Buffered Streaming Edge Partitioning

Adil Chhabra

August 17, 2023

3682160

Master Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

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

Co-Supervisor:
Marcelo Fonseca Faraj, Daniel Seemaier

Acknowledgments

My thesis supervisor, Prof. Dr. Christian Schulz, introduced me to the topic of graph partitioning and scalable graph algorithms in 2020, when he hired me as a student research assistant. I am forever indebted to his generosity, and to the opportunities that he has given to me over the past years. Under his guidance, I have not only developed as a programmer and researcher, but grown to become a better person - one who is able to work with focus, efficiency, and impact, while being attentive, kind and thoughtful to the people in my life. He is an inspiration for me.

I would also like to give a huge thanks to Marcelo Fonseca Faraj, who co-supervised this thesis. Whenever I required help, Marcelo would make himself available in an instant. His passion for programming, brilliant technical skills, and care for others is unmatched. If in these last years, I have adopted even 10% of these traits from Marcelo, I would consider my time in Heidelberg to be a success. Thank you also to Daniel Seemaier for co-supervising this thesis. Daniel's thoughtful feedback at every stage, and exceptional ability for scientific writing made this project possible. I also want to give a special mention to Henrik Reinstädtler, who offered support on various facets of this project. Every time I discussed my thesis with Henrik, he would give me excellent ideas.

Thank you also to Amna Pathan, who brought joy and excitement as I worked on the thesis. Amna encouraged me and helped me stay motivated. Finally, I would like to thank my parents, who have supported me through every decision I have made. Everything I do, I do to make them proud.

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, August 17, 2023

Adil Chhabra

Abstract

Partitioning a graph into balanced blocks is an important preprocessing step for distributed graph processing. In edge partitioning, the edge set of an input graph is partitioned into k roughly equal blocks while minimizing the replication of vertices across blocks. Streaming partitioners can partition huge graphs with fewer computational resources than in-memory partitioners. In this work, we propose a buffered streaming model for edge partitioning that sequentially loads batches of edges and permanently assigns them to blocks. For each batch, we construct a comprehensive graph representation that models adjacencies among edges and partition it using a multilevel scheme. Our approach produces state-of-the-art solution quality and is asymptotically independent of k in both runtime and memory consumption. We show experimentally that our algorithm yields better solution quality than all competing algorithms, and is substantially faster, and requires less memory, than comparable high-quality algorithms at large k values.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Complex, large graphs, often composed of billions of entities, are employed across multiple fields to model social, biological, navigational and technical networks. However, processing huge graphs requires extensive computational resources. Due to the recent stagnation of Moore's law, the primary approach to increasing computing power is to augment the number of available cores, processors, or networked machines, collectively referred to as processing elements (PEs), and leveraging parallel computation. When performing computations on massive graphs, therefore, graphs are distributed over multiple machines using distributed graph processing frameworks, like Pregel [47], GPS [58] and PowerGraph [26].

A powerful method to take advantage of parallelism is graph partitioning. Graph partitioning is used to model the distribution of large graphs across PEs to minimize communication volume [32] and to speedup jobs that have dependencies between computation steps [64]. Large graphs are partitioned into sub-graphs distributed among k PEs; each PE performs computations on a portion of the graph, and communicates with other PEs through message-passing. Graphs must be distributed across PEs such that each PE receives approximately the same number of components, and communication between PEs is minimized. The balanced graph partitioning problem thus optimizes for these objectives: a graph is partitioned into k blocks such that each block has roughly the same size to ensure balanced load distribution across PEs, and vertex or edge cuts are minimized to minimize communication between PEs. Traditionally, vertex partitioning has been used to distribute graphs across PEs, in which vertices are equi-partitioned to k blocks with the number of edges spanning partitions minimized. An alternate approach is to use edge partitioning to equi-partition edges to k blocks such that vertex replication is minimized, hence minimizing the communication needed to synchronize vertex copies. Graph partitioning is NP-complete [24] and there can be no approximation algorithm with a constant ratio factor for general graphs [10]. Thus, heuristic algorithms are used in practice.

A substantial amount of research has been dedicated to graph partitioning, developing three broad categories of partitioning algorithms: internal memory (shared-memory parallel) algorithms, which operate on graphs on a shared memory, streaming algorithms, which process graphs component by component using little memory, and distributed memory parallel algorithms, which partition graphs in-memory across multiple machines. Both internal memory and distributed memory algorithms, also called offline algorithms, can produce high-quality partitions. However, they face certain limitations that motivate the use of streaming algorithms, also referred to as online algorithms. While multi-level internal memory graph partitioners like KaHIP [59] and METIS [39] provide high-quality partitions, they require a single PE with sufficient memory, which is often infeasible for huge graphs. Further, they cannot be used for preprocessing in out-of-core algorithms, or for the initial distributive step of distributed memory algorithms. Distributed memory algorithms, on the other hand, can overcome memory constraints to partition huge graphs with high-quality solutions, however they require significant computational resources and potentially access to a supercomputer. Further, the initial step in distributed algorithms, to split the input graph across different machines, requires a preliminary partition that can be generated by a streaming algorithm to improve runtime and solution quality. Streaming partitioners are also used in distributed graph processing systems that utilize a load-compute-store logic such as MapReduce [19] and Giraph [18], and systems which support native graph-as-a-stream computations such as Kineograph [16], and Apache Flink [12].

While streaming partitioners can partition huge graphs quickly and with little memory, most produce a solution quality that is significantly lower than offline partitioners. The most popular streaming approach, the one-pass model, permanently assigns vertices to blocks during a single sequential pass over the graph's data stream. Examples of these include: 1) stateless partitioners, such as hashing and constrained partitioning algorithms, which ignore past partition assignments when making an assignment decision, and 2) stateful partitioners, such as Linear Deterministic Greedy (LDG) [65] which use the entire history of past assignments to make the next assignment decision. Stateless streaming partitioners typically produce the lowest quality solutions, but are very fast and light-weight. On the other hand, stateful partitioners achieve better solution quality but are slower than stateless partitioners. Nonetheless, even most stateful one-pass algorithms, like LDG, tend to produce relatively low-quality solutions due to sub-optimal initial assignment decisions, when little or no information is available regarding past assignments. While re-streaming to update block assignment can improve solution quality, it still falls short of offline approaches [53]. An alternative to one-pass streaming is buffered streaming, which addresses the issue of having little information for initial assignment decisions. Buffered streaming algorithms receive and store a buffer of vertices before making assignment decisions, thus providing some information about future vertices as well as past assignments. HeiStream [22] uses a buffered streaming approach for vertex partitioning that produces partitions of huge graphs with significantly higher quality than existing streaming vertex partitioners, while using a single machine without a lot of memory. The buffered approach in HeiStream, which we adapt to an edge partitioning setting, improves solution quality by

receiving and storing a buffer of vertices before making assignment decisions using a multilevel partitioning scheme.

Despite improvements in algorithms for vertex partitioning, the development of high-quality edge partitioners is motivated by research indicating that edge partitioning outperforms vertex partitioning on most real-world graphs [50], which tend to have skewed degree distributions across vertices. In particular, standard tools, e.g., [40] [41] for constructing a balanced vertex partition perform poorly on power-law graphs [1] [44] [46], which have many more vertices with low degree than high degree. In power-law graphs, the distributed processing runtime is negatively affected by high-degree vertices, which result in higher edge-cuts in vertex partitioning and thus more communication steps. Edge partitioning was introduced by Gonzalez et al. [26] to counter the shortcomings of vertex partitioning on power-law graphs. The rationale behind an edge partition is as follows: we allow a single vertex to span multiple machines, thereby offering more flexibility in load balancing, and we evenly distribute edges, resulting in reduced communication and storage overhead. As each edge is stored in exactly one block, changes to edge data do not need to be communicated.

The edge partitioning problem can be solved using hypergraph partitioners in shared memory, such as hMETIS [38] [40] and (mt)-KaHyPar [27] [29] [28]. These compute high quality solutions but require long runtime, as observed by Li et al. [45], who introduced the split-and-connect (SPAC) method for edge partitioning. In the SPAC method, the input graph G is transformed into a secondary graph G', where the original vertices are duplicated and connected with heavy dominant edges and unweighted auxiliary edges. A vertex partitioner run on G' can be used to compute an edge partitioning of the original input graph, with a faster runtime than hypergraph partitioning. Schlag et al. [62] adapt the SPAC method in a distributed memory parallel edge partitioning algorithm. We further adapt the SPAC method to a streaming setting through the creation of a contracted SPAC (CSPAC) graph for each batch of our buffered input, which we partition using a multilevel scheme similar to HeiStream.

As with streaming vertex partitioners, one-pass streaming edge partitioners, such as Degree Based Hashing (DBH) [69], which is a hashing-based stateless partitioner, or High-Degree (are) Replicated First (HDRF) [55], which uses a stateful greedy heuristic, yield low-quality solutions compared to in-memory partitioners. Mayer et al. [48] propose a buffered streaming approach for edge partitioning, Adaptive Window-based Streaming Edge partitioning (ADWISE), which uses a window of edges rather than an edge-by-edge stream for partitioning. Subsequently, Mayer et al. [50] introduced 2PS, a two-phase model for edge partitioning. In the first phase, 2PS uses a streaming clustering algorithm to gather information about the global graph structure. Then, in the second phase, it pre-partitions edges whose endpoints were in the same cluster, and computes assignments for the remaining edges using HDRF. The clustering phase enables 2PS to address the lack of information problem that one-pass algorithms face in initial assignment decisions, and offers more global information than the buffered approach used in ADWISE. With these phases, 2PS

produces partitions of higher solution quality than one-pass streaming algorithms, including DBH and HDRF, and also outperforms ADWISE for both solution quality and runtime.

While stateful streaming partitioners produce higher quality partitions than stateless partitioners, most existing stateful approaches result in a higher time complexity due to a linear dependency on the number of partitions k. While hash-based partitioning uses a constant-time hashing function for each edge assignment, resulting in a time complexity of $\mathcal{O}(|E|)$, most stateful streaming partitioners use a scoring function that is computed for every combination of edges and blocks, resulting in a time complexity of $\mathcal{O}(|E| \cdot k)$. Existing stateful partitioners thus require very long runtime when k is large. However, in recent years, there has been a significant growth in the size of real-world graphs, the complexity of computations, and the availability of machines with a large number of PEs, resulting in the increasing use of high k values in graph partitioning. These advancements may reduce the practical significance of current stateful streaming graph partitioning algorithms, due to their runtime becoming unfeasibly long for large k. Mayer et al. [51] address this problem with 2PS-L, an adaptation of 2PS that achieves linear runtime that is independent of k. Unlike other stateful partitioners, including 2PS with HDRF, 2PS-L uses a constant-time scoring function that only computes a score for two partitions. While 2PS-L achieves better solution quality than DBH and HDRF, its solution quality is significantly inferior to 2PS with HDRF. Thus, there remains potential to explore streaming edge partitioning algorithms that can produce high quality solutions for large k values without a runtime and memory dependency on k.

1.2 Our Contribution

In this work, we provide a buffered streaming model for edge partitioning that computes state-of-the-art solution quality and is independent of k in both memory and runtime. Our contributions are threefold:

1. We propose HeiStreamEdge, a buffered streaming model for edge partitioning that computes a balanced edge partition of the input graph with significantly better solution quality than existing state-of-the-art streaming edge partitioning algorithms. In this model, we load a buffer of vertices of size δ, which is passed as an input parameter. The parameter δ controls the amount of memory required by the partitioner. Next, we build a meaningful model from this buffer which incorporates past assignment decisions. This model is subsequently partitioned using a multilevel partitioning scheme. HeiStreamEdge is the highest quality streaming edge partitioner that additionally benefits from a runtime that is independent of the number of blocks k. Overall, we yield very low replication factor with a runtime of $\mathcal{O}(n + m)$. To the best of our knowledge, we are the only stateful streaming edge partitioner whose asymptotic memory consumption is also independent of k.

2. We introduce a novel transformation of a graph into its corresponding contracted split-and-connect (CSPAC) graph. This graph is obtained by contracting dominant edges of the split-and-connect (SPAC) graph proposed by Li et al. [45]. The CSPAC graph retains the approximation guarantees of the SPAC graph: a vertex partition of the CSPAC graph offers an edge partitioning of the input graph that is within a factor of $\mathcal{O}(\Delta\sqrt{\log n \log k})$ of the optimal edge partition of G, where Δ is the maximum degree of the graph. We prove this bound for the CSPAC graph and offer an algorithm to implement the transformation in $\mathcal{O}(n + m)$ time. The CSPAC graph has a much smaller size than the SPAC graph, and is therefore more efficient for computation.

3. We present extensive experimental evaluation on real-world graphs, including some huge graphs (up to 3.5 billion edges). Our results demonstrate that we have the best solution quality compared to existing streaming edge partitioners. We yield a replication factor that is approximately 8.3% better than the next best on average. Further, we are faster than other higher-quality solvers for larger k values ($k > 256$): among state-of-the-art partitioners that are faster for larger k values, we produce, on average, 57.6% better solution quality than the next best.

1.3 Structure

The remainder of the thesis is organized as follows: in Chapter 2, we describe the fundamental concepts related to our work, including the definition of the graph partitioning problem, and concepts related to vertex and edge partitioning, multilevel partitioning and the streaming approach. Chapter 3 discusses related research, beginning with an overview of fundamental work on streaming vertex partitioning, and later offering a deep-dive into relevant work on in-memory and streaming edge partitioning. In Chapter 4, we present our main contribution, HeiStreamEdge, a buffered streaming edge partitioner. After introducing the overall structure of our proposed approach, we discuss the construction of the novel CSPAC graph. Subsequently, we provide details of the graph model we build for partitioning, including multiple configurations for model construction. Next, we describe the vertex partitioning of our graph model using a multilevel partitioning scheme and present an adaptation for k-independent partitioning. In Chapter 5, we showcase the empirical evaluation of our proposed streaming edge partitioner. After describing our experimental methodology and the graph instances used, we present experiments to tune parameters and analyze the empirical effects of various configurations of our algorithm. Then, we compare our partitioner against the state-of-the-art, namely 2PS-HDRF and 2PS-L. Chapter 6 summarizes our work and contributions, and offers avenues for future work.

CHAPTER 2

Fundamentals

2.1 Graphs

A *graph* G is characterized by a set of vertices V, a set of edges $E \subset V \times V$ which represent relations between vertices, and two cost functions modeling vertex and edge weights. Let $c : V \rightarrow \mathbb{R}_{\geq 0}$ be a vertex-weight function, and let $\omega : E \rightarrow \mathbb{R}_{\geq 0}$ be an edge-weight function. As per standard convention, the number of vertices of a *graph* G is represented by n, and the number of edges is represented by m. An edge between two vertices u and v is shown as $(u, v) \in E$ and is defined as a forward edge if $u < v$ or a backward edge otherwise. In an undirected graph, an edge $(u, v) \in E$ implies the existence of a corresponding edge $(v, u) \in E$. An edge $e = (u, v)$ is said to be *incident* on vertices u and v. We generalize the cost functions c and w to sets, such that $c(V') = \sum_{v \in V'} c(v)$ and $w(E') = \sum_{e \in E'} \omega(e)$. The set $N(u) = \{u : u, v \in E\}$ denotes the neighbors of vertex u. The *degree* $d(u)$ of a vertex u is the number of its neighbors. The maximum degree among all vertices of a graph G is denoted by Δ. Likewise, the weighted degree of a vertex is the sum of the weights of its incident edges. A *graph* $S = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E(V' \times V')$.

2.2 Graph Partitioning

Given a number of *blocks* $k \in \mathbb{N}_{\geq 1}$, and an undirected graph with *positive* edge weights, the *graph partitioning* problem pertains to the partitioning of a graph into k smaller graphs by assigning the vertices (*vertex partitioning*) or edges (*edge partitioning*) of the graph to k mutually exclusive blocks, such that the blocks have roughly the same size and the particular objective function is minimized or maximized.

2.2.1 Graph Vertex Partitioning

The graph vertex partitioning problem asks for k *blocks* of vertices $V_1, ..., V_k$ that partition the vertex set V, that is,

1. $V_1 \cup \cdots \cup V_k = V$

2. $V_i \cap V_j = \emptyset \; \forall i \neq j$

Further, a *balance constraint* demands that all blocks have roughly equal size. More precisely, the sum of vertex weights in each block must not exceed a threshold associated with some imbalance ϵ, that is, for $\epsilon \in \mathbb{R}_{\geq 0}$, we must have $\forall i \in 1...k : c(V_i) \leq L_{max} = (1+\epsilon)\lceil \frac{c(V)}{k} \rceil$ where $c(V)$ is the sum of vertex weights for all vertices in V. A perfectly balanced partition is the case when $\epsilon = 0$. When $c(V_i) < L_{max}$, we say the block V_i is under loaded, and overloaded if $c(V_i) > L_{max}$.

As stated earlier, the *goal* of graph partitioning is to minimize or maximize a certain objective function. In the case of vertex partitioning, arguably the most prominent objective is to minimize the *edge-cut*. The *edge-cut* of a k-partition consists of the total weight of the *cut edges*, i.e., total weight of the edges crossing blocks. More formally, the *edge-cut* is defined as

$$\sum_{i<j} \omega(E_{ij})$$

in which $E_{ij} := \big\{\{u,v\} \in E : u \in V_i, v \in V_j, i \neq j\big\}$ is the *cut-set* (i.e., the set of all cut edges). We consider the weight of edges E_{ij} where $i < j$ to avoid double counting. While there are other more realistic (and often more complex) objective functions, we minimize the edge-cut as it is highly correlated with other formulations.

The problem of partitioning a graph into k blocks of roughly equal size, such that the *edge-cut* is minimized, is NP-hard as shown by Hyafil and Rivest [35] and Garey et al. [25].

A *clustering* is also a partition of the vertices of a graph into disjoint sets $C_1 \cup \cdots \cup C_l = V$. However, unlike *vertex partitioning*, in *clustering*, k is not usually known or given in advance, and the balance constraint is removed. A *size-constrained clustering* restricts the size of the blocks of a *clustering* by a given upper bound U such that $c(C_i) \leq U$.

We define an abstract view of the partitioned graph as a quotient graph Q, in which vertices represent the k blocks of partitions, and edges are induced by the connectivity between blocks, i.e., the cut-edges between blocks. More precisely, in a quotient graph, each vertex i of Q has weight $c(V_i)$ and there exists an edge between i and j if there is at least one edge in the original partitioned graph that runs between the blocks V_i and V_j. A pair of blocks that is connected by an edge in the quotient graph is called *neighboring* blocks. A vertex $v \in V_i$ that has a neighbor $w \in V_j$, $i \neq j$, is a boundary vertex.

2.2.2 Graph Edge Partitioning

Similar to the vertex partitioning problem, the *edge partitioning problem* asks for k *blocks* of edges $E_1, ..., E_k$ that partition the edge set E, that is,

1. $E_1 \cup \cdots \cup E_k = E$
2. $E_i \cap E_j = \emptyset \; \forall i \neq j$

The balance constraint is observed here as well, demanding that $\forall i \in 1...k : \omega(E_i) \leq L_{max} = (1+\epsilon)\lceil \frac{\omega(E)}{k} \rceil$ where $\omega(E)$ is the sum of edge weights of all edges in E. It ensures that the weights of the partitioned blocks do not exceed the expected sum of edge-weights multiplied by an imbalance factor ϵ in each block.

We define the set $V(E_i) = \left\{ v \in V \mid \exists u \in V : (u,v) \in E_i \vee (v,u) \in E_i \right\}$ for each partition E_i as the number of vertices in V that have at least one edge incident on them that was assigned to block E_i. Taking the sum of $|V(E_i)|$ over all k gives us the total number of vertex replicas generated by the partition. The primary objective function for edge partitioning, *replication factor* is then obtained by dividing the total number of vertex replicas by the number of vertices in the graph, i.e.,

$$RF(E_1, E_2, ...E_k) = \frac{1}{k} \sum_{i=1,...k} |V(E_i)|$$

The *goal* of edge partitioning is to minimize the *replication factor*. Intuitively, a minimized replication factor suggests that vertices are replicated in minimum blocks. Minimum vertex replication, in turn, results in lower synchronization overhead in distributed graph processing due to reduced exchange of vertex state across blocks. The runtime of distributed graph processing has been shown to have a direct correlation with replication factor in edge partitioning, as demonstrated in several studies [31] [70]. Like the vertex partitioning problem, edge partitioning of graphs into k blocks of roughly equal size such that replication factor is minimized is NP-hard [8] [70].

2.2.3 Multilevel Partitioning Scheme

As stated above, the graph partitioning problem is NP-hard. Approximation algorithms have been developed for graph partitioning, and are of high theoretical importance. However, they are often not implemented, or are too slow for large graphs compared to state-of-the-art graph partitioners [63]. Thus, mostly heuristics are used in practice.

A successful heuristic for partitioning large graphs is the multilevel graph partitioning (MGP) approach. In the MGP scheme, the input graph is recursively *contracted* to achieve smaller (coarser) graphs that reflect the same structure as the initial graph. This process is called *coarsening*. At the end of the *coarsening* phase, we arrive at the *coarsest graph*, on which we apply an *initial partitioning* algorithm. Thereafter, the coarsening is undone, and,

at each level, a *local search* method is used to improve the *initial partitioning* computed at the *coarsest level*. This stage of the MGP scheme is called uncoarsening, and it results in making the graph finer in every iteration till we arrive back at the input graph.

The crux of the MGP scheme is to retain the structure of the input graph during *coarsening*. For vertex partitioning, this can be achieved by *coarsening* through *contracting* clusters of vertices $C = C_1, ..., C_l$. *Contracting* a cluster of vertices consists of replacing each cluster C_i with a new vertex v. The weight of this vertex is set to the sum of the weight of all vertices in the cluster, i.e., $c(v) = \sum_{u \in C_i}(c(u))$. This new vertex is connected to all elements $w \in \bigcup_{i=1}^{l} N(C_i)$, $w(\{v, w\}) = \sum_{i=1}^{l} w(\{C_i, w\})$. In other words, the new vertex v is connected to all contracted vertices w induced by clusters C_j which contain a vertex that has an edge to some vertex in cluster C_i. This ensures that a partition at a coarser level that is transferred to a finer level maintains *edge-cut* and the balance of the partition. The uncoarsening phase, also known as *refinement*, undoes the contraction of a vertex back into its constituent cluster. During uncoarsening, local search is performed to reduce the objective function by moving vertices between blocks.

2.3 Streaming Computational Models

Streaming algorithms typically follow an iterative load-compute-store logic. The classic streaming model is the one-pass model. In this model, vertices of a graph are loaded one at a time, along with their adjacency lists. Then, some logic is applied to permanently assign each vertex to a block as it is visited. Here, assignment decisions for the current vertex depend on assignment decisions for previously visited vertices. Thus, the model has to store the assignments of all previously loaded vertices and hence needs $\Omega(n)$ space. A similar sequence of operations is used to partition a stream of edges of a graph on the fly. In the case of edge partitioning, edges of a graph are loaded one at a time along with their end-points. Then, some logic is applied to permanently assign them to blocks. The logic for one-pass partitioning can be a simple Hashing function [69] or a complex scoring of all blocks based on some objective function [55] [66]. In the latter case, each vertex is assigned to the block with the optimal score.

An extended version of the one-pass model is called the buffered streaming model, for example, the model used in HeiStream [22]. In this model, instead of loading vertices one at a time, we load a δ-sized buffer or batch of input vertices along with their edges. Here, we make block assignment decisions only after the entire batch has been loaded. In practice, the parameter δ can be chosen in accordance with memory available on the machine. In our contribution, like HeiStream, we use a fixed δ throughout the algorithm. For a predefined batch size of δ, we load and repeatedly partition $\lceil n/\delta \rceil$ batches.

