Streaming Matching and Edge Cover in Practice

S M Ferdous
Pacific Northwest National Laboratory, USA

Alex Pothen
Purdue University, USA

Mahantesh Halappanavar
Pacific Northwest National Laboratory, USA

Abstract

Graph algorithms with polynomial space and time requirements often become infeasible for massive graphs with billions of edges or more. State-of-the-art approaches therefore employ approximate serial, parallel, and distributed algorithms to tackle these challenges. However, such approaches require storing the entire graph in memory and thus need access to costly computing resources such as clusters and supercomputers. In this paper, we present practical streaming approaches for solving massive graph problems using limited memory for two prototypical graph problems: maximum weighted matching and minimum weighted edge cover. For matching, we conduct a thorough computational study on two of the semi-streaming algorithms including a recent breakthrough result that achieves a $1/(2+\varepsilon)$-approximation of the weight while using $O(n \log W/\varepsilon)$ memory (here n is the number of vertices and W is the maximum edge weight), designed by Paz and Schwartzman [SODA, 2017]. Empirically, we show that the semi-streaming algorithms produce matchings whose weight is close to the best 1/2-approximate offline algorithm while requiring less time and an order-of-magnitude less memory.

For minimum weighted edge cover, we develop three novel semi-streaming algorithms. Two of these algorithms require a single pass through the input graph, require $O(n \log n)$ memory, and provide a 2-approximation guarantee on the objective. We also leverage a relationship between approximate maximum weighted matching and approximate minimum weighted edge cover to develop a two-pass $3/2+\varepsilon$-approximate algorithm with the memory requirement of Paz and Schwartzman's semi-streaming matching algorithm. These streaming approaches are compared against the state-of-the-art 3/2-approximate offline algorithm.

The semi-streaming matching and the novel edge cover algorithms proposed in this paper can process graphs with several billions of edges in under 30 minutes using 6 GB of memory, which is at least an order of magnitude improvement from the offline (non-streaming) algorithms. For the largest graph, the best alternative offline parallel approximation algorithm (GPA+ROMA) could not finish in three hours even while employing hundreds of processors and 1 TB of memory. We also demonstrate an application of semi-streaming algorithm by computing a matching using linearly bounded memory on intersection graphs derived from three machine learning datasets, while the existing offline algorithms could not complete on one of these datasets since its memory requirement exceeded 1TB.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms; Theory of computation → Theory and algorithms for application domains; Computing methodologies; Computing methodologies → Shared memory algorithms

Keywords and phrases Matching, Edge Cover, Semi-Streaming Algorithm, Parallel Algorithms, Algorithm Engineering.

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.11

Supplementary Material *Software (Source Code)*: `https://github.com/smferdous1/GraST`

Funding *S M Ferdous*: Laboratory Directed Research and Development Program at PNNL.
Alex Pothen: U.S. Department of Energy SC-0022260.
Mahantesh Halappanavar: U.S. Department of Energy 17-SC-20-SC (ECP ExaGraph) at PNNL.

22nd International Symposium on Experimental Algorithms (SEA 2024).
Editor: Leo Liberti; Article No. 11; pp. 11:1–11:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Solving large-scale graph problems is at the forefront of various research fields, such as high performance computing, data science, and machine learning. An attractive computational model for large-scale graph computations is the *semi-streaming* model [14, 34], which promises to require significantly lower space than the traditional *offline* algorithms. In this paper, we study maximum weight Matching (MWM) and minimum weight edge cover (MWC) in the semi-streaming model. A matching (edge cover) in a graph is a subgraph where the degree of each vertex has an *upper* (*lower*) bound of one. On weighted graphs, for the MWM, we seek a subgraph with the maximum sum of weights of the edges, while for the MWC we minimize the sum of weights.

Matchings have been heavily researched in combinatorial optimization as they have rich algorithmic structures with many real-world applications. MWM can be solved in polynomial time, but the optimal algorithms are expensive and not suited for graphs with billions of edges or more. Consequently, for the past twenty years, offline approximation algorithms have been developed for the MWM problem [3, 39, 8, 36, 9]. See [31, 38] for surveys and a computational study of these algorithms. In the semi-streaming model, matching has also been extensively studied. The first semi-streaming algorithm for MWM is 1/6-approximate, and is due to Feigenbaum et. al. [14] (FB, henceforth). After several improvements, an algorithm with the approximation ratio of $\frac{1}{2+\varepsilon}$ was designed by Paz and Schwartzman [35] (PS, henceforth). Both algorithms use only $\mathcal{O}(n \text{ polylog}(n))$ bits of space, where n is the number of vertices of the graph.

Several natural questions arise: are these streaming algorithms memory-efficient in practice? By optimizing memory, do they sacrifice quality or runtime compared to offline approximation algorithms? In this paper we conduct a thorough computational study of two of the streaming matching algorithms (FB and PS) and compare them with four representative offline approximation algorithms (Greedy [3], PGA [8], GPA [31] and ROMA [36]). Our experiments reveal that the streaming algorithms (especially PS) stand out in terms of quality, memory, and runtime. We also show that the post-processing phase of the PS algorithm can be made parallel using a locally dominant strategy.

Edge cover has not been studied in the semi-streaming model, while in offline settings, there exist several approximation algorithms [26, 16, 15, 38]. For edge cover, we develop and implement three new semi-streaming algorithms (NN, OnePass and TwoPass) with approximation guarantees of $\frac{3}{2} + \varepsilon$ and 2. These algorithms are compared against a state-of-the-art offline algorithm, the primal-dual algorithm (PD [16]), which is 3/2-approximate.

We also consider an application for MWM, MWC and variants: construct sparse graph representations of large datasets, where each instance has several features. Such data sets are used for downstream applications such as semi-supervised learning [24] and privacy preservation [25]. A common practice here is to construct a complete graph with instances as the vertices and edges weighted by a similarity or dissimilarity measure computed from the features of their endpoints. This graph is then sparsified by selecting a subset of edges using variants of matching or edge cover. But this limits the size of data one can process since the complete graph has $\mathcal{O}(n^2)$ edges for n instances. This motivates the use of streaming algorithms, where we do not need to store the graph as it is generated but only a small footprint of it. Indeed, for one of our test problems, the offline algorithm runs out of memory, whereas for two others, we reduce the memory used by a factor of more than 100.

2 Preliminaries

Notations. Let $G = (V, E, w)$ be a simple undirected graph with vertex set V and edge set E, and let $w : E \to R_{>0}$ be a positive weight function defined on the edges. We denote $|V| = n$ and $|E| = m$ throughout the paper. Let W_{max} denote $\max_{e \in E}\{w(e)\}$, and W_{min} denote $\min_{e \in E}\{w(e)\}$. We assume both W_{max} and W_{max}/W_{min} are $\mathcal{O}(n^{\alpha})$, where α is a positive real-valued constant. This assumption lets us represent the weights and their sum in $\mathcal{O}(\log n)$ bits. An edge subset $F \subseteq E$ induces a subgraph of G with the endpoints of F as vertices and edges from F. We use $\mathcal{N}_F(.)$ to denote the neighborhood of a vertex or edge in the subgraph induced by F. For a vertex v, $\mathcal{N}_F(v)$ may represent the edges incident on v ($\{e \in E : v \cap e \notin \emptyset\}$) or vertices adjacent to v ($\{u \in V : \{u, v\} \in E\}$), and which definition is used will be clear from the context. Similarly, we define $\mathcal{N}_F(e)$ for an edge e. If f is a function and X is a set of elements on which f is defined, then $f(X) = \sum_{e \in X} f(e)$.

