Engineering Efficient Hypergraph b-Matching Semi-Streaming Algorithms

Daniel Heidemann

April 11, 2026

4262884

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

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

Co-Supervisor:
Henrik Reinstädtler

Acknowledgments

I want to thank Prof. Schulz for guiding me into the field of algorithm engineering and for providing me with the opportunity to write this thesis. Many thanks to Henrik for supervising this thesis and providing valuable feedback; not only during this work but throughout my studies. Finally, I am very grateful to my friends and family who have been there for me during my time in Heidelberg. I really enjoyed it.

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, April 11, 2026

Daniel Heidemann

Abstract

The semi-streaming b-matching problem on a weighted hypergraph $H = (V, E, \omega)$ asks for a subset of hyperedges $M \subseteq E$ that maximizes the total weight while ensuring that each vertex $v \in V$ is contained in at most $b(v)$ selected edges. This generalizes the standard matching problem, where $b(v) = 1$, and is motivated by large-scale applications in which the input is too large to store explicitly. In the semi-streaming model, the algorithm may use only $\mathcal{O}\big(n \cdot \mathrm{polylog}(n)\big)$ memory and is restricted to a small number of passes over the input stream.

This thesis studies how such algorithms can be implemented efficiently in practice for weighted hypergraph b-matching. Building on the semi-streaming framework of Huang and Sellier, we implement the streaming phase and greedy reconstruction phase and adapt them to a general hypergraph setting with vertex capacities. In addition, we explore several engineering-oriented improvements: a memory-efficient pruning strategy based on linked data structures and reference counting, a second-pass augmentation that reconsiders the input stream to recover additional matching weight, a bucketing approach that processes the stream in chunks, and a hybrid refinement phase based on Iterated Local Search by Großmann et al. We also investigate a parallel variant of the refinement step to better exploit multi-core hardware in a streaming context.

The implemented methods are evaluated on large benchmark hypergraphs with respect to running time, memory usage, and matching quality. The experiments show that the memory-efficient implementation reduces memory consumption in practice, that a second pass can substantially improve solution quality, and that bucketing combined with local search can yield further gains. At the same time, the results highlight the strong influence of edge ordering on the behavior of semi-streaming algorithms. Overall, the thesis demonstrates that semi-streaming hypergraph b-matching can be translated into practical implementations that balance memory efficiency and solution quality.

Contents

CHAPTER 1

Introduction

In the first chapter, we motivate the theoretical semi-streaming b-matching problem on hypergraphs by considering real-world use-cases in Section 1.1. Then, we present our contributions in Section 1.2 and structure the following chapters in Section 1.3.

1.1 Motivation

Datasets often contain complex, multi-way relationships. Standard graphs represent connections between exactly two entities. However, many real-world systems involve groups of multiple entities. *Hypergraphs* provide a natural framework for these interactions by allowing edges to connect any number of vertices.

Hypergraph b-matching generalizes many of these practical problems. Weighted set packing [32], resource allocation where resources have capacities [42], admission control in multi-tenant systems [17], higher-order clustering [22], and recommendation subroutines that must respect user/item budgets [1, 35]. Solving b-matching at scale, therefore, unlocks direct applications across social networks [41, 39], machine translation [29], ad allocation [6, 23], and beyond.

These datasets are large or arrive as streams (logs, clickstreams, network traces) and cannot be processed easily with large in-memory algorithms. The semi-streaming model, where the algorithm processes edges sequentially under memory roughly proportional to the maximum solution size $\mathcal{O}\big(n \cdot \text{polylog}(n)\big)$, is a useful computational abstraction for these settings. It captures trade-offs between passes over the data, memory usage, and approximation quality. While there is theoretical work on streaming and semi-streaming algorithms for graph and hypergraph problems, practical challenges remain in translating those results into implementations that are efficient and robust on real datasets.

1.2 Our Contribution

We propose concrete implementations of the Algorithms 1 and 2 by Huang and Sellier, introduced in [20]. Our approach handles the generalized b-matching problem on hypergraphs, where each vertex $v \in V$ has a capacity constraint $b_v \geq 1$.

To manage the incoming edge-stream in the semi-streaming context, we implement an architecture that is divided into a *streaming* phase and a *greedy construction* phase. For every vertex v, the system maintains a set of b_v queues, effectively representing the b_v available "slots" an edge may fill in the final b-matching. Our modular design allows for a $\mathcal{O}(1)$ admission check, leading to high streaming throughput.

Memory-Efficient Implementation. A significant challenge in semi-streaming is the linear growth of the candidate stack. We implement a memory-efficient variant of the streaming phase (see Algorithm 1 for $d = 1$) in Section 4.2 which reduces memory consumption by up to 30% compared to the non-discarding variant. This implementation replaces static vectors with doubly-linked lists and custom iterators to facilitate $\mathcal{O}(1)$ pruning of stale candidate edges. Our contribution includes a practical implementation of the theoretical space-saving threshold β. When a vertex's queue exceeds the size β, the "bottom" edges (those with the lowest reduced weights) are marked as *erasable*. To prevent premature deallocation of hyperedges that are still relevant to other vertices, we introduce a reference-counting mechanism. An edge is only physically removed from memory when it is marked as erasable and its *reference* count — the count of queues in which it currently resides at the front — reaches zero. In Section 5.2 we show that this engineering technique ensures that the algorithm operates within the predicted $\mathcal{O}\big(\log_{1+\varepsilon}(1/\varepsilon) \cdot |M_{\text{max}}| + \sum_{v \in V} b_v\big)$ space bounds while remaining fast.

Second Pass. In Section 4.3 we describe a technique where a second pass simply streams the whole set of edges again. Each edge that is currently disjoint from the stored b-matching but is matchable is added to the b-matching. This procedure only requires storing the current b-matching while the edge stream is not retained. It therefore respects the semi-streaming space bound. Because we only add edges that are disjoint from the existing b-matching, the total weight is monotonically non-decreasing. Leading to quality improvements of up to 94% compared to the first pass, and 93% compared to an offline greedy implementation in practice. The second pass can only preserve or increase the b-matching weight. In instances where the first pass made conservative choices due to ordering, the extra scan can recover edges that were previously blocked. The cost is only one additional linear scan of the input edges. We do not claim unconditional worst-case guaranties beyond these facts. The evaluation in Section 5.3.1 and 5.4, however, shows that the magnitude of the improvement depends on the instance size and ordering of incoming edges.

Bucketing. Another approach we use to tackle the ordering-problem already in the first pass is dividing the input stream into *buckets*. Over time, the gain an edge needs to bring to be chosen is much higher than in the beginning. Therefore, an edge that appears later, while being better than an earlier candidate, might not be selected. To improve the quality of the matching, we therefore propose dividing the input stream into $\log(|V|)$ buckets. After each bucket, a matching is determined and stored. Then, all queues as well as the stack are freed. From the results, we construct a new graph and determine the final matching. Especially in more complex instances, the evaluation in Section 5.3.1 shows that we can improve the solution quality of up to 98%. In Section 4.4, we will show that this technique does not violate the semi-streaming memory bound.

Integrated Iterated Local Search. While semi-streaming algorithms provide strong theoretical guaranties on approximation ratios, their empirical performance can often be improved through local optimizations. In Section 4.5 we propose a hybrid refinement phase that occurs after the *greedy construction* phase. Once the stream is processed and a maximal matching is extracted from the candidate stack using the reverse-pass greedy selection (Algorithm 2), we apply an Iterated Local Search (ILS), a metaheuristic that repeatedly perturbs and locally improves the current matching, refinement proposed by Großmann et al. [16]. Using the in-place ILS engine, we perform k iterations of local swaps and augmentations between the original b-matching and the candidate stack. Therefore, this refinement is performed on the induced subgraph stored in the semi-streaming space, meaning it does not violate the memory constraints of the model. As edge-sizes grow, in Section 5.3 we show that this hybrid approach can reliably improve the final b-matchings of each our semi-streaming variants up to 20%.

(Parallel) ILS on Buckets. In this refinement, we exploit the bucketed stream decomposition by running independent ILS engines on the matchings produced from each chunk (see Section 4.5). Each bucket yields a matching on which we perform an in-place ILS using only the semi-streaming memory already allocated. Because the local searches operate on disjoint matchings, they require no inter-process communication. To further enhance the running time, we can safely parallelize the local improvements. After the per-bucket refinements are complete, the matchings are combined into a complete matching of the whole instance. This improves our bucketing technique by up to 13%. The evaluation in Section 5.3 further shows that this technique brings a weight-gain of 94% compared to a single sequential ILS applied after the full stream. Adding an ILS refinement to the technique, we can report quality-improvements of up to 126% compared to the basic semi-streaming method proposed by Huang and Sellier [20].

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 introduces the fundamental concepts and notation used throughout the thesis. In particular, it provides the definitions of hypergraphs, b-matching, approximation guaranties, and the semi-streaming model that form the theoretical basis for the algorithms studied later on. Chapter 3 surveys the related literature and positions this work in the context of previous results on streaming and semi-streaming matching, with particular attention to the algorithms of Huang and Sellier [20] and to earlier engineering efforts for graph and hypergraph matching.

Chapter 4 presents the algorithmic core of the thesis. It explains the semi-streaming b-matching framework in detail and describes the concrete implementations of the streaming phase and greedy construction phase. In addition, it introduces the engineering extensions developed in this work, including the memory-efficient pruning mechanism, the second-pass augmentation strategy, the bucketing approach, and the hybrid refinement based on Iterated Local Search. Whenever appropriate, the Chapter is supported by pseudocode and illustrations that clarify the interaction between the individual components.

Chapter 5 evaluates the proposed implementations experimentally. It describes the benchmark instances, the implementation environment, and the evaluation methodology before analyzing the algorithms with respect to memory usage, running time, and matching quality. The Chapter further investigates how the results depend on the ordering of the input stream and on the structural properties of the underlying hypergraphs. Finally, Chapter 6 reflects on the main findings, discusses their implications, and summarizes the contributions of the thesis in the concluding section.

CHAPTER 2

Fundamentals

In the following Chapter, we will revisit the definitions used and needed in this thesis. We will start by defining general concepts and methods for the semi-streaming b-matching in Section 2.1. Afterwards, we present definitions regarding the implemented Algorithms 1 and 2 in Section 2.2.

2.1 General Definitions

Hypergraph. A *weighted undirected hypergraph* $H = (V, E, \omega)$ consists of a vertex set V with $|V| = n$ and a collection E of m hyperedges. Each hyperedge $e \in E$ is a subset of V and carries a positive weight given by the function $\omega \colon E \to \mathbb{R}_{>0}$. The size of an edge e is its number of vertices, denoted $|e|$, and the hypergraph's rank, the largest edge size, is $d := \max_{e \in E} |e|$. For brevity, we will often write "edge" instead of "hyperedge" when the hypergraph context is clear. To give some intuition on hypergraphs, we refer the reader to Figure 2.1. The difference from standard graphs is that an edge (blue) can connect any number of vertices (green).

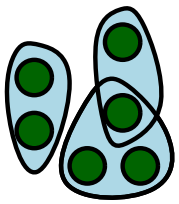

Figure 2.1: Example hypergraph with $d = 3$.

(b-)Matching. Let $G = (V, E)$ be a graph or hypergraph, and let $w : E \to \mathbb{R}$ be a weight function on the edges. A subset $M \subseteq E$ is called a *matching* if the chosen edges are pairwise disjoint, i.e. no vertex is contained in more than one edge of M. The *weight* of a matching M is $|M| := \sum_{e \in M} w(e)$, and a *maximum (weight) matching* is a matching that attains the largest possible value of $|M|$ over all matchings. A b-matching is *maximal* if no edge from $E \setminus M$ can be added to M without violating the b-matching condition $\forall v \in V : \left|\{ e \in M : v \in e \}\right| \leq b(v)$.