CHAPTER 3

Related Work

There has been extensive research on graph partitioning, so we refer the reader to relevant literature [11] [14]. Most high-quality vertex partitioners for real-world graphs use a multilevel scheme, including KaHIP [60], Jostle [68], METIS [39], Scotch [17], hMETIS [38] [40] and (mt)-KaHyPar [27] [29] [28]. In this chapter, we review relevant literature on streaming vertex partitioning as well as in-memory and streaming algorithms for edge partitioning.

3.1 Streaming Vertex Partitioning

Stanton and Kliot [65] propose graph vertex partitioning in the streaming model, offering several heuristics to solve it, including both one-pass methods, such as Hashing and Chunking, and buffered methods such as Greedy EvoCut. The proposed buffered heuristics all performed significantly worse than random partitioning. Stanton and Kliot's experiments found that the Linear Deterministic Greedy (LDG) heuristic performed the best in terms of minimizing edge-cut among all proposed heuristics. LDG minimizes edge-cut by prioritizing vertex assignments to blocks assigned to its neighbors, while using a penalty multiplier to control imbalance. To do so, it assigns each vertex to the block containing the most neighboring vertices: a vertex v is assigned to block V_i that maximizes $|V_i \cap N(v)| * \lambda(i)$, where $\lambda(i)$ is a penalty multiplier $(1 - \frac{|V_i|}{L_{max}})$. Weighing the score with a penalty multiplier avoids imbalance between blocks by penalizing larger blocks.

Tsourakakis et al. [66] introduce Fennel, a one-pass partitioning heuristic adapted from the clustering objective modularity [9]. Like LDG, Fennel minimizes edge-cuts by trying to place vertices in partitions with more neighboring vertices. It maintains a balancing threshold through an additive penalty, unlike the multiplicative penalty used in LDG. Fennel assigns a vertex v to a block V_i that maximizes the Fennel gain function $|V_i \cap N(v)| - f(|V_i|)$, where $f(|V_i|)$ is a penalty function to respect a balancing threshold. In particular, the penalty function is defined as $f(|V_i|) = \alpha\gamma|V_i|^{\gamma-1}$, where γ and α are tunable parameters

that control the amount of imbalance, as well as weights associated with maximizing the number of neighbors (when $\gamma = 1$), and minimizing the number of non-neighbors (when $\gamma = 2$) for the input vertex during partitioning. In other words, when $f(|V_i|)$ is constant, the objective function corresponds to the maximization of co-location with neighbors; when $f(|V_i|) = |V_i|$, the objective function corresponds to minimization of co-location with non-neighbors. After tuning experiments, Tsourakakis et al. propose setting $\gamma = 3/2$, and theoretically select $\alpha = m\frac{k^{\gamma-1}}{n^{\gamma}}$. While Tsourakakis et al. assume k is constant, and thus derive a complexity of $\mathcal{O}(n + m)$, since their algorithm iterates over all blocks k for each vertex, the complexity of the algorithm depends on k and is given by $\mathcal{O}(nk + m)$. Faraj and Schulz [22] propose a generalized weighted version of the Fennel gain function which we use in the initial partitioning step of our multilevel partitioning scheme. With a modification using priority queues, we compute the generalized weighted Fennel gain function for each vertex without visiting all k blocks. As such, our overall runtime is asymptotically independent of k, namely, $\mathcal{O}(n + m)$. Further information on our partitioning process, and the modification, is provided in Section 4.4.

Nishimura and Ugander [53] introduce a restreaming graph partitioning model, in which multiple passes through the input graph are performed, retaining vertex assignments across passes and thus allowing iterative improvements in partition quality. For all streaming runs except the initial one, future vertices in the stream have past block assignments that inform the partitioning heuristics. The authors propose ReLDG and ReFennel, restreaming versions of LDG and Fennel respectively. Besides recalling past vertex assignments, ReFennel also increases the weight of the balance penalty over the stream runs. In a study of the effects of vertex ordering on streaming graph partitioning, Awadelkarim and Ugander [4] introduce prioritized restreaming algorithms to optimize the ordering of the streaming process. They used ReLDG to test the effects of using either static or dynamic stream ordering; the latter allows for variation in the order of streamed vertices in between streaming runs, according to specific priorities to improve partition quality. The authors propose a dynamic vertex ordering prioritization called ambivalence, which outperformed their other reordering algorithms; ambivalence places vertices with "more ambivalent" block assignment decisions (often low-degree vertices) at the end of stream. These "more ambivalent" vertices are vertices for which the difference between the number of co-assigned neighboring vertices in their current block assignment and the best possible external assignment is small.

WStream, proposed by Patwary et al. [54], is a greedy streaming graph partitioning algorithm that uses a sliding stream window containing a few hundred vertices, which, similar to buffered partitioning, increases the information available for block assignments. However, the authors only evaluate WStream on graphs with a few thousand vertices. Faraj and Schulz [22] introduce HeiStream, a buffered streaming model that uses a sophisticated multilevel partitioning scheme. In HeiStream, vertices are streamed in batches of a user-definable number. A complex graph model is constructed for every batch which contains the streamed batch of vertices, their connections to past block assignment decisions, and optionally, their connections to future batch vertices. A multilevel partitioning scheme is

performed on the graph model, and subsequently, vertices of the batch are permanently assigned to blocks. HeiStream was demonstrated to outperform previous state-of-the-art streaming vertex partitioners in solution quality. In our work, we extend the approach used by HeiStream for edge partitioning.

3.2 In-memory Edge Partitioning

Edge partitioning has been solved directly with multilevel hypergraph partitioners by transforming the graph into its dual hypergraph representation. In this representation, edges of the graph become vertices of the hypergraph, and hyperedges are induced by incident edges of each vertex of the input graph. A vertex hypergraph partitioning of this dual hypergraph representation directly outputs an edge partition of the input graph. Popular hypergraph partitioners include PaToH [13] (originating from scientific computing), hMETIS [38] [40] (originating from VLSI design), KaHyPar [33] [2] [61] (general purpose, n-level), Mondriaan [67] (sparse matrix partitioning), MLPart [3] (circuit partitioning), Zoltan [41], and SHP [37] (distributed), UMPa [15] (directed hyper- graph model, multi-objective), and kPaToH (multiple constraints, fixed vertices) [5]. In hypergraph partitioning, the vertices of a hypergraph are equi-partitioned into k blocks, optimizing for the following two common objective functions: to minimize the number of cut-nets and the connectivity metric. Cut-net is a generalization of the edge-cut objective in graph partitioning applied to hypergraph partitioning. The connectivity metric is defined as $\gamma - 1$, where γ is the number of blocks connected by a net. Summed over all nets, connectivity models the total communication volume. In the case of edge partitioning with hypergraphs, optimizing the connectivity metric directly optimizes vertex cut of the underlying edge partitioning problem [14]. The multilevel scheme of these hypergraph partitioners, similar to those described above for graph partitioning, entail a recursive coarsening step, followed by direct partitioning on the coarsest hypergraph. This partition is then successively refined during the uncoarsening phase to return to the original hypergraph. Using this approach, hMETIS [38] [40] obtained edge partitioning with a 15% to 23% improvement in solution quality over the state-of-the-art at the time.

While multilevel hypergraph partitioning produces high-quality edge partitions, such partitioners tend to have long running time [45]. Li et al. [45] propose an alternative method of solving the edge partitioning problem, which produces comparable solution quality to hypergraph partitioning, but has a faster runtime. To compute an edge partitioning of G, Li et al. create a new graph G' from G with their proposed Split-and-Connect (SPAC) transformation. Subsequently, they use a vertex partitioner on G' which corresponds to an edge partitioning of the original input graph. Refer to Section 4.2 for a detailed explanation of the SPAC method of edge partitioning. Schlag et al. [62] adapt the SPAC method to a distributed memory parallel algorithm for edge partitioning, comparing multiple sequential and distributed graph, and hypergraph, partitioners for runtime, scalability, and solution quality. Their results showed that edge partitioning with parallel graph partitioners us-

ing the SPAC transformation outperforms distributed hypergraph partitioners. A notable exception is KaHyPar, a hypergraph partitioner that produces significantly better solution quality than alternatives. In particular, KaHyPar produces 20% better solutions than the best SPAC-based approach. However, it is an order of magnitude slower than the evaluated distributed algorithms. Our proposed approach uses a contracted version of the SPAC graph for streaming edge partitioning.

3.3 Streaming Edge Partitioning

Existing streaming edge partitioning heuristics can be divided into two categories: stateless approaches, like hashing and constrained partitioning algorithms, which ignore past partition assignments when making an assignment, and stateful approaches, like score-based heuristic models, which leverage past assignments to make the next assignment decision.

3.3.1 Stateless Streaming Edge Partitioning

Hash partitioning is a stateless, data-model agnostic partitioning method which can be used in vertex and edge partitioning. Hashing algorithms can be applied in a streaming setting, and can achieve good load balance across partitions if the predefined hash function guarantees uniformity. Hash partitioning uses a hashing function to map elements with distinct keys to different partitions: in edge partitioning, hashing maps a set of edges to partitions. The simple hashing technique for edge partitioning randomly assigns each edge to a partition: for each input edge $e \in E$, $E_i(e) = h(e)\ mod\ |k|$ is the identifier of the target partition E_i. Though hash partitioning is fast, this heuristic results in a large number of vertex replicas in general, and it performs poorly on graphs with skewed degree distributions, particularly power-law graphs [26]. Degree Based Hashing (DBH) [69] improves upon randomized hash partitioning, especially for power-law graphs, by taking vertex degree into account; DBH prioritizes cutting vertices with the highest degree, to minimize overall vertex cuts. For an input edge e, DBH computes the partial degree of its endpoint vertices u and v. After that, e is assigned to the partition ID computed as the hash of the vertex with the lowest degree.

Grid and PDS are constrained partitioning algorithms [36]. They limit vertex replication for each vertex $v \in V$ to only a small subset of partitions $S(v)$ among k partitions, called the constrained set of v. In Grid, all partition IDs are organized in a square matrix. The constrained set must guarantee the following properties; for each u, $v \in V$, (i) $S(u) \cap S(v) = \emptyset$; (ii) $S(u) \nsubseteq S(v)$ and $S(v) \nsubseteq S(u)$; (iii) $|S(u)| = |S(v)|$. It is easy to observe that this approach naturally imposes an upper bound on the replication factor. To position a new edge e connecting vertices u and v, it picks a partition from the intersection between $S(u)$ and $S(v)$ either randomly or by choosing the least loaded one. Different solutions differ in the composition of the vertex constrained sets. The grid solution arranges partitions in a $X \times Y$ matrix such that $k = XY$. It maps each vertex v to a matrix cell

using a hash function h. Then, $S(v)$ is the set of all the partitions in the corresponding row and column. In this way each constrained set pair has at least two partitions in their intersection. PDS generates constrained sets using Perfect Difference Sets [30]. This ensure that each pair of constrained sets has exactly one partition in the intersection. PDS can be applied only if $k = x^2 + x + 1$, where x is a prime number.

3.3.2 Stateful Streaming Edge Partitioning

Greedy [26] is a rule-based partitioning model that aims to minimize vertex replicas while maintaining a certain balance constraint across partitions. For every edge in the input stream, Greedy uses the following rules to evaluate the presence of the endpoint vertices in existing partitions:

1. If both endpoint vertices have been previously assigned to any common block, pick the least loaded common block.
2. If both endpoint vertices have been previously assigned to different blocks, and not assigned to any common blocks, pick the least loaded block from the union of all assigned blocks of the two vertices.
3. If only one of the endpoint vertices has been previously assigned, pick the least loaded block from the previously assigned partitions of that vertex.
4. If none of the vertices have been assigned, pick the least loaded block overall.

Gonzalez et al. [26] find that Greedy improves upon random placement with an order of magnitude reduction in the replication factor. Testing on five real-world graphs, the authors found that in all cases Greedy out-performs random placement.

Noting the primary motivation of edge partitioning to improve partition quality of power law graphs, Petroni et al. [55] propose High-Degree (are) Replicated First (HDRF), a streaming edge partitioning algorithm that exploits the skewed degree distribution of power law graphs by prioritizing vertex replicas of high-degree vertices. HDRF assigns an edge $e = (u, v)$ to the partition E_i that maximizes a scoring function $C^{HDRF}(u, v, E_i) = C_{REP}(u, v, E_i) + C_{BAL}(E_i)$, where $C_{REP}(u, v, E_i)$ is a degree-weighted replication score and $C_{BAL}(E_i)$ is a balancing score. $C_{REP}(u, v, E_i)$ is high if both vertices u and v incident to an edge e are in the vertex cover set of the same partition E_i. On the other hand, $C_{BAL}(E_i)$ is highest when E_i is the smallest partition (i.e., contains least number of edges). Here, C_{BAL} is controlled by a multiplicative parameter λ. When $\lambda = 0$, HDRF is agnostic of load balance. At $\lambda = 1$, HDRF represents the Greedy heuristic and for $\lambda > 1$, HDRF prioritizes balance proportional to λ. When λ approach ∞, HDRF produces random edge assignment. Furthermore, as stated previously, streaming partitioning is sensitive to the ordering of the graph stream. In particular, Petroni et al. note that a high locality of vertices in the graph data stream negatively impacts balancing. As such, they propose shuffling the graph before ingestion to combat the potentially

adverse effects of stream ordering. While shuffling mitigates the stream ordering problem, it causes poor memory access on the vertex-based partitioning state which results in slower execution. Zhang et al. [70] introduced Streaming Neighbor Expansion (SNE) which is a streaming version of the in-memory edge partitioner Neighbor Expansion (NE) [70] that utilizes sampling methods. SNE produces better solution quality than HDRF at the cost of increased memory consumption and runtime [50].

In contrast to one-pass streaming models like those mentioned above, Mayer et al. [48] introduce the Adaptive Window-based Streaming Edge partitioning algorithm (ADWISE), a window-based streaming edge partitioner. Window-based or buffered streaming produces better solution quality than one-pass streaming, as assignment decisions can be made from a set of edges within the buffer, rather than without any information about future edges, as in one-pass models. ADWISE uses a dynamic window size that adapts according to runtime constraints, and reports between $23 - 47\%$ improvement in solution quality over one-pass streaming edge partitioners. However, solution quality is dependent on window size, and so achieving a low replication factor necessitates a larger window and leads to a longer runtime. Like ADWISE, we use a buffered streaming model, however our model uses a fixed buffer size that can be specified by the user.

Mayer et al. [50] propose 2PS, a two-phase streaming algorithm for edge partitioning. The first phase uses a streaming clustering algorithm to gather information about the global graph structure; in the second phase, the graph is partitioned, using information obtained from clustering to make edge partitioning decisions. 2PS achieves better solution quality on real world graphs than DBH, and better runtime and solution quality than ADWISE and HDRF. The streaming clustering algorithm used in 2PS is an extension of an algorithm proposed by Hollocou et al. [34]. It follows the formal objective for clustering, which is to maximize modularity. Unlike streaming edge partition where edges are permanently assigned to blocks when visited, streaming clustering allows for cluster assignments of vertices to be updated when the vertex is revisited in a future edge. The clustering problem is typically less constrained than the partitioning problem: obtained clusters may be unbalanced and the number of clusters is not pre-set but rather determined by properties of the graph. Mayer et al. [50], however, limit cluster sizes such that the number of intra-cluster edges does not exceed the maximum size of any partition block controlled by the balancing constraint. With this, they are able to use clustering information to pre-partition edges whose endpoints were assigned to the same cluster, into the same block of the partition. The global information available through clustering thus enables the creation of pre-partitions with minimized replication factor, as the sub-set of edges partitioned at this stage do not have vertices that are replicated across blocks. In particular, the pre-partitioning algorithm performs a single pass over the edge stream: for each edge $e = (u, v)$, it checks if both incident vertices u and v are either in the same cluster or their clusters are assigned to the same partition E_i. If so, e is pre-partitioned and assigned to E_i. If E_i is at maximum capacity given the balancing constraint, e is not pre-partitioned, and instead partitioned in the final step. In the final partitioning step, all remaining edges are partitioned using the HDRF scoring function in yet another pass over the edges of the graph.

Subsequently, noting the increased importance of partitioning into a large number of blocks, Mayer et al. [51] proposed 2PS-L, an adaptation of 2PS that runs in time independent of k. 2PS-L retains the clustering phase and pre-partitioning step in 2PS, but introduces a new scoring function for the final partitioning step to remove its dependency on k and thus achieve a time complexity of $\mathcal{O}(|E|)$. The novel scoring function is constrained to only take into account two blocks, regardless of the value of k. Specifically, these are the two blocks associated with the clusters of the adjacent vertices u and v of an edge $e = (u, v)$. 2PS-L is shown to perform faster than all other stateful partitioners, particularly at larger k values. However, this comes at the cost of solution quality. For instance, the previously proposed 2PS with HDRF achieves 50% better solution quality over 2PS-L [51].

CHAPTER 4

Buffered Streaming Edge Partitioning

In this chapter, we describe our buffered model for computing edge partitions in a streaming setting. We begin by shortly outlining the overall structure of our algorithm in Section 4.1. Then, we provide details of the per-batch graph conversion into a novel contracted SPAC graph in Section 4.2 and prove theoretical approximation guarantees for it. Next, we present our ultimate graph model in Section 4.3 along with a description of its various configurations. Finally, we describe the partitioning scheme we apply on our graph model in Section 4.4. Overall we propose a high-quality stateful streaming edge partitioning approach that has a runtime complexity of $\mathcal{O}(n + m)$. Our approach is asymptotically independent of k in both memory (first of its kind) and runtime.

4.1 Overall Structure

In this section, we explain the overall structure of our proposed buffered streaming edge partitioner, HeiStreamEdge. The approach closely follows that of HeiStream for vertex partitioning by Faraj and Schulz [22]. We slide through the input graph G by repeating the following successive operations until all vertices of G are visited, at which point all edges of G are assigned to blocks. First, we load a batch G_b containing δ vertices along with their adjacency lists. This provides us with edges between vertices in the current batch, as well as edges to vertices in past and future batches (except for the first and last batch respectively). In our algorithm, we only consider edges between vertices in the current batch, and edges to vertices in previous batches (more information in sections to follow). Second, we transform our per-batch graph G_b into its corresponding contracted split-and-connect graph S^*. Third, we construct our graph model β from S^*. This model represents block assignments for edges that have already been partitioned, as well as edges of the current batch. Then, we partition β with a multilevel vertex partitioning algorithm to optimize for the generalized weighted Fennel objective function, adopting the approach used in HeiStream. As demonstrated by Schlag et al. [62], a good vertex partition of the split graph

intuitively leads to a good edge partition of the input graph. Finally, we permanently assign the vertices of the graph model β constructed from the current batch to their corresponding global edge IDs. Algorithm 1 summarizes the general structure of HeiStreamEdge which is showcased in Figure 4.1.

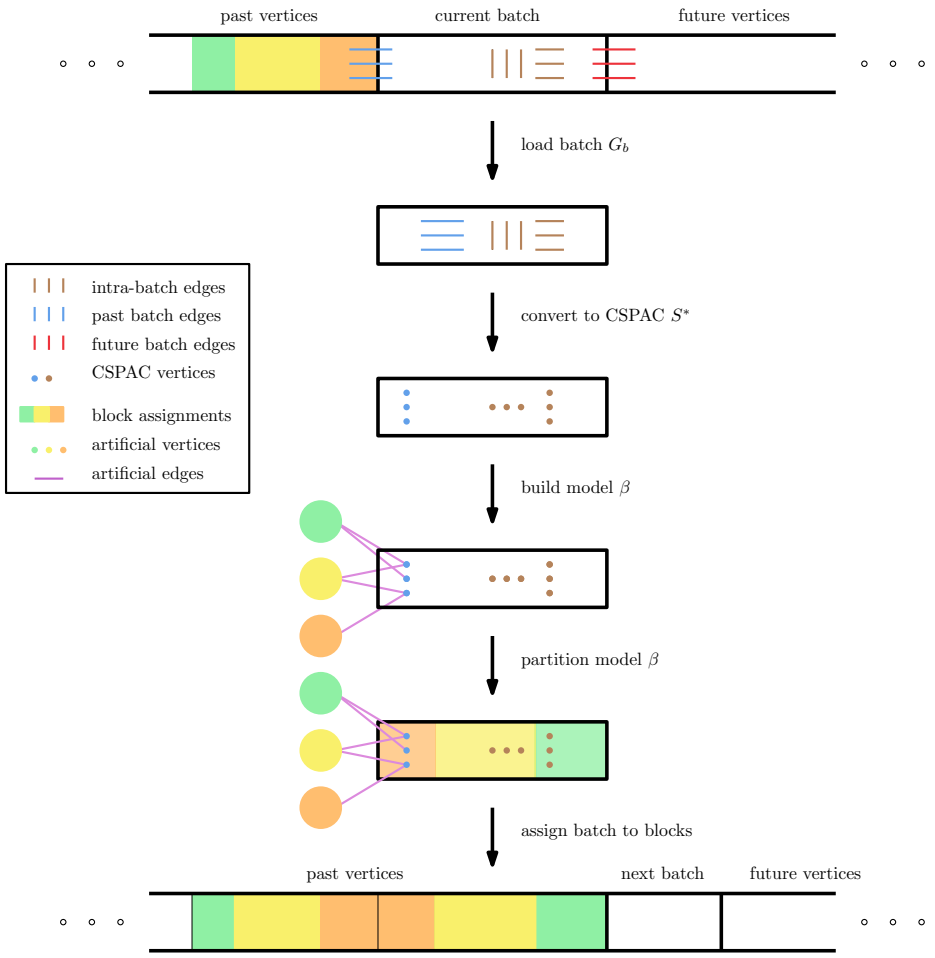


Figure 4.1: Detailed Structure of HeiStreamEdge. The algorithm starts by loading a batch graph of vertices and their corresponding edges to the current batch and previous batches. Next, it converts the batch graph into its corresponding contracted split-and-connect graph. It then builds a meaningful model from this, which is partitioned using a multilevel algorithm. Finally, edges of the loaded batch, which correspond to vertices of the partitioned batch model, are permanently assigned to blocks. The process is repeated for the next batch until the entire graph has been partitioned.

Algorithm 1: Overall Structure of HeiStream Edge

```
while G has not been completely streamed do
    Load batch of vertices G_b ignoring future edges
    Construct CSPAC S* from batch graph G_b
    Build model β from S*
    Run multilevel partitioning on model β
    Permanently assign edges of batch to blocks
```

4.2 CSPAC: Contracted Split-and-Connect Graph

In this section, we discuss the contraction of the Split-and-Connect (SPAC) graph introduced by Li et al. [45]. The contraction results in faster runtime due to the smaller size of the resulting graph model, relative to the uncontracted version. It also provides for better solution quality given our chosen partitioning scheme, due to all edges having equal weight.

In the original version of the SPAC method to compute an edge partition of an undirected, unweighted graph $G = (V, E)$, Li et al. [45] construct a SPAC graph $G' = (V', E')$ as follows:

Split Phase: For each vertex $v \in V$, create a set of $d(v)$ split vertices $S_v := \{v'_1, ..., v'_{d(v)}\}$ in G', $S_v \subset V'$ where $d(v)$ is the degree of v in G.

Connect Phase: For an edge $e = (u, v)$ in G, create a corresponding **dominant** edge $e' = (\mu'_i, v'_j)$ in G', such that $\mu'_i \in S_u$ and $v'_j \in S_v$ and $\omega(e') = \infty$. Both μ'_i and v'_j are connected by one and only one dominant edge e'. Further, connect split vertices in every set S_u to form a path of $d(u)$ vertices and $d(u) - 1$ edges (or $d(u)$ edges if a cycle is built instead of a path), called **auxiliary** edges e'' with $\omega(e'') = 1$.