MWM and MWC Problem. Given a graph $G = (V, E, w)$, a *matching* is a subset of edges $M \subseteq E$, where every vertex of G has *at most* one endpoint in M. A maximum weight matching (MWM) is a matching M^* of maximum sum of weights $w(M^*)$ among all matchings. In an *edge cover* $C \subseteq E$, every vertex of G has *at least* one endpoint in C. A minimum weight edge cover (MWC) is an edge cover C^* of minimum sum of weights $w(C^*)$. The primal-dual formulation of the MWM problem is shown in Equations (1) and (2).

$$\begin{aligned} \max \quad & \sum_{e \in E} w(e)x(e) \\ \text{s.t.} \quad & \sum_{e \in \delta(v)} x(e) \le 1 \;\; \forall v \in V \\ & x(e) \ge 0 \qquad \forall e \in E. \end{aligned} \tag{1}$$

$$\begin{aligned} \min \quad & \sum_{v \in V} y(v) \\ \text{s.t.} \quad & y(u) + y(v) \ge w(e) \;\; \forall e = (u, v) \in E \\ & y(v) \ge 0 \qquad \forall v \in V. \end{aligned} \tag{2}$$

The Semi-Streaming Model. For the semi-streaming MWM and MWC problems, in a pass, the edges of E are presented one at a time in an arbitrary order. We aim to compute a matching or edge cover in G at the end of the stream, using small memory and few passes. The semi-streaming algorithm is output-sensitive, i.e., it is allowed to use memory sizes for processing that are proportional (up to a polylog factor) to the size of the memory needed to store the output. For MWM and MWC, the final solution size is $\mathcal{O}(n)$, and hence the memory limit is $\mathcal{O}(n \text{ polylog}(n))$.

3 Related Work

Henzinger et al. first considered streaming graph problems, using sublinear space in [22]. Unfortunately, many graph problems, such as connectivity and finding paths, are provably intractable with sublinear space. Hence the Semi-streaming model [14, 34] was introduced, which relaxes the memory limit to $\mathcal{O}(n \text{ polylog}(n))$, where n is the number of vertices of the graph. Most of the semi-streaming graph algorithms are analyzed in either an *arbitrary* or *random* order of arrival of the elements. A random order stream assumes the streaming elements arrive uniformly at random, whereas in arbitrary order, the arrival of the stream could be adversarial. An early survey of semi-streaming graph problems is included in [33].

Matching problems are an active area of research in the semi-streaming model. For arbitrary edge arrival, Feigenbaum et al. [14] first studied semi-streaming MWM and presented

Algorithm 1 PS Matching($\{e\{u,v\}\}$,ε)

```
Input: Stream of edges $e\{u,v\}$, A constant $\varepsilon$
Output: Matching $M$, using $\mathcal{O}(\frac{n\log^2 n}{\varepsilon})$ bits space
$\forall v \in V : \phi(v) = 0$; $S \leftarrow \emptyset$; $M \leftarrow \emptyset$          ▷ Initialization
▷ Stream Process
for $e(u,v) \in E$ do
    if $w(e) > (1+\varepsilon)(\phi(u)+\phi(v))$ then
        $w'(e) = w(e) - (\phi(u)+\phi(v))$
        $\phi(u) = \phi(u) + w'(e)$; $\phi(v) = \phi(v) + w'(e)$
        S.push(e)
▷ Post Processing
while $S$ is not empty do
    $e(u,v) \leftarrow$ S.pop()
    if $(V(M) \cap \{u,v\}) = \emptyset$ then
        $M \leftarrow M \cup \{e\}$
```

a 1/6-approx matching, which was improved by several authors over time: [32, 41, 12, 6]. In a recent breakthrough, Paz and Schwartzman [35] showed that a simple algorithm could achieve a $1/(2+\varepsilon)$-approximation. Ghaffari and Wajc further simplified their algorithm and analysis in [18]. For arbitrary order of stream, this result remains the best till now. For random arrival, Gamlath et al. [17] broke the 1/2-approximate barrier and showed a $(1/2+\varepsilon)$-approximate algorithm in a single pass.

For the MWC problem, we are unaware of any semi-streaming algorithms in the literature. An edge cover can be formulated as a special case of a *set cover* problem, and for weighted set cover, Emek and Rosen [11] designed a single pass semi-streaming algorithm with approximation ratio $\sqrt{n}$. However, we report constant factor approximation algorithms for edge cover.

Software for modeling streaming graphs includes STINGER [10], graph-stream [28], and GraphStream [37]. Recent algorithmic and computational studies on streaming combinatorial problems include set-cover [4], connected components [40], and hypergraph partitioning [13]. However, we are not aware of any implementation and experimental study of streaming matching algorithms in the literature. Recent work on engineering algorithms for the dynamic matching problem with computational evaluations is presented in [1, 2, 20, 21].

4 Semi-streaming MWM Algorithms

We briefly describe the two semi-streaming MWM algorithms we implemented. Feigenbaum et al. [14] developed the first semi-streaming MWM algorithm (FB), which maintains a matching M, and modifies it when a sufficiently heavy edge arrives. When an edge $e\{u,v\}$ arrives, FB inspects whether $w(e) > 2 \cdot w(\mathcal{N}_M(e))$, where $\mathcal{N}_M(e)$ represents the neighboring edges of e already in M. If the inequality is true, e is inserted into M, and the edges in $\mathcal{N}_M(e)$ are removed; otherwise, e is ignored. The breakthrough $\frac{1}{2+\varepsilon}$-approximate semi-streaming algorithm (PS) for maximum weighted matching is due to Paz and Schwartzman [35]. The original algorithm was analyzed using local ratio techniques, but later Ghaffari and Wajc [18] provided a simpler primal-dual based analysis of the algorithm which we adopt here.

We now describe the idea underlying the Paz and Schwarzman algorithm. Consider the non-streaming setting first. The algorithm chooses an edge with positive weight, includes it in a stack for candidate matching edges, and subtracts its weight from neighboring edges. It repeats this process as long as edges with positive weights remain. At the end, we unwind the stack and greedily add edges in the stack to the matching. This means that once an edge is added to the matching, any neighboring edges in the stack cannot be added to the matching. To adapt this algorithm to the streaming setting, we keep an approximate dual variable for each vertex that accumulates the weight of edges incident on the vertex and added to the stack. When an edge arrives, we subtract the sum of dual variables of the endpoints of the edge from its weight, and if this reduced weight is positive, then it is added to the stack. If not, it is discarded. The rest of the algorithm proceeds as before. To bound the size of the stack to $O(n \log n)$, we need one more idea, which is to include an edge in the stack only if its weight is greater than a constant factor $(1+\varepsilon)$ times the sum of the approximate dual variables. It can be shown that if the edge weights are polynomial in n, then the size of the stack is bounded as desired, and that the approximation ratio becomes $1/(2+\varepsilon)$.

A formal description of the method is shown in Algorithm 1. It initializes the approximate dual variables (the vector ϕ) to zero, and then processes the streaming edges one by one. When an edge e arrives, the algorithm decides whether to store it in the set of candidate matching edges (the Stack S) or to discard it. This decision is based on whether the dual constraint (shown in the Algorithm) is approximately satisfied for this edge. If the edge is stored, we compute the reduced weight $w'(e) = w(e) - (\phi(u) + \phi(v))$ and add it to both $\phi(u)$ and $\phi(v)$. Ghaffari and Wajc [18] showed that as edges incident on a vertex v are inserted into the stack S, they have weights that exponentially increase with the factor $1+\varepsilon$. Thus for each vertex at most $\mathcal{O}(\log_{1+\varepsilon} W_{max}) = \mathcal{O}(\frac{\log W_{max}}{\varepsilon}) = \mathcal{O}(\frac{\log n}{\varepsilon})$ edges are stored in S (since we assume weights to be polynomial in n).

In the post-processing phase, the algorithm unwinds the stack and constructs a maximal matching, processing edges in the stack order, in serial. The post processing phase of PS can be parallelized by computing a locally dominant matching that preserves the approximation guarantee, which we discuss next.