Semi-Streaming Algorithms. If the input size exceeds a machine's available memory, a standard approach is to process the input in a streaming manner. There are multiple formal notions of streaming for graphs and hypergraphs. In the (semi-)streaming framework, the (hyper-)edges of a (hyper-)graph are typically presented one at a time, in an arbitrary, possibly adversarial order, and algorithms may be allowed several passes over this sequence. Under the semi-streaming model, the memory is generally bounded by the solution size. Consequently, for the general b-matching, we obtain $\mathcal{O}\big(\sum_{v\in V} b_v \cdot \operatorname{polylog}(n)\big)$ memory. We assume uniform capacities that are constant $\times n$, therefore $\mathcal{O}\big(n \cdot \operatorname{polylog}(n)\big)$.

Approximation Factors. Algorithms may be divided into three types: exact algorithms, heuristics without formal approximation guaranties, and approximation algorithms. The effectiveness of an approximation algorithm is evaluated by comparing the value of its output to that of an optimal solution. For a maximization instance I, denote the optimal objective value by $M(I)$. An algorithm $\mathcal{A}$ is said to provide an α-approximation if, for every instance I, it returns a solution of value at least $\alpha M(I)$, where $\alpha \in \mathbb{R}^{+}_{<1}$ is chosen as large as possible.

2.2 Algorithm Specific Definitions

Streaming Local-Ratio Selection. Let $H = (V, E, \omega)$ be a weighted hypergraph as described above, where each vertex $v \in V$ has a capacity b_v. For an edge $e \in E$ arriving in the stream, let $W(e)$ be the set of *witnesses*: the edges currently stored in the streaming buffer that are incident to e and occupy its required capacity. We define the *local cost* induced by the witnesses as $\Phi_e := \sum_{v\in e} \phi_v(e)$, where each $\phi_v(e)$ is the weight of the minimum-weight edge currently filling the capacity b_v at vertex v (or zero if the capacity has not yet been reached). The weight of the arriving edge is then decomposed as $\omega(e) = \Phi_e + \gamma_e$, where $\gamma_e := \max\{0, \omega(e) - \Phi_e\}$ is the *local gain* of edge e. In the streaming step, the edge e is selected and added to the buffer if and only if its local gain is strictly positive ($\gamma_e > 0$). If selected, the edge is recorded with a reduced weight proportional to γ_e, ensuring that the selection preserves the approximation invariant relative to the cumulative weight of the discarded witnesses.

Bucketing. Let the edge stream be partitioned into consecutive chunks of size $B = \lceil \log_\kappa(n) \rceil$, where $n = |V|$ and $\kappa \geq 2$ are fixed parameters. Bucketing denotes the strategy of processing the input stream chunk by chunk, running the streaming phase on each chunk independently, extracting a matching M_{chunk}, respectively and then clearing the temporary candidate structures before continuing with the next chunk. The resulting matchings are stored in a collection $\mathcal{M}$ and may later be combined into a final feasible solution.

CHAPTER 3

Related Work

This work is largely based on two Algorithms (1, 2) introduced by Huang and Sellier [20]. In the graph setting, they follow a series of papers [4, 8, 9, 13, 28, 30, 34, 40] with increasingly improved approximation guarantees for the matching problem in the semi-streaming model. Previous research has shown that a $2 + \varepsilon$ approximation guarantee is possible in a semi-streaming context for graphs. In [28], Levin and Wajc introduced a $3 + \varepsilon$ approximation for the general b-matching problem in the streaming context. Paz and Schwartzman [34] give a $2+\varepsilon$-approximation for graph matching; this result applies only to graphs. By using the same framework as in [28, 34], Huang and Sellier [20] propose a refined, queue-based data structure to store and manage edges selected in a local-ratio streaming phase. While achieving the same approximation guarantee as Paz and Schwartzman for 1-matching, Huang and Sellier manage to restrict their memory efficient optimization to $\mathcal{O}\big(\log_{1+\varepsilon}(1/\varepsilon) \cdot |M_{\text{max}}| + \sum_{v \in V} b_v\big)$ variables in memory, closing the gap to the simple matching problem in a semi-streaming context. Algorithm 1 has this memory optimization built in by setting $d = 1$. For convenience, we will refer to it as the *basic* ($d = 0$) and *memory efficient* ($d = 1$) implementation.

(b-)Matching in Graphs and Streaming. The matching problem and its capacity-constrained variant b-matching have been studied extensively in graphs. For ordinary graphs, Gabow [12] showed that b-matching can be reduced to standard matching by vertex splitting, although this is often impractical for large instances. On the approximation side, Mestre [31] proved that the greedy algorithm yields a half-approximation for graph b-matching, establishing a simple baseline that remains relevant in practice. In the streaming and semi-streaming setting, Feigenbaum et al. [9] present a $\frac{1}{6}$-approximation for weighted matching, McGregor [30] develop a multipass algorithm with a $\frac{1}{2+\varepsilon}$ guarantee, and Paz and Schwartzman [34] give the $\frac{1}{2+\varepsilon}$-approximation mentioned above. Ghaffari and Wajc [13] provide a simpler proof of the approximation ratio using a primal-dual analysis. Ferdous et al. [10] show empirically that the algorithm by Paz and Schwartzman can

compete quality-wise with offline $\frac{1}{2}$-approximation algorithms like GPA, while requiring less memory and time. Ferdous et al. [11] also present two semi-streaming algorithms for the related weighted k-disjoint matching problem, building upon the algorithms of Paz and Schwartzman and Huang and Sellier [20, 34] for streaming b-matching.

Hypergraph (b-)Matching. In hypergraphs, the matching problem becomes significantly harder. Hazan et al. [19] show strong inapproximability results for maximum set packing, which directly translate to hypergraph matching, and Hästad [21] proves similarly hard lower bounds for related non-uniform formulations. For weighted hypergraph b-matching, Krysta [25] gave a greedy $(k+1)$-approximation for k-uniform hypergraphs, while Parekh and Pritchard [33] improved this via linear programming. Koufogiannakis and Young [24] further studied distributed approximation algorithms for weighted uniform hypergraphs. Recent work by Reinstädtler et al. [36] proposes two one-pass streaming algorithms for hypergraph matching: one is an extension of [34] based on dual-variable stacks, and the other is a greedy swapping approach. These methods illustrate how classic semi-streaming strategies can be generalized to hypergraph settings, but they do not yet directly address general b-matching capacity constraints. Implementation-wise, their work lays the foundation for the experiments presented in this thesis. Engineering efforts for the offline general hypergraph b-matching have explored practical heuristics like data reduction and local search, improving the performance of weighted b-matching on real datasets [16]. We are unaware of any practical implementations of semi-streaming hypergraph b-matching algorithms.

Related $\mathcal{NP}$-hard Problems. Hypergraph b-matching generalizes the classical hypergraph matching problem, where the case $b = 1$ is already $\mathcal{NP}$-hard [36]. Accordingly, much of the hardness landscape surrounding hypergraph b-matching is captured by closely related set-packing formulations. In particular, hypergraph matching is tightly connected to k-set packing and k-dimensional matching, where one seeks a maximum family of pairwise disjoint k-sets. Recent work emphasizes that k-dimensional matching serves as a benchmark $\mathcal{NP}$-hard problem underlying these generalizations [27, 32]. The canonical 3-uniform special case, three-dimensional matching (3DM), is $\mathcal{NP}$-hard and is a standard source problem for reductions to hypergraph matching and set-packing variants [26].

CHAPTER 4

Semi-Streaming b-Matching Algorithm

This chapter gives a detailed description of the main ideas we propose in this thesis. First, we will introduce the base Algorithms 1 and 2 by Huang and Sellier [20] in Section 4.1. Then, we will cover the optimization strategies that we built on top. Starting with our memory efficient adaptation of Algorithm 1 ($d = 1$) in Section 4.2. We proceed by presenting two strategies tackling edge-ordering challenges: a naive second pass approach in Section 4.3 and a bucketing technique in Section 4.4. Using the ILS engine proposed by Großmann et al. [16], we will further refine the local matchings in Section 4.5.

4.1 Base Algorithms

We will now exemplify the base Algorithms (1, 2) in more detail. The semi-streaming b-matching algorithm follows a typical structure by dividing the process into a *streaming* phase and a *greedy construction* phase. Figure 4.1 and 4.2 illustrate the procedures presented here. We divide the work surfaces into two main areas labeled as *Queues* and *Stack*.

Streaming Phase. Huang and Sellier [20] present a basic algorithm and a refined, more memory efficient ($d = 1$) adaptation (see Algorithm 1). We will first describe the basic algorithm and adapt the memory efficient one on top in the next Section 4.2. The streaming phase fulfills one crucial task: Incoming edges (blue) from the hypergraph (top right) are processed and stored in the queues and stack (see Figure 4.1). Initially, while the stack is empty, memory is already allocated on the queues, reserving b_v queues for each vertex (green). The streaming phase maintains a collection of queues for each vertex $v \in V$, denoted as $Q_v = \{Q_{v,1}, \ldots, Q_{v,b_v}\}$, where initially $Q_{v,i} = \emptyset$ for all i. These queues represent distinct slots that facilitate a *local-ratio* selection policy, ensuring that the algorithm only retains $\mathcal{O}(n \cdot \text{polylog}(n))$ edges. To achieve this sparsity, the algorithm maintains a *reduced weight* for each slot to serve as a dynamic threshold for future edges. Specifically, for an incoming edge e, the algorithm identifies the minimum reduced weight among the

Algorithm 1 Streaming phase for weighted matching by Huang and Sellier [20].

$S \leftarrow \emptyset$
$\forall v \in V : Q_v \leftarrow (Q_{v,1} = \emptyset, \cdots, Q_{v,b_v} = \emptyset)$
for $e = e_t$, $1 \leq t \leq |E|$ an edge from the stream **do**
    for $u \in e$ **do**
        $w_u^*(e) \leftarrow \min\{w_u(Q_{u,q}.top()) : 1 \leq q \leq b_u\}$
        $q_u(e) \leftarrow q$ such that $w_u(Q_{u,q}.top()) = w_u^*(e)$
    if $w(e) > \alpha \sum_{u \in e} w_u^*(e)$ **then**
        $g(e) \leftarrow w(e) - \sum_{u \in e} w_u^*(e)$
        $S \leftarrow S \cup \{e\}$
        for $u \in e$ **do**
            $w_u(e) \leftarrow w_u^*(e) + g(e)$
            $r_u(e) \leftarrow Q_{u,q_u(e)}.top()$
            $Q_{u,q_u(e)}.push(e)$
            if $d = 1$ and $Q_{u,q_u(e)}.length() > \beta$ **then** ▷ remove some small element
                let e' be the $\beta + 1$-th element from the top of $Q_{u,q_u(e)}$
                mark e' as erasable, so that when it will no longer be on the top of any queue, it will be removed from S and from all the queues it appears in

