Boldi et al. "Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks". In: *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*. Ed. by Sadagopan Srinivasan et al. ACM, 2011, pp. 587–596. DOI: 10.1145/1963405.1963488.

[10] Ulrik Brandes et al. "On Modularity Clustering". In: *IEEE Transactions on Knowledge and Data Engineering* 20.2 (2008), pp. 172–188. DOI: 10.1109/TKDE.2007.190689.

[11] Thang Nguyen Bui and Curt Jones. "Finding Good Approximate Vertex and Edge Partitions Is NP-hard". In: *Inf. Process. Lett.* 42.3 (1992), pp. 153–159. DOI: 10.1016/0020-0190(92)90140-Q.

[12] Ümit V. Çatalyürek et al. "More Recent Advances in (Hyper)Graph Partitioning". In: *Acm Computing Surveys* 55.12 (2023), 253:1–253:38. DOI: 10.1145/3571808.

[13] Adil Chhabra et al. "Buffered Streaming Edge Partitioning". In: *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*. Ed. by Leo Liberti. Vol. 301. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 5:1–5:21. DOI: 10.4230/LIPICS.SEA.2024.5.

[14] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (2011), 1:1–1:25. DOI: 10.1145/2049662.2049663.

[15] Elizabeth D. Dolan and Jorge J. Moré. "Benchmarking Optimization Software with Performance Profiles". In: *Mathematical Programming* 91.2 (2002), pp. 201–213. 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*. IEEE, 2021, pp. 101–112. DOI: 10.1109/IISWC53511.2021.00020.

[17] Priyank Faldu, Jeff Diamond, and Boris Grot. "A Closer Look at Lightweight Graph Reordering". In: *IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019*. IEEE, 2019, pp. 1–13. DOI: 10.1109/IISWC47752.2019.9041948.

[18] Marcelo Fonseca Faraj and Christian Schulz. "Buffered Streaming Graph Partitioning". In: *ACM Journal of Experimental Algorithmics* 27 (2022), 1.10:1–1.10:26. DOI: 10.1145/3546911.

[19] Charles M. Fiduccia and Robert M. Mattheyses. "A Linear-Time Heuristic for Improving Network Partitions". In: *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*. Ed. by James S. Crabbe, Charles E. Radke, and Hillel Ofek. ACM/IEEE, 1982, pp. 175–181. DOI: 10.1145/800263.809204.

[20] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. "Some Simplified NP-complete Problems". In: *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*. Ed. by Robert L. Constable et al. ACM, 1974, pp. 47–63. DOI: 10.1145/800119.803884.

[21] Milad Rezaei Hajidehi, Sraavan Sridhar, and Margo Seltzer. "CUTTANA: Scalable Graph Partitioning for Faster Distributed Graph Databases and Analytics". In: *Proc. VLDB Endow.* 18.1 (Sept. 2024), pp. 14–27. DOI: 10.14778/3696435.3696437.

[22] Bruce Hendrickson and Robert Leland. "A Multilevel Algorithm for Partitioning Graphs". In: *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. Supercomputing '95. San Diego, California, USA: Association for Computing Machinery, 1995, 28–es. DOI: 10.1145/224170.224228.

[23] Nazanin Jafari, Oguz Selvitopi, and Cevdet Aykanat. "Fast Shared-Memory Streaming Multilevel Graph Partitioning". In: *J. Parallel Distributed Comput.* 147 (2021), pp. 140–151. DOI: 10.1016/J.JPDC.2020.09.004.

[24] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". In: *Siam Journal On Scientific Computing* 20.1 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997.

[25] George Karypis and Vipin Kumar. "Parallel Multilevel K-Way Partitioning Scheme for Irregular Graphs". In: *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, November 17-22, 1996, Pittsburgh, PA, USA*. IEEE Computer Society, 1996, p. 35. DOI: 10.1109/SC.1996.32.