4.1 Post-processing using Locally Dominant Algorithm

We now show that the post-processing phase of the PS algorithm (Algorithm 1) can be made parallel using locally dominant matchings. To do so, we maintain an array of n stacks $\mathcal{S}$ instead of a single stack S. In the streaming phase, for the eligible incident edges of a vertex v, $\mathcal{S}[v]$ stores the neighboring vertices of v. When an edge $\{u, v\}$ arrives if it is not discarded, we push v in $\mathcal{S}[u]$ and u in $\mathcal{S}[v]$. Let E_S be the set of edges stored in $\mathcal{S}$, i.e., $E_S \coloneqq \bigcup_{u \in V} \cup_{v \in \mathcal{S}[u]} \{u, v\}$.

In the post-processing phase, we use $\mathcal{S}$ to compute a maximal matching locally. Given a set of matching edges M, let $V(M)$ be the union of the endpoints of M, i.e., $V(M) = \cup_{\{u,v\} \in M} \{u, v\}$. An edge $e\{u, v\}$ is *available* w.r.t a matching M if e can be added to M without violating any matching constraints, i.e., $V(M) \cap \{e, v\} = \emptyset$. Given a matching M, we say an edge $e\{u, v\} \in E_S$ is *locally dominant* if the edge is available with respect to M and u is on the top of $\mathcal{S}[v]$ and v is on the top of $\mathcal{S}[u]$. The matching M is *locally dominant* if every edge of M is locally dominant when added to the matching. When the streaming phase ends, we compute a locally dominant matching in the subgraph induced by the edges in E_S.

This new post-processing does not change the approximation guarantee of the original algorithm. We reuse an observation and a corollary from [18].

▶ **Observation 1.** *For $v \in V$, $y(v) = (1+\varepsilon)\phi(v)$ is feasible in the dual LP (2).*

▶ **Corollary 2.** *Let $OPT(LP)$ denote the weight of an optimal solution of the LP (1). Any optimal matching M^* satisfies $w(M^*) \leq OPT(LP) \leq (1+\varepsilon)\sum_{v \in V} \phi(v)$.*

Let $\Delta\phi(e)$ for an edge e be the change to $\sum_v \phi(v)$ in Lines 6 in the Algorithm 2. We observe that if we decide to push the endpoints of e into $\mathcal{S}$, then the change in the total approximate

Algorithm 2 Semi-Stream Matching($\{e\{u,v\}\}$,ε)

```
Input: Stream of edges $e(u,v)$, a constant $\varepsilon$
Output: $\frac{1}{2+\varepsilon}$-approximate Matching $M$, using $\mathcal{O}(n \log^2 n/\varepsilon)$ bits space
 1: $\forall v \in V : \phi(v) = 0$; $M \leftarrow \emptyset$                         ▷ Initialization
 2: $\mathcal{S} \leftarrow$ Array of $n$ empty stacks
 3: for $e(u,v) \in E$ do                                                  ▷ Stream Process
 4:     if $w(e) > (1+\varepsilon)(\phi(u) + \phi(v))$ then
 5:         $w'(e) = w(e) - (\phi(u) + \phi(v))$
 6:         $\phi(u) = \phi(u) + w'(e)$; $\phi(v) = \phi(v) + w'(e)$
 7:         $\mathcal{S}[u]$.push($v$);$\mathcal{S}[v]$.push($u$)
 8: while $\{\exists e\{u,v\} \in E_S : e$ is locally dominant$\}$ do                 ▷ Post Processing
 9:     $M \leftarrow M \cup \{e\}$
10:     Remove $u$ from $\mathcal{S}[v]$ and $v$ from $\mathcal{S}[u]$
```

dual ($\sum_{v \in V} \phi(v)$) due to e is $\Delta\phi(e) = 2w'(e)$. Otherwise, $\Delta\phi(e) = 0$. At the end of the streaming phase, $\sum_{e \in E} \Delta\phi(e) = \sum_{v \in V} \phi(v)$.

The following Lemma proves a relationship between the weight of an edge e and the reduced weight of all of its neighbors in E_S that occur in the stream before e.

▶ **Lemma 3.** *Let the preceding edges of an edge $e\{u,v\} \in E_S$ be denoted by $\mathcal{P}(e) = \{e' | e' \cap e \neq \emptyset, e'$ is inspected no later than $e\}$. Then $w(e) \geq \frac{1}{2} \sum_{e' \in P(e)} \Delta\phi(e') = \sum_{e' \in P(e)} w'(e')$.*

Proof. Before inspecting e in the stream, let the values of $\phi(u)$ and $\phi(v)$ be $\phi'(u)$ and $\phi'(v)$, respectively. Let us first consider $\phi'(u)$, which is the accumulation of all the updates $w'(\{u,x\}) = \frac{1}{2}\Delta\phi(\{u,x\})$ for some vertex x, where the edge $\{u,x\}$ arrives before e in the stream. Hence, $\{u,x\}$ is in $\mathcal{P}(e)$, and v is positioned higher than x in $\mathcal{S}[v]$. A similar argument also holds for the edges $\{v,x\}$ arriving before e. So we have $\phi'(u) + \phi'(v) \geq \frac{1}{2} \sum_{e' \in \mathcal{P}(e) \setminus e} \Delta\phi(e')$. By construction (Line 5) of Algorithm 2, we have

$$w(e) = w'(e) + \phi'(u) + \phi'(v) \geq \frac{1}{2}\Delta\phi(e) + \frac{1}{2} \sum_{e' \in \mathcal{P}(e) \setminus e} \Delta\phi(e') = \frac{1}{2} \sum_{e' \in \mathcal{P}(e)} \Delta\phi(e'). \quad ◀$$

▶ **Lemma 4.** *The matching M produced by Algorithm 2 is $\frac{1}{2+\varepsilon}$-approximate.*

Proof. We denote the constant in the algorithm by ε'. We observe that each edge $e \in E_S$, which does not belong to the matching M, has a neighboring edge in M inspected later than e in the stream. Hence

$$\begin{aligned}
\sum_{e \in M} w(e) &\geq \frac{1}{2} \sum_{e \in M} \sum_{e' \in \mathcal{P}(e)} \Delta\phi(e') && \text{[from Lemma 3]} \\
&\geq \frac{1}{2} \sum_{e \in E_S} \Delta\phi(e) = \frac{1}{2} \sum_{e \in E} \Delta\phi(e) = \frac{1}{2} \sum_{v \in V} \phi(v) && [\Delta\phi(e) = 0 \text{ if } e \notin E_S] \\
&\geq \frac{1}{2(1+\varepsilon')} \sum_{v \in V} y(v) \geq \frac{1}{2(1+\varepsilon')} \sum_{e \in M^*} w(e) && \text{[from Obs. 1 and Cor. 2].} \quad ◀
\end{aligned}$$

Note that the $\varepsilon = 2\varepsilon'$ here is yet another constant. We could also express the approximation ratio as $\frac{1}{2} - \varepsilon$ as $\frac{1}{2+\varepsilon}$ is equivalent to it. The memory requirement of the new algorithm is the same as that of the serial one; since both these algorithms retain the same edges, and the new post-processing requires memory linear in the size of E_S.