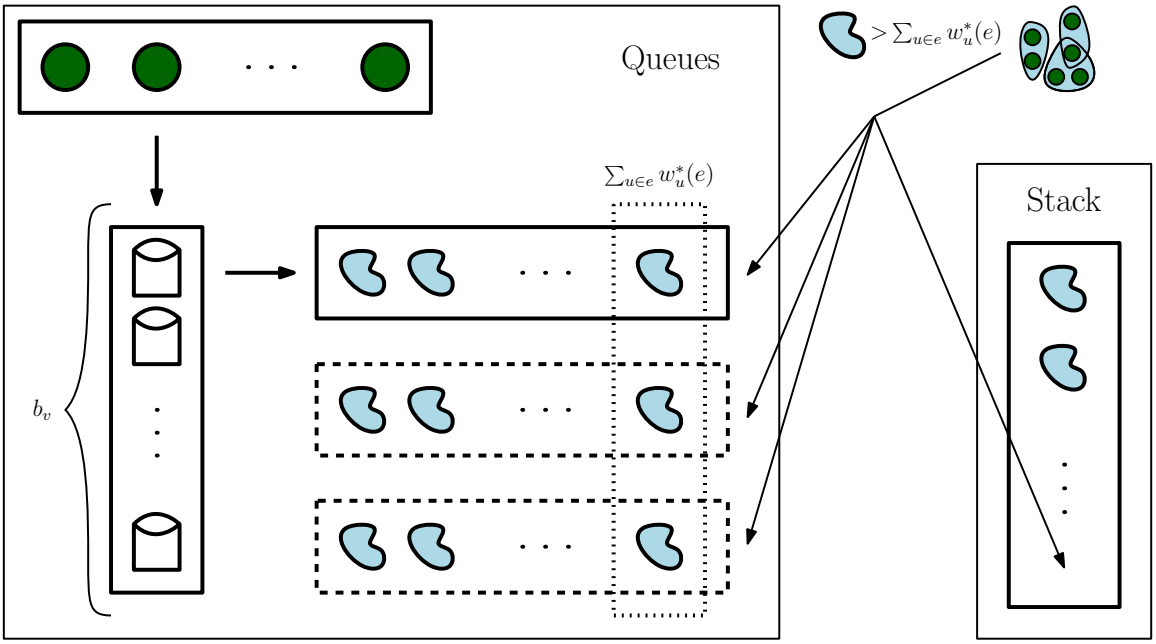

Figure 4.1: Streaming Phase from Algorithm 1.

slots for each endpoint $u \in e$, defined as $w_u^*(e)$. Simply put, the algorithm selects from each pin $u \in e$ the respective queue that offers the minimum current gain among all queues of v. The selected queues from all pins are illustrated as boxes in the *Queues* area to which the arrows point. An edge is admitted to the candidate set S only if its weight exceeds the aggregate threshold of its endpoints, i.e., $w(e) > \sum_{u \in e} w_u^*(e)$. Upon admission, we compute the marginal gain $g(e) = w(e) - \sum_{u \in e} w_u^*(e)$ and push e onto the specific queues that yielded $w_u^*(e)$. The reduced weight of these slots is then updated to $w_u(e) = w_u^*(e) + g(e)$, effectively accumulating the gain. This incremental thresholding ensures that each slot only accepts edges offering significant marginal improvement, thereby bounding the total number of stored edges while preserving the quality of the matching.

Greedy Construction Phase. The greedy construction phase creates a matching from the available edges selected during the streaming phase. As discussed previously, the threshold for edges arriving later in the stream to be considered for inclusion in the matching becomes progressively higher. This leads to an important assumption for the construction phase: Edges that appear later are probably better in terms of weight than earlier candidates. We therefore process the edges $e \in S$ in reverse order (LIFO). Please refer to Figure 4.2 for the following description of Algorithm 2. For the illustration, as the greedy construction phase works with the same data structure filled in during the streaming phase, we only need to add another area for the initially empty result set.

Algorithm 2 Greedy construction phase by Huang and Sellier [20].

```
1:  M ← ∅
2:  ∀e ∈ S : z_e ← 1
3:  for e ∈ S in reverse order do
4:      if z_e = 0 then continue                ▷ skip edge e if it is marked
5:      M ← M ∪ {e}
6:      for u ∈ e do
7:          c ← e
8:          while c ≠ ⊥ do
9:              z_c ← 0                         ▷ mark elements below e in each queue
10:             c ← r_u(c)
11: return M
```

For each edge on the stack, we store a boolean z_e, or more intuitively, whether it is *relevant* for the resulting matching M. After pushing a relevant edge e where $z_e = 1$ on M, all edges below e in the queues where e appears are marked as irrelevant ($z_e = 0$). By skipping them in the following iterations, the algorithm ensures it holds the b-matching constraint. The LIFO ordering is essential because the streaming phase ensures that edges arriving later are required to "pay" for potential gains of the edges they supersede. By following the predecessor pointers $r_u(c)$ and marking them as irrelevant, the algorithm

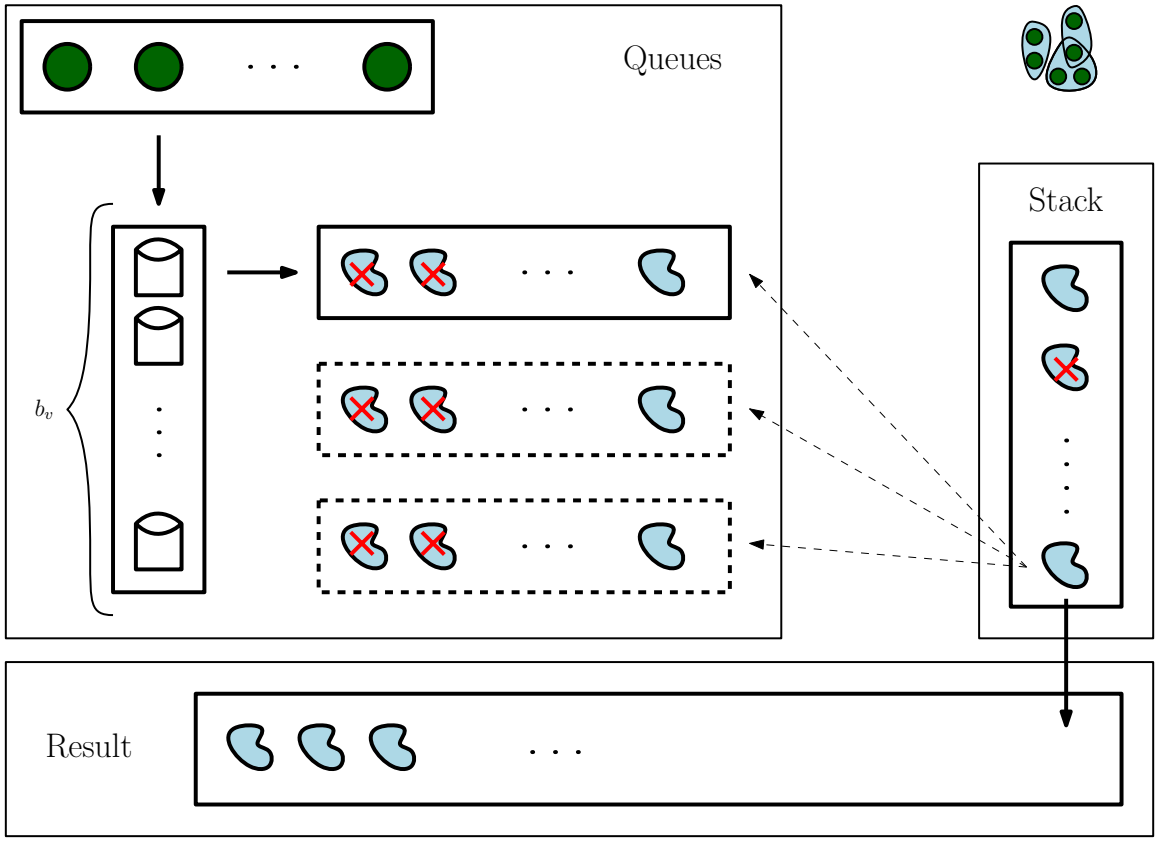

Figure 4.2: Greedy Construction Phase from Algorithm 2.

ensures that once a "slot" is occupied by a later edge, no earlier edges from that same queue can be selected. This mechanism naturally satisfies the vertex capacity constraints since each of the b_v queues for a vertex v can provide at most one edge to the matching M. Thus, the greedy phase efficiently extracts a feasible b-matching from the streaming candidates while preserving the approximation guaranties provided by the local-ratio method.

4.2 Memory-Efficient Pruning via Linked Structures

Huang and Sellier [20] propose a technique to efficiently remove edges from the storage that have no further impact on the resulting matching, freeing memory already in the streaming phase. As described in the introduction of this Chapter, the streaming phase relies on a set of queues, each representing a "slot" of the vertices' capacity that an edge may fill. Edges that outrank previous ones in terms of gained size reside at the top of these queues and define the current threshold for new arrivals. This leads to the main idea backing the proposed optimization. Edges at the back of these queues, which are no longer the "top" element for any of their incident vertices, eventually become redundant. Once an edge is sufficiently deep in all queues it belongs to, its potential contribution to the greedy phase is eclipsed by newer, heavier edges. Depending on a given $\varepsilon \leq \frac{1}{4}$, the algorithm maintains a "max-length" β for each queue, defined as:

$$\beta = 1 + \frac{2\log(1/\varepsilon)}{\log(1+\varepsilon)}.$$

The practical implementation of erasing edges while streaming comes with the overhead of two additional components in the data structure. We have included our specific implementation and interpretation in Algorithm 3. First, to ensure $\mathcal{O}(1)$ pruning of stale edges, the underlying queues and the candidate set S must be implemented as doubly-linked lists, allowing for the immediate removal of an element given its pointer. Secondly, for each edge e, we maintain a reference count $c(e)$ representing the number of queues where e is currently the front element. This counter is initialized to $|e|$ (the number of endpoints) and is decremented whenever e is superseded by a new incoming edge in a specific slot. An edge is safely erased from all structures if it is marked as *erasable* ($d(e) = 1$), meaning it has been pushed past the β-th position in at least one queue, and its reference count $c(e)$ reaches zero. This ensures that we only discard edges that are neither currently providing a threshold nor likely to be selected in the reverse-greedy phase due to the heavy weights of their successors. This engineering technique ensures the algorithm holds the desired memory bound of $\mathcal{O}\big(\log_{1+\varepsilon}(1/\varepsilon) \cdot |M_{\text{max}}| + \sum_{v \in V} b_v\big)$.

4.3 Weight Improvements via Second Pass

Due to the ordering of the incoming edges in the streaming phase, the local-ratio selection tends to make conservative choices in terms of edge selection. To address this, the second

Algorithm 3 Concrete memory-efficient streaming phase (adaptation of Algorithm 1 with $d = 1$ by Huang and Sellier [20]).

$S \leftarrow \emptyset$
$\forall v \in V : Q_v \leftarrow (Q_{v,1} = \emptyset, \cdots, Q_{v,b_v} = \emptyset)$
for $e = e_t$, $1 \leq t \leq |E|$ an edge from the stream **do**
    for $u \in e$ **do**
        $w_u^*(e) \leftarrow \min\{w_u(Q_{u,q}.top()) : 1 \leq q \leq b_u\}$
        $q_u(e) \leftarrow q$ such that $w_u(Q_{u,q}.top()) = w_u^*(e)$
    if $w(e) > \alpha \sum_{u \in e} w_u^*(e)$ **then**
        $g(e) \leftarrow w(e) - \sum_{u \in e} w_u^*(e)$
        $S \leftarrow S \cup \{e\}$
        $c(e) \leftarrow |e|$ ▷ initially on top of each appended queue
        $d(e) \leftarrow 0$
        for $u \in e$ **do**
            $w_u(e) \leftarrow w_u^*(e) + g(e)$
            $r_u(e) \leftarrow Q_{u,q_u(e)}.top()$
            $c(r_u(e)) \leftarrow c(r_u(e)) - 1$
            $Q_{u,q_u(e)}.push(e)$
            if $Q_{u,q_u(e)}.length() > \beta$ **then**
                $d(Q_{u,q_u(e)}[\beta + 1]) = 1$ ▷ set exceeding element erasable
            if $c(r_u(e)) = 0$ and $d(r_u(e)) = 1$ **then**
                $S \leftarrow S \setminus r_u(e)$ ▷ erase from stack
                for $u_r \in r_u(e)$ **do**
                    $Q_{u_r,q_{u_r}(r_u(e))} \leftarrow Q_{u_r,q_{u_r}(r_u(e))} \setminus r_u(e)$ ▷ erase from queues

pass performs a greedy augmentation by reconsidering edges from the original stream in their arrival order. This phase tries to fill the remaining capacity of each vertex v up to its budget $b(v)$ using edges that were not included in the initial selection. As for the naive nature, this approach lacks a guaranty of its improvement. For example, consider an edge connecting all vertices with remaining capacity but having a relatively small weight of 1. Ultimately, by blocking gain, other potentially better candidates may be brought in later in the stream. Nonetheless, this technique brings empirically better results in practice. The following pseudocode, Algorithm 4, details the logic of the augmentation phase, which operates within the standard semi-streaming space complexity.