Figure 4.2 illustrates the conversion of a toy graph G into its corresponding SPAC graph G'. Li et al. [45] propose that to partition the edges of G, perform a vertex partitioning of the SPAC graph G' obtained from G, and then construct the edge partitioning from this vertex partition. Since dominant edges have edge weight set to infinity, it is infeasible for the vertex partitioner to cut these edges. As a consequence, both endpoints of dominant edges are assigned to the same block. Ultimately, we obtain the edge partitioning of the input graph G by transferring the block incident on the endpoints of the dominant edge to the edge in G that induced the dominant edge.

This SPAC graph transformation model for edge partitioning is demonstrated to be highly effective empirically, and shown to have the best provable approximation factor under the same balancing constraints among alternate models [45]. Li et al. [45] proved that with this approach, they can achieve a partition that approximates the optimal solution for the balanced edge partitioning problem within a factor of $\mathcal{O}(\Delta\sqrt{\log n \log k})$, where Δ is the maximum degree of graph G.

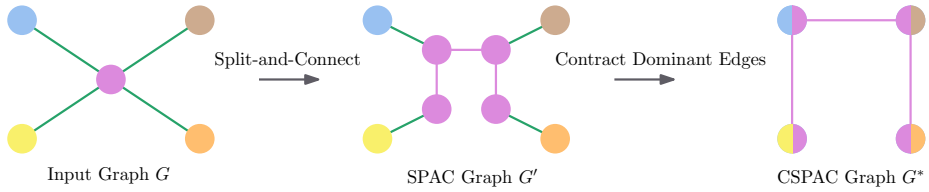


Figure 4.2: Contracted SPAC Graph Construction. Input graph G is converted to its SPAC graph G' by creating $d(v)$ split vertices for each $v \in G$. Each split vertex is identified by the color of the vertex that induces it. All edges of G form unique dominant edges of G' (colored green) between corresponding split vertices. Auxiliary edges form a path between split vertices in G' (colored purple). CSPAC graph G^* is obtained by contracting dominant edges of G'. Each vertex of G^* is formed by a pair of split vertices of different $v \in G$ (colored half-and-half to showcase origin). Edges of G^* are auxiliary edges of G'.

In our work, we develop a contracted version of the SPAC graph by contracting the dominant edges of the SPAC graph as shown in Figure 4.2. In this section, we define the **C**ontracted **SP**lit **A**nd **C**onnect graph, **CSPAC**, and prove that the theoretical approximation bounds for the SPAC graph transformation method also apply to the CSPAC graphs.

Definition 4.2.1 (CSPAC Graph G*)
A CSPAC graph $G^* = (V^*, E^*)$ is obtained from contracting the dominant edges of the SPAC graph $G' = (V', E')$ defined above, as shown in Figure 4.2.

A vertex $u^* \in V^*$ corresponds to some dominant edge $e' = (\mu'_i, v'_j) \in E'$, such that $\mu'_i \in S_u$ and $v'_j \in S_v$, and $u < v$ to avoid duplication. We denote such a vertex as $u^* = [\mu'_i, v'_j]$. Therefore $V^* = \{u^* \mid u^* = [\mu'_i, v'_j], e' = (\mu'_i, v'_j) \in E'\}$.

Further, there is an edge $e^* = (u^*, v^*) \in E^*$ if for $u^* = [\mu'_i, v'_j]$ with $\mu'_i \in S_u$ and $v'_j \in S_v$, there exists a vertex $v^* = [a, b] \in V^*$ with either $\{a \in S_u \vee b \in S_u\}$ or $\{a \in S_v \vee b \in S_v\}$.

The CSPAC construction has the following important properties:

1. Number of vertices $|V^*|$ in G^* is equal to the number of undirected edges $|E|/2$ in G. This is because every edge $e = (u, v) \in E$ induces a dominant edge $e' \in E'$ of the SPAC graph which is contracted to form a vertex $u^* \in V^*$ if $u < v$.

2. As every edge $e \in E$ forms one, and only one, dominant edge $e' \in E'$ of the SPAC graph G', every vertex u^* of G^* corresponds to one, and only one, edge of G. We denote the vertex u^* of G^* as the unique vertex induced by edge $e \in E$ with $G^*(e) = u^*$. Thus, $V^* = \{u^* | u^* = G^*(e)\ \forall e \in E\}$.

3. Edges $e^* = (u^*, v^*) \in E^*$ between vertices u^* and v^* in the CSPAC graph only exist if the two endpoints of e^* share split vertices derived from the same vertex $u \in V$, i.e., if both vertices u^* and v^* were contracted from dominant edges e'_1 and e'_2 of G' with $\mu'_i \in S_u$ and $\mu'_j \in S_u$, incident on e'_1 and e'_2 respectively.

Note that the CSPAC graph G^* is similar to the line graph L constructed from an input graph G. Given a graph G, a line graph L is constructed such that (i) every vertex of L is induced by a unique edge of G and (ii) two vertices of L are adjacent if the edges of G that induced them share an endpoint. Both G^* and L model adjacencies between edges of G, and have the same set of vertices (one for every edge of the G). However, they differ significantly in how their vertices are connected: for a set of vertices that are induced by edges of G that share an endpoint, we have a path in G^* and a clique in L. Because of this, a vertex partitioning of L does not offer the approximation guarantees for edge partitioning that the SPAC graph G', and the CSPAC G^* (proven in this section) can provide.

The CSPAC graph G^* can be constructed from the input graph G in two ways: (i) construct the SPAC graph and contract the dominant edges or (ii) construct the CSPAC graph directly from G. We provide algorithms for these constructions later in this section.

Next, similar to the SPAC method, in order to compute a valid edge partition of G, we compute a valid vertex partition of G^* using a vertex partitioner. From property 1 of the CSPAC graph, we know that every vertex of the CSPAC graph G^* corresponds to an edge of G, and from property 2, that this edge is unique. Thus, a vertex partitioning of G^* directly gives us an edge partition of G with the following definition:

Definition 4.2.2 (Edge Partition of G from Vertex Partition of G*)
Given an input graph $G = (V, E)$, and a CSPAC graph $G^* = (V^*, E^*)$ induced by G, assume you have a vertex partition $vp(G^*)$ of G^*, wherein every vertex $u^* \in V^*$ is assigned to block $i \in \{0, ..., k-1\}$ and k is the number of blocks of the edge partition. The edge partition $ep(G)$ of G is obtained as follows:

$$ep(G[e]) = vp(G^*)[u^*] = i \; \forall e \in E \wedge u^* = G^*(e).$$

As every vertex u^* in G^* represents a unique edge in G, $V^* = \{u^* | u^* = G^*(e) \; \forall e \in E\}$, and $vp(G^*)$ is a valid vertex partition with balance constraints, $ep(G)$ is a valid edge partition of G.

Given this construction and vertex partitioning of the CSPAC graph, we now prove that the theoretical approximation bounds for the SPAC graph method apply to the CSPAC method as well. We prove modifications or extensions of theorems originally presented in Li et al. [45] to show that the vertex partitioning of the CSPAC graph approximates the optimal edge partitioning solution of the input graph within a factor of $\mathcal{O}(\Delta\sqrt{\log n \log k})$.

Before proving the theorems, we need the following important lemma:

Lemma 4.2.1 (Edges of G* are auxiliary edges of G')
Given an input graph $G = (V, E)$, assume $G' = (V', E')$ is a SPAC graph constructed from G, and $G^* = (V^*, E^*)$ is the CSPAC graph obtained from contracting dominant edges of G'. Every edge $e^* \in E^*$ corresponds to a unique auxiliary edge in G', and $|E^*| = |\{e'' \in E' : e'' \text{ is an auxiliary edge}\}|$.

Proof. Every vertex $u^* \in V^*$ is obtained from contracting a dominant edge in G'. From the construction of G', we know that each dominant edge $e' = (\mu'_i, \nu'_j)$, with $\mu'_i \in S_u$ and $\nu'_j \in S_v$ is uniquely induced by a corresponding edge $e = (u, v) \in E$ of the input graph G.

From property 3 of the CSPAC graphs mentioned earlier, we know from construction that an edge $e^* = (u^*, v^*) \in E^*$ exists only if u^* and v^* were contracted from dominant edges e'_1 and e'_2 of G' with $\mu'_i \in S_u$ incident on e'_1 and $\mu'_j \in S_u$ incident on e'_2. In other words, both u^* and v^* must be obtained from a dominant edge that had two split vertices μ of the same vertex u in common. In G', such edges between two split vertices are precisely the auxiliary edges.

Next, we show that every edge $e^* \in E^*$ corresponds to a unique auxiliary edge in G'. Assume towards a contradiction that there are two edges $e^*_1 = (u^*, v^*) \in E^*$ and $e^*_2 = (a^*, b^*) \in E^*$ that correspond to the same auxiliary edge in G'. Let this auxiliary edge be $e'' = (\mu'_i, \mu'_l) \in E'$ that exists between two split vertices $\mu'_i, \mu'_l \in S_u$ of vertex $u \in V$. In G', by construction, μ'_i can have an auxiliary edge to another split vertex of u, say $\mu_s \in S_u$, or a dominant edge to a split vertex $\nu'_j \in S_v$ of a different vertex $v \in V$. Similarly, μ'_j can have an auxiliary edge to either another split vertex of u, say $\mu_p \in S_u$, or a dominant edge to a split vertex $w'_j \in S_w$ of another vertex $w \in V$. For e'' to be the edge $e^*_1 = (u^*, v^*)$ in G^*, we must have that $u^* = [\mu'_i, \nu'_j]$ and $v^* = [\mu'_l, w'_j]$ as these vertices can only be obtained from contracting the dominant edges incident on μ'_i and μ'_l respectively. Likewise, for e'' to be the edge $e^*_2 = (a^*, b^*)$ in G^*, we must have that $a^* = [\mu'_i, \nu'_j]$ and $b^* = [\mu'_l, w'_j]$ as they are the only dominant edges incident on μ'_i and μ'_l respectively. However, then $u^* = a^*$ and $v^* = b^*$. Therefore, $e^*_1 = e^*_2$ and we have arrived at a contradiction. So, every edge $e^* \in E^*$ must correspond to a unique auxiliary edge in G'.

Since in G^*, all dominant edges of G' are contracted to unique vertices, and all edges in G^* are unique auxiliary edges of G', it directly follows that $|E^*| = |\{e'' \in E' : e'' \text{ is an auxiliary edge}\}|$.

To see why, consider the following: if $|E^*| > |\{e'' \in E'\}|$ then there are two edges $e^*_1, e^*_2 \in E^*$ which correspond to the same auxiliary edge in E' which violates uniqueness. If $|E^*| < |\{e'' \in E'\}|$ then we have not connected two vertices in V^* that share two split vertices $\mu'_i, \mu'_j \in S_u$ of the same vertex $u \in V$ that had an auxiliary edge $e'' = (\mu'_i, \mu'_j)$ in G', which violates the construction of G^*. ■

With Lemma 4.2.1, we are ready to prove Theorem 4.2.1.

Theorem 4.2.1

Given an input graph G, assume G' is a SPAC graph constructed from G, and G^* is the CSPAC graph obtained from contracting dominant edges of G'. Let $vp(G^*)$ be a valid vertex partition of the CSPAC graph G^*, and let $ep(G)$ be a valid edge partition of G that is obtained from $vp(G^*)$ using the procedure in Definition 4.2.2.

Then, the edge partition cost of $ep(G)$, i.e., number of vertex replicas of G, is less than or equal to the vertex partition cost of $vp(G^*)$, i.e., total edge cut of G^*, denoted as

$$cost(ep(G)) \leq cost(vp(G^*)).$$

Proof. First, note that from Lemma 4.2.1, every edge $e^* = (u^*, v^*) \in E^*$ in G^* is an auxiliary edge $e'' \in E'$ of the SPAC graph G'. When we obtain a valid vertex partitioning $vp(G^*)$ of G^*, each cut edge e^* therefore corresponds to an auxiliary edge. Let $G^*(e_1) = u^*$, with $vp(G^*[u^*]) = i_1$ and $G^*(e_2) = v^*$, with $vp(G^*[v^*]) = i_2$, for $e_1, e_2 \in E$ and $i_1, i_2 \in \{0, 1, ..., k-1\}$. When translating $vp(G^*)$ to $ep(G)$, we set $ep(G[e]) = vp(G^*[G^*(e)]) = vp(G^*[u^*])$ as described in Definition 4.2.2. Therefore, every cut edge $e^* = (u^*, v^*)$ in $vp(G^*)$ results in the two edges e_1 and e_2 inducing u^* and v^* respectively to get assigned to different blocks i_1 and i_2 in $ep(G)$. Since e^* is an auxiliary edge and auxiliary edges exist between two split vertices $\mu'_i, \mu'_j \in S_u$ of the same vertex $u \in V$, both edges e_1 and e_2 must have u as one of their end points by construction of G'. As e_1 and e_2 are assigned to different blocks in $ep(G)$, vertex u is replicated in those two blocks. In conclusion, every edge cut in $vp(G^*)$ results in one additional replica in the vertex cut of $ep(G)$.

However, this is not true if the path between split vertices in G' is a cycle of more than two vertices. Say, for instance, vertex $u \in V$ has degree $d(u) = n$ and the n split vertices of u induce a cycle of auxiliary edges in G'. This also results in a cycle between n vertices in G^* due to the contraction of n dominant edges induced by edges from u to its n neighbors in G, and the unique correspondence of edges in E^* to auxiliary edges $e'' \in E'$. If and when the first of these auxiliary edges is cut in $vp(G^*)$, say $e_1^* = (u^*, v^*)$, we must have one additional edge of the cycle that gets cut, say, $e_2^* = (v^*, w^*)$. This is because if not, then u^* and w^* are in the same block i_1 of $vp(G^*)$ and v^* and w^* are in the same block i_2 of $vp(G^*)$. Since w^* can only be assigned to a unique partition in $vp(G^*)$, $i_1 = i_2$ and by transpose, $vp(G^*[u^*]) = vp(G^*[w^*]) = vp(G^*[v^*])$ which contradicts that $e_1^* = (u^*, v^*)$ is a cut edge. However, only the first edge cut e_1^* contributes to an additional replica in $ep(G)$. The additional cut edge e_2^* does not result in the number of replicas of u getting increased in $ep(G)$, as it is cut in addition to only the first edge cut in the cycle, thereby putting one split vertex of u in block i_2, and the rest in i_1. Concurrently, one edge of G is assigned to i_2 in $ep(G)$ and the rest in i_1. It is important to note that this additional edge cut occurs at most once per cycle, for the first edge cut. Any other cut edges in the cycle result in one additional replica of u in $ep(G)$, as is the case for paths.

Thus, in summary, the edge partition cost of $ep(G)$, i.e., number of vertex replicas of G, is less than or equal to the vertex partition cost of $vp(G^*)$, i.e., total edge cut of G^*, denoted as $cost(ep(G)) \leq cost(vp(G^*))$. ■

Before beginning the next proof, we explain how to obtain the CSPAC graph G^* directly from the input graph G, without first constructing the intermediate SPAC graph G' and then contracting the dominant edges.

Definition 4.2.3 (Direct construction of G* from G)
Given an input graph $G = (V, E)$, construct a CSPAC graph $G^* = (V^*, E^*)$ as follows:

1. *For every edge $e = (u, v) \in E$ ($u < v$), create a vertex $u^* \in V^*$. We ignore backward edges (from v to u) to avoid duplication of vertices in G^* for reverse edges of undirected graphs. This provides us with all vertices of G^*, with $|V^*| = |E|/2$.*

2. *Let T_u denote the set of all edges $e \in E$ which have vertex $u \in V$ as a common endpoint, i.e., $T_u := \{e = (u, v) \in E \mid v \in N(u)\}$. Further, let T_u^* be the set of all vertices in G^* that are induced by edges $e \in T_u$, i.e., $T_u^* = \{u^* \in V^* | u^* = G^*(e), e \in T_u\}$. We build a path between each vertex $u^* \in T_u^*$. These form the edges $e^* \in E^*$ of G^*.*

Note here that step 1 is effectively identical to contracting dominant edges $e' \in E'$ of the SPAC graph which are induced by some $e = (u, v) \in E$, with $u < v$. Additionally, step 2 results in the same edge set as if we contracted G'. Recall that $S_u \in G'$ contained one split vertex of u for every neighboring edge of u. Likewise, T_u^* contains one vertex u^* for every neighboring edge of u. Thinking in terms of contraction, every vertex $u^* \in T_u^*$ is obtained by contracting a dominant edge incident on a corresponding split vertex $\mu_i' \in S_u$. A path between vertices in T_u^* is precisely a path of auxiliary edges between S_u in G'. With this, we are ready to tackle the next proof.

Theorem 4.2.2
For any input graph G, there exists an optimal CSPAC graph W derived from G, such that the cost of an optimal vertex partition $vp_{opt}(W)$ of W is equivalent to the cost of the optimal edge partition $ep_{opt}(G)$ of G, i.e., $cost(ep_{opt}(G)) = cost(vp_{opt}(W))$. This particular CSPAC graph W is referred to as the ***contracted dual graph*** *for G.*

Proof. We provide a proof by constructing a CSPAC graph W from G, where it holds that $cost(ep_{opt}(G)) = cost(vp_{opt}(W))$. Assume that we have an optimal edge partition $ep_{opt}(G)$ of G. To generate the contracted dual graph W from G, first create a vertex $u^* = W(e) \in V^*$ for every edge $e \in E$. Next, partition the set $T_u := \{e = (u, v) \in E \mid v \in N(u)\}$ for every $u \in V$ into at most k subsets, $T_u(i), i = 1...k$, where $T_u(i) = \{e \in T(u) \mid ep_{opt}(G[e]) = i\}$. In other words, each $T_u(i)$ represents the set of edges incident on vertex u that are assigned to block i in the optimal edge partition of G. Additionally, define $T_u^*(i) = \{u^* \in V^* | u^* = W(e), e \in T_u(i)\}$ which is the set of vertices in W that are induced by edges in each $T_u(i)$.

Now, to build the edge set of W, first insert edges between vertices u^* in every non-empty set $T_u^*(i)$ to form a path P_i. Thus, we build up to k paths between vertices generated in W by edges in each $T_u(i)$. Next, connect path P_1 to P_2 to ... P_k to form a longer path. Note that this is still compatible with Definition 4.2.3, as the intra-$T_u^*(i)$ paths and the path between each P_i together form a longer path between each $u^* = W(e) \in V^*$ for every $e \in T_u$.

Assume we have an optimal vertex partition solution $t_{opt}(W)$ of W. From this, we construct a corresponding edge partition $ep_{t_{opt}}(G)$ using Definition 4.2.2. From Theorem 4.2.1, we know that the cost of the edge partition $ep_{t_{opt}}(G)$ is less than or equal to the cost of the vertex partition $t_{opt}(W)$: $cost(ep_{t_{opt}}(G)) \leq cost(t_{opt}(W))$.

From $ep_{opt}(G)$, we can construct a valid vertex partition $vp_t(W)$ of W by simply cutting only those edges that run between paths P_i, i.e., the edges connecting the different $T_u^*(i)$

sets. We do not cut any intra-$T_u^*(i)$ paths. Using this construction, it is easy to see that $cost(ep_{opt}(G)) = cost(vp_t(W))$.

Finally, by contradiction we show that $vp_t(W)$ has minimal cost among all other valid vertex partitions of W. Assume there is another vertex partition $vp_{t'}(W)$ such that $cost(vp_{t'}(W)) < cost(vp_t(W))$. Using Definition 4.2.2, we construct an edge partition of $ep_{t'}(G)$ of G from $vp_{t'}(W)$. From Theorem 4.2.1, we have that $cost(ep_{t'}(G)) \leq cost(vp_{t'}(W))$. So, $cost(ep_{t'}(G)) \leq cost(vp_{t'}(W)) < cost(vp_t(W)) = cost(ep_{opt}(G))$. This contradicts the optimality of $ep_{opt}(G)$. Therefore, $cost(vp_t(W)) = cost(vp_{t_{opt}}(W))$ and it follows that $cost(ep_{opt}(G)) = cost(vp_{t_{opt}}(W))$. ■

Theorem 4.2.3
Assume W is the contracted dual graph of input graph G. Let G^* represent any of the other CSPAC graphs generated from G with edges obtained from some path ordering between vertices in each $T_u^*, u \in V$. Further, let $vp_{opt}(G^*)$ be the optimal vertex partition of G^*, $vp_{opt}(W)$ be the optimal vertex partition of W, Δ be the maximum vertex degree of graph G. Then it holds that:

$$cost(vp_{opt}(G^*)) \leq (\Delta - 1)cost(vp_{opt}(W))$$

If we allow paths constructed between vertices in each T_u^* to be cycles, then

$$cost(vp_{opt}(G^*)) \leq (\Delta)cost(vp_{opt}(W))$$

.

Proof. For any G^* constructed from G, when connecting vertices in G^* with paths between vertices in T_u^* for each $u \in V$, the order in which vertices appear on the path is arbitrary. However, the set of vertices of G^* is identical to the set of vertices of W, where in both cases, vertices are induced by edges of the input graph G.

Assume we have an optimal vertex partition $vp_{opt}(W)$ of W. This can be converted into a valid vertex partition $vp_t(G^*)$ of G^* by cutting an edge $e^* \in E^*$ of G^* if its two incident end points are in different vertex partitions of $vp_{opt}(W)$. If we generate a path in G^* between vertices of some T_u^*, the path has a length of $d(u) - 1$, since $|T_u^*| = d(u)$, and each $u^* \in T_u^*$ maps to a unique $e = (u, v) \in E, v \in N(u)$. Alternatively, if we generate a cycle between vertices of some T_u^*, we have exactly $d(u)$ edges in the cycle. Therefore, for an arbitrary G^*, if a path of edges corresponding to some T_u^* is cut, at most $\Delta - 1$ edges are cut and if we have a cycle, then at most Δ edges are cut.

When converting $vp_{opt}(W)$ to $vp_t(G^*)$, a path between vertices in G^* of some T_u^* is cut only when the corresponding path is also cut for the same set of vertices in W. Therefore, the vertex partition cost of $vp_t(G^*)$ is at most $(\Delta - 1)$ times the vertex partition cost of $vp_{opt}(W)$ (or at most (Δ) times if the path is a cycle). Therefore, $cost(vp_t(G^*)) \leq (\Delta - 1)cost(vp_{opt}(W))$ or $cost(vp_t(G^*)) \leq \Delta cost(vp_{opt}(W))$ if the path is a cycle. Since $cost(vp_{opt}(G^*)) < cost(vp_t(G^*))$, the theorem is proved. ■.

Corollary 4.2.1
For any CSPAC graph G^* constructed from input graph G, there exists a polynomial time vertex partition solution $vp(G^*)$ such that, $cost(vp(G^*)) \leq \mathcal{O}(\sqrt{\log n \log k}) * \Delta * cost(vp_{opt}(W))$, where W is the contracted dual graph of G, and $vp_{opt}(W)$ is the optimal vertex partition of W.