Algorithm 3 Finding a locally dominant edge

```
function BESTMATE($u$, $\mathcal{S}[u]$)
    bestU = -1
    while $\mathcal{S}[u]$ is not empty do
        if $\mathcal{S}[u]$.top() is marked as deleted then
            $\mathcal{S}[u]$.pop()
            continue
        bestU = $\mathcal{S}[u]$.top()
        break
    return bestU
```

```
Input: a vertex $u$; the stacks $\mathcal{S}$; Current Matching $M$
Output: Matching endpoints for each vertex, $mate$
procedure LDCHECK($u$,$\mathcal{S}$,$M$)
    if $u$ is not marked as deleted then
        $v$ = BESTMATE($u$,$\mathcal{S}[u]$)
        if $v$ = -1 then
            Mark $u$ as deleted; return
        $bestV$ = BESTMATE($v$,$\mathcal{S}[v]$)
        if $u = bestV$ then                ▷ Locally dominant
            $mate[u] = v$; $mate[v] = u$
            $\mathcal{S}[u]$.pop(); $\mathcal{S}[v]$.pop()
            Mark $u$ and $v$ as deleted
```

4.2 Parallel Implementation

We have proved that any algorithm that computes a locally dominant matching is $\frac{1}{2+\varepsilon}$-approximate, and now we discuss the detailed implementation of our locally dominant algorithm that builds on a Procedure LDCHECK described next. In LDCHECK, for a vertex u, we find its best choice for mate, say v, in the order of its stack and check whether u is also the best choice for v. In that case, $\{u, v\}$ is a locally dominant edge. We apply the *matching operation* on this edge, which includes popping both stacks, updating the mate arrays, and marking u and v as deleted.

To be efficient, we maintain a *work queue* that stores the candidate vertices for matching (a vertex may be stored multiple times). Initially, this queue holds all the vertices. The algorithm continues by deleting a vertex from the work queue until the queue is empty and applies the LDCHECK procedure on this vertex. When an edge is matched, all of its *available* neighbors are inserted in the queue since, for these vertices, the corresponding stacks could potentially be modified. A vertex can be inserted into the queue at most $|\mathcal{N}_{E_s}(v)|$ times, and the total amortized work of a vertex (w.r.t all of its insertions in the queue) is $\mathcal{O}(|\mathcal{N}_{E_s}(v)|)$. The overall time complexity is thus linear in the size of E_S which is $\mathcal{O}(n \log n)$.

In the shared memory parallel implementation, we maintain queues for each of the processors. Initially, all the queues form a partition of V. Each processor in parallel runs the LDCHECK procedure on the vertices of its own queue and updates the queues similarly to the serial algorithm. In the parallel algorithm, a vertex can be in multiple queues and thus may call LDCHECK simultaneously. Moreover, both endpoints of an edge can initiate LDCHECK at the same time. To resolve these, we employ synchronization by defining a lock for each vertex. If an edge $\{u, v\}$ is locally dominant, instead of applying the matching operations immediately, we use the `test-and-lock` function to lock x, where $x = \min\{u, v\}$. If the lock test fails, then the vertex is ignored. Otherwise, we recheck the availability of u and v, perform the pop operation, update the matching, and mark u and v as deleted. The lock is released afterward. The detailed pseudocode of LDCHECK is in Algorithm 3 with the critical region highlighted.

The total work of the parallel algorithm is the same as the serial, i.e., linear in the size of edges in the stack. (The depth might not be $\mathcal{O}(\log n)$ even when the edge weights in the graph are random or the edges are streamed in random order since the stacks keep a non-random subset of edges). By induction on the edges matched by the post-processing algorithm of Paz and Schwartzman, we can show that the matchings obtained by the two algorithms are identical.

We implemented our parallel post-processing using the OpenMP library. Since the number of edges in the post-processing step is relatively small (geometric mean of the ratio of the stack size to the number of edges is only 2.65% for Large graphs), the added overhead of the parallel algorithm provides only slight benefits over the original post-processing, even when 128 processors are used. However, we expect that for massive graphs with trillions of edges, our parallel implementation could outperform the sequential one significantly.

5 Semi-streaming MWC Algorithms

5.1 Nearest Neighbor (NN) Algorithm

The 2-approximate Nearest Neighbor (NN) algorithm chooses an edge of minimum weight incident on a vertex to cover it. The NN algorithm can be implemented easily in a semi-streaming computational model. We maintain an array of edges (μ) of size n, representing a minimum weighted edge incident on a vertex, initialized to the empty set and updated in the course of the algorithm. When an edge $\{u, v\}$ arrives in the stream, we compare its weight with the weights of $\mu(u)$ and $\mu(v)$ and update them accordingly.

▶ **Lemma 5.** *The approximation ratio of the NN algorithm is 2, and its memory requirement is $\mathcal{O}(n \log n)$ bits.*

Proof. Let C^* be an optimum edge cover, and let C denote an edge cover computed by the NN algorithm. For any vertex $v \in V$, we denote by $\nu(v)$ an arbitrary edge in $\mathcal{N}_{C^*}(v)$. Since $\mu(v)$ is a minimum weighted edge incident on v, we have $w(\nu(v)) \geq w(\mu(v))$. An edge in the optimal solution can cover at most two vertices. Hence $w(C^*) \geq \frac{1}{2}\sum_{v \in V} w(\nu(v)) \geq \frac{1}{2}\sum_{v \in V} w(\mu(v)) = \frac{1}{2}w(C)$.

For each vertex v, NN stores $\mu(v)$, which requires $\mathcal{O}(\log n)$ bits, and hence the memory requirement. ◀

5.2 Two pass $\frac{3}{2} + \varepsilon$-approx (TwoPass) Algorithm

We will exploit a relationship between a minimum weighted edge cover (MWC) and a maximum weighted matching (MWM) to design a two-pass streaming algorithm. An MWC can be constructed by solving an MWM on a weight-transformed graph. For a vertex subset W, we will denote by $\mu(W)$ the set of minimum weight edges incident on vertices in W. We denote by $V(M)$ the set of endpoints of edges in a matching M. The following observation shows that some edges of the graph can be pruned for the MWC problem.

▶ **Observation 6.** *Any edge $e\{u, v\} \in E$ with weight $w(e)$ such that $w(e) \geq w(\mu(u)) + w(\mu(v))$ can be removed without changing the weight of an optimal edge cover.*

We define the transformed weight of an edge $\{u, v\}$ as $w'(\{u, v\}) = w(\mu(u)) + w(\mu(v)) - w(\{u, v\})$. By Observation 6, we may discard any edges with $w'(\{u, v\}) \leq 0$. Let E' be the set of edges in G after removing such edges, and $G' = (V, E', w')$ be the subgraph induced by E'. Let M^* be a maximum weight matching in G'. Then an edge cover can be constructed as $C^* = M^* \cup \mu(V \setminus V(M^*))$. We show in Lemma 7 that C^* is a minimum weight edge cover.

▶ **Lemma 7.** *Let M^* be a maximum weight matching obtained with the transformed weights w' from a graph $G(V, E, w)$. Then the edges in M^* together with a set of minimum weight eges incident on the unmatched vertices constitute a minimum weight edge cover C^* of the graph G.*

Proof. By construction of C^*, we have

$$\begin{aligned} w(C^*) &= w(M^*) + w(\mu(V \setminus V(M^*))) \\ &= \sum_{e=\{u,v\}\in M^*} w(\mu(u)) + w(\mu(v)) - w'(e) + w(\mu(V \setminus V(M^*))) \\ &= w(\mu(V)) - w'(M^*). \end{aligned}$$

Let C be an arbitrary edge cover, and $M \subseteq C$ be an arbitrary maximal matching (a matching to which we cannot add an edge without violating the matching conditions) in G'. We construct a new edge cover, C', with a possibly lower weight than C by $C' = M \cup \mu(V \setminus V(M))$. Since each of the edges in $C \setminus M$ covers a single vertex in $V \setminus V(M)$, we have $w(C') \leq w(C)$. It follows that $w(C) \geq w(C') = w(\mu(V)) - w'(M)$. This shows the equivalence of MWM and MWC. Since $w(\mu(V))$ is a constant, $w(C)$ is minimized when $w'(M)$ is maximized. ◀

In fact, this transformation is also approximation preserving. We state and prove the following theorem from [23].

