Semi-Streaming Algorithms for Hypergraph Matching

Henrik Reinstädtler
Heidelberg University, Germany

S M Ferdous
Pacific Northwest National Laboratory, Richland, WA, USA

Alex Pothen
Purdue University, West Lafayette, IN, USA

Bora Uçar
CNRS and LIP, ENS de Lyon, France
UMR5668 (CNRS, ENS de Lyon, Inria, UCBL1), France

Christian Schulz
Heidelberg University, Germany

Abstract

We propose two one-pass streaming algorithms for the $\mathcal{NP}$-hard hypergraph matching problem. The first algorithm stores a small subset of potential matching edges in a stack using dual variables to select edges. It has an approximation guarantee of $\frac{1}{d(1+\varepsilon)}$ and requires $\mathcal{O}((\frac{n}{\varepsilon})\log^2 n)$ bits of memory, where n is the number of vertices in the hypergraph, d is the maximum number of vertices in a hyperedge, and $\epsilon > 0$ is a parameter to be chosen. The second algorithm computes, stores, and updates a single matching as the edges stream, with an approximation ratio dependent on a parameter α. Its best approximation guarantee is $\frac{1}{(2d-1)+2\sqrt{d(d-1)}}$, and it requires only $\mathcal{O}(n)$ memory.

We have implemented both algorithms and compared them with respect to solution quality, memory consumption, and running times on two diverse sets of hypergraphs with a non-streaming greedy and a naive streaming algorithm. Our results show that the streaming algorithms achieve much better solution quality than naive algorithms when facing adverse orderings. Furthermore, these algorithms reduce the memory required by a factor of 13 in the geometric mean on our test problems, and also outperform the offline Greedy algorithm in running time.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Mathematics of computing → Hypergraphs

Keywords and phrases hypergraph, matching, semi-streaming

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.79

Related Version *Full Version*: `https://arxiv.org/abs/2502.13636` [38]

Supplementary Material *Dataset*: `https://doi.org/10.5281/zenodo.15778413`
Software: `https://github.com/HeiHGM/Streaming` [37]
archived at `swh:1:dir:4e4522550296fb38202457ce7371bf7034ae45d9`

Funding We acknowledge support by DFG grant SCHU 2567/8-1. Moreover, we like to acknowledge Dagstuhl Seminar 24201 on discrete algorithms on modern and emerging compute infrastructure. Alex Pothen's research was supported by the U.S. Department of Energy grant SC-0022260. S M Ferdous's research was supported by the Laboratory Directed Research and Development Program at PNNL.

33rd Annual European Symposium on Algorithms (ESA 2025).
Editors: Anne Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman; Article No. 79; pp. 79:1–79:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

We propose two streaming algorithms for the hypergraph matching problem, derive their approximation ratios, and implement them to evaluate their practical performance on a large number of test hypergraphs. Recall that hypergraphs are a natural extension of graphs and can help to model our ever evolving earth and society. In a hypergraph, a hyperedge is a subset of vertices and can contain any number of them, instead of just two. The hypergraph matching problem asks for a set of vertex-disjoint hyperedges. Two common objectives in the hypergraph matching problem are to maximize the number or total weight of the matching hyperedges. The hypergraph matching problem has applications ranging from personnel scheduling [17] to resource allocations in combinatorial auctions [21]. The hypergraph matching problem with either of the objective functions is $\mathcal{NP}$-complete [24], but a simple Greedy algorithm is $1/d$-approximate, where d is the maximum size of a hyperedge [5, 28].

There are a few papers with computational studies for the hypergraph matching problem, in a single CPU in-memory setting [11] and in a distributed computing setting [23]. However, little to no attention was paid to increasing data sizes and approximation guarantees. Streaming and semi-streaming algorithms address this trend of ever-increasing data size. In a streaming setting, hyperedges arrive one by one in arbitrary order. The amount of memory that can be used is strictly bounded by the size of the final solution. In the case of hypergraph matching, this is $\Theta(n)$, where n is the number of vertices in the hypergraph, because every vertex can be matched at most once. For a semi-streaming setting, this criterion is relaxed to allow for an additional polylog factor. Furthermore, establishing a bound on the degree of suboptimality is essential for evaluating the solution's effectiveness. For matchings in graphs, there is a semi-streaming algorithm having an approximation guarantee of $\frac{1}{2}$ by Paz and Schwartzman [33].

The algorithm by Paz and Schwartzman requires one variable per vertex, the dual variable, and uses a stack to store candidate matching edges. When an edge appears in the stream, the algorithm adds it to a stack if its weight dominates the sum of the duals of its vertices, and then updates the duals with the difference between the edge weight and the sum of duals. Otherwise, the edge is discarded. After all edges have been streamed, non-conflicting edges are added from the stack beginning at the top. Ghaffari and Wajc [18] give a simplified proof of the approximation guarantee of this algorithm using the primal-dual linear programming framework.

Our contributions are as follows. We first propose a novel streaming framework for hypergraph matching and prove an approximation guarantee in relation to the largest hyperedge size by extending the stack-based algorithm of Paz and Schwartzman [33] to hypergraphs. In essence, our algorithm puts hyperedges that potentially belong to a good matching on a stack, and in the end computes a matching out of those hyperedges. Given the maximum hyperedge size d (the maximum number of vertices in a hyperedge), we use primal-dual techniques to prove a $\frac{1}{d(1+\varepsilon)}$ approximation factor, where $\epsilon > 0$ is a parameter to be chosen. Our most memory-saving algorithm requires $\mathcal{O}(n \log^2 n/\varepsilon)$ bits of space. We then propose a second family of algorithms which do not need a stack and require less space and work by greedily swapping hyperedges from the current matching with the incoming ones. Like the first stack-based algorithm, this algorithm family requires $\mathcal{O}(|E| \cdot d)$ work. They have an approximation guarantee depending on a factor $\alpha > 0$, which can be tuned to result in a guarantee of $\frac{1}{(2d-1)+2\sqrt{(d-1)d}}$. In experiments, we show the competitiveness of our approaches and benchmark them on a set of social link hypergraphs and a large set of instances from hypergraph partitioning. We compare our algorithms with a Naive streaming algorithm that maintains a maximal matching in the hypergraph by adding a hyperedge from the stream to the match if it does not overlap in any vertex with the current matching edges, and a non-streaming Greedy algorithm. The stack-based algorithms reduce the memory consumption by up to 13 times in comparison to the non-streaming Greedy algorithm on social link hypergraphs. We investigate the impact of ordering the hyperedges in the stream and show that our stack algorithm can handle them better than the non-streaming Greedy and Naive streaming algorithms, while requiring only 26% more time than the Naive algorithm. We show the impact of the parameters α and ε on their respective algorithms.

The rest of the paper is organized as follows. After introducing the notation and related work in Section 2, we show our approximation guarantee for an adaptation of the Paz-Schwartzman semi-streaming algorithm and discuss further improvements in Section 3. Section 4 introduces our greedy swapping algorithm for streaming based on McGregors [31] algorithm. These approaches are then extensively evaluated by experiments in Section 5. We conclude in Section 6. The Appendix of the full version of the paper contains the proof of the approximation guarantee for the algorithm from Section 4, and additional statistics on the test problems as well.

2 Preliminaries

2.1 Basic Concepts

Hypergraphs. A *weighted undirected hypergraph* $H = (V, E, W)$ consists of a set V of n vertices and a set E of m hyperedges. Each hyperedge e is a subset of vertices and is assigned a positive weight by the weight function $W: E \to \mathbb{R}_{>0}$. The number of vertices in a hyperedge e is its size, and is denoted by $|e|$, and the maximum size of a hyperedge or rank of the hypergraph is denoted by $d := \max_{e \in E} |e|$. For clarity and brevity, we refer to a hyperedge simply as an edge when it is evident from the context that a hypergraph is under consideration.