Algorithm 4 Second pass.

```
1: M ← previous matching
2: for e = e_t, 1 ≤ t ≤ |E| an edge from the stream do
3:     if e ∉ M and ∀v ∈ e : |{e ∈ M : v ∈ e}| < b(v) then      ▷ is matchable
4:         M ← M ∪ {e}
5: return M
```

4.4 Buckets: Dividing the Input Stream in Chunks

As in the second pass improvement described in Section 4.3, the quality of streaming algorithms partially depends on the ordering of the incoming edges. We will now present another approach to addressing this issue by dividing the incoming edge-stream into $\log_\kappa(|V|)$ "buckets" with $\kappa \geq 2$. We essentially break the edge-stream after $\frac{|V|}{\log_\kappa(|V|)}$ edges have been processed by Algorithm 1. Then, determine a matching from the chunk using Algorithm 2, push it to the set $\mathcal{M}$, and clear the whole storage. Proceeding with the stream afterwards. Figure 4.3 shows the final composition to get a complete matching for the instance. By combining the individual matchings from each chunk in $\mathcal{M}$ into a single hypergraph, we can perform an offline greedy computation in the semi-streaming model. Determining the final matching M_{full}. Hence, using the same arguments as in [20], we obtain the following.

Theorem 1. *The division of the incoming edge-stream into* $\log_\kappa(|V|)$ *buckets does not violate the semi-streaming memory bound.*

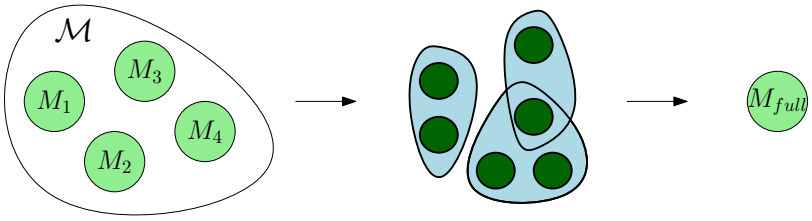

Figure 4.3: Example composition of multiple matchings M_1 - M_4 into one M_{full}.

Proof. After each chunk, we store only the resulting matching M_{chunk}. By resetting the temporary storage structures S and Q at the end of each chunk, the peak memory consumption for edge candidates is restricted to the local-ratio bound of a single partition. The persistent memory is then dominated by the list of matchings $\mathcal{M}$, which aggregates the results of each phase. Since each b-matching M_{chunk} is a feasible subgraph satisfying vertex capacities, its size is strictly bounded by $\sum_{v \in V} b_v$. Consequently, storing $\log_\kappa |V|$ such matchings requires $\mathcal{O}\big((\sum_{v \in V} b_v) \log_\kappa |V|\big)$ space, which remains within the semi-streaming limit of $\mathcal{O}\big(n \cdot \text{polylog}(n)\big)$. □

This allows the algorithm to effectively "forget" the high-volume candidate sets of previous chunks while retaining the most valuable edges in the form of compact, feasible matchings. Please refer to Algorithm 5 for the modified streaming procedure.

Algorithm 5 Streaming phase with bucketing (adaptation of Algorithm 1 with $d = 0$ by Hung and Sellier [20]).

$S \leftarrow \emptyset$
$\forall v \in V:\ Q_v \leftarrow (Q_{v,1} = \emptyset, \cdots, Q_{v,b_v} = \emptyset)$ ▷ b_v queues for vertex v
$B \leftarrow \lceil \log_\kappa n \rceil$ ▷ number of edges per chunk (bucketing)
$\mathcal{M} \leftarrow [\,]$ ▷ list of matchings appended after each chunk
$p \leftarrow 0$
for $e = e_t$, $1 \le t \le |E|$ an edge from the stream **do**
    for $u \in e$ **do**
        $w_u^*(e) \leftarrow \min\{w_u(Q_{u,q}.top()) : 1 \le q \le b_u\}$
        $q_u(e) \leftarrow q$ such that $w_u(Q_{u,q}.top()) = w_u^*(e)$
    if $w(e) > \alpha \sum_{u \in e} w_u^*(e)$ **then**
        $g(e) \leftarrow w(e) - \sum_{u \in e} w_u^*(e)$
        $S \leftarrow S \cup \{e\}$
        for $u \in e$ **do**
            $w_u(e) \leftarrow w_u^*(e) + g(e)$
            $r_u(e) \leftarrow Q_{u,q_u(e)}.top()$
            $Q_{u,q_u(e)}.push(e)$
    $p \leftarrow p + 1$
    if $p = B$ or $t = |E|$ **then** ▷ end of current chunk
        $M_{\text{chunk}} \leftarrow$ GREEDYCONSTRUCTION$(S, \{Q_v\}_{v \in V})$
        $\mathcal{M} \leftarrow \mathcal{M} \cup M_{\text{chunk}}$
        for all $v \in V,\ \forall q:\ Q_{v,q} \leftarrow \emptyset$ ▷ clear all queues
        $S \leftarrow \emptyset$ ▷ clear stack
        $p \leftarrow 0$ ▷ reset chunk count

4.5 Hybrid Refinement: Integrated Iterated Local Search

The solution quality of the resulting matching can often be enhanced by performing a post-processing local refinement on the initial output. By using the same framework as in [16], we leverage the edges S selected during the local-ratio phase by iteratively swap unmatched candidates with edges currently in the solution to find weight-increasing configurations. This technique ensures that, in the worst case, the current solution quality and weight are maintained while providing a mechanism for monotonic improvement.

The efficacy of this Iterated Local Search (ILS) is fundamentally dependent on the cardinality of the candidate set. A larger set of unmatched edges provides a richer neighborhood for the swapping procedure to explore. We therefore perform these optimizations on the basic variant of Algorithm 1 (where $d = 0$), which preserves all selected edges in storage rather than pruning them to meet a buffer constraint. By maintaining the full set S, the ILS can identify complex augmenting structures and local improvements that would otherwise be lost in the streaming pruning process. This exhaustive retention of candidate edges allows the refinement phase to more effectively bridge the gap between the fast streaming approximation and the global optimum.

(Parallel) ILS on Buckets The approximation quality of streaming algorithms is often inversely correlated with instance density, as a higher number of edges increases the likelihood that vertex capacities are filled by sub-optimal candidates. We find the bucketing technique specifically addresses this by periodically resetting the internal state (S, Q), allowing the algorithm to consider edges that would otherwise be discarded.

To further improve these "per-chunk" matchings, we perform independent ILS procedures on outsourced threads. Upon the completion of each chunk, the candidate set S and the preliminary matching are offloaded to a worker thread to explore weight-increasing swaps. This asynchronous refinement allows the algorithm to perform a more exhaustive search of the local neighborhood without interrupting the ingestion of the primary edge stream. Because each ILS operates on a discrete temporal partition, the process remains parallel and requires no inter-thread synchronization. This approach effectively utilizes multi-core architectures to enhance solution quality, a task typically difficult to achieve within the sequential constraints of the streaming model.

CHAPTER 5

Experimental Evaluation

In the following chapter, we will first present our experimentation setup in Section 5.1 and then evaluate the proposed algorithm implementations. With respect to metrics such as solution quality, memory consumption, and running time, we will discuss the following research questions:

- **RQ1:** How effective is the theoretical memory-efficient optimization in practice?
- **RQ2:** How do our algorithms compare with the offline greedy approach?
- **RQ3:** How does the ordering of the incoming edges affect the size of the resulting b-matching?

In Section 5.2, we will answer **RQ1**; Section 5.3 covers **RQ2**. Then, Section 5.4 will go into detail regarding edge ordering from **RQ3**.

5.1 Setup and Data Set

In the framework of Reinstädtler et al. [36], we implement our approaches in C++ using g++-14.2 with full optimization enabled (`-O3` flag). The experiments were executed on two identical machines, each equipped with 128 GB of main memory and a Xeon w5-3435X processor clocked at 3.10 GHz with a 45 MB L3 cache. We only compare results from runs executed on the same machine under the same compute job. To measure memory consumption, we use the Linux internals. The time required to load the hypergraph is excluded from our timing measurements. We schedule the experiments to run in parallel and randomize the order in which they were started. Each experiment is repeated 6 times, and the results are taken as the average. To compare methods, we employ performance profiles as recommended by Dolan and Moré [7]. The plots show the fraction of instances that are solved within a factor $\tau < 1$ of the best result for each instance. In the plot (e.g. Figure 5.2), an algorithm appearing in the top-left corner is considered the best performer.

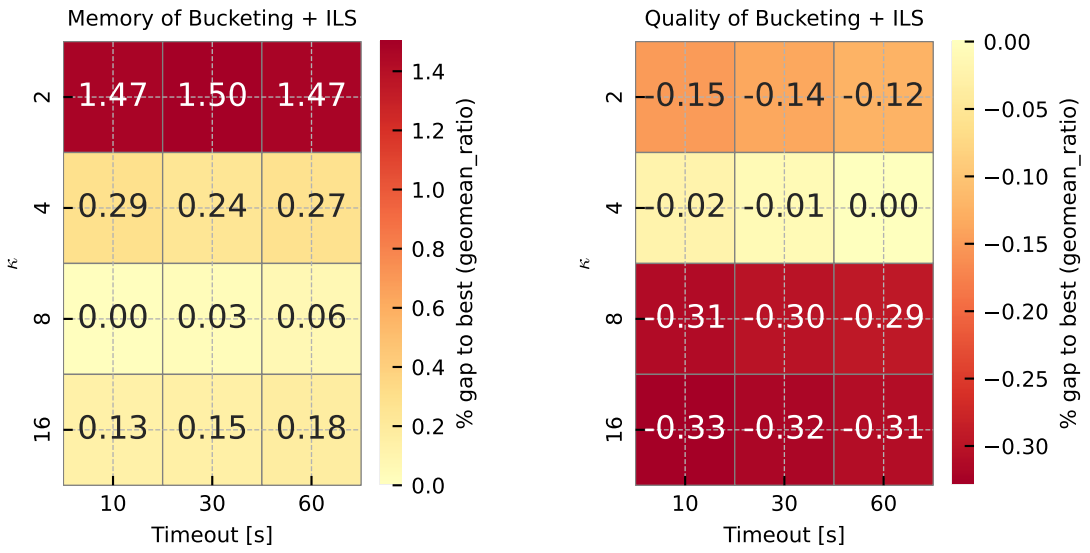

Figure 5.1: Sensitivity analysis using geometric mean ratios.

Our benchmark contains general hypergraphs typically used for partitioning, together with social link hypergraphs derived from question–answering websites. In the social link hypergraphs, a hypergraph b-matching can serve as a multi-faceted summary of the site since it captures the top b different categories or roles for a user [36]. In the hypergraph partitioning setting, matchings are used to contract the hypergraph within a multilevel scheme. In a social hypergraph, each page (for example, a post on StackOverflow or an article in Wikipedia) is represented as a hyperedge.

To address **RQ1** and **RQ2**, we use the hypergraph dataset L_{HG} collected previously for hypergraph partitioning [15]. This collection comprises 94 instances that span a broad range of applications, from DAC routability-driven placement [38] and general sparse matrices [5] to SAT solving [2]. As weights, we employ random values between 1 and 100 and experiment with a uniform node-capacity of 3. Note that the solution quality scales linearly with the node capacity. Therefore, a uniform capacity of 3, with large graph-sizes (edge/node count) in our dataset, represents real-world dimensions at best, considering our limitation of 128 GB of main memory. Capacities of 4 and upwards are not feasible on our machines. Uniform capacities of 1 represent the simple matching problem and are therefore much better solved by other algorithms. Refer to [36] for example. The instances include up to 1.4×10^8 hyperedges/vertices and exhibit a maximum hyperedge size of 2.3×10^6 vertices.