▶ **Theorem 8.** *Let M be a $(1-\alpha)$-approximate matching obtained with the transformed weights w' from a graph $G = (V, E, w)$. Then the edges in M together with a set of minimum weight edges incident on the unmatched vertices constitute a $(1+\alpha)$-approximate edge cover C of the graph G with respect to the original weights w.*

Proof. Let C^* denote an optimal edge cover with respect to the weights w and let M^* denote an optimal matching with respect to the weights w'. It is easy to verify that $w'(\{u,v\}) \leq w(\{u,v\})$ for all edges $\{u,v\}$. We have

$$w'(M^*) = \sum_{\{u,v\}\in M^*} w'(\{u,v\}) \leq \sum_{\{u,v\}\in M^*} w(\{u,v\}) = w(M^*) \leq w(C^*). \tag{3}$$

From the construction of C and the definition of M, we have

$$w(C) = w(M) + \mu(V \setminus V(M)) = \mu(V(M)) - w'(M) + \mu(V \setminus V(M)) = \mu(V) - w'(M).$$

Using the approximation ratio of the matching algorithm that computed M, we obtain

$$\begin{aligned} w(C) = w(\mu(V)) - w'(M) \ &\leq w(\mu(V)) - (1-\alpha)w'(M^*) \\ = w(\mu(V)) - w'(M^*) + \alpha\, w'(M^*) \ &= w(C^*) + \alpha\, w'(M^*) \ \leq (1+\alpha)\, w(C^*). \end{aligned}$$

In the second line, first we use the equation in the proof of Lemma 7, and next we use inequality (3). This completes the proof. ◀

We use this transformation to develop a two-pass (TwoPass) streaming edge cover algorithm. In the first pass, we find the edges $\{\mu(v) : v \in V\}$ using the NN Algorithm discussed in Section 5.1. In the next pass, we construct the reduced weight for an edge in the stream and employ the PS streaming matching algorithm discussed in Section 4. Any vertices v that are unmatched in the matching are then covered by the edges from $\{\mu(v)\}$. Since the PS algorithm is $(\frac{1}{2+\varepsilon}) \equiv (\frac{1}{2} - \varepsilon)$-approximate, by Theorem 8, the TwoPass algorithm is $(\frac{3}{2} + \varepsilon)$-approximate for a constant $\varepsilon \geq 0$. This algorithm requires an additional $\mathcal{O}(n \log n)$ bits space, to store the $\mu(.)$s, compared to the PS algorithm.

Algorithm 4 OnePass Edge Cover($\{e(u,v)\}$)

```
Input: Stream of edges $e(u,v)$
Output: 2-approximate Edge Cover $C$, using $\mathcal{O}(n \log n)$ bits space
 1: $\forall v \in V : \phi(v) = \infty$, tag(v)= 0, cover(v) = $\emptyset$, $\mu(v) = \emptyset$; $C \leftarrow \emptyset$    ▷ Initialization
 2: ▷ Stream Process
 3: for $e\{u,v\} \in E$ do
 4:     Update $\mu(u)$ and $\mu(v)$
 5:     if $e$ is a 2-covering edge then
 6:         for $x \in \{u,v\}$ do
 7:             if tag(x) = 2 then
 8:                 y = cover(x) \ x    ▷ y is the other endpoint of the edge that covers x
 9:                 Update cover(y), tag(y) and $\phi(y)$
10:             Update cover(x), tag(x) and $\phi(x)$
11:     else if $e$ is a 1-covering edge of $x$ then
12:         Update cover(x), tag(x) and $\phi(x)$
13: ▷ Post Processing
14: if $tag(v) = 1$ then
15:     Update $\phi(v)$ and cover(v)    ▷ $\phi(v)$ is updated for analysis only
16: $C = \bigcup_{v \in V} cover(v)$
```

5.3 One pass 2-approx (OnePass) Algorithm

We develop here another one-pass 2-approximation algorithm for the MWC problem. Although this OnePass algorithm provides the same worst-case approximation guarantee and has the same memory requirement as NN, empirically it obtains lower weights.

For each vertex $v \in V$, the algorithm maintains a potential function $\phi(v)$ initialized to ∞ and a $tag(v)$ initialized to zero. It also maintains an array *cover* for each vertex $v \in V$ that stores the covering edge of v. We say an edge is a *2-covering* edge if half of its weight is smaller than the current potentials of both of its endpoints. It is a *1-covering* edge if the weight of this edge is smaller than the potential of only one of the endpoints. When an edge $e\{u,v\}$ arrives in the stream, the algorithm updates $\mu(u)$ and $\mu(v)$, if necessary. Then it checks whether it is a 2-covering or 1-covering edge, and processes it as described below. If the edge satisfies neither condition, it discards the edge.

- e **is a 2-covering edge.** We update $\phi(u)$ and $\phi(v)$ values to $w(e)/2$, and tag u and v as vertices covered by a 2-covering edge by assigning $tag(u) = tag(v) = 2$. The algorithm also updates the values of $cover(u)$ and $cover(v)$ to e. If u or v was covered earlier by a 2-covering edge (Line 7 is true), we change the potential of the other endpoint, say y, to $w(\mu(y))$. We also set $cover(y)$ to $\mu(y)$, and tag y to be covered by a 1-covering edge.
- e **is a 1-covering edge.** Denote the covering endpoint by u. We update $cover(u) = \mu(u)$, the potential $\phi(u) = w(\mu(u))$, and set tag u to 1.

When the streaming phase ends, the $\mu(.)$ array stores a minimum weight edge incident on each vertex. In the post-processing phase, for each vertex v, we update $cover(v) = \mu(v)$, and $\phi(v) = w(\mu(v))$ if v is marked as covered by a 1-covering edge. Finally, $C = \bigcup_{v \in V} cover(v)$ is returned as the edge cover. We show the detailed pseudocode in Algorithm 4.

To prove the approximation guarantee, we start with a simple observation.

▶ **Observation 9.** $w(C) = \sum_{v \in V} \phi(v)$.

▶ **Lemma 10.** *After the post processing phase,* $\phi(u) \leq w(\mu(u))$ *for all* $u \in V$.

Proof. If u is covered by a 1-covering edge, then after the post-processing step, $\phi(u) = w(\mu(u))$. Hence assume that u is covered by a 2-covering edge, say $e\{u, v\}$. There are two cases to consider. i) When the edge $\mu(u)$ streams after $\{u, v\}$. Since $\mu(u)$ was not considered to cover u, we have $\phi(u) \leq w(\mu(u)$. ii) When the edge $\mu(u)$ is streamed before $\{u, v\}$. Consider the iteration after $\mu(u)$ arrives and before u is covered by $\{u, v\}$. If in any one of these iterations, u is covered by a 1-covering edge, $\phi(u)$ becomes $w(\mu(u))$. When $\{u, v\}$ arrives, $\phi(u)$ must be less than $w(\mu(u))$. On the other hand, if u is always covered by a 2-covering edge in all these iterations, then trivially $\phi(u) \leq w(\mu(u))$. ◀

▶ **Lemma 11.** *The* OnePass *algorithm is 2-approximate and uses $\mathcal{O}(n \log n)$ bits of memory.*

Proof. Let C^* be an optimal edge cover. By Observation 9,

$$w(C) \;=\; \sum_{u \in V} \phi(u) \;\leq \sum_{u \in V} w(\mu(u)) \;\leq\; 2w(C^*).$$

The last two inequalities follow from Lemma 10 and the proof of Lemma 5, respectively. ◀

For each vertex $v \in V$, the OnePass algorithm stores the edges $\mu(v)$, $cover(v)$, and a *flag* value representing whether this vertex is covered by 2-covering or 1-covering edge. Thus, the total memory per vertex is $\mathcal{O}(\log n)$ bits, and the memory required is $\mathcal{O}(n \log n)$ bits.