Matching. A subset of (hyper-)edges $M \subset E$ is a *matching*, if all (hyper-)edges in M are pairwise disjoint, i.e., only at most one (hyper-)edge is selected at every vertex. A matching M is called maximal, if there is no (hyper-)edge in E which can be added to M without violating the matching constraint. The weight of a matching is defined by $W(M) := \sum_{e \in M} W(e)$ and a maximum matching is a matching with the largest weight.

Related $\mathcal{NP}$-hard Problems. The unweighted hypergraph matching problem is closely related to the maximum independent set and the k-set packing problems. Both problems are $\mathcal{NP}$-hard [27]. An independent set in a graph is a subset of vertices in which no two vertices are adjacent. There is a simple transformation from hypergraph matching to maximum independent set using the line graph of the hypergraph; in the line graph, every hyperedge becomes a vertex, and two such vertices are connected if the corresponding hyperedges share a common vertex. Given a ground set S and some subsets $S_1, \ldots, S_n$, each of size at most k, the k-set packing problem asks to select the maximum number of disjoint subsets. It can be translated to the hypergraph matching setting, by choosing the set S to correspond to the vertices V, while the subsets $S_1, \ldots, S_n$ correspond to the hyperedges.

(Semi-)Streaming Algorithms. If the input size exceeds the memory of a machine, a typical solution is to stream the input. There are several definitions for streaming in graphs and hypergraphs. When (semi-)streaming, the (hyper-)edges of a (hyper-)graph are usually presented in an arbitrary (even adverse) order one-by-one in several passes. In this paper we consider having only one pass over the input. In a streaming setting, the memory is strictly bounded by the solution size and, hence, for matching in hypergraphs, it is limited by the number of vertices. When using the semi-streaming model the memory is bounded by $\mathcal{O}(n \cdot \text{polylog}(n))$.

Approximation Factors. Algorithms can be classified into three categories: exact algorithms, heuristics without approximation guarantees, and approximation algorithms. The quality of an approximation algorithm is measured by comparing its solution's value to that of an optimal solution. For an instance I of a maximization problem, let the optimum objective be denoted by $M(I)$. If an algorithm $\mathcal{A}$ is guaranteed to find a solution that is bigger than $\alpha M(I)$ for every instance, where $\alpha \in \mathbb{R}^+_{<1}$ and is chosen to be as large as possible, then $\mathcal{A}$ provides an α-approximation guarantee. In some communities, the convention is to report $\frac{1}{\alpha} > 1$ as the approximation ratio, although we do not follow it here. For an overview of techniques to design approximation algorithms, we refer the reader to Williamson and Shmoys [42].

(Integer) Linear Programs. Many optimization problems can be formulated as integer linear programs (ILP). In a maximization problem, an (integer) linear program finds an (integer) vector x with components x_i, that maximizes a linear cost function $\sum c_i x_i$ such that a constraint $Ax \leq b$, where A is a matrix, is satisfied, with typically additional constraints on the components of x, e.g., $x_i \geq 0$. If the variables of the problem are integer, some problems are $\mathcal{NP}$-hard, while other problems (where the constraint matrix is unimodular) are solvable in polynomial time [43]. When the integer constraint is dropped, any linear program is solvable in polynomial time [44]. For every linear program one can find a dual problem [42]. Given a maximization problem as described before, the dual problem is to find a vector $y_i \geq 0$, that minimizes $\sum b_i y_i$ subject to $A^T y \geq c$. The weak duality theorem [42] states that for any primal maximization problem, the dual minimization problem for any feasible solution has an objective value larger than the optimal solution of the primal problem. The strong duality theorem states that if the primal problem has an optimal solution then the dual is solvable as well and the optimum values are the same. For a more detailed introduction, we refer the reader to the Appendix of [42].

2.2 Related Work

Matching is a well-studied problem in computer science, and here we give a brief overview of matchings in graphs and hypergraphs.

Matching in Graphs and Streaming. The polynomial-time complexity of matchings in graphs is one of the classical results in theoretical computer science [12]. While Preis [36] presents the first linear time $\frac{1}{2}$-approximation, Drake and Hougardy [10] show a simpler algorithm with the same approximation ratio by path growing (PGA) in linear time. A number of other 1/2-approximation algorithms have been developed, including the proposal-based Suitor algorithm of Manne and Halappanavar [29]. Pettie and Sanders [34] propose a $\frac{2}{3} - \varepsilon$ approximation with expected running time of $\mathcal{O}(m \log \frac{1}{\varepsilon})$. The GPA algorithm by Maue and Sanders [30] bridges the gap between greedy and path-searching algorithms, showing that a combination of both works best in practice. Pothen, Ferdous and Manne [35] survey these approximation algorithms. Birn et al. [6] develop a parallel algorithm in the CREW PRAM model with $\frac{1}{2}$-approximation guarantee and $\mathcal{O}(\log^2 n)$ work. Feigenbaum et al. [14] present a $\frac{1}{6}$-approximation for the weighted matching problem in the semi-streaming setting using a blaming-based analysis. McGregor [31] develop a multipass algorithm that returns a $\frac{1}{2+\varepsilon}$ approximation in $\mathcal{O}(\varepsilon^{-3})$ rounds, with the initial matching having an approximation guarantee of $\frac{1}{3+2\sqrt{2}}$. Paz and Schwartzman [33] give a $\frac{1}{2+\varepsilon}$-approximation algorithm, which employs a dual solution to admit candidate edges into a stack, while also updating the dual solution. The matching is constructed by removing edges from the top of the stack, and those that do not violate the matching property are added to the solution. The resulting matching is not necessarily maximal. Ghaffari and Wajc [18] provide a simpler proof of the approximation ratio using a primal-dual analysis. Ferdous et al. [15] 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. [16] 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 [25, 33] for streaming b-matching.

Hypergraph Matching. Hazan et al. [24] prove that for the maximum k-set packing problem there is no approximation within a factor of $\Omega(k/\log k)$ unless $\mathcal{P} = \mathcal{NP}$. This directly translates to d-uniform cardinality hypergraph matching, where every edge has size d, with $k = d$ and the number of edges selected is maximized. Dufosse et al. [11] investigate reduction rules and a scaling argument for finding large matchings in d-partite, d-uniform hypergraphs. There are several approximation results and local search approaches, most notably by Hurkens and Schrijver [26] and Cygan [7] with an approximation guarantee of $\mathcal{O}((\frac{d+1+\varepsilon}{3}))$. Hanguir and Stein [23] propose three distributed algorithms to compute matchings in hypergraphs, trading off between quality guarantee and number of rounds needed to compute a solution.

The Greedy algorithm for maximum weight hypergraph matching, which considers hyperedges for matching in non-increasing order of weights, is $1/d$-approximate, where d is the maximum size of a hyperedge [5, 28]. For the weighted k-set packing problem Berman [4] introduces a local search technique. Improving on these results, Neuwohner [32] presents a way to guarantee an approximation threshold of $\frac{k}{2}$. We are not aware of any practical implementations of these techniques. For the more general weighted hypergraph b-matching problem, Großmann et al. [22] present effective data-reduction rules and local search methods. In the online setting, when hyperedges arrive in an adversarial order, and one must immediately decide to include the incoming hyperedge or not in the matching, Trobst and Udwani [40] show that no (randomized) algorithm can have a competitive ratio better than $\frac{2+o(1)}{d}$.

We are unaware of any studies or implementations for streaming hypergraph matching.

3 Stack-based Algorithm

We now present our first algorithm to tackle the hypergraph matching problem in the semi-streaming setting. Our algorithm uses dual variables to evaluate whether a streaming hyperedge is good enough to be retained in a stack. Once the stream has been ingested, the stack is evaluated from top to bottom to determine a matching, achieving an approximation guarantee of $\frac{1}{d(1+\varepsilon)}$. We also discuss a more permissive and lenient update function that allows more hyperedges into the stack. Finally, we discuss the space complexity of our algorithms.