Proof. This directly follows from the Theorem 1.1 in [42] which states that there exists a polynomial time algorithm for the balanced vertex partition problem with an approximation factor of $\sqrt{\log n \log k}$. So, assuming the algorithm gives us the vertex partition $vp(G^*)$, we have that

$$cost(vp(G^*)) \leq \mathcal{O}(\sqrt{\log n \log k}) \cdot cost(vp_{opt}(G^*)).$$

Then from Theorem 4.2.3, we get

$$\begin{aligned} cost(vp(G^*)) &\leq \mathcal{O}(\sqrt{\log n \log k}) \cdot cost(vp_{opt}(G^*)) \\ &\leq \mathcal{O}(\sqrt{\log n \log k}) \cdot (\Delta) \cdot cost(vp_{opt}(W)). \end{aligned}$$

■

Theorem 4.2.4
Let G be the input graph, let W be the contracted dual graph of G, and G^* be any CSPAC graph constructed from G. Further, let $ep(G)$ be a balanced edge partition of G obtained from a valid balanced vertex partition $vp(G^*)$ of G^* that satisfies Corollary 4.2.1. It holds that $cost(ep(G)) \leq \mathcal{O}(\sqrt{\log n \log k}) \cdot \Delta \cdot cost(ep_{opt}(G))$, and consequently there exists a polynomial algorithm for the balanced edge partition problem with an approximation factor of $\mathcal{O}(\Delta\sqrt{\log n \log k})$.

Proof. We have an input graph G, a corresponding CSPAC graph G^*, a valid balanced vertex partition $vp(G^*)$ of G^* that satisfies Corollary 4.2.1, and an optimal balanced vertex partition $vp_{opt}(G^*)$ of G^*. Construct a valid balanced edge partition $ep(G)$ from $vp(G^*)$. Further, let W be the contracted dual graph of G, and $vp_{opt}(W)$ be the optimal balanced vertex partition of W. Then it follows that,

$$\begin{aligned} cost(ep(G)) &\leq cost(vp(G^*)) && \text{Theorem 4.2.1} \\ &\leq \mathcal{O}(\sqrt{\log n \log k}) \cdot cost(vp_{opt}(G^*)) && \text{Corollary 4.2.1} \\ &\leq \mathcal{O}(\sqrt{\log n \log k}) \cdot (\Delta) \cdot cost(vp_{opt}(W)) && \text{Theorem 4.2.3} \\ &\leq \mathcal{O}(\sqrt{\log n \log k}) \cdot (\Delta) \cdot cost(ep_{opt}(G)) && \text{Theorem 4.2.2} \end{aligned}$$

■

We have therefore shown, by proving the above theorems that were originally proven for the SPAC graph G' by Li et al. [45], that there exists a polynomial time algorithm involving the vertex partitioning of a CSPAC transformation graph that approximates the balanced edge partitioning problem up to a factor of $\mathcal{O}(\Delta\sqrt{\log n \log k})$.

Before closing this section, we provide a proof regarding the number of auxiliary edges of a SPAC graph G' of G where the auxiliary edge paths are cycles induced by split vertices, which ultimately refers to the number of edges of the CSPAC graph G^*.

Theorem 4.2.5
Let $G = (V, E)$ be the input graph, let $G' = (V', E')$ be the SPAC graph of G where the paths between split vertices are cycles, and let $G^* = (V^*, E^*)$ be the CSPAC graph constructed by contracting dominant edges of G'. The number of edges of a SPAC graph G' is $|E'| = (3 * m) - 2 * (n_{d1} + n_{d2})$ where n_{d1} is the number of degree 1 vertices in V, and n_{d2} is the number of degree 2 vertices in V. Then, the number of auxiliary edges of $G' = |E'| - m = |E^*|$.

Proof. Let $G = (V, E)$ be the input graph with n vertices and m edges. When constructing G', we first add $d(u)$ split vertices in a set S_u for every $u \in V$, $S_u \subset V'$. In the connect phase, we add dominant edges for every edge $e \in E$ to S', resulting in m edges in S'. Next, since the auxiliary edge paths are chosen to be cycles, we add up to $d(u)$ auxiliary edges for every set $S_u \in V'$, $u \in V$. If every vertex $u \in V$ has $d(u) \geq 3$, we would get exactly $d(u)$ auxiliary edges for every $S_u \in V'$, $u \in V$. Summing up over all $u \in V$, we would have $\sum_{u \in V} d(u) = 2 * m$ auxiliary edges. However, vertices $v \in V$ with $d(v) = 1$ or $d(v) = 2$ do not add $d(v)$ auxiliary edges among their split vertices S_v. Vertices with $d(v) = 1$ contribute no auxiliary edges, as they are only split into one split vertex which has a dominant edge incident on it. Vertices with $d(v) = 2$ contribute exactly 1 auxiliary edge between the two split vertices they induce. Let n_{d1} be the number of degree 1 vertices in V, and let n_{d2} be the number of degree 2 vertices in V. We can now take away the miscounted auxiliary edges for $d(v) = 1$ and $d(v) = 2$ vertices by subtracting $2 * (n_{d1} + n_{d2})$ from $2 * m$. This gives us $2 * m - 2 * (n_{d1} + n_{d2})$ auxiliary edges in E'. In total, we have $|E'|$ = number of dominant edges + number of auxiliary edges = $m + 2 * m - 2 * (n_{d1} + n_{d2})$ = $3 * m - 2 * (n_{d1} + n_{d2})$. Then, from Lemma 4.2.1, it follows that $E^* = |E'| - m$. ■

Over the course of the various proofs, we also demonstrated fundamental ideas and definitions that guide the algorithms we implemented to construct a SPAC and CSPAC graph of a given input graph G. We now describe our algorithms to achieve this.

4.2.1 SPAC Graph Construction and Contraction

Our first approach to build a CSPAC graph was to construct a SPAC graph from the input graph, and then to contract the dominant edges. Let the graph to be edge partitioned be $G = (V, E)$. Then, as Figure 4.1 illustrates, our input graph refers to the graph of the current batch of our buffered approach $G_b = (V_b, E_b)$. This batch graph G_b is built as follows: during IO, we read δ vertices (our chosen buffer size) and their adjacency lists from the graph's data stream. Then, V_b consists of the δ vertices of the current batch, remapped from 0 to $\delta - 1$, as well as p vertices of past batches that have edges to the current batch, remapped from δ to $\delta + p - 1$. Likewise, E_b consists of edges to vertices in the current batch and to vertices of previous batches, i.e.,

Algorithm 2: SPAC Construction

input : Graph G_b, rev_edges
output: SPAC Graph S

```
1  foreach vertex u ∈ V_b do
2      foreach edge e ∈ E_b(u) = {e_u ... e_u + d(u) − 1} do
3          S.insertvertex(u')                    // u' = split vertex
4          S.insertDominantEdge(u', rev_edges[e], ∞)
5          if d(u) > 1 then
               // compute target vertices for aux. edge
6              u'_next := e − e_u + 1 mod d(u)
7              u'_prev := e − e_u − 1 mod d(u)
8              S.insertAuxiliaryEdge(u', u'_next, 1)
9              if u'_prev ≠ u'_next then
10                 S.insertAuxiliaryEdge(u', u'_prev, 1)
```

$E_b = \{(u, v) \in E \mid u \in \text{Current Batch} \wedge v \in \text{Current or Previous Batch}\}$. We ignore edges to future batches for consistency as explained in more detail in Section 4.3.

We store our batch graph G_b in a standard adjacency array representation. While building the current batch as a graph G_b, we fill an array rev_edges of size $|E_b| = m_b$ that stores for every edge $e = (u, v) \in E_b$, the corresponding index of $e_{rev} = (v, u) \in E_b$ in the adjacency array of edges. We find rev_edges in linear time, by maintaining consistency in edge insertion during IO. In particular, when visiting edge $e = (u, v)$ during IO, we insert both $e = (u, v)$ and $e_{rev} = (v, u)$ into E_b. Here, for the reverse edge, we already update $rev\_edges[e_{rev}] = e$. Later, when visiting edges of our batch for a second time, we update $rev\_edge[e] = e_{rev}$ upon visiting e_{rev}. This allows us to eventually access the reverse edge in $\mathcal{O}(1)$ time in our algorithm.

We use the algorithm shown in Algorithm 2 to construct our SPAC graph S for a given input graph G_b. The algorithm builds S as follows: for the split phase, recall that a SPAC graph contains a set S_u of $d(u)$ split vertices for each vertex $u \in V_b$. We thus simply pass over all vertices u of G_b, and for each outgoing edge $e = (u, v) \in E_u$ we insert a split vertex u' into S (line 3).

For the connect phase, which occurs concurrently here, as we visit all edges from u, we look through edges $e_u, ..., e_{d(u)-1}$, where e_u is the index of the first outgoing edge of u in the adjacency array of edges. For every edge $e = (u, v)$, we insert a dominant edge with edge-weight ∞, from u' to $rev\_edges[e]$ into S (line 4). Recall that dominant edges are induced by undirected edges $\{u, v\}$ in G, and they connect split vertices in S_u and S_v such that each split vertex is incident to precisely one (undirected) dominant edge. To this end, coordination is required between split vertices S_u of u and S_v of v. When constructing a dominant edge for $e = (u, v)$, we achieve this coordination with the help of $rev\_edges[e]$, which provides us with the edge ID $e_{rev} = (v, u)$ of the corresponding reverse edge. Ultimately,

this gives us a dominant edge in S for each edge of the input graph, while maintaining the unique mapping of dominant edges to split vertex pairs, through pairing edge IDs in the adjacency array of edges.

Finally, to insert the auxiliary edges, we define a path to construct between each split vertex as required in a SPAC graph construction. In our case, for easier implementation, auxiliary edges with edge-weight 1 are inserted between split vertices $v \in S_v$ to connect them to an induced cycle. We compute the ID of the next (u'_{next}) and previous (u'_{prev}) split vertex in the cycle for vertices of degree greater than 1 (line 6 and 7). Here, $u'_{next} = e - e_u + 1 \; mod \; d(u)$, where e_u is the edge ID of the first outgoing edge from u in the adjacency array of edges. Using modulo, we obtain a unique identifier for every auxiliary edge in the cycle of S_u vertices from $0...d(u) - 1$. For vertices of degree 2, $u'_{next} = u'_{prev}$, and thus we only insert one auxiliary edge (u, u'_{next}) without loss of generality. Otherwise, for vertices with degree 3 or above, we always have two auxiliary edges (u', u'_{next}) and (u', u'_{prev}).

After visiting all vertices, we have constructed a complete and correct SPAC graph S from G_b. The functions `insertVertex`, `insertDominantEdge` and `insertAuxiliaryEdge` all occur in $\mathcal{O}(1)$ time as they all insert the passed values into some array. Thus, the overall runtime of Algorithm 2 is $\mathcal{O}(n_b + m_b)$ where $n_b = |V_b|$ and $m_b = |E_b|$. This procedure was parallelized by Schlag et al. [62], and proven to produce a valid SPAC graph from a graph G with a runtime of $\mathcal{O}(m/p + \log p)$ for p processors.

The next step is to contract the dominant edges to build the CSPAC graph from the SPAC graph we have now constructed. To achieve this efficiently, we assign all pairs of split vertices that share a dominant edge into a unique block $i \in \{0...m_b/2 - 1\}$, one for each undirected edge of G_b. Then, we contract vertices in every block to form a quotient graph, where vertices are the contracted dominant edges, and edges are the cut edges between blocks, which are the auxiliary edges between split vertices. We compute the contraction in an additional $\mathcal{O}(n_b + m_b)$ time. This gives us the desired CSPAC graph S^*.

4.2.2 Direct CSPAC Construction

A naturally faster alternative to constructing a SPAC graph and then contracting the dominant edges is to directly obtain the CSPAC graph S^* from G_b, thereby skipping the construction of the SPAC graph S. Algorithm 3 shows how we perform this construction. Here, c refers to an array that stores the corresponding vertex ID in S^* for every edge in E_b. These are determined in the order in which the edges appear in the adjacency array.

The algorithm to directly obtain S^* from G_b works as follows: for every vertex $u \in V_b$, we visit all its outgoing edges $e = (u, v) \in E_b$. We only consider edges where $u < v$ to avoid symmetry. First, we insert a vertex $u^* = S^*(e)$ into S^*, which is the unique vertex induced by e in S^*. Recall that u^* would otherwise be obtained by contracting a dominant edge (u', v') between some $u' \in S_u$ and $v' = rev\_edges[e] \in S_v$ of the SPAC graph S.

Algorithm 3: CSPAC Construction

input : Graph G_b, rev_edges, c
output: CSPAC Graph S^*

1 **foreach** *vertex* $u \in V_b$ **do**
2 **foreach** *edge* $e = (u, v) \in E_b(u) = \{e_u ... e_u + d(u) - 1\}$ **do**
3 **if** *u < v* **then**
4 $e_v := G_b$.getFirstEdge(*v*)
5 $u^* := c[e]$ // u^* = contracted split vertex
6 S^*.insertVertex(u^*) // induced by current edge e
   // $u^* := [u', v']$ for some $u' \in S_u, v' \in S_v$
7 **if** *d(u) > 1* **then**
   // compute target vertices for aux. edge
   // from u' among S_u
8 $u^*_{next} := c[(e - e_u + 1)\ mod\ d(u) + e_u]$
9 $u^*_{prev} := c[(e - e_u - 1)\ mod\ d(u) + e_u]$
10 S^*.insertAuxiliaryEdge(u^*, u^*_{next}, *1*)
11 **if** $u*_{prev} \neq u^*_{next}$ **then**
12 S^*.insertAuxiliaryEdge(u^*, u^*_{prev}, *1*)
13 **if** *d(v) > 1* **then**
   // compute target vertices for aux. edge
   // from v' among S_v
14 $v^*_{next} := c[(rev\_edges[e] - e_v + 1)\ mod\ d(v) + e_v]$
15 $v^*_{prev} := c[(rev\_edges[e] - e_v - 1)\ mod\ d(v) + e_v]$
16 S^*.insertAuxiliaryEdge(u^*, v^*_{next}, *1*)
17 **if** $v*_{prev} \neq v^*_{next}$ **then**
18 S^*.insertAuxiliaryEdge(u^*, v^*_{prev}, *1*)

Next, we must insert edges from u^* to its neighbors in S^*. Earlier in this section, we demonstrated that the edges of the CSPAC graph are the auxiliary edges of the SPAC graph S. To add these edges here, we need to consider the auxiliary edges which would be connected to both the split vertices u' and v' of the SPAC graph S that would otherwise be contracted to obtain u^*. In order to do so, we consider both endpoints of the current edge $e = (u, v)$. To obtain the auxiliary edges from u', we do the following: first, we determine if u has any other edges (line 7), if not, then there is no auxiliary edge, as we are currently at the only edge incident on u. Otherwise, we compute the next and previous split vertices in the cycle of S_u as we did for the SPAC graph conversion (lines 8 and 9). However, unlike the SPAC graph, here we require the ID of the contracted dominant edge connected to u^*, and not of the split vertices themselves. In other words, we need $u^*_{next} = [u'_{next}, a']$ which is obtained from contracting the dominant edge between u'_{next} and some $a' \in S_a$, and $u^*_{prev} = [u'_{prev}, b']$ which is obtained from contracting the dominant edge between u'_{prev} and some $b' \in S_b$.

We solve this problem by computing $u'_{next} = (e - e_u + 1)\ mod\ d(u)$ where e_u is the edge ID of the first edge from u in the adjacency array of edges. Next, we add back e_u to this to fetch the exact location of the edge in the adjacency array of edges that induces the unique dominant edge incident on u'_{next}. Finally, we get the vertex ID of u^*_{next} by passing this edge ID into the array c (line 8). Similarly, we obtain u^*_{prev}.

This entire procedure to get the auxiliary edges from u is repeated for the other endpoint v. However, we do not have access to the edge ID of the edge (v, u), which was simply the current e for (u, v). This is therefore obtained from the array rev_edges which stores the edge ID of the reverse edge for every $e \in E_b$. Also, we get e_v, i.e., the edge ID of the first edge from v in the adjacency array of edges, from the pointer to this position for vertex v. From here, we compute, for example, $v'_{next} = (rev\_edges[e] - e_v + 1)\ mod\ d(v)$ in the same manner as u'_{next}, and the rest follows analogously (line 16 and 17).

The overall runtime of direct CSPAC construction with Algorithm 3 is the same as SPAC construction with Algorithm 2, that is, $\mathcal{O}(n_b + m_b)$. Using direct conversion we save the additional cost factor $\mathcal{O}(n_b + m_b)$ of contraction.

4.3 Model Construction

In this section, we provide an overview of the graph model β that we construct by extending the CSPAC graph S^*. It is this graph β that we subsequently partition using an adaptation of the HeiStream vertex partitioning scheme described in Section 4.4. We begin by motivating the need for this extension and then offer various configurations for achieving it.

To see why we need to extend S^* to β, it is important to note, as Figure 4.1 illustrates, that we ignore edges to future batches in our model altogether. This is done for two primary reasons: first, considering edges in the input graph $G = (V, E)$ in a fixed order of intra-batch edges and past batch edges only ensures consistency with respect to mapping vertex IDs u^* in the per-batch CSPAC graph S^* to their corresponding global edge IDs in E. This is because, with this ordering, no edge $e = (u, v) \in E$ is stored in two separate batch graphs G_b, and consequently, every edge is processed as a vertex in a batch CSPAC graph S^* exactly once. Secondly, while we have information about block assignment decisions for edges visited in previous batches, we have no such information at hand about future batches. This allows us to exploit block assignment decisions made in previous batches, to provide a more global view to our vertex partitioner while in a streaming setting. In this section, we demonstrate how we exploit past assignment decisions in our model to improve solution quality.

We initialize β as the CSPAC graph S^* for the current batch. If the current batch is not the first batch, we add k artificial vertices to the model which represent the k partition blocks in their current state, i.e., filled with edges assigned to them from previous batches. The weight of each artificial vertex i is set to the weight of the block V_i.

In HeiStream for vertex partitioning, a vertex of the current batch is connected to an artificial vertex i, if it has a neighbor from a previous batch that has been assigned to block V_i.

However, we do not have such direct access to block assignments in edge partitioning: during IO, we only know the vertices incident on the edge from the current batch to the previous batch. To solve this limitation, we maintain an array B of size n throughout the streaming process which stores for each vertex $u \in V$, the blocks that were assigned to edges incident on it. Thus, when we assign a partition ID i to an edge (u, v) in a batch, we insert i to $B[u]$ and $B[v]$. Later in this section, we discuss various configurations for B that provide for a runtime, memory and solution-quality trade-off.

Now, we connect vertices of the CSPAC graph S^* to artificial vertices as follows: during the construction of S^*, we store for every vertex $u^* \in S^*$ induced by edge $e_b = (u_b, v_b) \in E_b$, the original vertex IDs corresponding to the edge $e_b = e = (u, v) \in E$ of input graph G. After inserting u^* and its incident auxiliary edges $e^* = (u^*, v^*)$ in S^*, we insert an artificial edge from u^* to each artificial vertex i such that $i \in B[u]$. Note that for any vertex $u \in V$, $B[u] \neq \emptyset$ if and only if u is a vertex of a previous batch. This is because no edge to a vertex v of a future batch would have been considered previously. Further, as we consider only forward edges (edges (u, v) where $u < v$), it must be the case that if the edge is an edge to a previous batch, vertex u was visited previously while vertex v is a member of the current batch. Thus, when inserting artificial edges to the k artificial vertices, we only check for blocks $i \in B[u]$ for a vertex $u^* = S^*(e), e = (u, v) \in E$.

Next, we discuss the various configurations for B as showcased in Figure 4.3. Later, in Section 5.3.2, we present an experimental comparison of the following modes.

4.3.1 Maximal Mode

In maximal mode, for every $u \in V$, we store all unique blocks that were assigned to edges incident on u, i.e., $e \in E(u)$, as we visit them, in $B[u]$. In the worse case, this requires $\mathcal{O}(min\{nk, m\})$ memory. For the current batch, we insert an artificial edge from $u^* \in V^*$, where $u^* = S^*(e), e = (u, v) \in E, u \in$ previous batch, to all artificial vertices $i \in B[u]$, as demonstrated in depiction (a) of Figure 4.3. This suggests the vertex partitioner to assign u^* to one of these $B[u]$ blocks to avoid replicating the vertex on a new block. The vertex partitioner is able to comfortably manage balance constraints as it has access to multiple blocks that u^* could potentially be assigned to.

This mode produces the best solution quality compared to the alternatives, as it is the most complete. However, it has the largest memory requirement. It also has slower partitioning runtime due to a larger size of the resulting graph model β, which can have up to k artificial edges per vertex induced by an edge to a vertex of a previous batch.

4.3.2 r-Subset Mode

The r-Subset mode is similar to the maximal mode, in that, for every $u \in V$, we store in $B[u]$ all unique blocks that were assigned to edges incident on u as we visit them. However, when inserting artificial edges from a vertex $u^* \in S^*$, $u^* = S^*(e)$, $e = (u, v) \in E$,

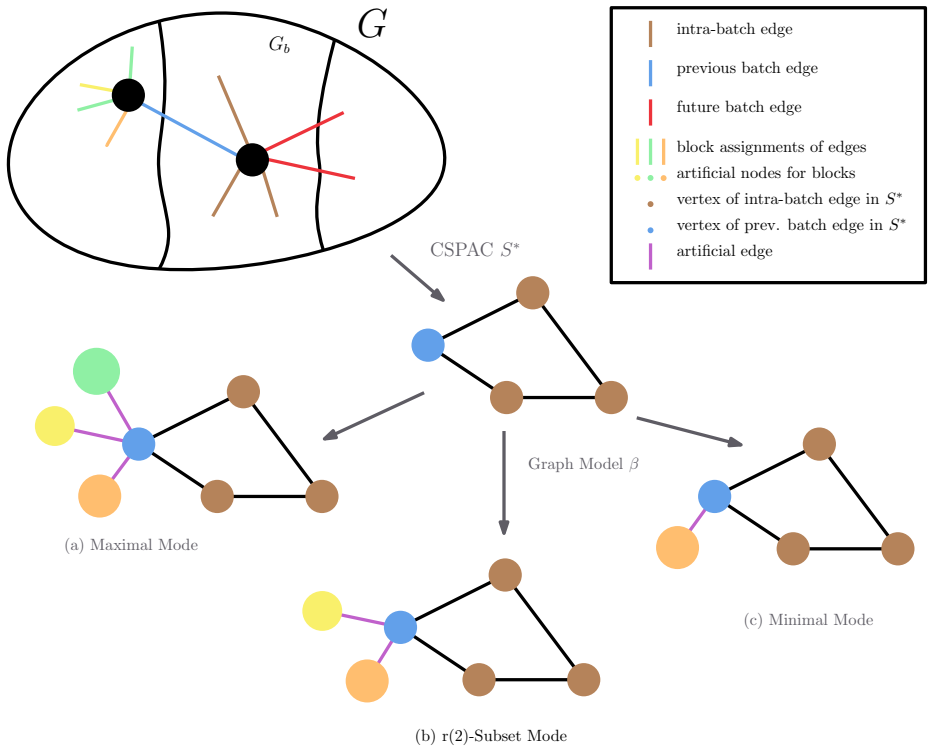


Figure 4.3: Graph Model β Construction. β is obtained by appending past assignment decisions to S^*. If a vertex of the current batch graph $u \in G_b$ has an edge $e = (u, v)$ to a previous batch (colored blue), we connect the CSPAC vertex u^* induced by e to blocks assigned to edges incident on v as follows: (a) Maximal Mode: connects all blocks incident on v (b) r-Subset Mode: Connect r random blocks incident on v (c) Minimal Mode: Connect only the latest block assigned to the most recently partitioned edge incident on v.

$u \in$ previous batch, we insert a sample of r artificial edges to r random artificial vertices among $i \in B[u]$. This is shown for $r = 2$ in depiction (b) of Figure 4.3.

While $B[u]$ has a memory footprint of $\mathcal{O}(min\{nk, m\})$ like in maximal mode, the resulting graph model β is smaller due to at most $r < k$ random artificial edges per vertex induced by a previous batch edge. This leads to faster partitioning runtime, but with a marginal sacrifice on solution quality.