We justify the parameters of our bucketing-based algorithms with the sensitivity analysis in Figure 5.1, which reports memory consumption (left) and matching quality (right). The experiments are run on the L_{HG} data set, repeated three times, use random weights between 1 and 100, and are compared via geometric mean ratios. Each cell is normalized against the best-performing configuration, shown as 0.0.

For memory usage, the ILS timeout has no noticeable effect. The bucket size κ has a much stronger impact. With $\kappa = 2$, memory consumption is worse than with $\kappa = 8$, since larger chunks are kept in storage for longer. For larger values such as $\kappa = 16$, memory usage increases again because smaller chunks weaken the local-ratio filtering. Overall, $\kappa = 8$ gives the best balance.

The quality results show the same pattern. The ILS timeout again has little influence. The best matching quality is obtained for $\kappa = 4$. However, this setting requires more memory. While $\kappa = 2$ achieves slightly better quality than $\kappa = 8$, the latter uses considerably less memory. In fact, $\kappa = 8$ is the only setting in which we outperform the basic algorithm in both quality and memory. We therefore use $\kappa = 8$ in our experiments.

We evaluated **RQ3** using different social link hypergraphs with a uniform capacity of 3. The (THREADS) graphs model participating users as vertices, whereas the (TAGS) graphs model the tags of the posts as vertices. We use three StackExchange networks collected by Benson et al. [3] and set view counts as weights using data from [37]. For the StackOverflow instance, the weights were obtained by querying the public dataset on BigQuery [14], and [36] produced an additional instance from that source. We also use a hypergraph from the English Wikipedia dump, constructed in [36], where categories are treated as vertices and articles as hyperedges. They selected a category as a vertex only if it occurred at least 25 times, yielding 293K vertices for 8M articles. The access frequency of each article in December 2024 is used as the weight.

Statistical Analysis. In **RQ1** and **RQ2**, we want to further evaluate how graph properties, such as maximum edge-size or density of the graph, correlate with the gathered results in the L_{HG} hypergraphs. As the set consists of 94 instances, we can make meaningful statements using standard Pearson and Spearman correlation coefficients. The main goal is to get a single-valued answer to the question of whether the rise of variable x means y rises or falls as well. Therefore, both yield values between -1 and 1, indicating a negative or positive correlation, respectively. For greater detail on the comparison between those coefficients, we refer the reader to [18]. We proceed by comparing a candidate to an interesting competitor, calculating the relative improvement made on each instance and trying to find a correlation between the magnitude of the relative improvements and other graph properties. Interesting findings are marked in yellow and further interpreted in the accompanying text. Instances where the competitor performs better are considered negative improvements. This way, all 94 instances of the L_{HG} data set have an impact on the coefficients, making the results statistically robust.

Notations. The combined basic implementation of Algorithm 1 and 2 by Huang and Sellier [20] in the introduction of Chapter 4 is denoted as Basic. Our memory-efficient adaptation in Section 4.2 is then described as MemEff. The second-pass approach in Section 4.3 is referred to as TwoPass, while our bucketing-technique from Section 4.4 is implemented in Bucketing. This is further refined by ParallelILS from Section 4.5. Note

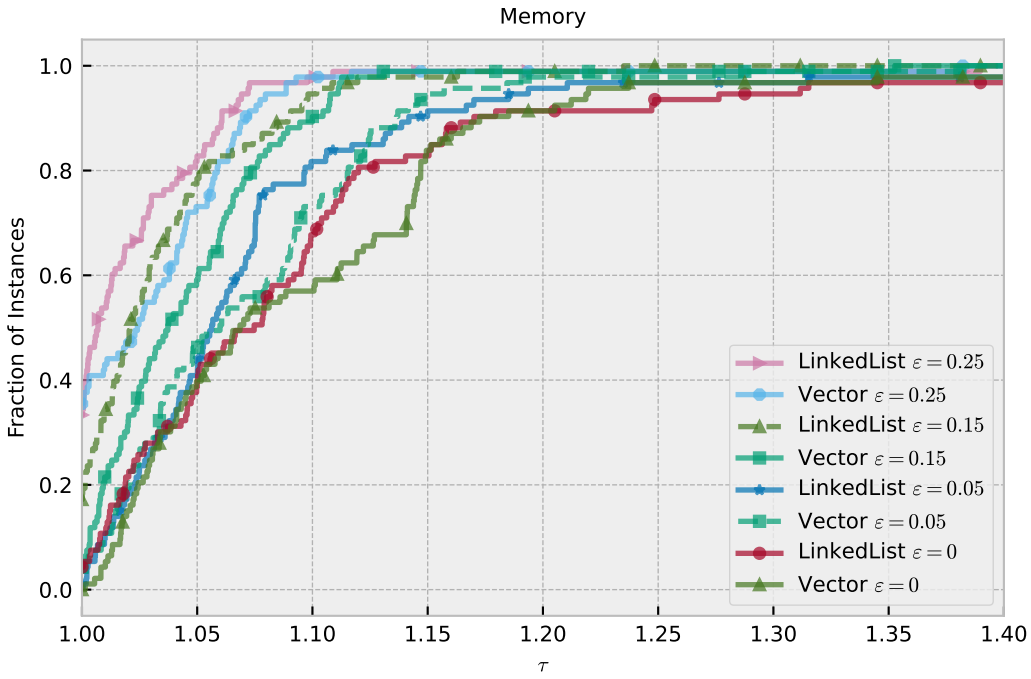

Figure 5.2: Performance Profile for the memory consumption.

that every time we write + ILS after an implementation, we use the engine by Großmann et al. [16] to perform a local-search on the matching yielded by the referred implementation.

5.2 Memory Efficiency in Practice

This section answers **RQ1** by comparing the algorithms Basic and MemEff. As discussed in Section 4.2, the implementation of the theoretical memory-efficient optimization comes with the additional overhead of doubly-linked lists and other variables to track "erasability". In the following, we will evaluate the implementation of our concrete memory efficient adaptation presented in Algorithm 3, denoted as LinkedList, and compare the results for different ε against the basic Algorithm 1 ($d = 0$) implemented with static vectors. Therefore, it is denoted as Vector. We will now further evaluate how this implementation detail affects memory consumption in practice. Note that LinkedList and Vector always calculate the same solution quality for the same $\varepsilon \leq \frac{1}{4}$, as shown in [20].

Figure 5.2 disproves our concern about the overhead of doubly-linked lists. The memory profile for the memory-efficient implementation (LinkedList) is always better on $0.05 \leq \varepsilon \leq 0.25$ than that of the non-discarding vector-based implementation. This leads to the conclusion that the memory freed by the removal of edges in storage dominates the memory additionally allocated for linked-lists. Naturally, this gap closes as we approach

$\varepsilon = 0$, as the max-queue-length β increases for smaller ε (refer to Section 4.2). In the following paragraph, we will further verify the intended behavior of the memory-efficient optimization by determining which graph-properties lead to the highest memory-advantage for $\varepsilon = \frac{1}{4}$.

Vector vs. LinkedList. The linked-list based implementation achieves lower memory consumption on instances characterized by large average hyperedge size and high edge-to-node ratios, as reflected in Table 5.1 by consistently strong positive correlations across both Pearson ($r = 0.57$) and Spearman ($\rho = 0.67$) for average edge size, and Spearman $\rho = 0.55$ for edges per node. Conversely, instances with many nodes and small hyperedges favor the vector-based implementation, with Pearson and Spearman reporting $r = -0.43$ and $\rho = -0.40$ for log-transformed node count. These results are broadly consistent with the main idea behind the memory-efficient optimization. The design goal is to reduce memory by physically removing edges whose weight contribution has become negligible, with the expectation that this is most beneficial when many such edges accumulate. This is precisely the case in dense instances with high edge-to-node ratios and large hyperedges, where more edges compete for queue positions and evictions are frequent. The fact that average edge size is the single strongest predictor of improvement, and that the effect is consistent across both Pearson and Spearman, suggests the correlation is not driven by outliers but reflects a structural property of the algorithm. The negative correlation with node count is a natural counterpart: sparse graphs with many nodes produce fewer evictions, and the per-node overhead of the linked-list representation is not amortized, causing regressions. The alignment between predicted and observed behavior across instance families thus supports the soundness of the optimization strategy, while also identifying sparse, node-heavy instances as a regime where the overhead of the linked-list representation outweighs its benefits.

Metric	Pearson	Spearman
Avg. edge size	0.57	0.67
Log node count	-0.43	-0.40
Log edge count	-0.03	-0.09
Node count	-0.32	-0.40
Edge count	0.02	-0.09
Max. edge size	-0.00	0.06
Edges per node	0.18	0.55
Avg. degree	-0.25	-0.35

Table 5.1: Correlation coefficients for a selection of graph properties. Vector vs. linked-list implementation on $\varepsilon = 0$. Positive correlation indicates higher memory consumption.

5.3 Comparison to Offline Greedy

In this section, we compare the offline greedy algorithm proposed by Großmann et al. [16] to our implementation of the semi-streaming algorithm by Huang and Sellier [20] and also put our optimization strategies in scope. In the following Figures 5.3, 5.4 and 5.5,

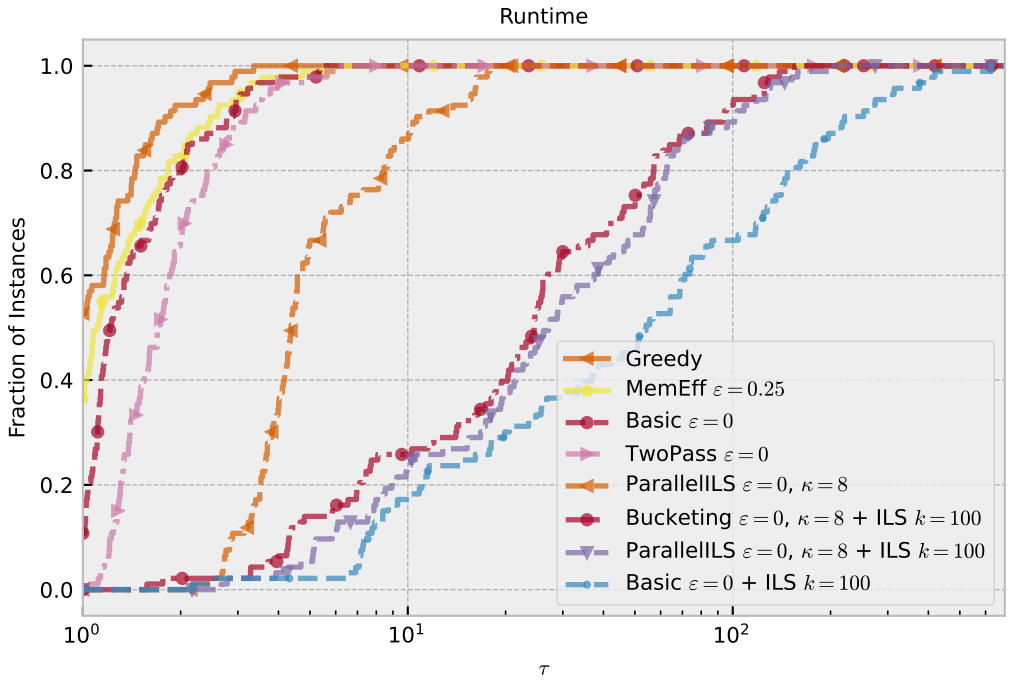

Figure 5.3: Performance Profile for the running time. Note the log-scale on the x-axis.

the offline greedy algorithm is denoted as Greedy, while Basic and MemEff label the semi-streaming variant (refer to the introduction of Chapter 4 for a quick walkthrough). TwoPass is described in Section 4.3; Section 4.4 introduces the Bucketing technique, which is further refined by ParallelILS in Section 4.5.