Algorithm 1 shows our framework for computing a hypergraph matching in a streaming setting. This algorithm is an extension of an algorithm by Paz and Schwartzman [33] proposed for graphs. The algorithm starts with an empty stack and keeps a dual variable ϕ_v for each vertex v of the hypergraph. Throughout the algorithm the stack contains candidate hyperedges for inclusion in a matching.

For each streamed hyperedge e, the algorithm checks if the weight of the dual variables of e's vertices and thereby the solution can be improved by adding e to the stack. More precisely, with $\Phi_e := \sum_{v \in e} \phi_v$, the algorithm checks if $W(e) \geq \Phi_e(1+\varepsilon)$. The parameter $\varepsilon \in \mathbb{R}_{\geq 0}$ is used to trade quality for memory. A smaller ε yields a better approximation guarantee, while a higher ε yields a smaller memory consumption. With $\varepsilon = 0$, the algorithm no longer provides memory guarantees. If $W(e) \geq \Phi_e(1+\varepsilon)$, then e is added to the stack, and the dual variables of the vertices of e are then updated using an update function. The update functions that we consider take $\mathcal{O}(1)$ time per vertex.

After all hyperedges have been streamed, a matching containing only the hyperedges stored in the stack is computed. To do that, our algorithm takes the hyperedges in reverse order from the stack and adds non-overlapping hyperedges to the matching. Note that hyperedges processed earlier that have conflicting heavier (later) hyperedges will be ignored. This is crucial to prove the performance guarantee later. The amount of total work needed for scanning one hyperedge e is $\mathcal{O}(|e|)$, because we need to sum up the dual variables of e's vertices and update them.

The update functions determine whether an approximation guarantee can be provided and directly impact the results. Upon processing a hyperedge e, the update function applied to the dual variable of a vertex $v \in e$ can use the prior value ϕ_v and sum $\Phi_e := \sum_{v \in e} \phi_v$ as well as e's size and weight. We define the following update function for proving the approximation guarantee

$$\phi_v^{\text{new}} := \delta_{\text{g}}(e, \phi_v, \Phi_e, W(e)) := \phi_v + \overbrace{W(e) - \Phi_e}^{w'_e}. \tag{1}$$

The δ_{g} function is exactly the function used by Paz and Schwartzman [33]. For a hyperedge e added to the stack, the $\phi(v)$ value of its endpoints is increased by the potential gain in matching weight, $w'_(e)$. This follows since $\phi(v)$ stores the gain in weight of all earlier edges incident on v that have been added to the stack thus far.

We introduce a different update function later.

At the core of this method are the variables ϕ_v for each v and the update mechanism. These core components and local ratio techniques are used to prove the approximation guarantee in the original work [33]. We follow the structure of the proof derived from a primal-dual analysis [18].

3.1 Approximation Guarantee

We make use of the primal-dual framework for showing the approximation guarantee. For general linear programming concepts, we refer the reader to books [1, 44], and for an excellent overview of primal-dual approximation method to the book [42].

We show that using δ_{g} update function (1) in our algorithm leads to a $\frac{1}{d(1+\varepsilon)}$-approximation, when the stack is unwound and hyperedges are selected in that order which they are on the stack. When streaming the hyperedges in descending order of weights, we can prove that the Greedy algorithm achieves a $1/d$ approximation factor.

We now proceed with a primal-dual analysis of an ILP formulation of the hypergraph matching problem. In our formulation, there is a binary decision variable associated with each hyperedge to designate if that hyperedge is selected to be in the matching. The objective function is to maximize the sum of the weights of the selected hyperedges. The constraint is to select at most one hyperedge containing a given vertex. The linear programming relaxation, shown in Figure 1a, is obtained by dropping the binary constraint on the hyperedge variables.

Algorithm 1 Simple Streaming Algorithm.

1: **procedure** StackStreaming($H = (V, E, W)$)
2: $S \leftarrow emptystack$
3: $\forall v \in V : \phi_v = 0$
4: **for** $e \in E$ in any (even adverse) order **do**
5: $\Phi_e \leftarrow \sum_{v \in e} \phi_v$
6: **if** $W(e) < \Phi_e(1 + \varepsilon)$ **then**
7: next
8: **end if**
9: $S.push(e)$
10: **for** $v \in e$ **do**
11: $\phi_v \leftarrow \delta(e, \phi_v, \Phi_e, W(e))$ {update}
12: **end for**
13: **end for**
14: $M \leftarrow \emptyset$
15: **while** $S \neq \emptyset$ **do**
16: $e \leftarrow S.pop()$
17: **if** $\forall f \in M : f \cap e = \emptyset$ **then**
18: $M \leftarrow M \cup \{e\}$
19: **end if**
20: **end while**
21: **end procedure**

The objective value of the relaxation is naturally greater or equal than the integer version of the linear program. The dual problem of the relaxed hypergraph matching is given in Figure 1b. Following the weak duality theorem for linear programs, we know that any feasible solution of the dual has an objective value greater or equal to the objective value of any feasible primal solution. Furthermore, the optimal value of the linear program $\text{OPT}(LP)$ is equal for both problems (strong duality). The first step for the proof is to check that the variables $(1 + \varepsilon) \sum_{v \in e} \phi_v$ (from Algorithm 1) constitute a valid dual solution.

▶ **Observation.** *Function δ_g and Algorithm 1 generate valid dual solutions for all $\varepsilon \geq 0$.*

Proof. For each hyperedge e not on the stack, there was enough weight in the ϕ_v values of its vertices, when e was scanned in Line 6. In the update for every added hyperedge to the stack, all vertices ϕ_v values are increased by $w(e) - \Phi_e$ such that clearly the sum of vertex ϕ_v values is higher than the weight of the hyperedge just added. Therefore, for any hyperedge it holds $\sum_{v \in e} \phi_v \geq W(e)$, satisfying the dual equation (2). ◀

Such a valid dual solution has a greater objective value then the optimum solution of the relaxed dual, and the LP duality theorem gives an upper bound for every matching, including the optimal one M^* by $W(M^*) \leq \text{OPT}(LP) \leq (1 + \varepsilon) \sum_v \phi_v$.

Now, we connect the changes to the dual variables with the hyperedges that have already been processed. Define

$$\Delta\phi^e = \begin{cases} \sum_{v \in e} (\delta_\text{g}(e, \phi_v, \Phi_e, W(e)) - \phi_v) = |e| \left(W(e) - \Phi_e\right) & \text{if } e \in S \\ 0 & \text{else} \end{cases} \quad (3)$$

as the change to the dual $\sum_v \phi_v$ by inspecting e. We give a bound for the change of the dual variable w.r.t. the preceding hyperedges in Lemma 1.

Figure 1 Primal and Dual LPs for Hypergraph Matching.