6 Experimental Results

We have implemented the streaming matching and edge cover algorithms discussed in the previous sections and performed a thorough comparison of the algorithms with a number of *offline* approximation algorithms that are implemented on a node of a cluster computer to meet the memory requirements of the large graphs we work with. For larger graphs, the shared memory parallel (OpenMP) version of the ROMA algorithm (Table 2) is used, since it is compute-intensive. All other algorithms are sequential. The algorithms are executed on a node of a community cluster computer with 128 cores in the node, where the node is an AMD EPYC 7662 with 1 TB of total memory. The algorithms are implemented in C++17 and compiled with g++9.3.0 using the `-O3` optimization flag.

We provide two options for streaming of the edges: one option directly reads the edges from a file one by one, which we call TrueStream, while the other reads all the edges into memory first and then simulates the streaming algorithm by reading edges from memory, which we call SimStream. The TrueStream setting is used for comparing the overall runtime and memory of the competing algorithms and the SimStream is used for comparing the actual processing time of the algorithms without the graph reading and construction time. The SimStream also reveals the performance of the streaming algorithms executing in an offline setting. For reporting memory, we use the `getrusage` system call to retrieve the maximum resident set size (RSS) during the program's execution. We refer to the title page for the source code repository that contains our implementation of the streaming algorithms.

6.1 Dataset and Benchmark Algorithms

Our testbed (shown in Table 1) consists of three datasets. In Table 1a, we show the SMALL dataset, which includes ten weighted graphs from the Suitesparse Matrix Collection [7] frequently used for comparing matching algorithms [27, 19]. In Table 1b, we show the LARGE benchmark data that consists of six of the *largest* undirected graphs in the Suitesparse Matrix Collection [7], with each graph having more than 1 billion edges. We assign uniform random

Table 1 Graph statistics of our dataset. K, M and B stand for thousand, million and billion, respectively.

(a) Small graph instances. All the graphs are weighted.

Graph	n	m	Avg. Deg.	Max. Deg.	Min. Deg.
astro-ph	16,706	121,251	14.52	360	0
Reuters911	13,332	148,038	22.21	2,265	0
cond-mat-2005	40,421	175,691	8.69	278	0
gas_sensor	66,917	818,224	24.45	32	7
turon_m	189,924	778,531	8.20	10	1
Fault_639	638,802	13,303,571	41.65	266	0
mouse_ gene	45,101	14,461,095	641.27	8,031	0
bone010	986,703	23,432,540	47.50	62	11
dielFil.V3real	1,102,824	44,101,598	79.98	269	8
kron.logn21	2,097,152	91,040,932	86.82	213,904	0

(b) Large graph instances. U: Unweighted, W: Weighted Graph.

Graph	n	m	Avg. Deg.	Max. Deg.	Min. Deg.
mycielskian20 (U)	786.43 K	1.36 B	3446.42	393,215	19
com-Friendster (U)	65.61 M	1.81 B	55.06	5,214	1
GAP-kron (W)	134.22 M	2.11 B	31.47	1,572,838	0
GAP-urand (W)	134.22 M	2.15 B	32	68	6
MOLIERE_2016 (W)	30.24 M	3.34 B	220.81	2,106,904	0
Agatha_2015 (U)	183.96 M	5.79 B	62.99	12,642,631	1

(c) ML Datasets

Dataset	# of items	# of features	# of edges
IMDb	45,039	30,000	992.94 M
MNIST	60,000	784	1.80 B
DBpedia	545,721	10,000	40.42 B

weights in the range $[1, 10^6]$ for the three unweighted graphs. Table 1c lists three popular machine learning datasets. The MNIST is a hand-written digit recognition dataset, while the two others are text classification data. IMDb [30] is a binary sentiment analysis dataset, and DBpedia [29, 42] is ontology classification data.

We compare the streaming algorithms with a few representative approximate offline algorithms from the literature listed in Table 2. These algorithms are practical to implement [31] and have a similar approximation guarantee as the streaming ones. The parameter ε in the PS algorithm is set to 10^{-3} unless otherwise specified. Following [31], we ran only one phase of ROMA (n random augmentations) on an initial matching generated by GPA algorithm (GPA+ROMA). The PGA, GPA and parallel ROMA implementations are due to Manne and Berge [5]. For a detailed description of these approximation algorithms, we refer the reader to [31, 38].

6.2 Matching Results

Figure 2 shows the relative weight, memory and runtime comparisons of the streaming and offline matching algorithms for the Small (Figure 2a) and Large (Figure 2b) datasets. The relative quantities are computed w.r.t. a baseline algorithm. For weights, the baseline is GPA+ROMA, while for the runtime and memory it is the FB algorithm. The number of runs for each graph and algorithm pair is set to five for the Small dataset and three for the Large graphs. The relative quantity of interest is obtained by computing the mean

Table 2 Benchmark Approximation Offline Algorithms

Algorithm	Approx. Ratio	Time Complexity
Greedy [3]	1/2	$\mathcal{O}(m \log n)$
Global Path Algo. (GPA) [31]	1/2	$\mathcal{O}(m \log n)$
Path Growing Algo. (PGA) [8]	1/2	$\mathcal{O}(m)$
Random Order Augmentation Matching Algo. (ROMA) [36]	2/3 - ε	$\mathcal{O}(m \log 1/\varepsilon)$
Primal-dual `MWC` (PD) [16]	3/2	$\mathcal{O}(mn)$

value of the quantity over the runs and then taking its ratio with the mean baseline value. The top-left plots in both of the subfigures show the *box plots* of the weights achieved by the algorithms across all the graphs relative to the weights of GPA+ROMA. For reporting relative runtime and memory results we show *line plots*, where the x-axis represents the graph instances (sorted from high to low w.r.t. to the edges) and the y-axis represents the relative memories (bottom-left) or runtimes (top and bottom right) of the algorithms compared. The two right plots in both Subfigures in Figure 2 report the relative runtimes. Of these, the top right ones show the total runtimes that include *reading, graph constructions, initialization, processing, and post-processing* time. The bottom-right plots report only the algorithmic time *excluding* reading and graph construction time. For this experiment (bottom-right plots), when streaming algorithms are executed, we used the SimStream setting. For all other experiments, we used TrueStream streaming option. For the largest instance in our data set, the Agatha_2015 graph, the parallel ROMA did not terminate within three hours of runtime. For this instance, we chose GPA weight to be the baseline. We report the quantitative results of the baseline algorithms for our dataset in Table 3.

Quality. From the top left plots in Figure 2, we see that the GPA+ROMA obtains higher weights than other algorithms on almost all the graphs. On the Small dataset, GPA and Greedy perform the next best with similar median relative weights. Then we have the PGA and the streaming algorithm PS. All these algorithms achieved relative weight $\geq 95\%$ of GPA+ROMA. The FB algorithm performs the worst achieving around 90% of the relative weights in terms of median. On the Large dataset, the weight differences from the GPA+ROMA are more pronounced. In terms of median relative weights, the streaming PS and the GPA perform similarly with around 90% in the median. The next are the Greedy and PGA. This result suggests that, for our benchmark instances, the algorithms follow their worst case approximation guarantee in a relative manner. The GPA+ROMA is $2/3 - \varepsilon$-approximate, and it achieves the best weight, while all the other offline algorithms are 1/2-approximate. The quality of PS is better than the FB and comparable to the other half approximate offline algorithms. We note that for a few graphs (e.g., myc20), the streaming algorithms perform significantly worse than the others.

Memory. For memory usage (bottom left plots), as expected, we see that the streaming algorithms significantly outperform the offline ones. For both the datasets, the FB algorithm requires the smallest memory while the PS follows it closely. The Greedy algorithm is the next, which requires 4× to 256× more memory than FB across the Small graphs. For Large graphs, the range is $[16, 2048]\times$. All other offline algorithms require approximately twice the memory of the Greedy. This experiment suggests the streaming algorithms are extremely memory efficient and well-suited to compute matchings in massive graphs.