Running time. Figure 5.3 shows how the implementations perform in terms of running time. We can observe two main streams: First, the Greedy offline algorithm as the top performer, closely followed by our memory efficient implementation MemEff with $\varepsilon = 0.25$ and Basic. The next notable finding is the very similar performance of the algorithms that undertake an ILS in the end. Their worse and very similar running time can be explained by a timeout of 30 seconds for the ILS. ParallelILS then performs operations between these streams as it runs parallelized individual ILS engines on submatchings, performing local optimizations while streaming.

Memory. The memory performance profile in Figure 5.4 also meets our expectations. With MemEff performing as the best candidate, we can report that our Bucketing technique, even with an ILS engine on top, performs better than the standard Basic algorithm for $\varepsilon = 0$. Another important observation we can make is that each semi-streaming approach performs better in terms of memory, as they reach 1.0 faster than the Greedy al-

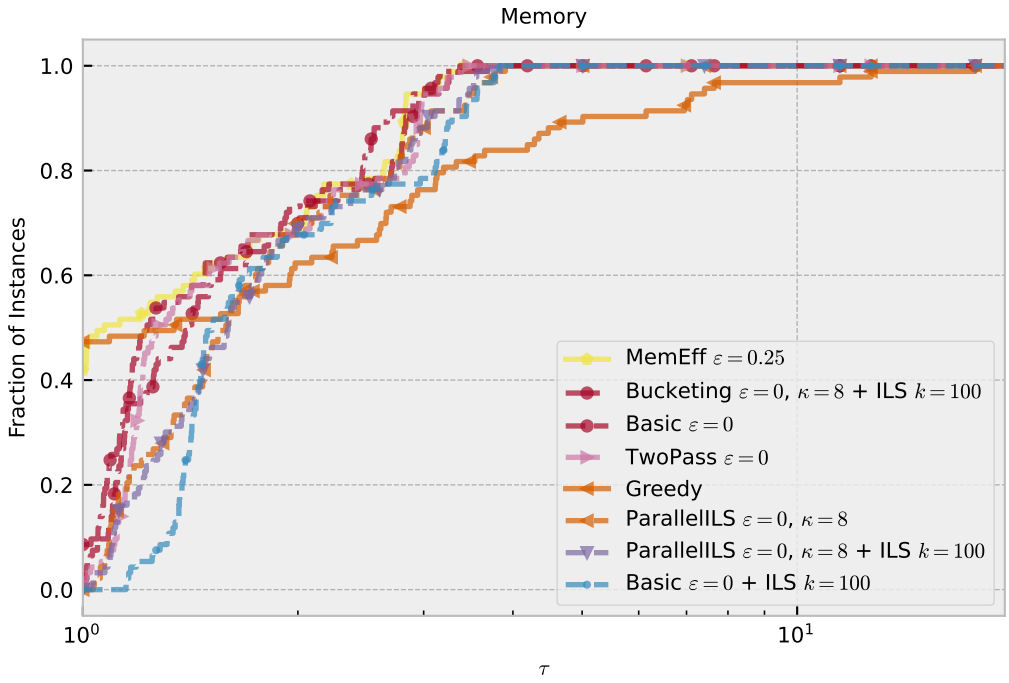

Figure 5.4: Performance Profile for the memory consumption. Note the log-scale on the x-axis.

gorithm. This confirms the correctness of the algorithms in the semi-streaming model.

5.3.1 Quality

The weight of the b-matching often employs the most important metric, as it provides information on the final quality of the respective algorithms. Naturally, as Greedy is allowed to consider all edges in the hypergraph for the resulting matching, it performs best among all single-pass semi-streaming approaches. However, streaming the entire edge-stream again and filling gaps in our previous b-matching using the TwoPass approach turns out to be effective. In 73% of the instances, we are able to report improvements of up to 94.74% compared to offline Greedy. Note here that this approach works very well on our 94 instances of the L_{HG} data set. What the plot also indicates is that, even though it is better in most instances, it lacks in providing a usable worst-case performance, as Greedy reaches 1.0 earlier. This is due to the naive nature of the approach, further explained in Section 4.3. Similar to the previous section, we will now use correlation coefficients to further evaluate these interesting findings in the solution quality yielded by meaningful competitors.

Bucketing vs. Basic. First, we will compare our Bucketing $\varepsilon = 0, \kappa = 8 +$ ILS $k = 100$ approach to the Basic $\varepsilon = 0$ approach. This comparison is interesting as the perfor-

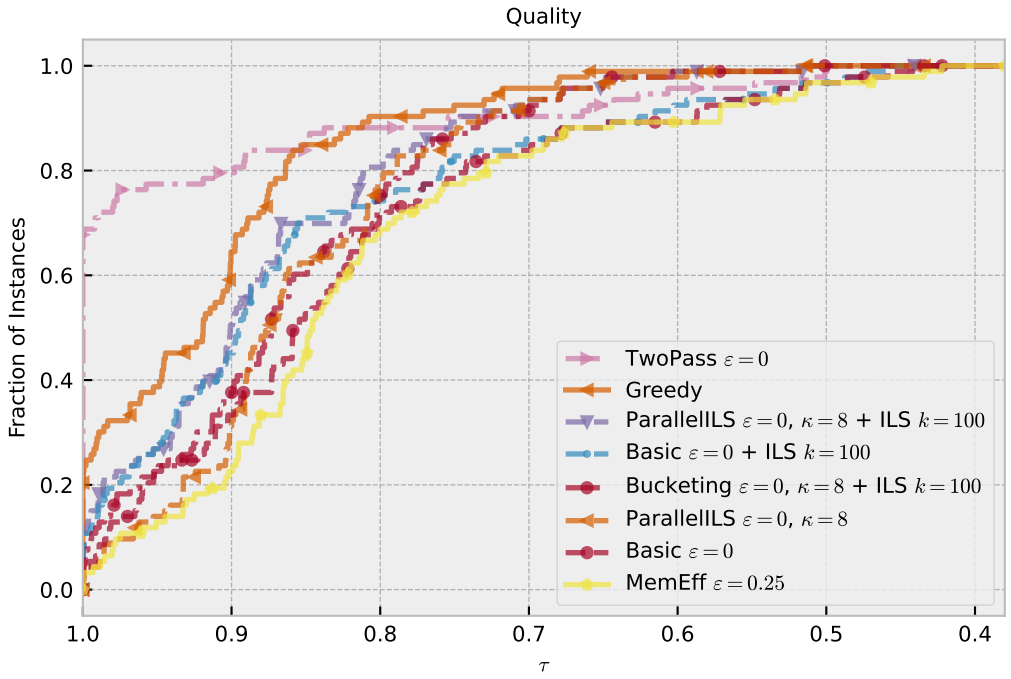

Figure 5.5: Performance Profile for the quality of the b-matching.

mance profiles gathered suggest that our bucketing technique performs better in terms of memory (see Figure 5.4) while producing higher quality results overall (see Figure 5.5). This holds true, as we can report improvements in 64% of the instances, reaching a top weight-improvement of 122.25%. For comparison, the top improvement of our competitor reaches 11.2% on its best instance. Table 5.2 contains the correlation coefficients, showing how the improvements made by our bucketing technique correlate with different graph properties. We will now discuss two main takeaways that the values show. First, Pearson suggests a rather strong correlation of $r = 0.43$ towards a higher average edge size. However, Spearman provides not only a lower correlation but also slightly disagrees. This means the relationship is not monotonic and may be driven by a few influential points or a roughly linear trend that does

Metric	Pearson	Spearman
Avg. edge size	0.44	-0.06
Log node count	-0.38	-0.29
Node count	-0.21	-0.29
Log edge count	-0.26	-0.08
Edge count	-0.15	-0.08
Max. edge size	-0.05	0.16
Edges per node	-0.02	0.23
Avg. degree	-0.01	0.08

Table 5.2: Correlation coefficients for a selection of graph properties. Bucketing vs. basic implementation. Positive correlation indicates higher weight gains.

not hold consistency. Secondly, considering (log) node counts: as size increases, the percentage gain from our bucketing algorithm tends to fall. This time, Pearson and Spearman agree, making the result fairly robust. In conclusion, we can say that among the instances where our bucketing technique wins, its advantage tends to be larger on smaller graphs and may significantly increase with average edge size; however, the average edge size result is not stable.

TwoPass vs. Greedy. As evaluated in Figure 5.5, streaming edges a second time can significantly improve the b-matching yielded after a single stream. As TwoPass $\varepsilon = 0$ lacks a provable worst-case guaranty, we will try to find further correlations in the graph structures that explain the weight-gains compared to the offline Greedy implementation. A closer look at the instance-level correlations in Table 5.3 suggests that the benefit of a second streaming pass is not uniform but instead depends on the structural properties of the hypergraph. The relative gain of TwoPass is moderately positively correlated with graph size, with the strongest signals appearing for (log) node count and (log) edge count, indicating that larger instances tend to leave more room for the second pass to recover additional matching weight beyond Greedy. In contrast, the average edge size is negatively correlated with the improvement. This time, it is not only reported by Pearson at $r = -0.24$ but is also backed by Spearman at $\rho = -0.42$, validating its monotonic correlation. This suggests that TwoPass is most effective on instances with comparatively smaller hyperedges, where the second scan can better exploit the residual structure left unresolved by the first pass.

Metric	Pearson	Spearman
Avg. edge size	-0.24	-0.42
Log node count	0.38	0.37
Node count	0.27	0.37
Log edge count	0.30	0.29
Edge count	0.26	0.29
Max. edge size	0.09	0.34
Edges per node	0.02	-0.12
Avg. degree	-0.12	-0.04

Table 5.3: Correlation coefficients for a selection of graph properties. Two pass vs. Offline greedy implementation. Positive correlation indicates higher weight gains.

5.4 Impact of Edge-Ordering

As the following experiments will show, the order in which the edges are presented to the streaming algorithm has a significant influence on the used memory, running time, and the resulting solution quality. We therefore, again, compared a representative selection of our algorithms to the offline Greedy approach using different $\varepsilon \leq \frac{1}{4}$.

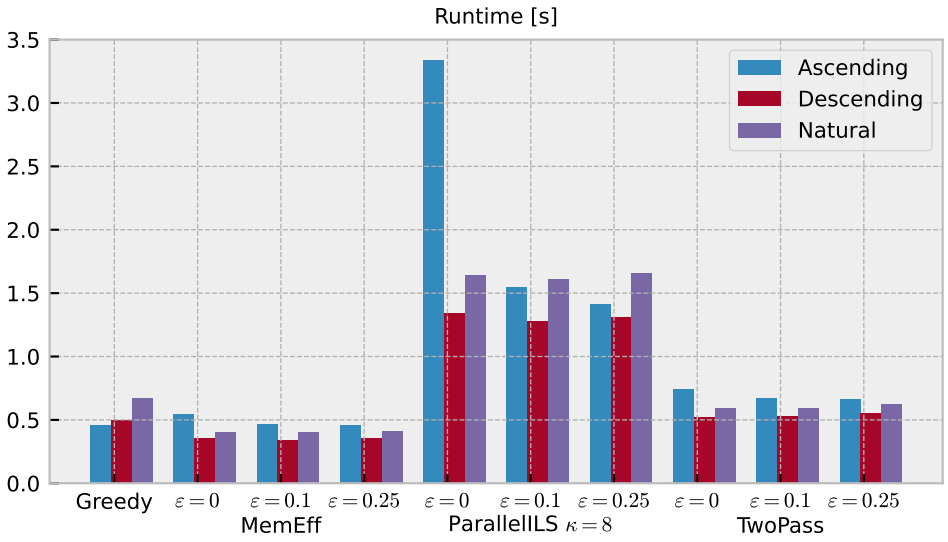

Figure 5.6: Geometric mean of the running time on the social-link hypergraphs.