(a) LP

$$\begin{aligned} &\text{maximize} \quad \sum_{e\in E} W(e)x_e \\ &\text{subject to} \\ &\quad \forall v\in V\colon \sum_{e\ni v} x_e \le 1 \\ &\quad \forall e\in E\colon x_e \ge 0. \end{aligned}$$

(b) Dual LP

$$\begin{aligned} &\text{minimize} \quad \sum_{v\in V} \phi_v \\ &\text{subject to} \\ &\quad \forall e\in E\colon \sum_{v\in e} \phi_v \ge W(e) \\ &\quad \forall v\in V\colon \phi_v \ge 0. \end{aligned} \tag{2}$$

▶ **Lemma 1.** *For a hyperedge e, let $W'_e := W(e) - \Phi_e$. For each hyperedge $e \in E$ added to stack S, if we denote its preceding neighboring hyperedges (including itself) by $\mathcal{P}(e) := \{c \mid c \cap e \neq \emptyset, c \text{ added before } e\} \cup \{e\}$, then $W(e) \ge \sum_{e'\in\mathcal{P}(e)} \frac{1}{d}\Delta\phi^{e'} = \sum_{e'\in\mathcal{P}(e)} W'_{e'}$.*

Proof. From the definition (3), we have $\Delta\phi^e = |e|\, W'_e$ for δ_g, because of line 11 of Algorithm 1. $\Phi_e := \sum_{v\in e}\phi_v$ is defined as the previous value of the dual variables before inspecting e. Each of these dual values ϕ_v consists of the sum $\sum_{c\in\mathcal{P}(e) \text{ s.t. } v\in c} W'_c$ for all preceding hyperedges. This leads to $\Phi_e = \sum_{v\in e}\phi_v \ge \sum_{c\in\mathcal{P}(e)\setminus\{e\}} \frac{1}{|e|}\Delta\phi^{e'} \ge \sum_{c\in\mathcal{P}(e)\setminus\{e\}} \frac{1}{d}\Delta\phi^{c}$. So we can conclude $W(e) = W'_e + \Phi_e \ge \frac{1}{d}\Delta\phi^e + \sum_{e'\in\mathcal{P}(e)\setminus\{e\}} \frac{1}{d}\Delta\phi^{e'}$. ◀

The previous bound relates the weight of a hyperedge to the change in dual variables by its predecessors. In conclusion, we show that our algorithm returns a $\frac{1}{d(1+\varepsilon)}$-approximation.

▶ **Lemma 2.** *Algorithm 1 with δ_g function guarantees a $\frac{1}{d(1+\varepsilon)}$-approximation.*

Proof. We show a lower bound on the weight of any matching M constructed by the algorithm. For any hyperedge e not in the stack, $\Delta\phi^e = 0$, as when a hyperedge is not pushed into the stack, no dual variables are updated. Furthermore, any hyperedge in the stack that is not included in the matching must be a previously added neighbor of a matching hyperedge as defined in Lemma 1. Therefore, Lemma 1 applies, and the weight can be lower-bounded. The sum of changes $\sum_e \Delta\phi^e$ to ϕ is equal to the sum of dual variables $\sum_v \phi_v$ at the end. We have

$$W(M) = \sum_{e\in M} W(e) \overset{\text{L. 1}}{\ge} \sum_e \frac{1}{d}\Delta\phi^e \ge \frac{1}{d}\sum_e \Delta\phi^e = \frac{1}{d}\sum_v \phi_v \overset{\text{LP Duality}}{\ge} \frac{1}{d(1+\varepsilon)} W(M^*).$$

◀

3.2 Improving Solution Quality

Now we look into optimizing the solution quality. The design space for optimizations is vast, therefore we focus on simple yet effective techniques. We propose a second update function allowing more hyperedges than δ_g into the stack, with the same approximation guarantee.

Lenient Update Function. The δ_g in Equation 1 builds a dual solution much larger than needed. For every successfully added hyperedge, the difference between the current dual solution and the weight of the hyperedge is added to every vertex of the hyperedge. We address this by combining it with a scaling argument. The resulting function is

$$\delta_{\text{lenient}}(e, \phi_v, \Phi_e, W(e)) := \phi_v + (W(e) - \Phi_e)/\,|e|\,. \tag{4}$$

This function produces a valid dual solution, and Lemma 1 also holds. For every previously added neighboring hyperedge e' the change $\Delta\phi^e = W'_{e'}$ was distributed over all vertices of the hyperedge, so $\Phi_e \geq \sum_{e' \text{ added before}} \frac{1}{d}\Delta\phi^{e'}$. Lemma 2 follows and gives us the desired approximation factor of $\frac{1}{d(1+\varepsilon)}$.

3.3 Space Complexity Analysis

The space complexity of δ_{g} and δ_{lenient} can be deduced by simple counting arguments. Let W be the maximum normalized weight of a hyperedge in the hypergraph, i.e., $W := \frac{\max_{e\in E} W(e)}{\min_{e\in E} W(e)}$, and let W be $\mathcal{O}(\text{poly}(n))$. We discuss both update functions separately.

Guarantee Function. On every vertex we can observe up to $1 + \log_{1+\varepsilon}(W)$ incrementing events, because every change in the dual variables has to be bigger by a factor of $(1+\varepsilon)$. This causes the stack to contain $\mathcal{O}(\sum_v(1 + \log W/\varepsilon))$ vertices in its edges. Each vertex in an edge requires $\log n$ bits. Therefore, the overall space complexity is $\mathcal{O}((1/\varepsilon)n\log^2 n)$ bits since W is $\mathcal{O}(poly(n))$.

Lenient Function. This function updates ϕ_v for every vertex in an edge e added to the stack by $\frac{W(e)-\Phi_v}{d}$, resulting in $1 + d\cdot\log_{1+\varepsilon}(W)$ possible increases to reach the total sum of W. Following the same argument, the stack's space complexity in bits is $\mathcal{O}((1/\varepsilon)nd\log^2 n)$. Note that with this update function the algorithm is semi-streaming only if d is $\mathcal{O}(\text{polylog}(n))$.

The time complexity per edge is in $\mathcal{O}(d)$ and $\Omega(md)$ overall in the scanning phase, as for every edge its vertices need to be scanned and optionally updated once. The unwinding of the stack takes d checks per hyperedge, when reusing the memory of ϕ_v from the previous step. Overall, since the stack size is naturally bounded by m, the complexity of this algorithm is $\mathcal{O}(md)$. In Section 5.2 we show the difference in the stack size in experiments under several orderings of the input.

4 Greedy Swapping Algorithm

We now propose a second streaming algorithm that computes, stores, and updates a matching in the hypergraph as the edges stream. It is conceptually similar to a streaming matching algorithm for graphs [31]. It requires a constant amount of memory per vertex, has an approximation factor that depends on the maximum size of a hyperedge d and a parameter α and obtains high-quality matchings in practice.

The proposed approach is described in Algorithm 2. We store for every vertex v a reference to the current matching hyperedge containing v; the $\perp$ sign symbolizes that no matching hyperedge contains v. For simplifying the presentation, we define $W(\perp) = 0$. When we inspect a hyperedge e, we compute the sum of the weights of its adjacent hyperedges that are currently in the matching. If the weight of the incoming hyperedge is larger than $(1+\alpha)$ times the previous conflicting hyperedges, we first remove the previous hyperedges from all their vertices. Afterwards we can set the reference to the new incoming hyperedge. The overall space consumption of this algorithm is $\mathcal{O}(n)$. There are n references involved, and each hyperedge of size d holds the d vertices referencing it. Removing a hyperedge is linear in d, because up to d vertices in $\mathcal{B}_v$ must be set to $\perp$; it follows that the total work is $\mathcal{O}(d\cdot|E|)$.

We show its approximation guarantee of $\frac{1}{(1+\alpha)(\frac{d-1}{\alpha}+d)}$, which is optimal for $\alpha = \sqrt{(d-1)/d}$, in the Appendix of the full version. In our experimental evaluation, we look at various values of $\alpha \in \{0, 0.1, 1\}$ and $\alpha = \sqrt{(d-1)/d}$. Note that for $\alpha = 0$ the algorithm has no guarantee.