4.3.3 Minimal Mode

In minimal mode, for every $u \in V$, we store in $B[u]$ only the *latest* block that was assigned to the most recently visited edge incident on u. Every time we partition an edge $e = (u, v)$ visited in the current batch into block i, we update $B[u] = B[v] = i$. Later, when inserting artificial edges from $u^* \in S^*$, $u^* = S^*(e), e = (u, v) \in E, u \in$ previous batch, we have precisely one artificial edge to the artificial vertex $i = B[u]$ as showcased in depiction (c) of Figure 4.3.

Naturally, this has the smallest memory requirement among alternatives. Here, B has size exactly n and there is no dependency on k. This also impacts our graph model β, which has the smallest possible size when inserting artificial edges. With minimal mode, β has about the same size per batch regardless of k, while in maximal mode, number of edges in β grows with k.

It is worth noting that with minimal mode, we have the least amount of information regarding past block assignments compared to alternatives. One might expect more vertex replicas, or consider challenges with respect to the balancing constraint when making an assignment decision for a vertex u^* of our current batch. However, this is not the case. If vertex u^* is connected to a single artificial vertex i, as is the case here, our vertex partitioning scheme highly favors assigning u^* to block V_i. This leads to the same edge-cut of β, as if we assigned u^* to any block V_i with $i \in B[u]$ of the maximal mode. Further, balance constraints are kept in check by always updating $B[u]$ with the *latest* block assigned to an edge incident on u.

4.4 Vertex Partitioning: Multilevel Weighted Fennel

After we have our graph model β, we apply the multilevel weighted fennel vertex partitioning scheme developed by Faraj and Schulz [22] on it. In this section, to be self contained, we briefly describe the vertex partitioning scheme used in HeiStream, and provide a modification we adopt to lose the dependency on k in the partitioning runtime. When used in conjunction with the minimal mode described in Section 4.3.3, the runtime of our entire edge partitioning process is asymptotically independent of the choice of k.

Our vertex partitioner uses a multilevel partitioning scheme as described in Section 2.2.3. In this scheme, β is recursively contracted to achieve smaller graphs which reflect the same structure as β. We then compute an initial partitioning of the smallest graph that minimizes

edge-cut, and begin uncoarsening the graph. Recall that from Theorem 4.2.1, we can infer that minimizing edge-cut of the vertex partitioning of the CSPAC graph (in this case, β) leads to a minimization of the number of replicas in the induced edge partitioning of the input graph G_b. At each level of uncoarsening, we perform local search to improve the initial partitioning done at the coarsest level.

The coarsening phase in HeiStream is an adaptation of the size-constrained label propagation approach [52] that supports artificial vertices of β. In the coarsening phase, to obtain a graph hierarchy, the algorithm computes a size-constrained cluster on each level, and then contracts each cluster into a single vertex. This process is recursively repeated until the graph reaches a small enough size. Here, we ensure that a partition of a coarse graph, in terms of edge-cut and balance, corresponds to a partition of all the finer graphs in the hierarchy.

HeiStream computes these clusters at each level with label propagation [56], while adhering to size constraints to avoid large clusters based on the approach in [52]. The algorithm works in rounds as follows: in the first step, we insert every vertex in its own cluster. In subsequent rounds, we traverse all vertices of the graph. When a vertex u is visited, it is moved to the cluster V_i that maximizes $\omega(\{(u,v)|v \in N(u) \cap V_i\})$, i.e., u is assigned to the cluster that has the strongest connection to u. If there are multiple blocks with equally strong connection where u could be moved, we break the ties randomly. At most L rounds are performed to obtain clusters of good quality, where L is a tuning parameter.

Further, HeiStream ignores artificial vertices and artificial edges of β during label propagation in order to preserve this information at the coarsest level where initial partitioning is performed, and to avoid two artificial vertices from getting contracted together. Overall, the coarsening process is recursively repeated until the graph has fewer vertices than $\mathcal{O}(max(\frac{|\beta|}{2xk}, xk))$ where x is a tuning parameter, at which point it is considered small enough for initial partitioning. For large enough buffer sizes, i.e., for δ large enough, this threshold is $\mathcal{O}(\frac{|\beta|}{k})$.

Once we are at the coarsest level β_c, we are ready to compute the initial partitioning. In this step, all vertices of β_c, except artificial vertices, are assigned to one of k blocks. This is achieved using the generalized Fennel algorithm proposed by Faraj and Schulz [22]. More specifically we run the generalized Fennel algorithm with an explicit balancing constraint L_{max} which serves as an upper bound for block weights. A vertex u of β_c is assigned to the block i that maximizes

$$\sum_{v \in V_i \cap N(u)} \omega(u,v) - c(u)f(c(V_i)),$$

where $f(c(V_i)) = \alpha * \gamma * c(V_i)^{\gamma-1}$, α, and γ refer to fennel alpha and gamma respectively, and $c(V_i \cup u) \leq L_{max}$. In HeiStream, the algorithm at this point considers all possible blocks $i \in \{1...k\}$, and therefore the initial partitioning step has a dependency on k. We modify this step to make initial partitioning drop the linear dependency on k, as will be described later in this section. Additionally, in HeiStream, β is simply the current batch G_b with artificial vertices and edges, whereas in our case, β is the CSPAC graph

of G_b, namely, S^* with artificial vertices and edges. Thus, we cannot assume the suitability, as HeiStream does for consistency, of the choice of fennel $\alpha = \sqrt{k}\frac{m}{n^{3/2}}$ proposed by Tsourakakis et al. [66], where m and n are the the number of edges and vertices of the input graph G respectively. This problem will also be discussed in more detail later in the section.

When initial partitioning is completed, we transfer the current solution to the next finer level by assigning block i of the coarse cluster to its constituent vertices of the next level. At each level of the graph hierarchy, we apply a local search algorithm. This local search algorithm is similar to the size-constrained label propagation algorithm we used in the contraction phase but with a different object function. In particular, when visiting a vertex u, we remove it from its current block and then assign it to its neighboring block (blocks assigned to vertices $v \in N(u)$) that maximizes the generalized Fennel gain function described above. Note that while in initial partitioning all possible blocks are considered for a vertex, here we only look at neighboring blocks. This ensures that each round of uncoarsening can be implemented to run in linear time in the size of the current level. Again, HeiStream does not allow artificial vertices to be moved during the uncoarsening phase, but artificial vertices and artificial edges are used to compute the generalized Fennel gain function of other vertices.

The overall runtime of coarsening and uncoarsening sums up to be linear in the size of the batch β, while the overall running time of initial partitioning depends linearly on both the size of β and k. Assuming geometrically shrinking graphs throughout the hierarchy, and that buffer size δ is sufficiently larger than number of blocks k, the overall runtime to partition a batch is $\mathcal{O}(n + m)$. However, the requirement that δ is sufficiently larger than k must be dropped in order to ensure that runtime does not increase linearly in the size of k for a fixed buffer size δ. We provide a modification of initial partitioning that removes the linear dependency on k by adopting the approach used in [21] for streaming hypergraph partitioning.

4.4.1 k-Independent Initial Partitioning

In this section, we describe an updated implementation of initial partitioning in HeiStream. Recall that, for every vertex $u \in \beta$, HeiStream finds the block with the highest score among all blocks $i \in \{1...k\}$ (Section 3.1). In the HeiStream approach, we evaluate the score for each of the k blocks for every vertex, resulting in $\mathcal{O}(nk)$ evaluations in total. Eyubov et al. [21] offer a more efficient alternative which avoids evaluating each block for a vertex to compute their hypergraph gain function FREIGHT. We adopt their approach to lose the linear dependency on k in our runtime. The core idea is as follows.

For the current vertex, we separate the blocks V_i for which $V_i < L_{max}$ into two disjoint sets K_1 and K_2. A block $V_i \in K_1$ if a neighbor of u was assigned to block V_i, and otherwise $V_i \in K_2$. With this, we can split the generalized Fennel gain function as follows: Find the blocks V_{max} and V'_{max} that maximize the following equations:

$$V_{max} = \arg\max_{i \in K_1} \left\{ \sum_{v \in V_i \cap N(u)} \omega(u,v) - c(u) f(c(V_i)) \right\} \tag{4.1}$$

$$V'_{max} = \arg\max_{i \in K_2} \left\{ \sum_{v \in V_i \cap N(u)} -c(u) f(c(V_i)) \right\} \tag{4.2}$$

Then, the block that maximizes the generalized Fennel gain function is $\max(V_{max}, V'_{max})$.

Equation 4.2 is the same as Equation 4.1, but we lose the term $\omega(u,v)$ as there exists no $v \in \beta \wedge v \in K_2$ such that $e = (u,v) \in \beta$ by definition of K_2. Then, since $c(u)$ is constant, finding the block $V_i \in K_2$ that maximizes Equation 4.2 is equivalent to finding the block V_i that minimizes $f(c(V_i)) = \alpha * \gamma * c(V_i)^{\gamma-1}$, i.e., the block $V_i \in K_2$ with the minimum block weight $c(V_i)$. By maintaining a priority queue, we can keep track of the minimum weight block among all k blocks in $\mathcal{O}(\log k)$ time with a binary heap or $\mathcal{O}(1)$ time with a bucket priority queue. Then, we no longer need to evaluate all k blocks for every vertex: we only evaluate blocks assigned to neighbors of u, and the minimum weight block among all blocks (which may already by assigned to a neighbor of u). This gives us the optimal block V_i that maximizes the generalized Fennel gain function for every vertex u in $\mathcal{O}(d(u) + \log(k))$ time with a binary heap priority queue, or $\mathcal{O}(d(u))$ for a bucket priority queue like the one suggested by [21]. Thus, we get an overall linear complexity of $\mathcal{O}(m+n)$. With this updated approach, our runtime is independent of the choice of k.

4.4.2 Choice of Fennel Alpha

In HeiStream for vertex partitioning of an input graph $G = (V, E)$, with n vertices and m edges, Fennel α is set to $\alpha = \sqrt{k}\frac{m}{n^{3/2}}$ as advocated by Tsourakakis et al. [66]. Tsourakakis et al. make this choice of α, regardless of whether it may be sub-optimal, as it provides for a proper scaling of the objective function. In particular, for this α, the Fennel optimization problem is reduced to minimizing a natural normalization of the objective function. While this choice of α results in a good solution for vertex partitioning of G, it can potentially be quite sub-optimal for edge partitioning of G with the CSPAC graph model. This is perhaps because in our model, we instead compute a vertex partitioning of the CSPAC graph G^*, which has $n^* = m/2$ vertices and $m^* = \{\text{number of auxiliary edges of SPAC graph } G'\}$ edges. For instance, for the in-2004 graph, which is a Web graph, the choice of α could result in a difference of up to 20% in solution quality.

While we know n^*, we cannot directly obtain m^* without visiting all vertices of the graph. This is because the number of auxiliary edges of the CSPAC graph G^* is equal to $(3 * m) - (2 * (n_{d1} + n_{d2}) - m$, where n_{d1} is the number of vertices in V with degree 1, and n_{d2} is the number of vertices in V with degree 2, as shown in Theorem 4.2.5. Without

visiting all vertices, we cannot obtain this value. Thus, we need some way to approximate it. Here, we provide the various α values we tested.

Static Alpha: In this version we keep the α value constant throughout all batches. It is set to $\alpha = \sqrt{k}\frac{m_{approx}}{n^{*3/2}}$, where $n^* = m/2$, $m_{approx} = y * m$, and y is a tuning parameter. We set values of y in the range $[1, 3]$ as the number of auxiliary edges is upper bounded by 3 * m.

Batch Alpha: Unlike static alpha, when using batch alpha, we update the Fennel α for every batch. After computing the CSPAC graph S^* for our batch graph G_b, we set $\alpha = \sqrt{k}\frac{m_s}{n_s^{3/2}}$, where n_s is the number of vertices of S^*, i.e., the number of edges of G_b, and m_s is the number of edges of S^*, i.e., auxiliary edges of the SPAC graph of G_b. As we compute these values for every batch, we can also update α accordingly.

Dynamic Alpha: Dynamic alpha, like batch alpha, updates the Fennel α for every batch. However, in this setting, we begin by setting α equal to static alpha with $y = 3$. Then, as we proceed through batches, we revise m_{approx} by computing the number of degree 1 and degree 2 vertices encountered, and subsequently update α. As we advance, we get to a better approximation of m^*, which is perfect for the final batch. In the experimentation Section 5.3.3, we showcase a comparison of these different choices for the Fennel α.

CHAPTER 5

Experimental Evaluation

After describing our overall streaming edge partitioning approach in Chapter 4, we now provide an experimental evaluation in this section. We begin by introducing our experimental methodology, and enlist the graph instances we used. Then, we showcase a series of tuning experiments to identify parameters with which to compose HeiStreamEdge. We conclude the section by providing a comparison with the current state-of-the-art.

5.1 Methodology

We implemented HeiStreamEdge inside the KaHIP framework (using C++) and compiled it using gcc 9.3 with full optimization enabled (-O3 flag). For comparison with competitors, we obtained implementations of the Two-Phase Streaming algorithm (including both Two-Phase-HDRF and Two-Phase-Linear) and Hybrid Edge Partitioner (HEP) by Mayer et al. [49] [51], KaHIP, and the Streaming Neighbor Expansion (SNE) algorithm by Zhang et al. [70], from their official repositories. Mayer et al. [50] also provide implementations of HDRF and DBH that we used for further comparison. All experiments were performed on the same machine. The machine has two sixteen-core Intel(R) Xeon(R) Silver 4216 processors running at 2.10GHz, 93GB of main memory, and 16 MB of L2-Cache. It runs Ubuntu GNU/Linux 20.04.1 LTS and Linux kernel version 5.4.0-65-generic. We set the number of blocks of partition, $k = \{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384\}$ for all experiments except those on huge graphs. We allow an imbalance of 3% for all experiments (and all algorithms). Further, we configure all competitor algorithms with the optimal settings provided by the authors. 2PS-HDRF, 2PS-L, HDRF and DBH are compiled with the flag for the number of blocks for each experiment, to optimize for memory consumption. The provided code for 2PS-HDRF, 2PS-L, and HDRF in the official repository sets a hard-coded soft limit on the number of partitions to 256. We override this limit to test the algorithms for larger block partitions. For HDRF, we set $\lambda = 1.1$, and for SNE, we use a cache size of $2 * |V|$. HEP can be configured with a parameter τ, which controls

the proportion of edges that are partitioned in-memory. We set $\tau = 1$ for our experiments, which, as stated by the authors of HEP, makes HEP's memory overhead similar to that of a streaming partitioner [49]. We refer to HEP with $\tau = 1$ as HEP(1). Finally, we run the edge partitioner in KaHIP with the fastsocial configuration. All competitors other than KaHIP ingest the graph in a binary edgelist format with 32-bit vertex IDS. This allows for faster IO during program execution. Additionally, all these programs offer a converter which is capable of loading a graph in the standard edgelist format, converting it into the binary edgelist format, and writing it to memory before proceeding. For a fair comparison with our proposed HSE partitioner, which reads graphs in the METIS format [39], all graphs are passed into competitor algorithms in the standard edgelist format, and the time for conversion is included in the runtime for these partitioners.

Our experiments are focused on computing the overall running time and/or replication factor depending on the objective. Unless stated otherwise, we perform 10 repetitions per algorithm and per instance using different random seeds for initialization. We then compute the arithmetic average of the computed objective functions and runtime per instance. When further averaging over all instances, we use the geometric mean to give every instance the same influence on the final score. Further, we average all results of each algorithm grouped by k, to explore performance with increasing k values. Let the runtime or replication factor be denoted by the score σ_A for some k_i-partition generated by an algorithm A. We express this score relative to others using one or more of the following tools, as used by Faraj and Schulz [22] in HeiStream:

- *improvement* over an algorithm B, computed as a percentage $(\frac{\sigma_A}{\sigma_B} - 1) * 100\%$;
- *ratio* over optimal, computed as $(\frac{\sigma_A}{\sigma_{max}})$ where σ_{max} is the best score (maximum or minimum) for k_i among all algorithms including A
- *relative* value over an algorithm B, computed as $\frac{\sigma_A}{\sigma_B}$

Additionally, we present pair-wise performance profiles by Dolan and Moré [20] for benchmarking our algorithms. These profiles relate the running time (resp. solution quality) of the slower (resp. worse) algorithm to the faster (resp. better) one on a per-instance basis, rather than grouped by k. Their x-axis shows a factor τ while their y-axis shows the percentage of instances for which an algorithm has up to τ times the running time (resp. solution quality) of the faster (resp. better) algorithm.

5.2 Instances

Our graph instances for experiments are shown in Table 5.1. We obtain these through various sources [6] [23] [43] [57]. All instances evaluated have been used for benchmarking in previous works on graph partitioning. The roadNet graphs, wiki graphs, web-Google, web-NotreDame, and all social, co-purchasing, and autonomous systems graphs were obtained from the publicly available SNAP dataset [43]. For testing with HeiStreamEdge, we

converted these graphs to a vertex-stream format (METIS) while removing parallel edges, self loops, and directions, and assigning unitary weight to all vertices and edges. These METIS graphs were then converted back into the edgelist format that SNAP uses to be compatible with all competitor algorithms. We also used graphs from the 10th DIMACS Implementation Challenge, namely eu-2005, in-2004 and uk-2007-05 [7]. Any remaining graphs are available on the network repository website [57]. From these graph instances, we construct three disjoint sets: a tuning set for parameter study experiments, a test set for comparison against state-of-the-art and a set of huge graphs, for which in-memory partitioners ran out of memory on our machine. While streaming, we use the natural order of the vertices in these graphs.

Graph	n	m	Type
Tuning Set			
hcircuit	105 676	203 734	Circuit
coAuthorsCiteseer	227 320	814 134	Citations
coAuthorsDBLP	299 067	977 676	Citations
com-DBLP	317 080	1 049 866	Social
roadNet-PA	1 088 092	1 541 898	Roads
web-Google	356 648	2 093 324	Web
amazon0601	403 394	2 443 408	Co-Purch.
com-Youtube	1 134 890	2 987 624	Social
Amazon-2008	735 323	3 523 472	Similarity
soc-lastfm	1 191 805	4 519 330	Social
as-Skitter	554 930	5 797 633	Aut. Syst.
italy-osm	6 686 493	7 013 978	Roads
in-2004	1 382 908	13 591 473	Web
ML_Laplace	377 002	13 656 485	Mesh
Huge Set			
it-2004	41 291 594	1 027 474 947	Web
com-friendster	65 608 366	1 806 067 135	Social
sk-2005	50 636 154	1 810 063 330	Web
uk-2007-05	105 896 555	3 301 876 564	Web

Graph	n	m	Type
Test Set			
Dubcova1	16 129	118 440	Mesh
DBLP-2010	300 647	807 700	Citations
com-Amazon	334 863	925 872	Social
web-NotreDame	325 729	1 090 108	Web
citationCiteseer	268 495	1 156 647	Citations
wiki-Talk	232 314	1 458 806	Web
roadNet-TX	1 379 917	1 921 660	Roads
amazon0312	400 727	2 349 869	Co-Purch.
amazon0505	410 236	2 439 437	Co-Purch.
roadNet-CA	1 965 206	2 766 607	Roads
G3_circuit	1 585 478	3 037 674	Circuit
soc-flixster	2 523 386	7 918 801	Social
great-britain-osm	7 733 822	8 156 517	Roads
FullChip	2 986 999	11 817 567	Circuit
coPapersDBLP	540 486	15 245 729	Citations
coPapersCiteseer	434 102	16 036 720	Citations
eu-2005	862 664	16 138 468	Web
soc-pokec	1 632 803	22 301 964	Social
wiki-topcats	1 791 489	25 444 207	Social
circuit5M	5 558 311	26 983 926	Circuit
com-LJ	3 997 962	34 681 189	Social
soc-LiveJournal1	4 846 609	42 851 237	Social
Ljournal-2008	5 363 186	49 514 271	Social
ca-hollywood-2009	1 069 126	56 306 653	Roads
Flan_1565	1 564 794	57 920 625	Mesh
Bump_2911	2 852 430	62 409 240	Mesh
com-Orkut	3 072 441	117 185 083	Social
HV15R	2 017 169	162 357 569	Mesh

Table 5.1: Graphs for experiments

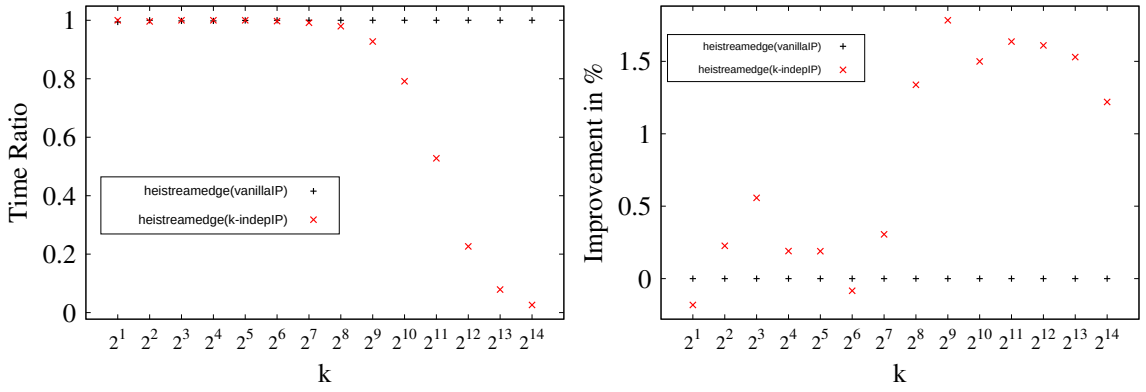
5.3 Tuning: Parameter Study

In this section, we provide experiments to tune parameters and modes used in HeiStreamEdge, and to explore its performance under various configurations. All tests are performed on the Tuning Set in Table 5.1. Our strategy is to tune a single parameter in every experiment, while keeping all others constant. We start with the following baseline configuration: β construction does not use any mode, i.e., the per-batch graph model is the CSPAC graph itself, without any extensions, fennel alpha is set to batch alpha, 5 rounds of coarsening label propagation, 5 rounds of uncoarsening local search label propagation, and $x = 4$ in the expression of the coarsest model size. After each tuning experiment, we update the baseline configuration with the best found parameter. We run all tuning experiments on HeiStreamEdge as defined in Chapter 4, with a buffer size of $\delta = \{32768, 131072, 262144\}$. As we found that the choice of the best tuning parameter was independent of buffer size, in this section, we only showcase experiments with $\delta = 32768$ for brevity. The results of each tuning experiment are shown through plots presenting, on the y-axis, (a) the ratio of time required and (b) the percentage improvement in replication factor relative to the baseline configuration against the number of blocks k on the x-axis.

5.3.1 Initial Partitioning: k-Independent Adaptation

Before showcasing experiments to tune parameters, we highlight the improvement obtained from using the k-independent initial partitioning approach described in Section 4.4.1 relative to not using it (referred to here as vanilla initial partitioning). In our implementation, we use a binary heap priority queue to obtain the block with the smallest weight in $\mathcal{O}(1)$ time. Our expectation is to retain solution quality, while achieving a high speedup for larger k-values. The results of this experiment are shown in Figure 5.1.