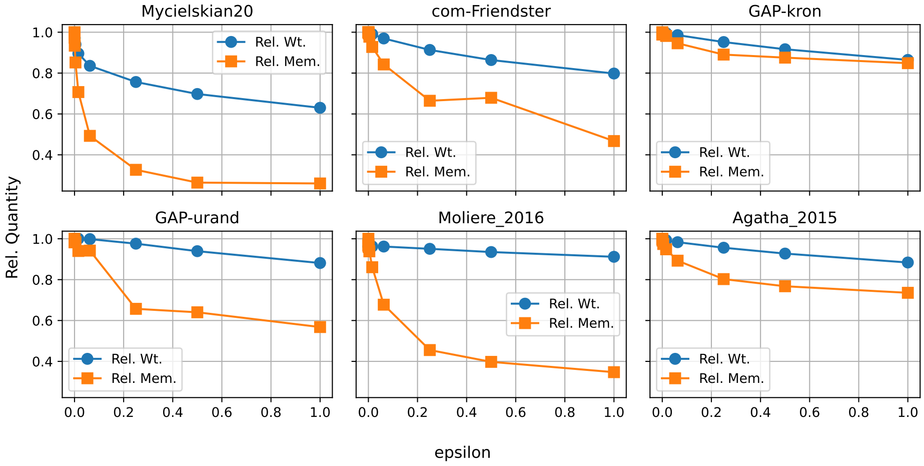


Figure 1 Memory and weight changes of the streaming matching with varying ε on the LARGE dataset. The relative weights and memory are wrt the weights and memory using $\varepsilon = 0$.

Runtimes. Finally, we discuss the runtime results. All these algorithms, except the Greedy, GPA and GPA+ROMA, have linear time complexity. In terms of total runtime in the SMALL dataset (top-right plot in Figure 2a), we see that all the algorithms except GPA and GPA+ROMA are comparable, with PS completing faster for a few of the instances than FB. The GPA and GPA+ROMA require almost twice the time of FB for most of the instances, except for kron, where the GPA+ROMA is 8× slower. For the LARGE dataset, the streaming and the Greedy algorithms are comparable, except for the myc20 graph, where Greedy requires twice the time of the streaming algorithms. When we look into the algorithmic times of these competing algorithms (bottom-right plots in Figure 2, we see clear separations. FB is the fastest, and PS requires at most twice as much time as FB in most of the instances for SMALL graphs, while for LARGE graphs both streaming algorithms are similar. Next is the PGA requiring 2× to 4× more time than the FB for both SMALL and LARGE datasets. The Greedy algorithm is 2× to 16× slower than the FB on the SMALL dataset, while on LARGE ones, it is up to 32× slower. GPA and GPA+ROMA are the two slowest algorithms, where for the SMALL dataset the ranges of relative runtimes are $[4, 32]$ and $[32, 128]$, respectively. For the LARGE graph experiments, GPA needs 8× to 32× more time to complete than the baseline, whereas for GPA+ROMA the range is $[8, 64]\times$. We note that for the large graphs, the ROMA timing reported is from the parallel algorithm using 128 threads, and it did not finish for the Agatha_2015 graph in 3 hours.

These experiments suggest that the streaming algorithms are suitable for solving massive graph problems since they are more memory efficient and faster than the offline ones. We highlight the PS algorithm to be the overall winner since it often achieves quality close to the best weights while significantly outperforming the offline algorithms for memory and runtimes.

Effect of ε on PS. Figure 1 shows the effect of the parameter ε on the streaming matching algorithm, PS. We choose ε from the set $\{0, 2^{-x}\}$, where $x = \{16, 14, 12, 10, 8, 6, 4, 2, 1, 0\}$. The value of ε influences both the approximation ratio and the memory requirement of the algorithm. With increasing ε, we see that both the memory requirement and the weight of the matching decrease. But in almost all cases, the change in the weight is smaller than the decrease in the memory. This experiment suggests that we can substantially decrease memory without significantly decreasing the matching weight.

6.3 Edgecover Results

We now explain Figure 3, which shows the edge cover results for our streaming algorithms(NN, OnePass and TwoPass) and the offline primal-dual algorithm (PD). The experimental setup, number of runs, and subplot description of the figure are exactly the same as the matching experiments described in Section 6.2. For weights, we choose PD to be the baseline, while the runtime and memory results are relative to the NN algorithm.

Quality. The two top-left plots of Figure 3 show the relative weights of the edge cover algorithms. Here, the lower is the better. We see that on both Small and Large datasets, our TwoPass algorithm finds edge cover with smaller weights than the offline PD algorithm. The OnePass is better than NN algorithm in terms of median weight for both datasets, while the NN is the worst in terms of quality.

Memory. The bottom-left plots in the two subfigures of Figure 3 present the relative memory of the edge cover algorithms. The NN requires the least amount of memory space to execute, while both the OnePass and TwoPass need approximately twice the memory across all the datasets. An exception is the mouse_gene dataset, where the TwoPass algorithm takes 8× memory than the NN. The offline PD algorithm is significantly more memory-demanding than the streaming ones. For Small dataset, the PD algorithm requires 8-512× more memory, while for the Large dataset, the range is $[16, 8192]$.

Runtime. The two right plots on the two subfigures of Figure 3 show the relative total runtime and algorithmic time taken by the algorithms. NN and OnePass behave similarly for both datasets, whereas TwoPass is about twice slower. TwoPass reads the graph twice from the file, which dominates the overall runtime. The offline PD is faster than the TwoPass algorithm and comparable to the NN. If we consider only the algorithmic time, the ranking order of the algorithms is: NN, OnePass, TwoPass and the offline PD for both Small and Large datasets.

These experiments highlight the applicability of our streaming algorithm to solve large-scale MWC problems. The geometric means of the relative quantities of MWM and MWC algorithms on both datasets are listed in Table 4.

6.4 Baseline Results

Table 3 presents the weight, time and memory quantities for the baseline algorithms for our experiment. Table 4 shows the geometric means of all the algorithms for both MWM and MWC across the Small and Large datasets. We can conclude that the streaming algorithms reduce the memory required by the offline algorithms by one or two factors of ten. They are also faster than the offline algorithms, and the weights computed by the PS matching algorithm and the TwoPass edge cover algorithms are close to the values obtained by the best performing offline algorithms.

6.5 Matching on ML Dataset

Next we show results on graphs generated from Machine Learning datasets. To compute an item intersection graph from a dataset, for each distinct pair of items, we create an edge with an edge weight calculated from the two feature vectors of the items. Note that the item

Table 3 Baseline algorithm results for matching and edge cover.

	MWM				MWC			
	Weight	Total	Proc.	Mem	Weight	Total	Proc.	Mem
Graph		Time(s)	Time(s)	(MB)		Time(s)	Time(s)	(MB)
Base. Alg.	GPA+	FB	FB	FB	PD	NN	NN	NN
astro-ph	6.44e+3	0.056	0.001	0.52	3.80e+3	0.05	0.001	0.52
Reuters911	2.55e+4	0.050	0.002	0.52	1.02e+4	0.04	0.001	0.52
cond-mat-2005	1.94e+4	0.086	0.001	0.77	1.17e+4	0.07	0.002	0.77
turon_m	5.84e+4	0.547	0.006	3.08	5.62e+4	0.37	0.006	3.08
gas_sensor	1.56e+3	0.925	0.005	1.30	2.06e+1	0.49	0.007	1.29
Fault_639	2.57e+18	6.730	0.103	10.09	6.17e+14	6.43	0.094	10.15
mouse_gene	1.33e+3	7.719	0.114	1.03	6.49e+2	7.05	0.070	1.04
bone010	1.84e+9	15.555	0.113	15.40	4.87e+7	14.25	0.128	15.52
dielFilterV3real	1.76e+5	27.512	0.270	17.17	3.03e+0	26.51	0.322	17.31
kron_g500-logn21	4.63e+5	30.595	1.577	32.34	1.14e+6	27.74	1.167	32.60
mycielskian20	3.22e+11	207.199	6.016	12.33	1.12e+9	206.96	6.526	12.42
com-Friendster	1.99e+13	479.481	45.842	1,001.44	1.18e+13	452.40	50.864	1,009.27
GAP-kron	2.79e+9	887.629	52.938	2,048.34	3.76e+9	880.56	55.430	2,064.35
GAP-urand	1.58e+10	933.648	55.375	2,048.34	8.78e+8	955.00	67.333	2,064.35
MOLIERE_2016	1.67e+6	1,710.526	75.416	461.76	2.74e+6	1,698.38	94.983	465.37
AGATHA_2015	4.88e+13	1,531.040	165.304	2,807.40	1.76e+13	1,435.73	224.678	2,829.34

Table 4 Geometric mean of quantities relative to baseline algorithms for the MWM and MWC problems. For baseline algorithms, see Table 3.