Algorithm 2 Swapping streaming algorithm.

```
procedure SwapSet((H = (V, E, W), α))
  ∀v ∈ V : B_v = ⊥{ Initialize best hyperedge to empty}
  W(⊥) := 0
  for e ∈ E in arbitrary order do
    C ← ⋃_{v∈e} B_v
    Φ_e ← W(C){ weight of hyperedges to be removed.}
    if W(e) ≥ (1 + α) · Φ_e then
      for v ∈ e do
        if B_v ≠ ⊥ then
          for w ∈ B_v do
            B_w ← ⊥ { Unmatch vertices in B.}
          end for
        end if
        B_v ← e
      end for
    end if
  end for
  M ← ⋃_{v∈V} B_v
  return M
end procedure
```

5 Experimental Evaluation

We now evaluate our algorithms with respect to solution quality, running time, and memory usage. Specifically, we address the following research questions:

- **RQ1:** How does the ordering of the hyperedges affect our metrics (memory, running time, and quality)?
- **RQ2:** How do other instance properties affect our algorithms in their memory needs?
- **RQ3:** How do our algorithms compare with offline greedy or naive streaming approaches?

5.1 Setup and Data Set

We implemented our approaches in `C++` using `g++-14.2` with full optimization turned on (`-O3` flag). We tested on two identical machines, equipped with 128 GB of main memory and a Xeon w5-3435X processor running at 3.10 GHz having a L3 cache of 45 MB each. We repeat each experiment three times, and the results are compared only if experiments are run on the same machine and the same compute job. For memory consumption, we used the jemalloc malloc implementation [13]. The time needed for loading the hypergraph is not measured. We scheduled eight experiments (RQ1) and ten experiments (RQ2) to run in parallel, and the order of the experiments was randomized. In order to compare the results we use performance profiles as suggested by Dolan and Moré [9]. We plot the fraction of instances that could be solved within a factor $\tau < 1$ of the best result per instance. In the plot (Figure 7), the algorithm towards the top left corner is the best performer.

Our benchmark includes general hypergraphs that are primarily used for partitioning and social link hypergraphs generated from question-answering websites. In social link hypergraphs, a hypergraph matching can be used to summarize the overall websites, as it represents a subset of disjoint pages from different categories or users. In hypergraph partitioning, matchings can be used to contract the hypergraph in a multilevel scheme. In a social hypergraph, each page (e.g., a post on StackOverflow or an article in Wikipedia) is considered as a hyperedge. The (Threads) graphs model participating users as vertices, whereas the (Tags) ones model the tags of the posts as vertices. We use three stackexchange networks by Benson et al. [3], and set the number of views as weights from [39]. For the StackOverflow instance, the weights are set by querying the public dataset from BigQuery [19]. We created an additional instance from this source. We also generated a new hypergraph from the English Wikipedia dump, where the categories represent the vertices and the articles represent the hyperedges. We selected a category as a vertex if it has at least 25 mentions, which resulted in 293K vertices for 8M articles. The access frequency of each article in December 2024 is set as weight. We answer **RQ1** and **RQ3** with this data set.

We use the hypergraph data set L_{HG} collected earlier [20] for hypergraph partitioning to address **RQ2** and **RQ3**. The set consists of 94 instances, spanning a wide range of applications from DAC routability-driven placement [41], general matrices [8] to SAT solving [2]. As weights we use $\max|e| - |e|$. This function maximizes the cardinality (number of edges matched). These instances have up to 1.4×10^8 hyperedges/-vertices and a maximum hyperedge size of 2.3×10^6 vertices. More statistics are in the Appendix of the full version.

For the stack-based approach of Section 3, we implemented the Guarantee and Lenient update functions. For simplicity, we refer to the algorithm with the stricter Guarantee function as Stack and the one with the lenient function as StackLenient. Both approaches can be configured by their ε parameter. The algorithms from Section 4 are named SwapSet. The SwapSet algorithm has a parameter $\alpha \geq 0$. For comparison, we use a non-streaming Greedy algorithm that sorts the hyperedges based on weight in descending order and greedily adds them to a matching. We also implemented a Naive streaming algorithm that maintains a maximal hypergraph matching (i.e, it includes a hyperedge if it is feasible w.r.t the current matching) in the order that they are streamed.

5.2 Impact of Streaming Order

We now investigate the impact of different streaming orders on our algorithms (**RQ1**). To this end, we stream the hyperedges of the social-link hypergraphs in three ways: by ascending weight, descending weight, and the original input order. The original input order is the order given by either the original files [3] or, in the case of the wikipedia instance, the order that the articles appear in the dump.

Memory. Figure 2 shows the geometric mean memory consumption for all approaches grouped by the ordering. Interestingly, for lower values of ε, the memory consumption is significantly higher for the ascending order, up to 2.97 times over the descending order (StackLenient $\varepsilon = 0$). For the swapping based algorithms (SwapSet) the differences between the orderings are only minimal, nearly reaching the memory consumption of the Naive algorithm. The Greedy algorithm utilizes the same amount of memory for all orderings, 13.43 times more memory than the stack approaches in the geometric mean. The ordering heavily affects our stack-based algorithms, but they still require less memory than the Greedy approach. The SwapSet algorithms are not significantly impacted by the order. Both results are in line with the theoretical results derived in Section 3.3 and 4.

Running time. In Figure 3 the geometric mean of the running times of the compared algorithms are shown. The Naive streaming algorithm is the fastest, requiring similar time over every ordering. The ascending ordering requires more time for both greedy swapping and

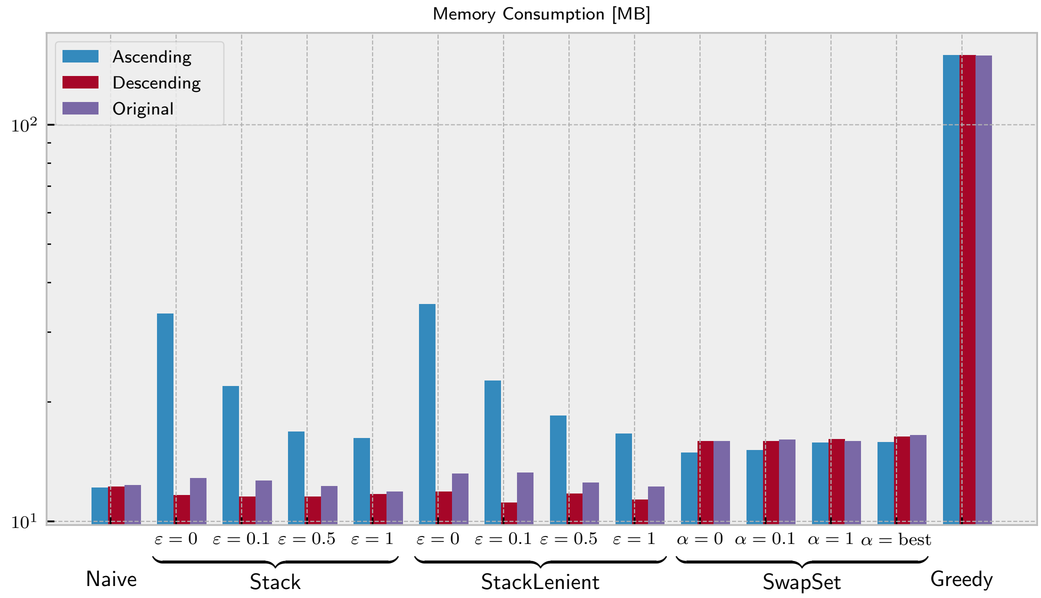

Figure 2 Geometric mean of the memory consumption on the social-link hypergraphs. Note the log-scale on the y-axis.

stack-based approaches since more replacements in the swapping algorithm and placements in the stack happen in that order. The stack-based approaches have a running time only 26% higher than the Naive. We observe that preordering the hyperedges speeds up the offline Greedy algorithm by a factor of 2.73 in comparison to the original ordering, but it is still slower than the stack-based algorithms. This is due to our use of std::sort that is partly optimized for ordered data. The running time of the SwapSet algorithms is comparable to the running time of Greedy with natural ordering. The cost for voiding edges in the SwapSet algorithm is higher than simply pushing edges on the stack. In general, the streaming process of Stack and StackLenient is similar to that of Naive. The small overhead is rooted in the required pass through the stack to build the matching. The ordering affects how many edges are added to the stack (see the following Memory section) and so the running time as well, but not as much as in the Greedy algorithm.