[26] Brian W. Kernighan and Shen Lin. "An Efficient Heuristic Procedure for Partitioning Graphs". In: *Bell System Technical Journal* 49.2 (1970), pp. 291–307. DOI: 10.1002/J.1538-7305.1970.TB01770.X.

[27] Jérôme Kunegis. "KONECT: The Koblenz Network Collection". In: *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*. Ed. by Leslie Carr et al. International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 1343–1350. DOI: 10.1145/2487788.2488173.

[28] Fabrice Lécuyer et al. "Tailored Vertex Ordering for Faster Triangle Listing in Large Graphs". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2023, Florence, Italy, January 22-23, 2023*. Ed. by Gonzalo Navarro and Julian Shun. SIAM, 2023, pp. 77–85. DOI: 10.1137/1.9781611977561.CH7.

[29] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. June 2014. URL: http://snap.stanford.edu/data.

[30] Henning Meyerhenke, Peter Sanders, and Christian Schulz. "Parallel Graph Partitioning for Complex Networks". In: *IEEE Trans. Parallel Distributed Syst.* 28.9 (2017), pp. 2625–2638. DOI: 10.1109/TPDS.2017.2671868.

[31] Joel Nishimura and Johan Ugander. "Restreaming Graph Partitioning: Simple Versatile Algorithms for Advanced Balancing". In: *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. Ed. by Inderjit S. Dhillon et al. ACM, 2013, pp. 1106–1114. DOI: 10.1145/2487575.2487696.

[32] Lara Ost, Christian Schulz, and Darren Strash. "Engineering Data Reduction for Nested Dissection". In: *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*. Ed. by Martin Farach-Colton and Sabine Storandt. SIAM, 2021, pp. 113–127. DOI: 10.1137/1.9781611976472.9.

[33] Md Anwarul Kaium Patwary, Saurabh Kumar Garg, and Byeong Kang. "Window-Based Streaming Graph Partitioning Algorithm". In: *Proceedings of the Australasian Computer Science Week Multiconference, ACSW 2019, Sydney, NSW, Australia, January 29-31, 2019*. ACM, 2019, 51:1–51:10. DOI: 10.1145/3290688.3290711.

[34] Ryan A. Rossi and Nesreen K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. Ed. by Blai Bonet and Sven Koenig. AAAI Press, 2015, pp. 4292–4293. DOI: 10.1609/AAAI.V29I1.9277.

[35] Laura A. Sanchis. "Multiple-Way Network Partitioning". In: *IEEE Trans. Computers* 38.1 (1989), pp. 62–81. DOI: 10.1109/12.8730.

[36] Peter Sanders and Christian Schulz. "Engineering Multilevel Graph Partitioning Algorithms". In: *Algorithms - ESA 2011 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*. Ed. by Camil Demetrescu and Magnús M. Halldórsson. Vol. 6942. Lecture Notes in Computer Science. Springer, 2011, pp. 469–480. DOI: 10.1007/978-3-642-23719-5_40.

[37] Peter Sanders and Christian Schulz. "Scalable Generation of Scale-Free Graphs". In: *Inf. Process. Lett.* 116.7 (2016), pp. 489–491. DOI: 10.1016/J.IPL.2016.02.004.

[38] Christian Schulz and Darren Strash. "Graph Partitioning: Formulations and Applications to Big Data". In: *Encyclopedia of Big Data Technologies*. Ed. by Sherif Sakr and Albert Zomaya. Cham: Springer International Publishing, 2018, pp. 1–7. DOI: 10.1007/978-3-319-63962-8_312-2.

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

[40] Ole Tange. *GNU Parallel 20231022 ('al-Aqsa Deluge') [Stable]*. Zenodo, Oct. 2023. DOI: 10.5281/zenodo.10035562.

[41] Charalampos E. Tsourakakis et al. “FENNEL: Streaming Graph Partitioning for Massive Scale Graphs”. In: *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*. Ed. by Ben Carterette et al. ACM, 2014, pp. 333–342. DOI: 10.1145/2556195.2556213.