Running time. As pointed out in Section 5.3, fixed timeouts have a vast influence on the running time of algorithms working with local ILS optimizations. ParallelILS is the candidate in this case. We can observe two main, expected behaviors in Figure 5.6. First, presenting edges in ascending order means presenting the probably worst candidates first to the algorithm. Performing an ILS on such closely uniform chunks is therefore much harder and takes longer to achieve better results. This is also true for edges presented in descending order; however, in this case, the local matching may already be near optimal. Secondly, especially for ascending order, the running time vastly improves for larger ε. This behavior results from the fact that higher ε constitute higher thresholds for edges in terms of being considered. Many edges may be skipped initially as they do not provide enough additional gain to the current matching. This, in turn, reduces the set of edges for the ILS to work on and makes it increasingly faster to compute.

Compared to Greedy, only MemEff performs mostly better. However, the trend for the streaming approaches follows the same pattern as described above. Ascending order performs worse but improves with larger approximation bounds, while edges presented in descending order speed up the algorithm, as many edges are discarded already in the local-ratio phase. The natural ordering always settles in the middle.

Memory. The same reason that led us to higher running times also leads to higher memory consumption. The local-ratio technique assumes that edges provide random gains in no particular order. This way, in natural ordering, many edges are filtered in the streaming phase. However, when edges are presented in ascending order, most edges provide a higher gain than their prior candidates, i.e., prior neighbors in the hypergraph. This leads

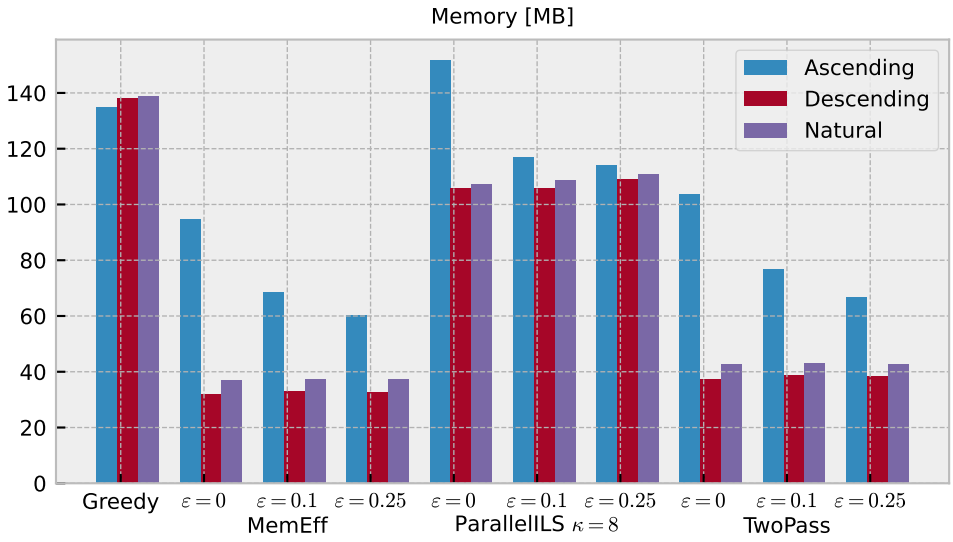

Figure 5.7: Geometric mean of the memory consumption on the social-link hypergraphs.

to most edges being considered for the resulting matching, undermining the theoretical semi-streaming memory bound in practice.

In Figure 5.7, considering the evaluations for the ascending order of the streaming algorithms at $\varepsilon = 0$, one can easily examine the trend of using close to as much memory as the offline Greedy approach. Natural and Descending ordering on MemEff and TwoPass provide the memory consumption expected in the semi-streaming model, using only a fraction of the memory allocated in the Greedy approach. The reason we report a higher overall memory consumption for ParallelILS arises from two main factors: first, the parallelized technique used to outsource individual ILS engines. While the edge stream continues, previous memory remains allocated for use on a different thread for local optimizations. The memory consumption would therefore drop if everything were performed sequentially, resulting in a trade-off of higher running times. Secondly, the theoretical memory performance of our bucketing technique is slightly worse than the original algorithm, dominated by the per-chunk matchings as shown in Section 4.4. However, the worse result we get on this small set of graphs does not seem to be representative in practice, as the results of **RQ2** in Section 5.3 indicate an overall better performance compared to the basic variant.

Quality. Notable are the larger resulting b-matchings in Figure 5.8 for the edge-streams presented in ascending order in the semi-streaming approaches. However, as we previously examined, especially in the analysis regarding memory consumption, these better results come with high memory consumption in practice. By using memory that is quasi-linear to the number of edges rather than nodes, the results become infeasible in the semi-streaming model. Comparing MemEff and ParallelILS leads to a good observation regarding the

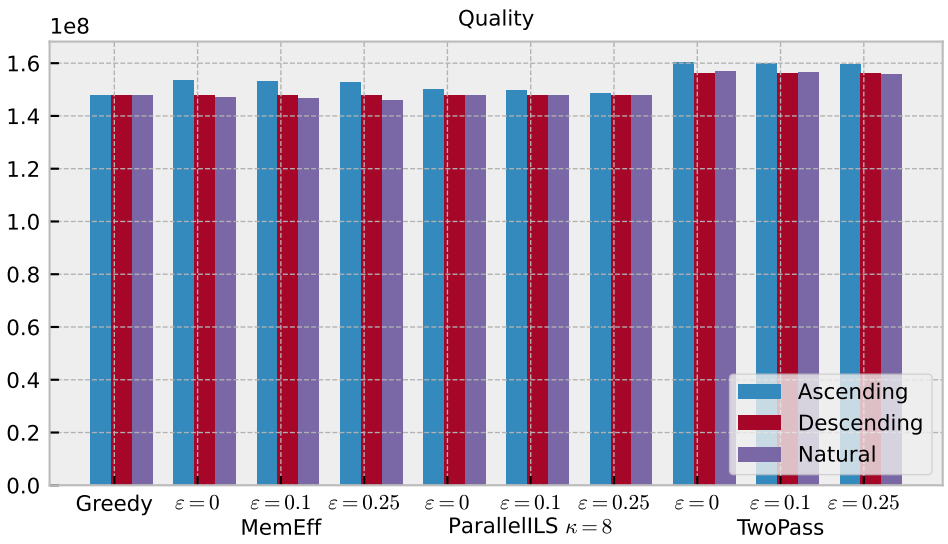

Figure 5.8: Geometric mean of the size of the b-matchings on the social-link hypergraphs.

natural ordering. Our bucketing technique used for the ParallelILS implementation yields empirically better and constant results for higher ε. This results from the consideration of good candidates which are otherwise discarded by large threshold rising during the edge-stream. Similar to the previous Section 5.3, TwoPass yields the overall best quality by reconsidering all unmatched edges, strongly improving the matching achieved by MemEff in practice.

CHAPTER 6

Discussion

This chapter concludes our findings on the topic of engineering efficient hypergraph b-matching algorithms in the semi-streaming model. We revisit the whole paper in Section 6.1 and give a short outlook on future avenues of work with our results in Section 6.2.

6.1 Conclusion

This thesis translated the semi-streaming hypergraph b-matching framework of Huang and Sellier [20], introduced and formalized in Chapter 4, into a concrete implementation and experimental evaluation. The main question was whether the theoretical Algorithms 1 and 2 can be implemented faithfully and improved in practice without leaving the semi-streaming setting. The experiments in Chapter 5 show that this is indeed possible: the theoretical core is robust, but practical performance remains sensitive to input order, memory management, and the use of additional passes or post-processing.

A first observation is that the memory-efficient implementation described in Section 4.2 is not only a theoretical refinement but also a practical one. The linked-list-based variant MemEff reduces memory consumption compared to the static-vector baseline, confirming that the additional bookkeeping is compensated by earlier pruning. This shows that implementation details in the streaming phase substantially affect performance.

The results also show that the streaming order is a dominant factor. The same algorithm can behave very differently under ascending, natural, and descending edge orders. Under ascending order, more edges survive the local-ratio threshold, increasing memory usage and running time. Under descending order, many edges are discarded early, which improves efficiency but may limit later improvements. This reflects a fundamental feature of semi-streaming b-matching algorithms: their guaranties are order-agnostic, but their empirical behavior is not.

The comparison with the offline greedy approach clarifies the role of the proposed extensions. The baseline semi-streaming implementations Basic and MemEff are competi-

tive in memory and running time, but their solution quality is constrained by the one-pass streaming paradigm. The second-pass heuristic in Section 4.3 provides a simple way to recover additional weight by reconsidering feasible edges after the first pass. This can yield substantial gains, although the benefit depends on the instance and should be seen as an empirical improvement rather than a worst-case guaranty.

The bucketing strategy in Section 4.4 addresses the same order sensitivity from another angle. By processing the stream in chunks and resetting the intermediate state, it reduces dependence on the earliest edges and can improve the final matching weight. Its main drawback is that the global behavior becomes less direct to analyze since the final result is assembled from several partial matchings.

The hybrid refinement with Iterated Local Search, introduced in Section 4.5 and extended in Section 4.5, highlights the balance between approximation structure and practical optimization. The semi-streaming phase provides a feasible initial solution, while ILS can exploit local exchanges not captured by the streaming heuristic alone. The parallel variant further shows that the implementation can use modern hardware effectively, although this introduces additional time and memory trade-offs. In this sense, the ILS extensions are best understood as performance-oriented refinements rather than part of the core semi-streaming guaranty.

Another important result is that streaming-based methods can be competitive with the offline greedy baseline from Chapter 3. The offline greedy method remains strong in running time and is often difficult to beat; yet the thesis shows that the two-pass and bucketed variants can match or even outperform it on the measured objective. This demonstrates that the semi-streaming model is not merely a memory-saving compromise, but can also be a practical alternative when combined with careful engineering.

The correlation analysis in Sections 5.2 and 5.3 suggests that larger instances often leave more room for the second-pass strategy, while smaller average hyperedges tend to benefit more from augmentation relative to the offline greedy baseline. The relationship between structural properties and improvement is not always monotonic, indicating that performance depends on several interacting factors, including density, edge-size distribution, and stream order.

Overall, this thesis shows that semi-streaming hypergraph b-matching is a strong example of the interaction between theoretical algorithm design and systems engineering. The formal results in Chapter 2 provide the basis for feasibility and approximation, Chapter 4 turns these ideas into implementable strategies, and Chapter 5 confirms their practical value. The main lesson is that the best-performing approach depends on the desired balance between memory consumption, running time, and matching quality. Taken together, these findings indicate that the practical success of the approach stems less from any single optimization than from the cumulative effect of several modest design choices.

6.2 Future Work

Future work could extend these ideas in several directions. Natural next steps include experiments on non-uniform capacities, adaptive chunk sizes for bucketing, adaptive choices of ε, and more sophisticated refinement strategies that preserve the semi-streaming memory model. It would also be interesting to investigate whether the observed correlations from Sections 5.2 and 5.3 can be turned into predictive rules for choosing between Basic, MemEff, TwoPass, and ParallelILS on a per-instance basis. Such work would further narrow the gap between theoretical semi-streaming guaranties and robust large-scale hypergraph optimization in practice.

Zusammenfassung

Das Semi-Streaming-b-Matching-Problem auf einem gewichteten Hypergraphen $H = (V, E, \omega)$ fordert eine Teilmenge von Hyperkanten $M \subseteq E$, die das Gesamtgewicht maximiert und gleichzeitig sicherstellt, dass jeder Knoten $v \in E$ in höchstens $b(v)$ ausgewählten Kanten enthalten ist. Dies verallgemeinert das Standard-Matching-Problem, bei dem $b(v) = 1$ ist, und ist durch groß angelegte Anwendungen motiviert, bei denen die Eingabe zu groß ist, um explizit gespeichert zu werden. Im Semi-Streaming-Modell darf der Algorithmus nur $\mathcal{O}\big(n \cdot \mathrm{polylog}(n)\big)$ Speicherplatz beanspruchen und ist auf eine geringe Anzahl von Durchläufen über den Eingabestrom beschränkt.