Quality. We present the geometric mean of the weights of matchings in Figure 4. As expected, the Greedy algorithm is not affected by the ordering since it computes its own order in the beginning. The quality of Naive algorithm is heavily affected by orderings, where the best weight is achieved when the hyperedges are streamed in descending order. On ascending and original order, the quality of the Naive is significantly worse. Under the descending orderings, the swapping algorithm (SwapSet) produces the same results as the Greedy algorithm, whereas the ascending order produces worse quality results in the SwapSet algorithm. The Stack and StackLenient approaches are more robust under different orderings and compute better quality matchings in the adversarial ascending order compared to other algorithms..

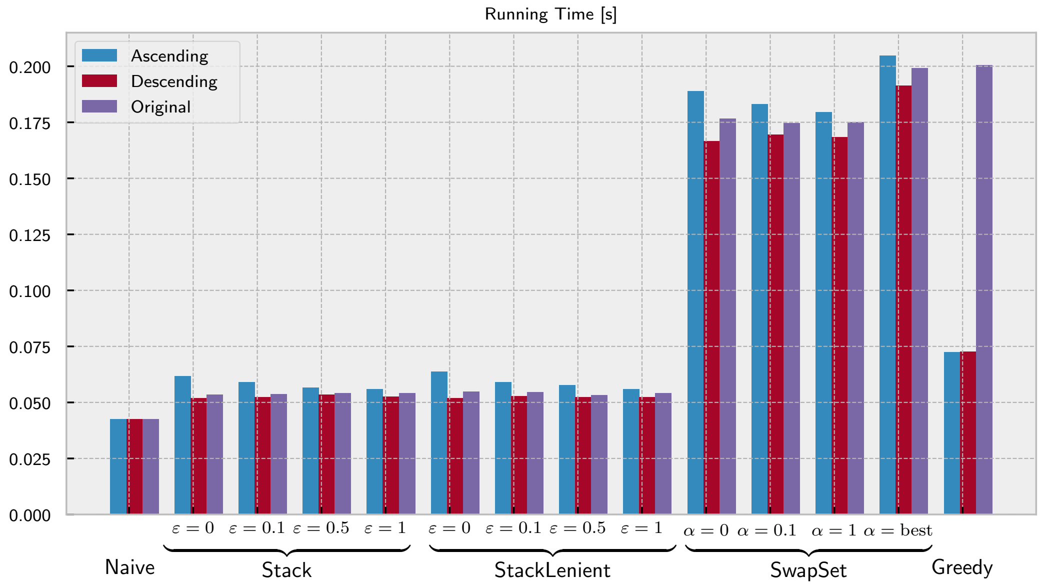

Figure 3 Geometric mean of running times on the social-link hypergraphs.

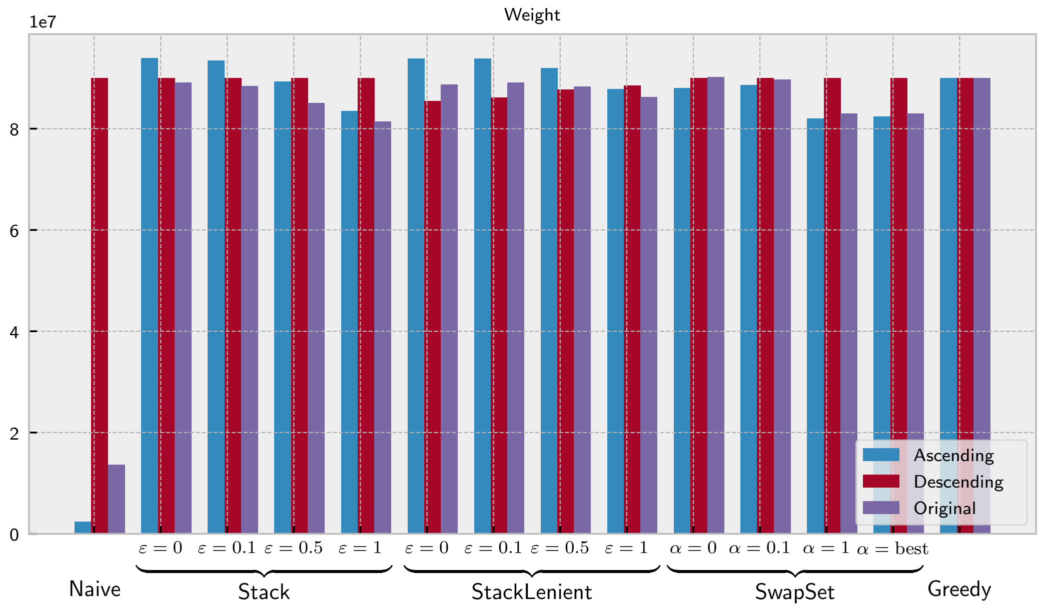

Figure 4 Geometric mean of weights on the social-link hypergraphs.

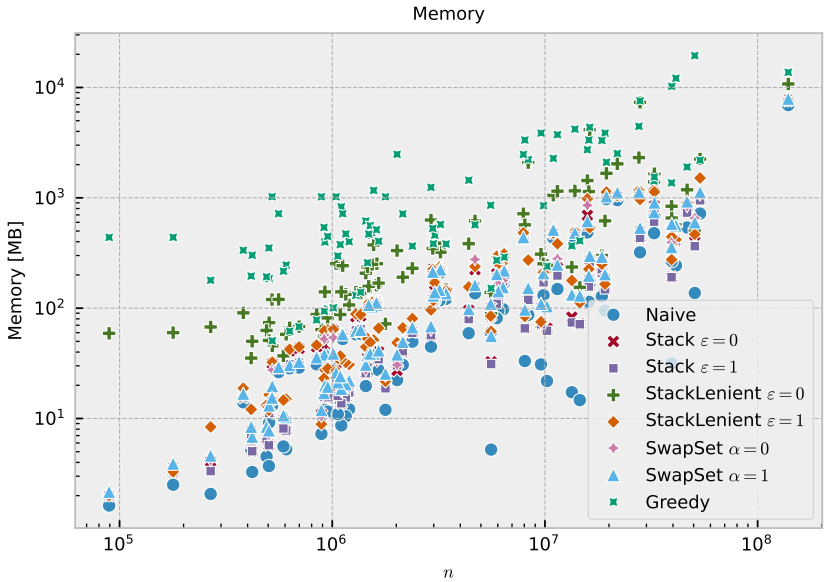

Figure 5 Geometric mean of the memory consumption on hypergraphs from partitioning plotted against the number of vertices.

5.3 Impact of Other Properties

We now test whether some other properties of the instances have an impact on the memory consumption of the algorithm (**RQ2**). Namely, we check if the number of vertices and the number of pins (sum of all edge sizes) are correlated with the memory consumption. We use the data set by Gottesbüren et al. [20] from hypergraph partitioning, which contains many larger-scale instances. This data set contains 94 diverse instances, making it more suitable for statistical testing. The ordering is the original ordering as given by the files in [20]. Instance details can be found in the Appendix of the full version. A hyperedge e in these instances has a weight of $(\max_{e \in E} |e|) - |e|$, optimizing the number of hyperedges matched. We set ε (and also α) to $\{0, 1\}$ since they represent the two extremes in terms of memory usage.