As expected, using k-independent initial partitioning, we obtain a tremendous speedup for larger k values, while maintaining solution quality. Figure 5.1a displays that the inflection point for speedup using the k-independent initial partitioning version is $k = 2^8 = 256$; the speedup accelerates as k increases, with the k-independent initial partitioning version taking only 7.9% and 2.6% of the time required by vanilla initial partitioning, for $k = 2^{13} = 8192$ and $k = 2^{14} = 16384$ respectively. Across all k values, we achieve 46% faster runtime on average over the baseline. For values of $k = 256$ or higher, k-independent initial partitioning is 76.2% faster on average than vanilla initial partitioning. Figure 5.1b shows that the k-independent initial partitioning version produces higher solution quality than vanilla initial partitioning for the majority of k values tested. Across all instances, the k-independent initial partitioning version produces 0.84% better solution quality on average than vanilla initial partitioning. Surprisingly, we find that solution quality is on average 1.54% better for $k > 256$ when using k-independent initial partitioning. This is likely due to the effects of randomness in tie breaking when finding the block with the minimum



(a) Running time ratio.

(b) Quality improvement over vanilla IP.

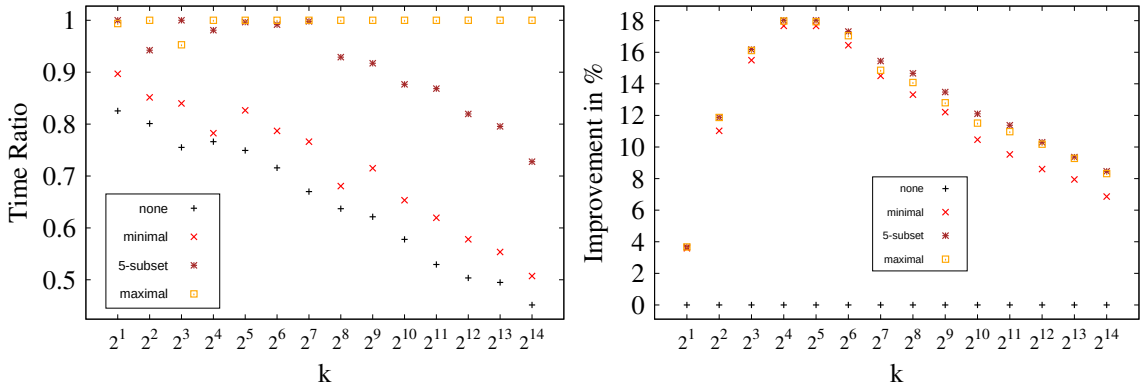
Figure 5.1: Tuning Experiment: Initial Partitioning. For all plots depicting time ratio and percentage improvement in replication factor, the running time ratio is calculated against the longest average runtime for each k, and the percentage improvement in replication factor is calculated against a chosen baseline. Here, we select vanilla initial partitioning (vanillaIP) of HeiStream as the baseline for comparison.

weight. In our implementation of k-independent initial partitioning, we prioritize blocks assigned to neighbors of a vertex u (set K_1 from Section 4.4.1) in the case of a tie with blocks outside of the neighborhood (set K_2). On the other hand, vanilla initial partitioning considers all blocks with equal importance in the case of a tie. This likely is the cause for the marginal increase in solution quality with k-independent initial partitioning. Based on these results, we compose our baseline configuration with k-independent initial partitioning instead of vanilla initial partitioning.

5.3.2 Model Construction Modes

Our first tuning experiment relates to choosing a suitable per-batch graph model mode among the options described in Section 4.3. As described, the baseline configuration uses no mode, i.e., it does not have any artificial vertices or edges. This experiment revises this baseline to use the maximal, minimal, and r-Subset modes and compares these with each other and with the no-mode configuration.

The results are shown in Figure 5.2, which demonstrate that using any of the modes significantly improves solution quality - thus highlighting the necessity of using a model mode over using no mode - while increasing runtime, with the minimal mode producing the fastest runtime among the different model modes. Figure 5.2b shows that all modes produce a higher solution quality than using no mode, with only a small difference in solution quality between modes. On average across all instances and all k values, we achieve an improvement of 11.73% to 12.8%, depending on the mode, over not using any artifi-



(a) Running time ratio.

(b) Rep. factor improvement over using no mode.

Figure 5.2: Tuning Experiment: Graph Model Modes. We arbitrarily show results for $r = 5$ for the r-Subset mode, as a similar trend is observed for other r values as well.

cial extensions to the per-batch CSPAC graph. This is because the modes are designed to leverage block assignment decisions from previous batches when partitioning the current batch as was described in Section 4.3. A surprising result here is that minimal mode performs extremely well, despite being significantly more light-weight than maximal mode and r-Subset mode. Minimal mode improves solution quality by 11.73% over the baseline, while maximal mode produces only 0.73% better solution quality on average over minimal mode.

Figure 5.2a demonstrates that, while using no mode (which results in the smallest graph model per batch) is the fastest, minimal mode is the fastest among the different mode choices. Relative to using no mode, minimal mode requires 11% more time on average, while maximal mode and 5-Subset require 56% and 43% more time respectively. Minimal mode is 28.9% faster than maximal mode.

Based on these results, we update our baseline configuration with minimal mode. Besides offering a substantial increase in solution quality while being faster than other modes, minimal mode also has a much lower memory overhead compared to r-Subset and maximal mode, as we store only one block per vertex v (instead of up to $min\{k, d(v)\}$ blocks).

5.3.3 Fennel Alpha

Next, we perform a comparison of the different choices of fennel alpha as discussed in Section 4.4.2. We run our updated baseline configuration, including minimal mode and k-independent initial partitioning, with static, batch and dynamic alpha. Figure 5.3 shows the impact of the alpha choices on replication factor. As we do not observe a significant difference in runtime between the choices for fennel alpha, we do not display comparisons for runtime.

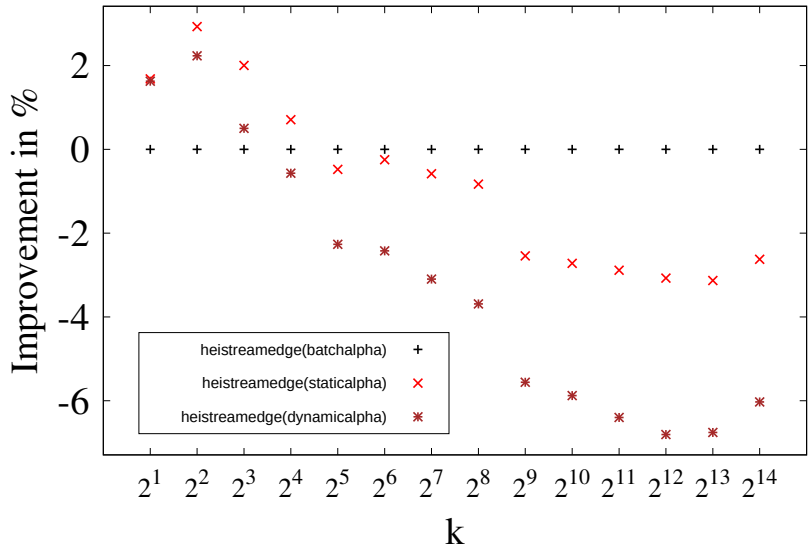


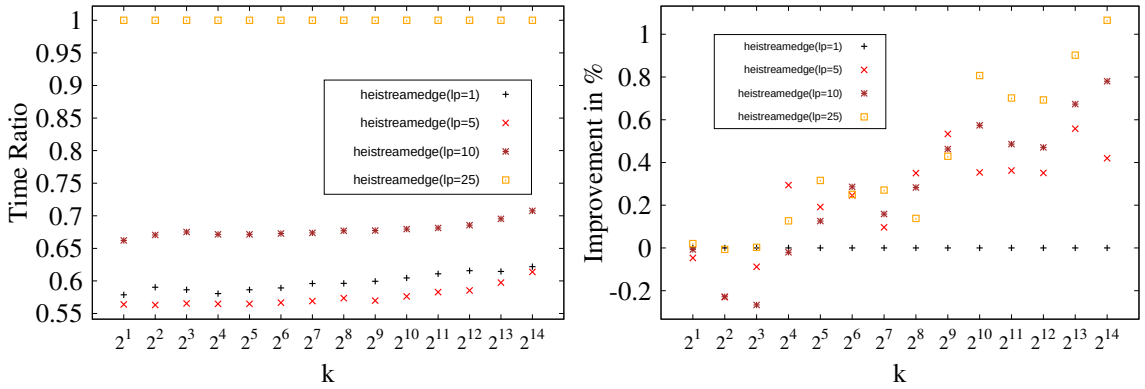
Figure 5.3: Tuning Experiment: Fennel Alpha. Replication factor improvement with choice of Fennel α. Here, batch alpha is selected as the baseline.

Figure 5.3 demonstrates that batch alpha provides the best solution quality for a majority of k values, particularly for k values above 32. On average, across all instances and all k values, batch alpha produces 0.86% and 3.27% better solution quality than static alpha and dynamic alpha respectively. For k values above 32, these averages increase to 2.1% and 5.2% for static alpha and dynamic alpha respectively. As batch alpha is shown to produce the best solution quality, particularly for larger k values, we retain batch alpha as our choice for fennel alpha in the baseline configuration for subsequent tuning experiments.

5.3.4 Label Propagation

We evaluate how the number of label propagation rounds, which is responsible for computing clusters during the contraction phase of our multilevel partitioning scheme, impacts solution quality and runtime. We run the current baseline configurations of HeiStreamEdge with 1, 5, 10, and 25 rounds of label propagation, and report the results in Figure 5.4. We find that increasing the number of label propagation rounds does not have a significant impact on solution quality, and using 5 rounds produces the fastest runtime.

In Figure 5.4b, we observe a very small improvement in solution quality as the number of rounds increases. Taking 1 round as the baseline, we notice a maximum improvement of approximately 1% when we use 25 rounds. On average, across all instances and k values, we find that 25 rounds of label propagation results in an improvement of only 0.41% in solution quality over 1 round of label propagation. Using 5 or 10 rounds gives an average



(a) Running time ratio.

(b) Rep. factor improvement over 1 round.

Figure 5.4: Tuning Experiment: Label Propagation Rounds.

improvement of 0.24% and 0.27% respectively. The solution quality improvement over 1 round is not significant, so we consider the runtime to make a decision.

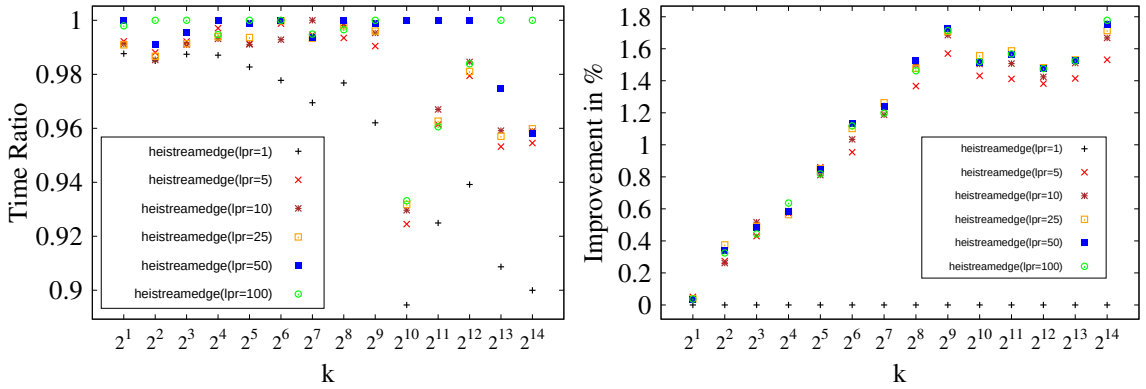
In general, we expect runtime to increase as we increase the number of rounds of label propagation. While this is observed for 10 and 25 rounds, using 5 rounds of label propagation gives a lower overall runtime compared to 1 round of label propagation, as shown in Figure 5.4a. This is because with 5 rounds, improvement in cluster quality over 1 round results in lesser vertex movements in the later stages of the multilevel scheme and thus offsets the time required by additional iterations of label propagation. Our results show that on average, 10 rounds take 13.51%, and 25 rounds take 67.28% longer than 1 round respectively, while using 5 rounds of label propagation is 3.76% faster than using 1 round.

We retain 5 rounds of label propagation in our baseline configuration, as our results demonstrate that this configuration produces the fastest runtime, while the impact of label propagation rounds on solution quality is non-significant.

5.3.5 Local Search Label Propagation

After deciding on the number of rounds of label propagation during coarsening, we consider the number of rounds of local search label propagation during uncoarsening. We evaluate the solution quality and runtime impact of increasing the number of rounds of local search label propagation by testing 1, 5, 10, 25, 50 and 100 rounds of local search label propagation with our current configuration. Figure 5.5, displays the results of these experiments, indicating that both solution quality and runtime increase with the number of rounds of local search label propagation.

Figure 5.5b demonstrates that increasing the number of local search rounds improves solution quality over 1 round, and this improvement increases with k. Using 5 rounds of local search achieves, on average, a 1% improvement in solution quality over 1 round.



(a) Running time ratio.

(b) Rep. factor improvement over 1 round.

Figure 5.5: Tuning Experiment: Local Search Rounds.

However, the rate of improvement slows down as the number of rounds increases beyond 5. Using 10 rounds produces an average improvement in solution quality of 0.05% over 5 rounds, and using 100 rounds of local search results in a negligible average improvement of only 0.09% over 5 rounds.

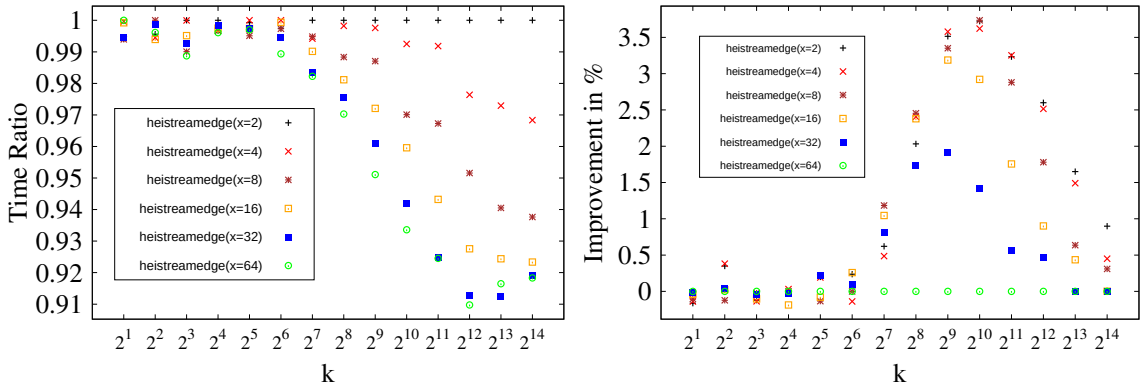
Figure 5.5a shows that, in general, runtime increases with the number of local search rounds used. We find that using 100 rounds of local search takes on average 3.62% longer than 1 round, while 5 and 10 rounds take 2.5% and 2.68% longer respectively.

Since improvement in solution quality above 10 rounds is negligible, and runtime increases with the number of rounds, we restricted our choice between 5 or 10 rounds. We chose 10 rounds as, compared to 5 rounds, it offers marginally better solution quality for large k values with a minimal runtime overhead.

5.3.6 Coarsest Model Size

Our final tuning experiment pertains to the parameter x associated with the expression $\max(\frac{|\beta|}{2xk}, xk)$, which determines the size of the coarsest graph in our multilevel scheme. We run experiments for $x = 2^i$, with $i \in \{1, ..., 6\}$. Results are reported in Figure 5.6.

We observe from Figure 5.6b that solution quality is not impacted by x for smaller k values till $k = 2^6$. Then we observe solution quality improving between $k = 2^7$ and $k = 2^{10}$, before tapering down again. For instance, on average across all k values, setting $x = 2$ improves the solution by 1.33% over $x = 64$. However, between $2^7 \leq k \leq 2^{10}$, $x = 2$ improves solution quality by 2.46% over $x = 64$ on average, while between $2^{11} \leq k \leq 2^{14}$, $x = 2$ improves solution quality by 2.09% over $x = 64$ on average. This trend is because: firstly, for smaller k values, any of the tested x values still produces a large enough coarse graph that initial partitioning generates a good global result. Then as k increases, larger x values make the coarse graph small enough that initial partitioning



(a) Running time ratio.

(b) Rep. factor improvement over $x = 64$

Figure 5.6: Tuning Experiment: Coarsest Graph Size = $\max(\frac{|\beta|}{2xk}, xk)$.

produces relatively lower solution quality. Finally, when k becomes large enough relative to x, i.e., $k >> x$, then k dominates the expression of the coarsest graph size, resulting in a small (or large) enough coarsest graph regardless of the choice of x for solution quality to not get significantly impacted.

On the other hand, we see in Figure 5.6a that $x = 64$ has the least runtime, while $x = 2$ has the most, particularly for larger k values. Again, we note however that the runtime difference peaks at around $k = 2^{12}$ before reducing again. On average, $x = 2$ has 3.88% more runtime than $x = 64$. However, for $k \leq 256$, $x = 2$ only requires 0.85% more runtime than $x = 64$. This is again because smaller x, relative to k, produces larger coarsest graphs, resulting in more initial partitioning runtime. As k becomes much larger than x, the difference between our various test x begins to reduce since k dominates the expression of the coarsest graph size.

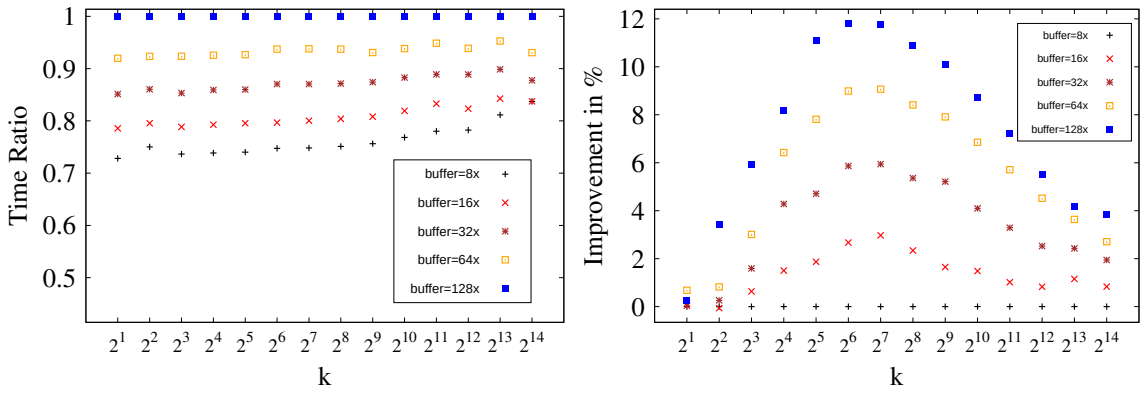
As our runtime is independent of k, we are already able to deliver very fast performance for larger k values. Therefore, we are able to sacrifice some runtime for larger k values for higher solution quality. Therefore, we choose $x = 2$ as our parameter.

5.4 Exploration

After performing the above tuning experiments, we compose HeiStreamEdge for further exploration with the parameters shown in Table 5.2. In the exploration experiments, we start by investigating how buffer size impacts solution quality and runtime. We start with a buffer size of $8x$, where $x = 1024$ vertices, and repeatedly double the buffer size until any graph in the tuning set in Table 5.1 fits entirely in a single buffer. In our case, therefore, the maximum buffer size explored is $128x = 131072$. The resulting plots are shown in Figure 5.7.

Parameter	Choice
Mode	Minimal
Initial Partitioning	k-Independent
Fennel α	Batch α
Label Propagation Rounds	5
Local Search Rounds	10
Coarsest Graph x	2

Table 5.2: HeiStreamEdge parameters chosen after tuning experiments.



(a) Running time ratio.

(b) Rep. factor improvement over buffer size = 8x.

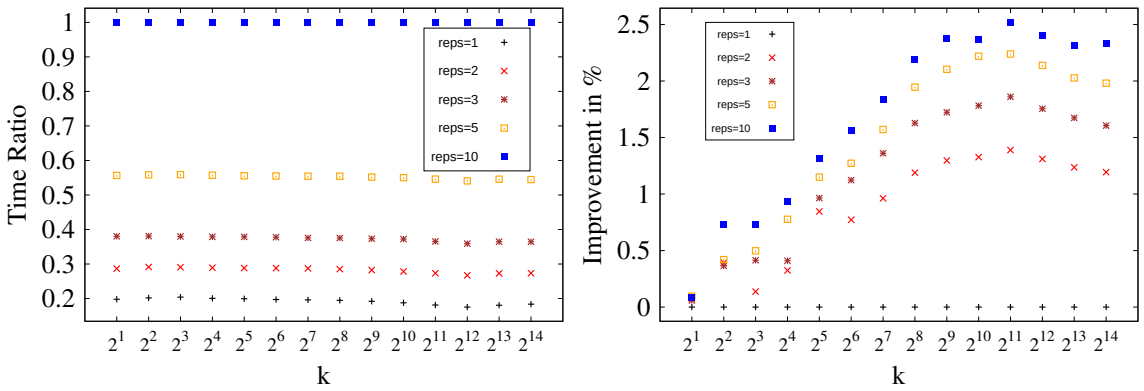
Figure 5.7: Exploration Experiment: Buffer Size = δ. Here, $x = 1024$.

We note that, in general, solution quality and runtime both increase with an increase in buffer size. On average, each time we double the buffer size, solution quality increases by approximately 1.4% over the baseline of $\delta = 8x$. Concurrently, each time the buffer size is doubled, runtime increases on average by approximately 6.2% over the baseline. For instance, $\delta = 16x$ produces a solution quality improvement of 1.35%, and a runtime increase of 6.1% over the baseline on average. When setting $\delta = 128x$, solution quality increases by 7.3%, and runtime increases by 31.23% over the baseline on average. This pattern occurs because a larger buffer size grants our multilevel partitioner the ability to leverage more comprehensive and complex graph structures. As a result, there is a trade-off between solution quality and resource consumption: we can improve replication factor at the cost of memory and runtime.

As we observed in the coarsest model size tuning experiment in Figure 5.6b, we note in Figure 5.7b that the improvement in replication factor with increasing buffer size begins to narrow with increasing k. Likewise, we see in Figure 5.7a that time ratio begins to converge at larger k values. In the expression of the coarsest model size of our multilevel partitioning scheme, i.e., $\max(\frac{|\beta|}{2xk}, xk)$, $|\beta|$ depends on the buffer size: the greater the

number of vertices we read in a batch, the larger is the resulting graph model. At small enough k values relative to the size of the input graph, the first term, $\frac{|\beta|}{2xk}$, determines the size of the coarsest graph: as $|\beta|$ increases, so too does the size of the coarsest graph, which results in higher quality partitions overall. We find, however, that for most of our tuning instances, at high k values, the size of the coarsest graph on which initial partitioning is computed, is controlled by xk instead of $\frac{|\beta|}{2xk}$. Thus, beyond some k value, depending on the size of the graph instance, buffer size no longer influences the size of the coarsest graph. This, in turn, reduces the impact of buffer size on both solution quality and runtime. For huge graphs, we still expect both solution quality and runtime to keep growing with an increase in buffer size, even for larger k values. In general, these results warrant an exploration into a potentially dynamic coarsest graph tuning parameter which can change the size of the coarsest graph depending on the size of the input graph and k. This will be described further in Section 6.2 on future work.

Based on our exploration of buffer size, we recommend using the largest possible buffer size given the memory constraints of the machine, for the best solution quality. To further improve solution quality, without additional memory requirement, we investigate the possibility to perform the entire multilevel partitioning process for each batch multiple times. We test this by repeating the multilevel partitioning 1, 2, 3, 5 and 10 times for every batch. Results are shown in Figure 5.8. As these plots demonstrate, the first repeat generates a limited solution quality improvement of 0.8% over the baseline on average. Thereafter, we get marginal returns, with ten repetitions improving the solution quality by just 1.69% on average over one repetition. On the other hand, performing each additional repetition, on average, requires approximately 45% more time than performing one repetition. In summary, we note that the limited improvement in solution quality with more repetitions is not worth the substantial increase in runtime.