Diese Arbeit untersucht, wie solche Algorithmen in der Praxis für gewichtete b-Matching-Probleme in Hypergraphen effizient implementiert werden können. Aufbauend auf dem Semi-Streaming-Framework von Huang und Sellier implementieren wir die Streaming-Phase und die Greedy-Rekonstruktionsphase und passen sie an einen allgemeinen Hypergraphenkontext mit Knotenkapazitäten an. Darüber hinaus untersuchen wir mehrere technisch orientierte Verbesserungen: eine speichereffiziente Pruning-Strategie auf Basis verknüpfter Datenstrukturen und Referenzzählung, eine Second-Pass-Erweiterung, die den Eingabestrom erneut berücksichtigt, um zusätzliche Matching-Gewichte zu gewinnen, einen Bucketing-Ansatz, der den Strom in Blöcken verarbeitet, sowie eine hybride Verfeinerungsphase auf Basis der iterierten lokalen Suche. Wir untersuchen zudem eine parallele Variante des Verfeinerungsschritts, um moderne Multi-Core-Hardware besser auszunutzen.

Die implementierten Methoden werden anhand großer Benchmark-Hypergraphen hinsichtlich Laufzeit, Speicherverbrauch und Abgleichqualität bewertet. Die Experimente zeigen, dass die speichereffiziente Implementierung den Speicherverbrauch in der Praxis reduziert, dass ein zweiter Durchlauf die Lösungsqualität erheblich verbessern kann, und dass eine Kombination aus Bucketing und lokaler Suche weitere Verbesserungen bewirken kann. Gleichzeitig verdeutlichen die Ergebnisse den starken Einfluss der Kantenreihenfolge auf das Verhalten von Semi-Streaming-Algorithmen. Insgesamt zeigt die Arbeit, dass Semi-Streaming-Hypergraph-b-Matching in praktische Implementierungen umgesetzt werden kann, die ein Gleichgewicht zwischen Speichereffizienz und Lösungsqualität herstellen.

Bibliography

[1] Oren Anava, Shahar Golan, Nadav Golbandi, Zohar Shay Karnin, Ronny Lempel, Oleg Rokhlenko, and Oren Somekh. Budget-constrained item cold-start handling in collaborative filtering recommenders via optimal design. In Aldo Gangemi, Stefano Leonardi, and Alessandro Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 45–54. ACM, 2015.

[2] Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. The SAT competition 2014. `https://satisfiability.org/competition/2014/`, 2014. Accessed: 2026-04-08.

[3] Austin R. Benson, Rediet Abebe, Michael T. Schaub, Ali Jadbabaie, and Jon M. Kleinberg. Simplicial closure and higher-order link prediction. *Proc. Natl. Acad. Sci. USA*, 115(48):E11221–E11230, 2018.

[4] Michael S. Crouch and Daniel M. Stubbs. Improved streaming algorithms for weighted matching, via unweighted matching. In Klaus Jansen, José D. P. Rolim, Nikhil R. Devanur, and Cristopher Moore, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, Barcelona, Spain, September 4-6, 2014*, LIPIcs, pages 96–104. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014.

[5] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[6] John P. Dickerson, Karthik Abinav Sankararaman, Aravind Srinivasan, and Pan Xu. Balancing relevance and diversity in online bipartite matching via submodularity. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 1877–1884. AAAI Press, 2019.

[7] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002.

[8] Leah Epstein, Asaf Levin, Julián Mestre, and Danny Segev. Improved approximation guarantees for weighted matching in the semi-streaming model. *SIAM J. Discret. Math.*, 25(3):1251–1265, 2011.

[9] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2-3):207–216, 2005.

[10] S. M. Ferdous, Alex Pothen, and Mahantesh Halappanavar. Streaming matching and edge cover in practice. In Leo Liberti, editor, *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, 2024, Vienna, Austria*, volume 301 of *LIPIcs*, pages 12:1–12:22, Dagstuhl, Germany, 2024. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[11] S. M. Ferdous, Bhargav Samineni, Alex Pothen, Mahantesh Halappanavar, and Bala Krishnamoorthy. Semi-streaming algorithms for weighted k-disjoint matchings. In Timothy M. Chan, Johannes Fischer, John Iacono, and Grzegorz Herman, editors, *32nd Annual European Symposium on Algorithms, ESA 2024, September 2-4, 2024, Royal Holloway, London, United Kingdom*, volume 308 of *LIPIcs*, pages 53:1–53:19, Dagstuhl, Germany, 2024. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[12] Harold N Gabow. An Efficient Reduction Technique For Degree-constrained Subgraph and Bidirected Network Flow Problems. In *STOC*, pages 448–456, 1983.

[13] Mohsen Ghaffari and David Wajc. Simplified and space-optimal semi-streaming (2+epsilon)-approximate matching. In Jeremy T. Fineman and Michael Mitzenmacher, editors, *2nd Symposium on Simplicity in Algorithms, SOSA 2019, San Diego, CA, USA, January 8-9, 2019*, OASIcs, pages 13:1–13:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[14] Google Cloud Platform. Bigquery public datasets. `https://cloud.google.com/bigquery/public-data`. Accessed: 2026-03-13.

[15] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. *ACM Trans. Algorithms*, 20(1):9:1–9:54, 2024.

[16] Ernestine Großmann, Felix Joos, Henrik Reinstädtler, and Christian Schulz. Engineering hypergraph b-matching algorithms. *J. Graph Algorithms Appl.*, 30(1):1–24, 2026.

[17] Bin Han, Vincenzo Sciancalepore, Di Feng, Xavier Costa-Perez, and Hans D Schotten. A utility-driven multi-queue admission control solution for network slicing. In *IEEE INFOCOM 2019-IEEE conference on computer communications*, pages 55–63. IEEE, 2019.

[18] Jan Hauke and Tomasz Kossowski. Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data. *Quaestiones geographicae*, 30(2):87–93, 2011.

[19] Elad Hazan, Shmuel Safra, and Oded Schwartz. On the Complexity of Approximating K-Set Packing. *computational complexity*, 15(1):20–39, 2006.

[20] Chien-Chung Huang and François Sellier. Semi-streaming algorithms for submodular function maximization under b-matching, matroid, and matchoid constraints. *Algorithmica*, 86(11):3598–3628, 2024.

[21] Johan Håstad. Clique Is Hard to Approximate Within $N^{1-\epsilon}$. *Acta Mathematica*, 182(1):105 – 142, 1999.

[22] Bogumił Kamiński, Valérie Poulin, Paweł Prałat, Przemysław Szufel, and François Théberge. Clustering via hypergraph modularity. *PloS one*, 14(11):e0224307, 2019.

[23] Nitish Korula, Vahab S. Mirrokni, and Morteza Zadimoghaddam. Online submodular welfare maximization: Greedy beats 1/2 in random order. *SIAM J. Comput.*, 47(3):1056–1086, 2018.

[24] Christos Koufogiannakis and Neal E. Young. Distributed Fractional Packing and Maximum Weighted b-Matching via Tail-recursive Duality. In Idit Keidar, editor, *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, volume 5805 of *Lecture Notes in Computer Science*, pages 221–238, 2009.

[25] Piotr Krysta. Greedy Approximation via Duality for Packing, Combinatorial Auctions and Routing. In Joanna Jedrzejowicz and Andrzej Szepietowski, editors, *Mathematical Foundations of Computer Science 2005, 30th International Symposium, MFCS 2005, Gdansk, Poland, August 29 - September 2, 2005, Proceedings*, volume 3618 of *Lecture Notes in Computer Science*, pages 615–627, 2005.

[26] Shrinu Kushagra. Three-dimensional matching is np-hard. *CoRR*, abs/2003.00336, 2020.

[27] Euiwoong Lee, Ola Svensson, and Theophile Thiery. Asymptotically optimal hardness for k-set packing and k-matroid intersection. In Michal Koucký and Nikhil Bansal, editors, *Proceedings of the 57th Annual ACM Symposium on Theory of Computing, STOC 2025, Prague, Czechia, June 23-27, 2025*, pages 54–61. ACM, 2025.

[28] Roie Levin and David Wajc. Streaming submodular matching meets the primal-dual method. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 1914–1933. SIAM, 2021.

[29] Hui Lin and Jeff A. Bilmes. Word alignment via submodular maximization over matroids. In *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA - Short Papers*, pages 170–175. The Association for Computer Linguistics, 2011.

[30] Andrew McGregor. Finding graph matchings in data streams. In Chandra Chekuri, Klaus Jansen, José D. P. Rolim, and Luca Trevisan, editors, *Approximation, Randomization and Combinatorial Optimization, Algorithms and Techniques, 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2005 and 9th InternationalWorkshop on Randomization and Computation, RANDOM 2005, Berkeley, CA, USA, August 22-24, 2005, Proceedings*, Lecture Notes in Computer Science, pages 170–181. Springer, 2005.

[31] Julián Mestre. Greedy in Approximation Algorithms. In Yossi Azar and Thomas Erlebach, editors, *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 528–539, 2006.

[32] Meike Neuwohner. Passing the limits of pure local search for weighted k-set packing. In Nikhil Bansal and Viswanath Nagarajan, editors, *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023, Florence, Italy, January 22-25, 2023*, pages 1090–1137. SIAM, 2023.

[33] Ojas Parekh and David Pritchard. Generalized Hypergraph Matching via Iterated Packing and Local Ratio. In Evripidis Bampis and Ola Svensson, editors, *Approximation and Online Algorithms - 12th International Workshop, WAOA 2014, Wrocław, Poland, September 11-12, 2014, Revised Selected Papers*, volume 8952 of *Lecture Notes in Computer Science*, pages 207–223, 2014.

[34] Ami Paz and Gregory Schwartzman. A (2+ϵ)-approximation for maximum weight matching in the semi-streaming model. *ACM Trans. Algorithms*, 15(2):18:1–18:15, 2019.

[35] Yunke Qu, Tong Chen, Quoc Viet Hung Nguyen, and Hongzhi Yin. Budgeted embedding table for recommender systems, 2024.

[36] Henrik Reinstädtler, S. M. Ferdous, Alex Pothen, Bora Uçar, and Christian Schulz. Semi-streaming algorithms for hypergraph matching. In Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms, ESA 2025, Warsaw, Poland, September 15-17, 2025*, LIPIcs, pages 79:1–79:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025.

[37] StackExchange. Stackexchange data explorer. `https://data.stackexchange.com/`. Accessed: 2026-03-13.

[38] Natarajan Viswanathan, Charles J. Alpert, Cliff C. N. Sze, Zhuo Li, and Yaoguang Wei. The DAC 2012 routability-driven placement contest and benchmark suite. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, DAC '12, pages 774–782, New York, NY, USA, 2012. ACM.

[39] Junliang Yu, Hongzhi Yin, Jundong Li, Qinyong Wang, Nguyen Quoc Viet Hung, and Xiangliang Zhang. Self-supervised multi-channel hypergraph convolutional network for social recommendation. In *Proceedings of the web conference 2021*, pages 413–424, 2021.

[40] Mariano Zelke. Weighted matching in the semi-streaming model. *Algorithmica*, 62(1-2):1–20, 2012.

[41] Zi-Ke Zhang and Chuang Liu. A hypergraph model of social tagging networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2010(10):P10005, October 2010.

[42] Botao Zhu and Xianbin Wang. Hypergraph-aided task-resource matching for maximizing value of task completion in collaborative iot systems. *IEEE Transactions on Mobile Computing*, 23(12):12247–12261, December 2024.