Number of Vertices. Figure 5 shows the memory consumption plotted against the number of vertices in each instance. The plot is on a log-log-scale. Naturally, Naive requires the least amount of memory. Our SwapSet algorithm follows, as well as the Stack algorithm and the StackLenient. Finally, the offline Greedy requires even more memory. This is also backed by the theoretical results of Sections 3.3 and 4. The higher the number of vertices, the smaller is the difference in magnitudes between the approaches. This is due to the additional memory needed for the list of finally matched hyperedges and the overhead in some allocations. The measured peak allocation may contain some overallocated memory caused by the growing result vectors.

Number of Pins. In Figure 6, a plot of the geometric mean memory consumption against the number of pins (sum of all edge sizes) for our approaches and the competitors is shown. Additionally, we added a linear regression line for the Greedy algorithm's memory consumption. The linear model achieves an R^2 score of 0.96 when (randomly) splitting the data set into a

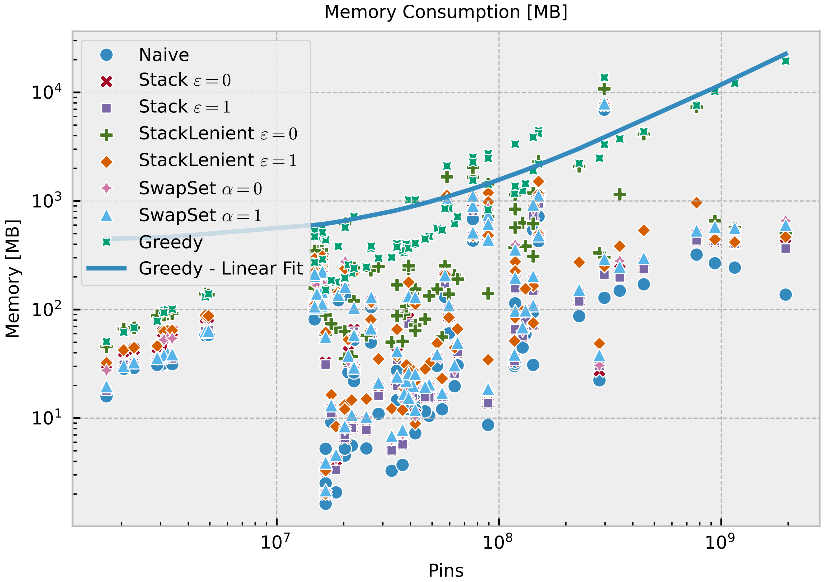

Figure 6 Geometric mean of the memory consumption on the hypergraphs from partitioning. Linear Regression line for the offline Greedy algorithm.

training ($n = 75$) and evaluation test set ($n = 19$), on the latter. This shows that the greedy algorithm requires memory linear in the number of pins since it loads the whole hypergraph at the start. Our proposed algorithms require less memory, especially the SwapSet algorithms nearly match the memory consumption of the Naive approach.

Other Properties. We also tested the dependence of the memory consumption on the average hyperedge size and the number of hyperedges, but found no pattern. Note that the number of pins is equal to the product of the number of edges and the average hyperedge size.

5.4 Comparison with Offline Greedy and Naive Streaming

In this section, we compare the algorithms on the instances stemming from hypergraph partitioning (**RQ3**). The Results on the social set of instances can be found in Section 5.2. Figure 7 shows a performance profile for the cardinality/size of the matchings for our algorithms as well as the Naive streaming and offline Greedy algorithm. The StackLenient variant with $\varepsilon = 0$ computes the biggest matchings and is the best performing algorithm, followed by SwapSet $\alpha = 0$. The Greedy algorithm has similar results to StackLenient $\varepsilon = 1$. The Stack $\varepsilon = 1$ and SwapSet $\alpha = 1$ are our worst-performing algorithms, having results very close to the Naive streaming algorithm. Their higher parameters ($\alpha = \varepsilon = 1$) cause the algorithm to converge to the naive algorithm.

6 Conclusion

We have proposed two (semi-)streaming algorithms for hypergraph matching. The first, inspired by Paz-Schwartzman [33], uses a stack and adds hyperedges to it, based on dual variables it keeps track of and updates according to an update function. The approximation

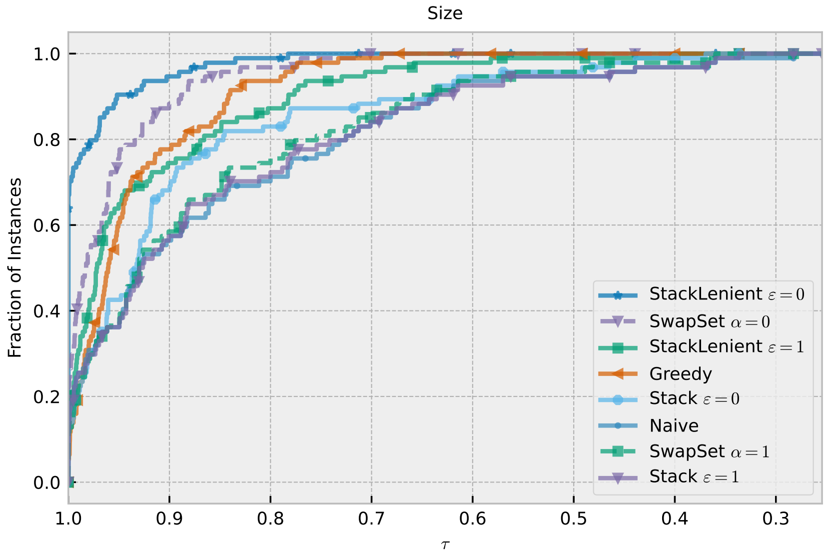

Figure 7 Performance Profile for the size of the matching in hypergraphs used in partitioning.

guarantee for this algorithm is $\frac{1}{d(1+\varepsilon)}$, and its running time per hyperedge is linear in hyperedge size. We have proposed two other update functions to be used in this algorithm. The proposed update functions result in a space complexity of $\mathcal{O}((1/\varepsilon)n\log^2 n)$ and $\mathcal{O}((1/\varepsilon)nd\log^2 n)$ bits. The second proposed algorithm works by greedily swapping out hyperedges and maintaining only one solution, requiring only $\mathcal{O}(n)$ memory. Inspired by McGregor's $\frac{1}{3+2\sqrt{2}}$-approximation guarantee [31], we have shown that if every swap increases the weight by at least $(1+\alpha)$, the algorithm has an approximation guarantee of $\frac{1}{(1+\alpha)(\frac{d-1}{\alpha}+d)}$. The best choice is $\alpha = \sqrt{(d-1)/d}$.

In extensive experiments, we have shown the competitiveness of the proposed algorithms in comparison to the standard non-streaming Greedy and a Naive streaming approach with respect to running time, memory consumption, and quality. We showed that the order of the hyperedges in the stream directly impacts the solution quality and that our stack algorithms handle worst-case orderings (like ascending weights for the Naive algorithm) even better than the offline Greedy algorithm. The running times of our Stack algorithm are only 26% higher than the Naive algorithm. Furthermore, we validated that the memory consumption for both families of algorithms is not linear in the number of pins, as it is for the Greedy algorithm; for the Stack approaches on the social link hypergraphs it is 13 times lower than Greedy. Lastly, we report that our algorithm obtains considerably better cardinality matchings in general hypergraphs from partitioning tasks.

Avenues of future work include improving the solution quality and extending to problems with relaxed capacity constraints. We aim to develop a streaming algorithm that can efficiently handle instances with capacity $b(v) > 1$ at each vertex, while maintaining a reasonable approximation ratio and computational overhead.

References

1 Mokhtar S Bazaraa, John J Jarvis, and Hanif D Sherali. *Linear Programming and Network Flows*. John Wiley & Sons, 2011.

2 Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. The SAT competition 2014. `http://www.satcompetition.org/2014/index.shtml`, 2014.

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. doi:10.1073/pnas.1800683115.