(a) Running time ratio.

(b) Rep. Factor improvement over 1 repetition.

Figure 5.8: Exploration Experiment: Repetition of entire multilevel partitioning process.

5.5 Comparison against State-of-the-Art

We now provide experiments in which we compare HeiStreamEdge (from here on referred to as HSE) against the current state-of-the-art algorithms for streaming edge partitioning. These experiments were performed on the Test Set and the Huge Set of graphs in Table 5.1. We compose HSE with the configuration shown in Table 5.2 obtained from the tuning experiments, and test various buffer sizes.

We primarily compare our performance against the two state-of-the-art streaming edge partitioners, namely, Two-Phase-HDRF (2PS-HDRF) and Two-Phase-Linear (2PS-L). Additionally, we run comparisons with DBH, HDRF and Streaming Neighborhood Expansion (SNE). Aside from streaming algorithms, we also perform experiments with KaHIP, an in-memory partitioner, and Hybrid Edge Partitioner (HEP), which, as the name suggests, partitions a portion of the edges in-memory and the remaining edges with a streaming algorithm, namely, HDRF. DBH, which is based on hashing, produces the highest (worst) replication factor and shortest runtime in the group. This is expected as it is a stateless streaming partitioner. Among the stateful streaming algorithms, SNE delivers better solution quality than HDRF at the cost of higher runtime and memory [51], and 2PS-HDRF outperforms both HDRF and SNE in terms of replication factor and runtime [50]. While 2PS-L produces lower solution quality than 2PS-HDRF, it was shown to be significantly faster than all other stateful streaming and in-memory partitioners [51]. This is because the runtime of 2PS-L, unlike 2PS-HDRF, is independent of k. HEP was shown to produce better replication factor than HDRF, DBH and SNE [49].

5.5.1 Pair-wise Comparisons on Test Set

In this section, we present the results of our experiments comparing HSE to state-of-the-art streaming edge partitioners, namely 2PS-HDRF, 2PS-L, DBH and HDRF. We exclude comparisons with SNE as it fails to execute for $k > 127$, and has been shown to be slower, and produce worse solution quality than 2PS-HDRF [50]. Further, we relegate comparisons with in-memory (KaHIP) and hybrid partitioners (HEP) to the appendix. Here, we report comparisons with HSE using a buffer size of $\delta = 32x$, where $x = 1024$, i.e., a buffer size of 32768. For comparisons, we present pairwise performance profiles, as described in Section 5.1, of each algorithm compared with HSE. For comparison with 2PS-HDRF and 2PS-L, we additionally present comparisons of runtime and solution quality grouped by k. For runtime, we plot the ratio of time taken by the slower algorithm compared to the faster one against k values on the x-axis. For solution quality, we take 2PS-HDRF (resp. 2PS-L) as the baseline and plot the percentage improvement in replication factor of HSE against k values on the x-axis.

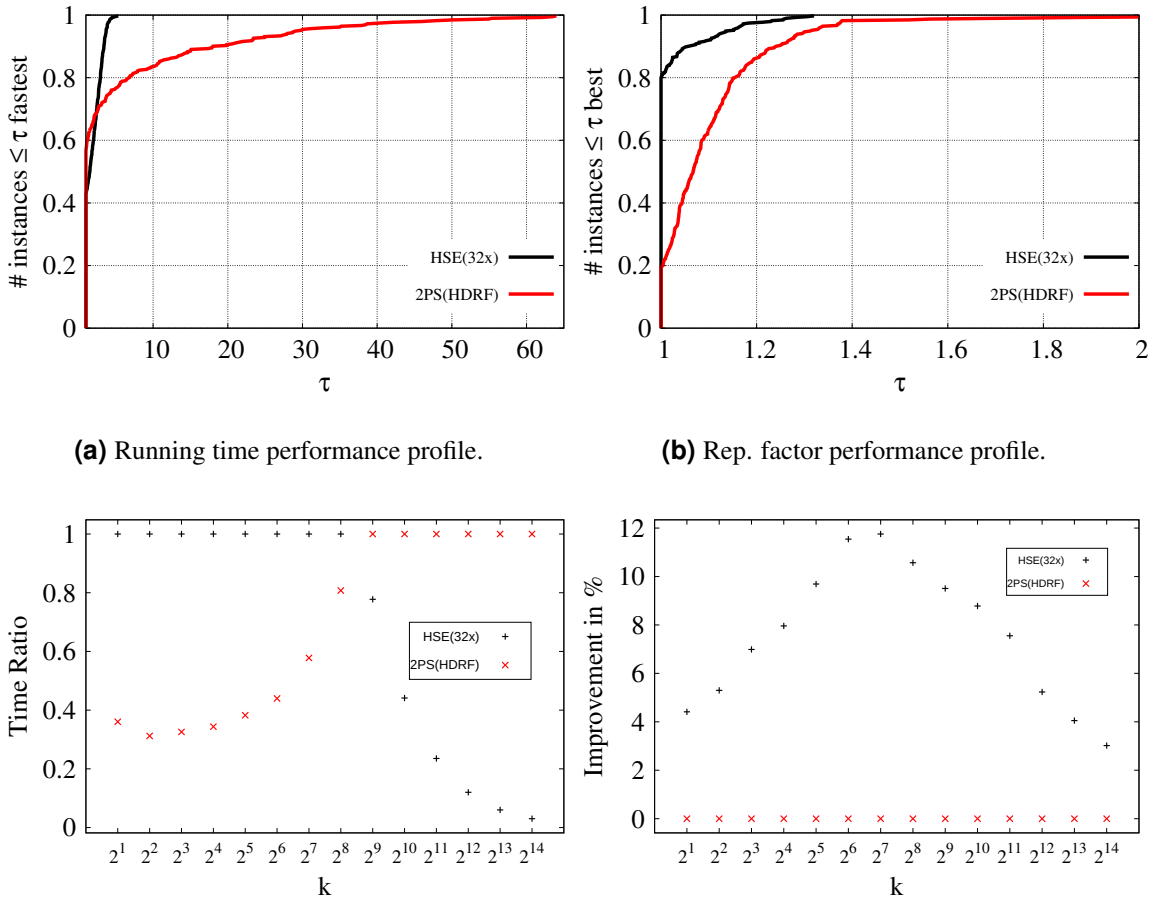
We find that HSE, with $\delta = 32x$, produces better solution quality than state-of-the-art streaming edge partitioners for all k values, with an average improvement in solution quality of 7.56% compared to 2PS-HDRF, which produces the next best solution quality (See

Figure 5.9). Additionally, as the runtime of HSE is not linearly dependent on k, HSE is substantially faster for large k values (particularly above 256) compared to 2PS-HDRF and HDRF. While the runtime of 2PS-L is also independent of k, and is shorter than that of HSE, HSE produces significantly higher solution quality than 2PS-L with an average improvement of 48.51% percent over 2PS-L (See Figure 5.10). Further, HSE's improvement in solution quality over 2PS-L increases with k, making it more suitable for large k values. Thus, HSE is the only existing streaming edge partitioner that maintains high solution quality while producing fast solutions for large k. Additionally, while HSE has the highest memory consumption on average across all instances, its memory consumption is not asymptotically dependent on k. As a consequence, HSE consumes less memory than 2PS-HDRF, 2PS-L, and HDRF for larger k values. For instance, among test set instances, at $k = 16384$, HSE has a peak memory consumption of 4.14GB, while 2PS-HDRF, 2PS-L, and HDRF consume up to 6.76, 6.63GB, and 6.37GB respectively. Only DBH, which is a stateless partitioner, consumes less memory than HSE at larger k, requiring only a maximum of 321.18MB across test set instances.

In Figure 5.9, we present detailed comparisons between HSE and 2PS-HDRF for runtime and solution quality, demonstrating that HSE is significantly faster than 2PS-HDRF for high k values, and HSE produces higher solution quality than 2PS-HDRF for all k values. Figure 5.9c demonstrates that, while HSE is slower than 2PS-HDRF for k values lower than 2^8, $k = 2^8$ is an inflection point after which HSE is faster than 2PS-HDRF, with speedup increasing with k. On average, for $k \geq 2^9 = 512$, HSE takes 83.86% less time than 2PS-HDRF. As shown in Figure 5.9a, HSE produces the fastest runtime in approximately 42% of all instances, and is, at worst, approximately 5 times slower than 2PS-HDRF, whereas, 2PS-HDRF is up to approximately 63 times slower than HSE at worst. Figure 5.9d shows that HSE produces on average better solution quality than 2PS-HDRF for all k values. As displayed in Figure 5.9b, HSE produces better solution quality than 2PS-HDRF in approximately 80% of all instances.

Figure 5.10, which compares HSE and 2PS-L, shows that while 2PS-L is faster than HSE for most k values, HSE produces substantially better solution quality, with improvements in solution quality increasing with k. For instance, for $k \geq 512$, HSE delivers 68.46% better solution quality than 2PS-L on average. Figure 5.10a shows that 2PS-L is faster than HSE for over 90% of all instances, however, Figure 5.10c demonstrates that the difference in runtime between 2PS-L and HSE begins to converge for k values larger than 256. Further, HSE is faster than 2PS-L for $k = 2^{14}$. Figure 5.10b shows that HSE produces solutions of higher quality than 2PS-L for about 99% of all instances. As shown in Figure 5.10d, HSE's improvement in solution quality over 2PS-L increases substantially with increasing k values.

Figure 5.11 depicts comparisons of HDRF and DBH with HSE. Figures 5.11b and 5.11d show that HSE produces better solution quality than HDRF and DBH for all instances. As shown in Figure 5.11c, DBH is faster than HSE for about 99% of all instances, as expected since DBH is a hashing algorithm. A runtime comparison of HSE and HDRF, depicted in Figure 5.11a, demonstrates that HSE is slower than HDRF approximately 40% of the time.



(a) Running time performance profile.

(b) Rep. factor performance profile.

(c) Running time ratio.

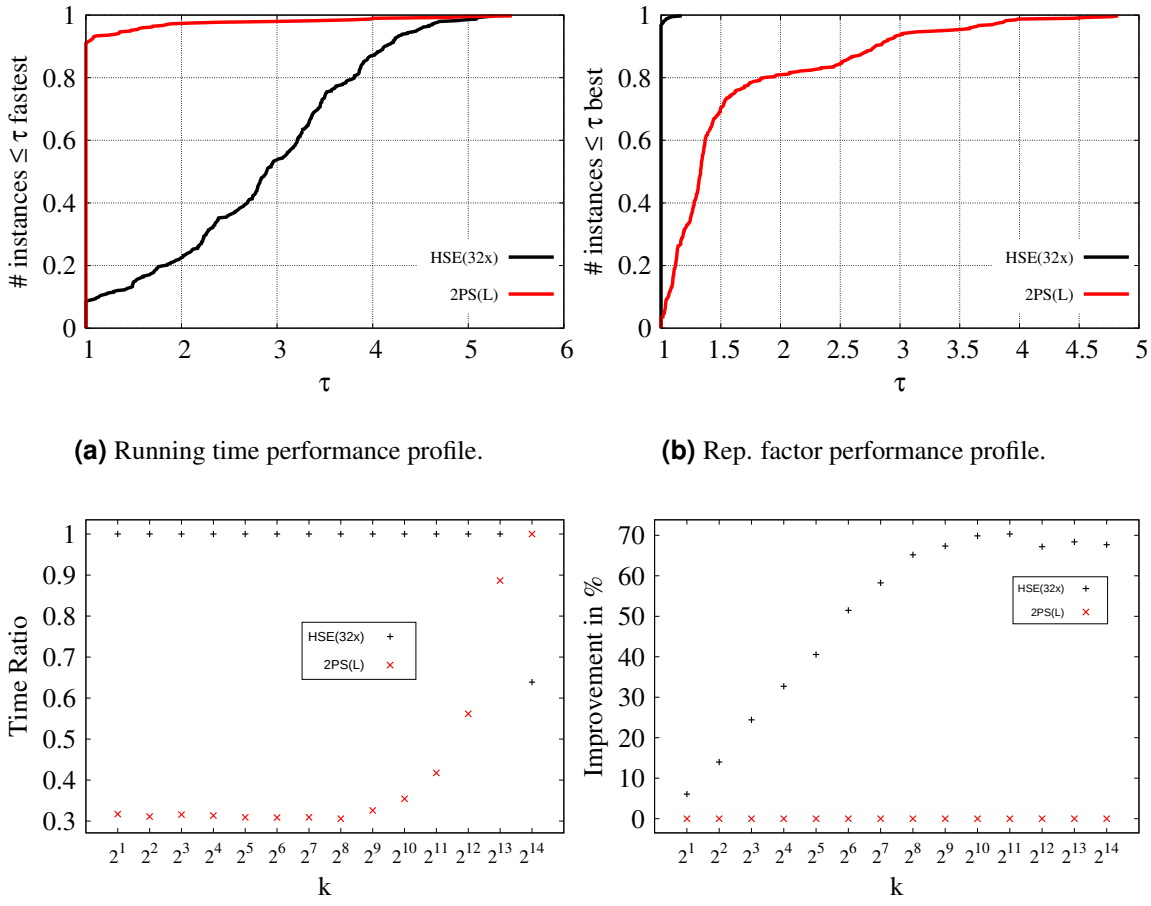
(d) Rep. factor improvement over 2PS-HDRF.

Figure 5.9: Comparison of 2PS-HDRF with HSE(32x) where x = 1024, i.e., buffer size is 32768.

However, similar to the comparison between HSE and 2PS-HDRF, HSE is significantly faster than HDRF for larger k values. In particular, for $k \geq 512$, HSE takes 87.5% less time than HDRF on average.

5.5.2 Comparison With Large Buffer Size

While all the comparisons presented thus far were computed with HSE with a buffer size of $32x$ ($x = 1024$) in this section we present a comparison of HSE with a buffer size of 512x, i.e., HSE(512x), with HSE(32x) and 2PS-HDRF. The results, as shown in Figure 5.12 demonstrate that HSE's improvement in solution quality over 2PS-HDRF is further pronounced at larger buffer sizes, at the cost of higher runtime and memory overhead. Notably, however, at large k values runtime does not increase with buffer size, and HSE(512x),



(a) Running time performance profile.

(b) Rep. factor performance profile.

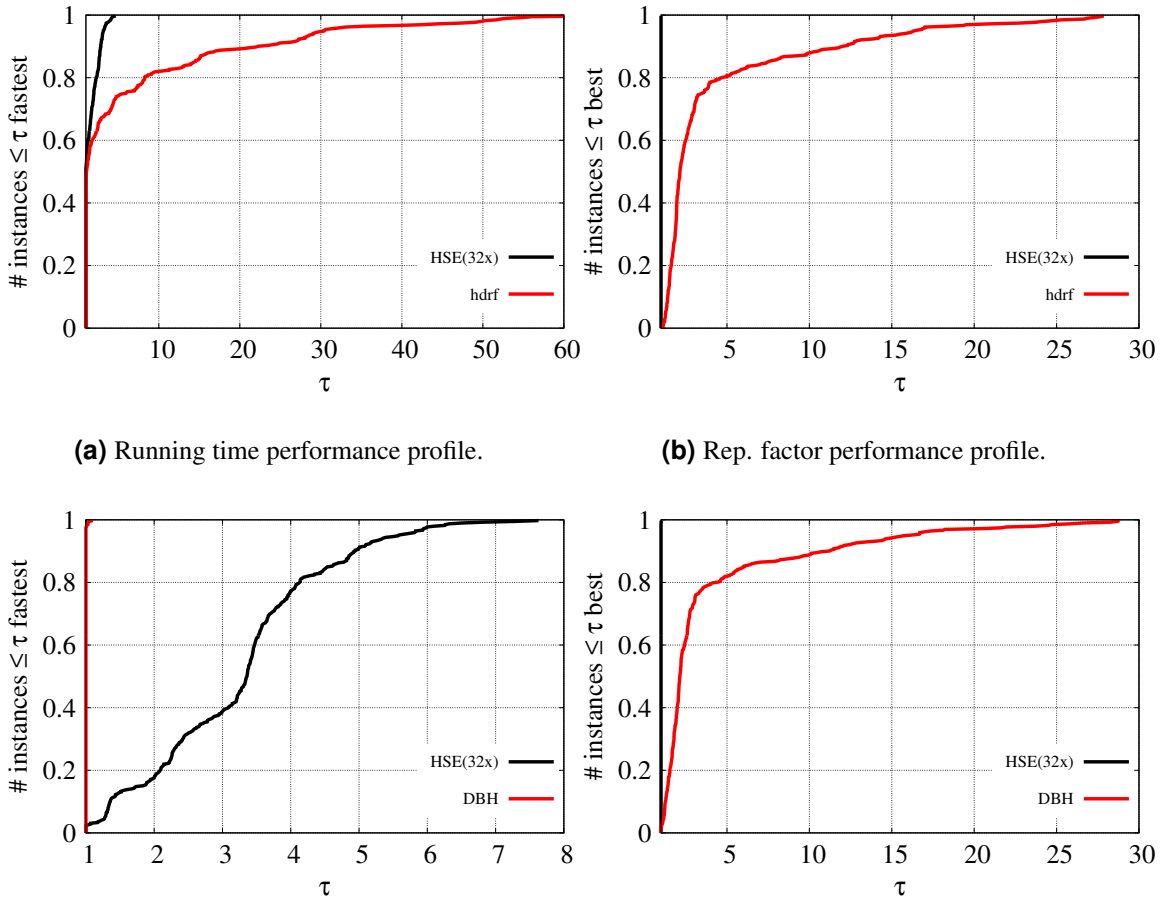
(c) Running time ratio.

(d) Rep. factor improvement over 2PS-L.

Figure 5.10: Comparison of 2PS-L with HSE(32x) where x = 1024, i.e., buffer size is 32768.

like HSE(32x), is substantially faster than 2PS-HDRF. These results confirm the results of the exploration experiments described in Section 5.4. The same trend is achieved in comparisons with other state-of-the-art streaming edge partitioners, though the results are omitted here for brevity.

Figure 5.12b shows that HSE(512x) achieves a higher average solution quality across all k values than both HSE(32x) and 2PS-HDRF. Using a buffer size of 512x, HSE achieves on average 13.95% higher solution quality than 2PS-HDRF. As shown in Figure 5.12a, for small k values the increased buffer size of 512x results in longer running time, however, for large k, HSE(512x) is only marginally slower than HSE(32x) and still substantially faster than 2PS-HDRF.



(a) Running time performance profile.

(b) Rep. factor performance profile.

(c) Running time performance profile.

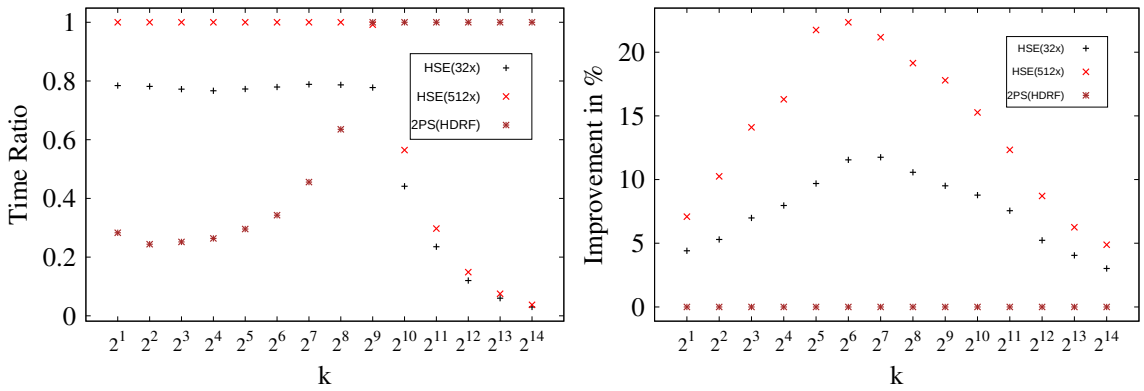
(d) Rep. factor performance profile.

Figure 5.11: Comparison of HDRF (a)(b) and DBH (c)(d) with HSE(32x) where x = 1024, i.e., buffer size is 32768.

5.5.3 Performance on Huge Set

Next, we present findings from our experiments testing HSE and state-of-the-art streaming edge partitioners on huge graphs. Computing high-quality partitions of huge graphs on small machines is the primary use case of streaming algorithms. Our experiments are run on the huge graphs listed in Table 5.1, and are performed on the same machine as our previous experiments. We ran experiments for HSE, 2PS-HDRF, 2PS-L and DBH with $k = \{8, 32, 128, 256, 512, 1024, 2048\}$. We did not repeat each test with different seeds as in previous experiments.

The results are recorded in Table 5.3. They show that HSE produces better solution quality than all competitor algorithms for most instances, providing the best solution quality for all k values for three of the four huge graphs, and the best solution quality for most k values,



(a) Running time ratio.

(b) Rep. factor improvement over 2PS-HDRF

Figure 5.12: Comparison of HSE(32x), HSE(512x) and 2PS-HDRF.

and especially large k, for the fourth graph. HSE computes, on average across all instances and graphs, 8.3% higher solution quality than the next highest solution quality by 2PS-HDRF, and 57.6% and 83.8% higher solution quality than 2PS-L and DBH respectively. HSE is, on average, slower than all competitor algorithms. While it is slower than 2PS-L and DBH for all instances, it is faster than 2PS-HDRF for large k values. For $k = 1024$ and larger, it is faster than 2PS-HDRF for all instances, and for $k = 512$ it is faster than 2PS-HDRF in three of the four graphs. On average, across all graphs for $k = 512$ or larger, HSE is 43.6% faster than 2PS-HDRF, and up to 84.6% faster at $k = 2048$. Our findings on huge graphs confirm that HSE outperforms all competitors for solution quality, and is substantially faster on large graphs for larger k values than 2PS-HDRF, the algorithm with the next best solution quality. HSE produces significantly higher solution quality than 2PS-L, the only stateful streaming edge partitioner whose runtime does not depend on k. Lastly, as observed for test set instances, HSE has larger memory consumption on average across the instances. However, due to an asymptotic dependency on k in memory consumption, 2PS-HDRF and 2PS-L eventually require more memory than HSE for large enough k, relative to the size of the graph instance. For example, both 2PS-HDRF and 2PS-L were not able to partition uk-2007-05 at $k = 8192$ without exceeding memory available on the machine, while HSE and DBH ran successfully.