4 Piotr Berman. A $d/2$ approximation for maximum weight independent set in d-claw free graphs. In Magnús M. Halldórsson, editor, *Algorithm Theory - SWAT 2000, 7th Scandinavian Workshop on Algorithm Theory, Bergen, Norway, July 5-7, 2000, Proceedings*, volume 1851 of *Lecture Notes in Computer Science*, pages 214–219, Berlin, Heidelberg, 2000. Springer. doi:10.1007/3-540-44985-X_19.

5 B. Besser and M. Poloczek. Greedy matching: Guarantees and limitations. *Algorithmica*, 77:201–234, 2017. doi:10.1007/S00453-015-0062-2.

6 Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, volume 8097 of *Lecture Notes in Computer Science*, pages 659–670. Springer, Springer, 2013. doi:10.1007/978-3-642-40047-6_66.

7 Marek Cygan. Improved approximation for 3-dimensional matching via bounded pathwidth local search. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 509–518. IEEE, IEEE Computer Society, 2013. doi:10.1109/FOCS.2013.61.

8 Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011. doi:10.1145/2049662.2049663.

9 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 91(2):201–213, 2002. doi:10.1007/s101070100263.

10 Doratha E. Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Inf. Process. Lett.*, 85(4):211–213, 2003. doi:10.1016/S0020-0190(02)00393-9.

11 Fanny Dufossé, Kamer Kaya, Ioannis Panagiotas, and Bora Uçar. Effective heuristics for matchings in hypergraphs. In Ilias S. Kotsireas, Panos M. Pardalos, Konstantinos E. Parsopoulos, Dimitris Souravlias, and Arsenis Tsokas, editors, *Analysis of Experimental Algorithms - Special Event, SEA^2 2019, Kalamata, Greece, June 24-29, 2019, Revised Selected Papers*, volume 11544 of *Lecture Notes in Computer Science*, pages 248–264. Springer, Springer, 2019. doi:10.1007/978-3-030-34029-2_17.

12 Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965. doi:10.4153/CJM-1965-045-4.

13 Jason Evans. jemalloc. http://jemalloc.net/, 2006–. Accessed: 2025-04-15.

14 Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Departmental Papers (CIS)*, page 236, 2005. doi:10.1016/j.tcs.2005.09.013.

15 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. doi:10.4230/LIPIcs.SEA.2024.12.

16 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. doi:10.4230/LIPIcs.ESA.2024.53.

17 Aurélien Froger, Olivier Guyon, and Eric Pinson. A set packing approach for scheduling passenger train drivers: the French experience. In *RailTokyo2015*, 2015.

18 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, January 8-9, 2019, San Diego, CA, USA*, volume 69 of *OASIcs*, pages 13:1–13:8, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.SOSA.2019.13.

19 Google Cloud Platform. Bigquery public datasets. https://cloud.google.com/bigquery/public-data. Accessed: 2025-04-19.

20 Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. *CoRR*, abs/2303.17679, 2023. doi:10.48550/arXiv.2303.17679.

21 Georg Gottlob and Gianluigi Greco. Decomposing combinatorial auctions and set packing problems. *J. ACM*, 60(4):24:1–24:39, 2013. doi:10.1145/2508028.2505987.

22 Ernestine Großmann, Felix Joos, Henrik Reinstädtler, and Christian Schulz. Engineering hypergraph b-matching algorithms. *arXiv preprint arXiv:2408.06924*, 2024. doi:10.48550/arXiv.2408.06924.

23 Oussama Hanguir and Clifford Stein. Distributed algorithms for matching in hypergraphs. In Christos Kaklamanis and Asaf Levin, editors, *Approximation and Online Algorithms - 18th International Workshop, WAOA 2020, Virtual Event, September 9-10, 2020, Revised Selected Papers*, volume 12806 of *Lecture Notes in Computer Science*, pages 30–46. Springer, Springer, 2020. doi:10.1007/978-3-030-80879-2_3.

24 Elad Hazan, Shmuel Safra, and Oded Schwartz. On the complexity of approximating k-set packing. *Comput. Complex.*, 15(1):20–39, 2006. doi:10.1007/s00037-006-0205-6.

25 Chien-Chung Huang and François Sellier. Semi-streaming algorithms for submodular function maximization under b-matching constraint. In Mary Wootters and Laura Sanità, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2021, August 16-18, 2021, University of Washington, Seattle, Washington, USA (Virtual Conference)*, volume 207 of *LIPIcs*, pages 14:1–14:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.APPROX/RANDOM.2021.14.

26 Cor A. J. Hurkens and Alexander Schrijver. On the size of systems of sets every t of which have an SDR, with an application to the worst-case ratio of heuristics for packing problems. *SIAM J. Discret. Math.*, 2(1):68–72, 1989. doi:10.1137/0402008.

27 Richard M. Karp. Reducibility among combinatorial problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 219–241. Springer, 2010. doi:10.1007/978-3-540-68279-0_8.

28 B. Korte and D. Hausmann. An analysis of the greedy algorithm for independence systems. *Annals of Discrete Mathematics*, 2:65–74, 1978.

29 Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 519–528. IEEE Computer Society, 2014. doi:10.1109/IPDPS.2014.61.

30 Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In Camil Demetrescu, editor, *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*, volume 4525 of *Lecture Notes in Computer Science*, pages 242–255. Springer, Springer, 2007. doi:10.1007/978-3-540-72845-0_19.

31 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*, volume 3624 of *Lecture Notes in Computer Science*, pages 170–181. Springer, Springer, 2005. doi:10.1007/11538462_15.

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, SIAM, 2023. doi:10.1137/1.9781611977554.ch41.

33 Ami Paz and Gregory Schwartzman. A $(2+\epsilon)$-approximation for maximum weight matching in the semi-streaming model. *ACM Trans. Algorithms*, 15(2):18:1–18:15, December 2019. doi:10.1145/3274668.

34 Seth Pettie and Peter Sanders. A simpler linear time $2/3$-ε approximation for maximum weight matching. *Information Processing Letters*, 91(6):271–276, 2004. doi:10.1016/j.ipl.2004.05.007.

35 Alex Pothen, SM Ferdous, and Fredrik Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28:541–633, 2019. doi:10.1017/S0962492919000035.

36 Robert Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In Christoph Meinel and Sophie Tison, editors, *STACS 99, 16th Annual Symposium on Theoretical Aspects of Computer Science, Trier, Germany, March 4-6, 1999, Proceedings*, volume 1563 of *Lecture Notes in Computer Science*, pages 259–269. Springer, Springer, 1999. doi:10.1007/3-540-49116-3_24.

37 Henrik Reinstädtler, S M Ferdous, Alex Pothen, Bora Uçar, and Christian Schulz. HeiHGM/Streaming. Software, DFG-SCHU 2567/8-1, swhId: swh:1:dir:4e4522550296fb38202457ce7371bf7034ae45d9 (visited on 2025-09-05). URL: https://github.com/HeiHGM/Streaming, doi:10.4230/artifacts.24674.

38 Henrik Reinstädtler, S M Ferdous, Alex Pothen, Bora Uçar, and Christian Schulz. Semi-streaming algorithms for hypergraph matching, 2025. doi:10.48550/arXiv.2502.13636.

39 StackExchange. Stackexchange data explorer. https://data.stackexchange.com/. Accessed: 2025-04-19.

40 Thorben Tröbst and Rajan Udwani. Almost tight bounds for online hypergraph matching. *arXiv preprint arXiv:2402.08775*, 2024. doi:10.1016/j.orl.2024.107143.

41 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. doi:10.1145/2228360.2228500.

42 David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011. URL: http://www.cambridge.org/de/knowledge/isbn/item5759340/?site_locale=de_DE.

43 Laurence A Wolsey. *Integer Programming*. John Wiley & Sons, 2020.

44 Stephen J. Wright. *Primal-Dual Interior-Point Methods*. SIAM, 1997. doi:10.1137/1.9781611971453.