Graph	k	HSE(256x)		2PS-HDRF		2PS-L		DBH	
		RF	RT(s)	RF	RT(s)	RF	RT(s)	RF	RT(s)
com-friendster	8	2.85	7342.51	**2.64**	1663.15	3.25	1569.51	3.15	1288.86
	32	5.51	7441.01	**5.21**	2022.79	6.99	1577.613	7. 12	1106.69
	128	**8.30**	7624.01	9.00	3240.04	12.17	1671.237	13.76	1121.26
	256	**10.43**	7869.91	11.12	4534.69	15.05	1703.251	17.56	1136.48
	512	**13.07**	8313.52	13.19	7173.64	17.69	1739.348	21.05	1141.5
	1024	**13.06**	8319.1	15.16	12517.5	20.03	1824.121	23.86	1135.8
	2048	**14.14**	8290.15	16.95	22542.7	21.87	2012.934	25.83	1278.63
in-2004	8	**1.12**	1049.49	1.15	364.56	1.55	351.942	2.29	323.984
	32	**1.08**	1103.68	1.21	410.067	2.12	355.2272	4.80	326.647
	128	**1.09**	1055.59	1.27	452.48	2.40	358.9895	7.34	323.336
	256	**1.09**	1056.67	1.31	670.65	2.52	362.0369	9.39	323.373
	512	**1.20**	1165.29	1.35	1233.1	2.98	371.6388	11.94	262.147
	1024	**1.25**	1259.41	1.39	2268.03	3.46	395.2373	14.98	291.517
	2048	**1. 33**	1065.18	1.51	4892.87	4.33	450.19	17.42	303.434
uk-2007-05	8	**1.03**	3896.43	1.12	1240.08	1.56	1178.78	2.24	1187.32
	32	**1.05**	3851.37	1.16	1508.18	2.05	1364.442	4.57	1077.38
	128	**1.09**	3844.68	1.22	2049.12	2.64	1142.67	6.98	1006.03
	256	**1.12**	3878.09	1.26	2728.56	3.06	1154.225	8.90	846.077
	512	**1.17**	3850.61	1.30	4304.9	3.60	1178.537	11.26	1073.75
	1024	**1.22**	3860.52	1.38	8187.78	4.39	1521.79	14.05	1151.92
	2048	**1.28**	3853.47	1.51	16997.7	5.58	1453.76	16.37	1165.52
sk-2005	8	**1.12**	2341.45	1.20	635.067	1.74	623.442	2.46	525.734
	32	**1.15**	2287.56	1.25	752.249	3.06	760.468	5.10	676.57
	128	**1.24**	2057.34	1.33	933.922	4.67	648.516	7.31	549.877
	256	**1.30**	2047.7	1.38	1791.23	5.65	657.114	8.94	551.149
	512	**1.37**	2270.33	1.51	3030.68	7.04	684.072	10.82	687.486
	1024	**1.43**	2266.85	1.72	6303.33	8.93	776.8	12.86	561.702
	2048	**1.52**	2077.48	1.99	13497.4	9.79	939.648	14.89	576.28
Geo. Mean		1.98	2929.35	2.16	2451.50	4.67	878.96	12.25	688.66

Table 5.3: Results of experiments on the Huge Set in Table 5.1. Here, we compare HSE(256x), i.e, HSE with a buffer size of $256 \cdot 1024$, with state-of-the-art streaming edge partitioners on huge graph instances. We exclude HDRF as it is outperformed in both runtime and solution quality by 2PS-HDRF, while showing a similar trend against HSE. The best solution quality for each instance is emboldened.

CHAPTER 6

Discussion

6.1 Conclusion

In this work, we proposed HeiStreamEdge, a buffered streaming edge partitioner that achieves state-of-the-art results in solution quality. HeiStreamEdge uses a buffered approach, where batches of vertices, along with their edges, are loaded and partitioned sequentially. From each loaded batch of vertices, we generate a novel Contracted Split-and-Connect (CSPAC) graph, which is derived from the Split-and-Connect transformation introduced by Li et al. [45]. In this CSPAC graph, vertices are edges of the input graph, and edges model adjacencies among edges of the input graph. We prove that a vertex partition of the CSPAC graph yields an edge partition of the input graph with an approximation factor of $\mathcal{O}(\Delta\sqrt{\log n \log k})$ of the optimal edge partition, where n is the number of vertices, Δ is the maximum degree of the input graph, and k is the number of blocks of partition. The CSPAC graph for each batch is extended using artificial edges and vertices representing block assignments of edges visited in previous batches. We then partition the extended CSPAC graph using a multilevel scheme, wherein the initial partitioning is computed using a modified version of the generalized Fennel algorithm proposed by Faraj and Schulz [22]. This modification allows our overall runtime to be asymptotically independent of k. In particular, we achieve a linear runtime of $\mathcal{O}(n + m)$. Additionally, to the best of our knowledge, we are the only stateful streaming edge partitioner whose memory consumption is asymptotically independent of k. As a consequence, we not only achieve the best solution quality for all k values, but also, we are faster than most, and require less memory than all alternate stateful streaming edge partitioners for large k values.

6.2 Future Work

Our work opens up several lucrative avenues for future work, to further improve solution quality, or to save running time. Firstly, we can apply restreaming to yield a lower (better) replication factor. To implement restreaming, the general structure of HeiStreamEdge would remain as proposed. However, after the first round of streaming, we would adapt the per-batch graph model to also include edges to future batches, for which, we would now have block assignment decisions. These assignment decisions can be incorporated into our graph model using the modes described in Section 4.3. Note that currently, we only consider edges to previous batches, as we only have partitioning decisions for previously visited edges.

Secondly, we can save on overall running time by implementing asynchronous graph model construction and partitioning. After the first batch of vertices is loaded, and the corresponding CSPAC graph model is constructed from it, we can simultaneously partition the batch, and construct the graph model for the following batch. In this case, we expect a slightly worse solution quality, as we cannot incorporate block assignment decisions of the preceding batch to the current batch for which we are developing the graph model. On the other hand, we can achieve a speedup up to a factor of 2, as for most instances, graph model construction and partitioning required similar amount of time in our experimentation. This asynchronous execution, however, comes at the cost of greater memory consumption, as at any given moment, we are dealing with two batch graph models loaded into memory.

Thirdly, building on the asynchronous approach, another direction for future work is a parallel implementation of our overall buffered streaming model, wherein every processing element (PE) can operate on its own buffer of vertices along with their edges. In this case, however, we would not be able to extend the CSPAC graph for each batch with block assignment decisions of edges in other batches, which may be getting processed simultaneously on another PE. Thus, the approach would lead to a worse replication factor. Taking the "no model mode" configuration we tested in Section 5.3.2 as an analogous scenario where we incorporate no past batch assignment decisions, we can expect replication factor to be approximately 10.5% worse than the minimal mode configuration (Section 4.3) we compose HeiStreamEdge with in this work to leverage past block assignment decisions.

Finally, to improve solution quality for large k values, we suggest the exploration of a dynamic parameter x in the expression of the size of the coarsest graph model of our multilevel partitioning scheme, i.e., $\max(\frac{|\beta|}{2xk}, xk)$. As demonstrated in the coarsest graph size tuning experiment (Section 5.3.6) and the buffer size experiment (Section 5.4), to continue improving solution quality for large k values, we require the size of the coarsest graph to be suitably large, relative to the size of the input graph and choice of k. This would allow the initial partitioning step to be performed on a graph that contains more complex global graph structures, thereby resulting in better solution quality overall. To this end, function tuning experiments, or potentially machine learning, can be deployed to identify an optimal function for x given k and the input graph size (or buffer size).

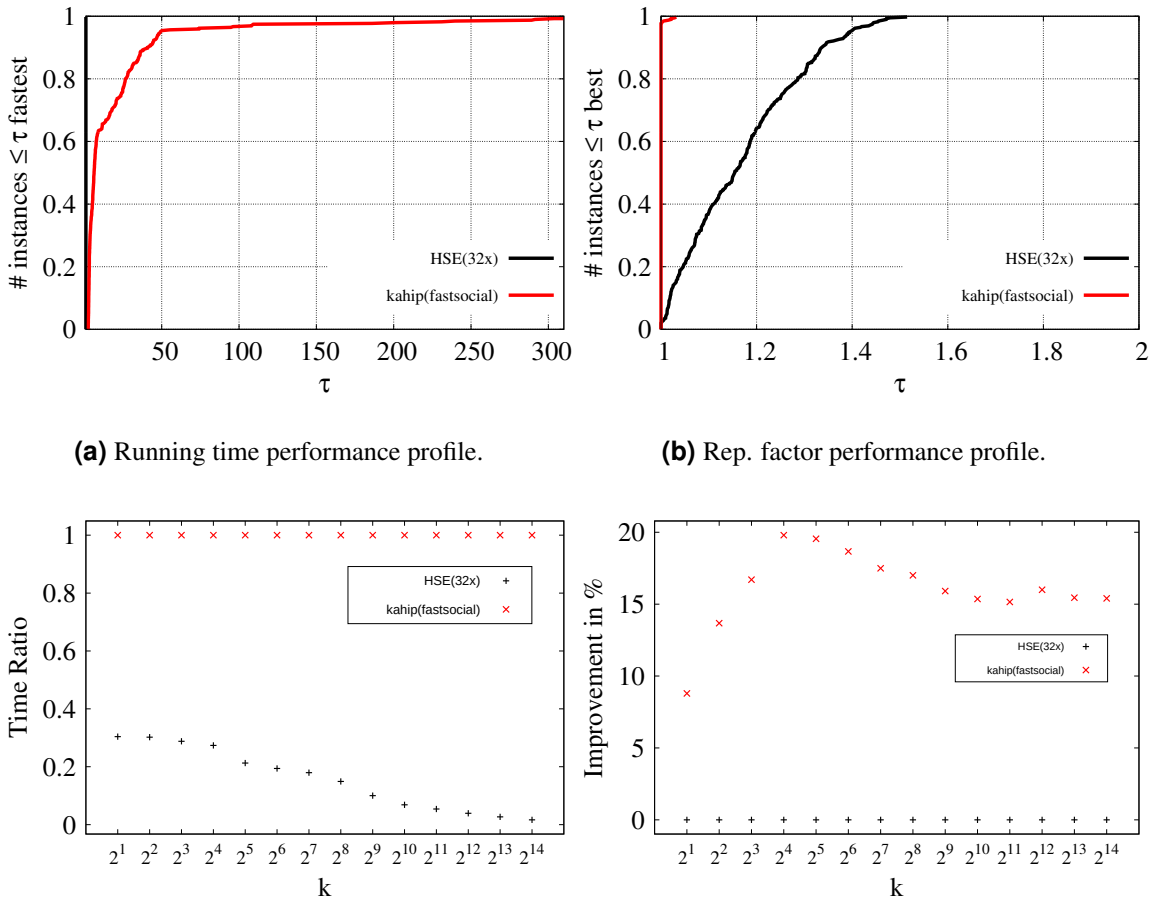
Further Results

Comparison with KaHIP

As shown in Figure A1, HSE(32x) is faster than KaHIP for all instances, but produces worse solution quality. On average, across all k values, HEP(32x) is 88.75% faster than KaHIP(fastsocial), however, KaHIP(fastsocial) produces 16.04% better solution quality. Comparing with an increased buffer size of 1024x, KaHIP(fastsocial) produces a 8.97% better solution quality, but HSE(1024x) is 85.26% faster on average across all k values.

Comparison with HEP

Figure A2 shows experimental results comparing HSE(32x) with HEP(1) on the Test Set in Table 5.1 (excluding Dubcova1 for which HEP(1) failed to execute for $k \geq 2048$). Here, HEP(1) refers to the Hybrid Edge Partitioner [49] with $\tau = 1$. By setting $\tau = 1$, we instruct HEP to partition all edges $e = (u, v)$, for which $d(u)$ and $d(v)$ are greater than $\tau \cdot \phi_d$ (if $\tau = 1$, simply ϕ_d), where ϕ_d is the mean degree of all vertices of the input graph, with a streaming algorithm, and the remaining edges with an in-memory partitioner. We find that HSE(32x) produces 4.48% better solution quality than HEP(1) on average across all k values, while being 53.84% faster. Figure A2b demonstrates that HSE(32x) produces better solution quality for approximately 60% of all instances. Figure A2c shows that while HEP(1) is faster for k values below $k = 2^8 = 256$, HSE(32x) is significantly faster above that. In particular, for $k \geq 512$, HSE(32x) is 92.82% faster than HEP(1) on average.



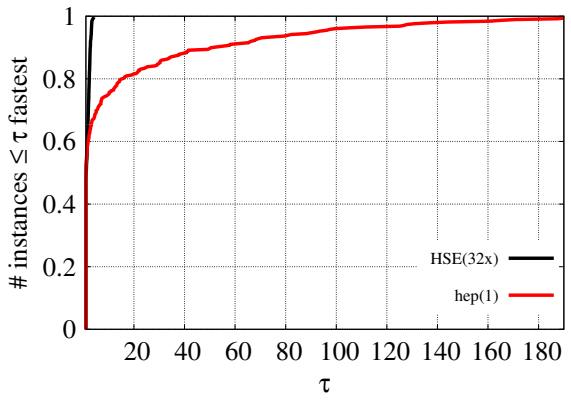
(a) Running time performance profile.

(b) Rep. factor performance profile.

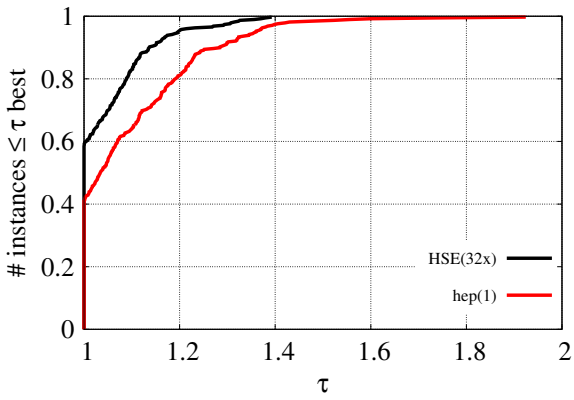
(c) Running time ratio.

(d) Rep. factor improvement over HSE(32x).

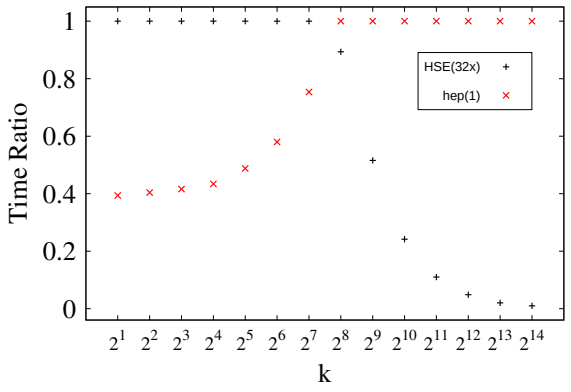
Figure A1: Comparison of KaHIP with fastsocial configuration, with HSE(32x) where x = 1024, i.e., buffer size is 32768 on the Test Set in Table 5.1.



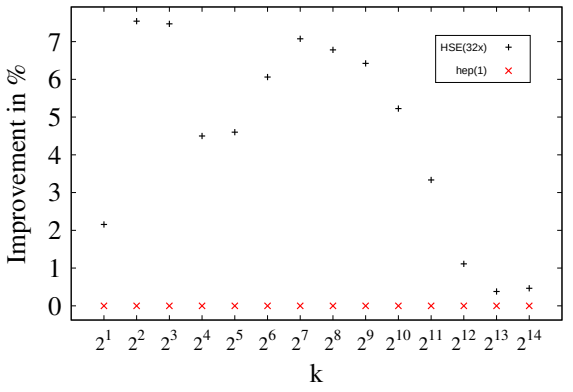
(a) Running time performance profile.



(b) Rep. factor performance profile.



(c) Running time ratio.



(d) Rep. factor improvement over HEP(1).

Figure A2: Comparison of HEP composed with $\tau = 1$ with HSE(32x) where x = 1024, i.e., buffer size is 32768 on the Test Set in Table 5.1.

Zusammenfassung

Die Partitionierung eines Graphen in ausgeglichene Blöcke ist ein wichtiger Vorverarbeitungsschritt für die verteilte Verarbeitung von Graphen. Bei der Kantenpartitionierung wird die Kantenmenge eines Eingangsgraphen in k ungefähr gleich große Blöcke aufgeteilt, wobei die Replikation von Knoten, ein Maß für die Qualität einer Partionierung, über die Blöcke hinweg minimiert wird. Streamende Partitionierer können riesige Graphen mit weniger Rechenressourcen partitionieren als In-Memory Partitionierer. In dieser Arbeit schlagen wir ein gepuffertes Streaming-Modell für die Kantenpartitionierung vor, das schrittweise Stapel von Kanten lädt und ihnen dauerhaft Blöcke zuweist. Für jeden Stapel konstruieren wir eine umfassende Graphenrepräsentation, die Adjazenzbeziehungen zwischen den Kanten modelliert, und partitionieren sie mithilfe eines mehrstufigen Schemas. Unser Ansatz ist sowohl in Laufzeit als auch Speicherbedarf asymptotisch unabhängig von k. In Experimenten zeigen wir, dass unser Algorithmus bessere Lösungsqualität liefert als alle konkurrierenden Algorithmen und bei großen k-Werten wesentlich schneller ist und weniger Speicher benötigt als vergleichbare qualitativ hochwertige Algorithmen.

Bibliography

[1] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, page 124, USA, 2006. IEEE Computer Society.

[2] Yaroslav Akhremtsev, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Engineering a direct *k*-way hypergraph partitioning algorithm. In Sándor P. Fekete and Vijaya Ramachandran, editors, *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017*, pages 28–42. SIAM, 2017.

[3] C.J. Alpert, Jen-Hsin Huang, and Andrew Kahng. Multilevel circuit partitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17:655 – 667, 09 1998.

[4] Amel Awadelkarim and Johan Ugander. Prioritized restreaming algorithms for balanced graph partitioning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, pages 1877–1887, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Cevdet Aykanat, Berkant Cambazoglu, and Bora Uçar. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68:609–625, 05 2008.

[6] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. *Benchmarking for Graph Clustering and Partitioning*, pages 161–171. Springer New York, New York, NY, 2018.

[7] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.

[8] Florian Bourse, Marc Lelarge, and Milan Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1456–1465, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Gorke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008.

[10] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, May 1992.

[11] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham, 2016.

[12] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4):28–38, 2015.

[13] Ümit Çatalyürek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, jul 1999.

[14] Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *ACM Comput. Surv.*, 55(12), mar 2023.

[15] Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya, and Bora Uçar. Umpa: A multi-objective, multi-level partitioner for communication minimization. In *Graph Partitioning and Graph Clustering*, 2012.

[16] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. Association for Computing Machinery.

[17] C. Chevalier and F. Pellegrini. PT-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8):318–331, jul 2008.

[18] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, aug 2015.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, jan 2008.

[20] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, Jan 2002.

[21] Kamal Eyubov, Marcelo Fonseca Faraj, and Christian Schulz. FREIGHT: Fast Streaming Hypergraph Partitioning. In Loukas Georgiadis, editor, *21st International Symposium on Experimental Algorithms (SEA 2023)*, volume 265 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[22] Marcelo Fonseca Faraj and Christian Schulz. Buffered streaming graph partitioning. *ACM J. Exp. Algorithmics*, 27, oct 2022.

[23] Daniel Funke, Sebastian Lamm, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 336–347, 2018.

[24] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. Association for Computing Machinery.

[25] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete graph problems. *Theor. Comput. Sci.*, 1:237–267, 1976.

[26] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, October 2012. USENIX Association.

[27] Lars Gottesbüren, Tobias Heuer, and Peter Sanders. Parallel flow-based hypergraph partitioning. In *20th International Symposium on Experimental Algorithms (SEA 2022)*, volume 233, pages 5:1–5:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[28] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable shared-memory hypergraph partitioning. In *23rd Workshop on Algorithm Engineering and Experiments (ALENEX 2021)*, pages 16–30. SIAM, 2021.

[29] Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Shared-memory n-level hypergraph partitioning. In *24th Workshop on Algorithm Engineering and Experiments (ALENEX 2022)*. SIAM, 01 2022.

[30] H. Halberstam and R. R. Laxton. Perfect difference sets. *Glasgow Mathematical Journal*, 6(4):177–184, 1964.

[31] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis S. Liu, Georgios Theodoropoulos, and Wentong Cai. Distributed edge partitioning for trillion-edge graphs. *Proc. VLDB Endow.*, 12(13):2379–2392, 2019.

[32] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534, nov 2000.

[33] Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[34] Alexandre Hollocou, Julien Maudet, Thomas Bonald, and Marc Lelarge. A streaming algorithm for graph clustering. *CoRR*, abs/1712.04337, 2017.

[35] Laurent Hyafil and Ronald L Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. *Tech. rep. IRIA â€“ Laboratoire de Recherche en Informatique et Automatique*, 1973.

[36] Nilesh Jain, Guangdeng Liao, and Theodore L. Willke. Graphbuilder: Scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, New York, NY, USA, 2013. Association for Computing Machinery.

[37] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. Social hash partitioner: A scalable distributed hypergraph partitioner. *Proc. VLDB Endow.*, 10(11):1418–1429, 2017.

[38] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[39] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[40] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *Proceedings - Design Automation Conference*, 11, 12 1998.

[41] R.T. Heaphy R.G. Bisseling K.D. Devine, E.G. Boman and Ümit Çatalyürek. Parallel hypergraph partitioning for scientific computing. *International Conference on Parallel and Distributed Processing (IPDPS)*, 20:124–124, 2006.

[42] Robert Krauthgamer, Joseph Naor, and Roy Schwartz. Partitioning graphs into balanced components. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 942–949. SIAM, 2009.

[43] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[44] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[45] Lingda Li, Robel Geda, Ari Hayes, Yanhao Chen, Pranav Chaudhari, Eddy Zhang, and Mario Szegedy. A simple yet effective balanced edge partition model for parallel computing. *ACM SIGMETRICS Performance Evaluation Review*, 44:6–6, 06 2017.

[46] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, apr 2012.

[47] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. Association for Computing Machinery.

[48] Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 685–695, 2018.

[49] Ruben Mayer and Hans-Arno Jacobsen. Hybrid edge partitioner : Partitioning large power-law graphs under memory constraints. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1289–1302. Association for Computing Machinery, New York, 2021.

[50] Ruben Mayer, Kamil Orujzade, and Hans-Arno Jacobsen. 2ps: High-quality edge partitioning with two-phase streaming. *CoRR*, abs/2001.07086, 2020.

[51] Ruben Mayer, Kamil Orujzade, and Hans-Arno Jacobsen. Out-of-core edge partitioning at linear run-time. In *38th IEEE International Conference on Data Engineering,*

ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022, pages 2629–2642. IEEE, 2022.

[52] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Partitioning complex networks via size-constrained clustering. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms*, pages 351–363, Cham, 2014. Springer International Publishing.

[53] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1106–1114, New York, NY, USA, 2013. Association for Computing Machinery.

[54] Md Anwarul Kaium Patwary, Saurabh Garg, and Byeong Kang. Window-based streaming graph partitioning algorithm. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '19, New York, NY, USA, 2019. Association for Computing Machinery.

[55] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. Hdrf: Stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, pages 243–252, New York, NY, USA, 2015. Association for Computing Machinery.

[56] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Phys. Rev. E*, 76:036106, Sep 2007.

[57] 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*, 2015.

[58] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, New York, NY, USA, 2013. Association for Computing Machinery.

[59] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magnús M. Halldórsson, editors, *Algorithms – ESA 2011*, pages 469–480, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[60] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms*, pages 164–175, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[61] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via n-level recursive bisection. *CoRR*, abs/1511.03137, 2015.

[62] Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Darren Strash. Scalable edge partitioning. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019*, pages 211–225. SIAM, 2019.

[63] Christian Schulz. *High Quality Graph Partitioning*. PhD thesis, 2013.

[64] Christian Schulz and Darren Strash. *Graph Partitioning: Formulations and Applications to Big Data*, pages 1–7. Springer International Publishing, Cham, 2018.

[65] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. Association for Computing Machinery.

[66] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 333–342, New York, NY, USA, 2014. Association for Computing Machinery.

[67] Brendan Vastenhouw and Rob Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47, 06 2002.

[68] Chris Walshaw and Mark Cross. Jostle: Parallel multilevel graph-partitioning software - an overview. *Mesh Partitioning Techniques and Domain Decomposition Techniques*, 01 2007.

[69] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 27. Curran Associates, Inc., 2014.

[70] Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. Graph edge partitioning via neighborhood heuristic. In *KDD 2017 - Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 605–614. Association for Computing Machinery, August 2017. Publisher Copyright: © 2017 Copyright held by the owner/author(s).; 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2017 ; Conference date: 13-08-2017 Through 17-08-2017.