		Small Dataset				Large Dataset			
	Algorithm	Rel.	Rel.	Rel.	Rel	Rel.	Rel.	Rel.	Rel
		Wt.	Mem.	Tot. Tm.	Proc. Tm.	Wt.	Mem.	Tot. Tm.	Proc. Tm.
MWM	FB	0.89	1.00	1.00	1.00	0.76	1.00	1.00	1.00
	PS	0.96	1.70	0.91	1.13	0.90	2.44	0.98	1.26
	Greedy	0.96	21.22	1.13	6.92	0.86	73.41	1.23	5.69
	PGA	0.95	61.63	1.08	2.56	0.85	231.26	2.12	2.60
	GPA	0.98	66.81	1.48	17.77	0.89	233.68	2.72	12.13
	GPA+ROMA	1.00	66.74	2.12	60.82	1.00	272.40	3.56	24.13
MWC	NN	1.26	1.00	1.00	1.00	1.05	1.00	1.00	1.00
	OnePass	1.20	2.17	1.02	1.94	1.02	2.54	1.07	1.51
	TwoPass	0.97	2.57	2.28	2.57	0.98	2.38	2.01	2.02
	PD	1.00	34.55	1.32	7.31	1.00	75.01	1.04	3.36

Table 5 The weight, time, and memory requirement of item intersection graphs for the PS algorithm, and relative results for the Greedy algorithm. The memory for PS is in Megabytes.

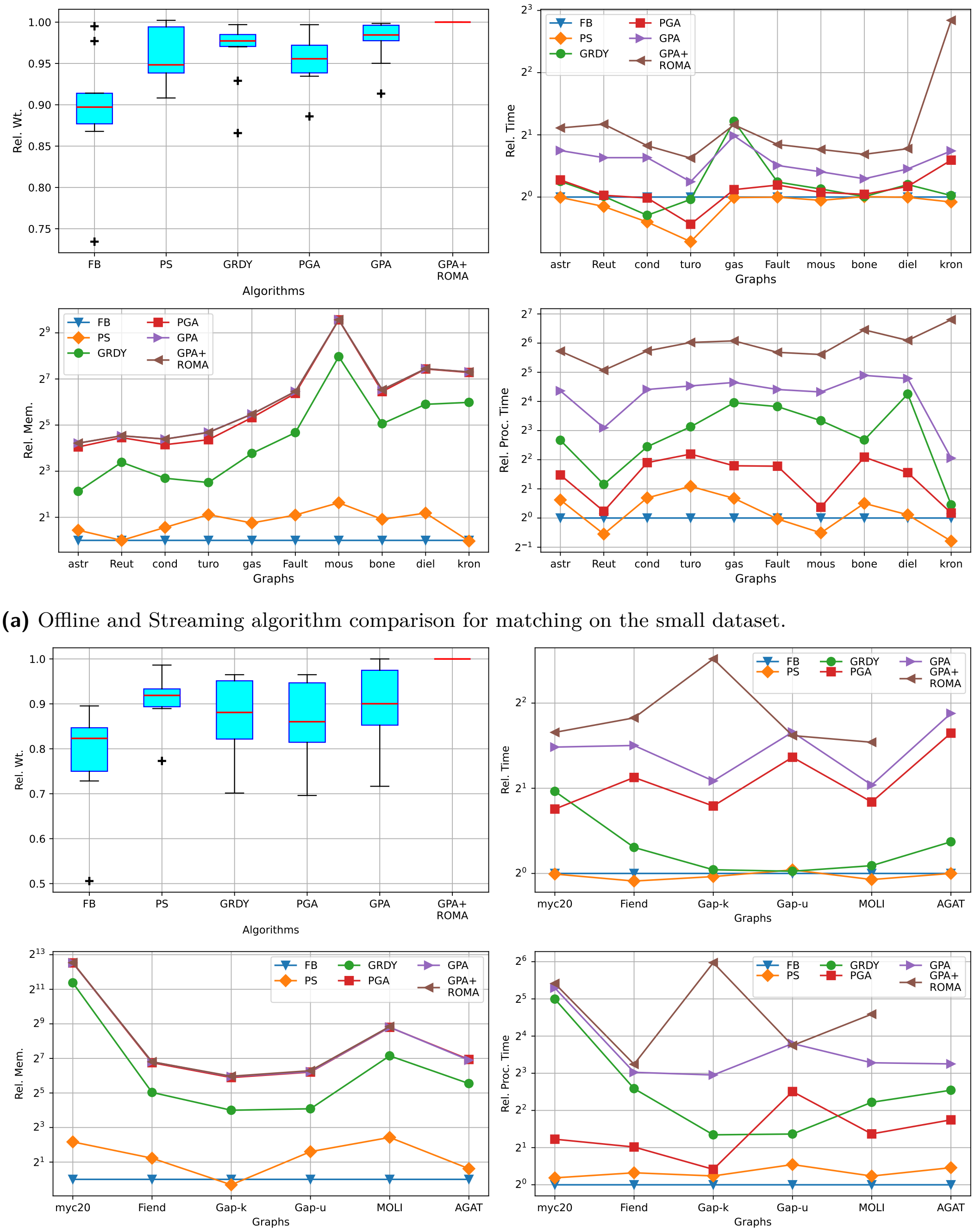
	PS ($\varepsilon = .001$)				Greedy		
Dataset	Wt.	Time	Mem.	Edge-Retn.	Wt.	Rel.	Rel.
		(Sec.)	(MB)	Ratio. (%)	Impr. (%)	Time	Mem.
IMDb	4.42e+3	901.61	98.15	0.02	8.17	1.37	167.92
MNIST	1.43e+12	1,715.88	254.91	0.18	3.81	1.18	129.35
DBpedia	1.25e+5	33,062.95	318.65	0.01	Out of Memory		

intersection graph is dense; indeed, it could be a complete graph. For our experiment, we have used cosine similarity for the IMDb and DBpedia datasets since they are text data. We compute edge weights for the MNIST by subtracting the squared Euclidean distance of the feature vectors from the maximum possible squared distance. Since the size of each image is 28×28 pixels (the size of the feature vector is 784), with pixel value in [0,255], the maximum possible squared distance for MNIST is $784 * 255^2$. The streaming algorithm assumes these edges are generated one by one and passed to the algorithm. But for the offline algorithms, we are required to generate all the edges and then apply the algorithm to them. We compare the PS algorithm (streaming) and the Greedy algorithm (offline), the latter chosen since it is most memory-efficient among offline algorithms.

In Table 5, we observe that while Greedy gets better weights for two of the datasets (8% and 3% increase in weights, respectively), it requires two orders of magnitude memory than the Streaming algorithm. For the DBpedia dataset, where the item intersection graphs contain more than 40 billion edges, the Greedy algorithm failed to terminate since it needed more than 1 TB of memory. This experiment also shows the utility of the streaming algorithm (particularly the PS algorithm) for dense graphs. The *edge-retention ratio*, defined as the ratio of the number of edges in the stack to the total number of edges (in percent), is extremely low, as shown in Table 5.

7 Conclusion and Future work

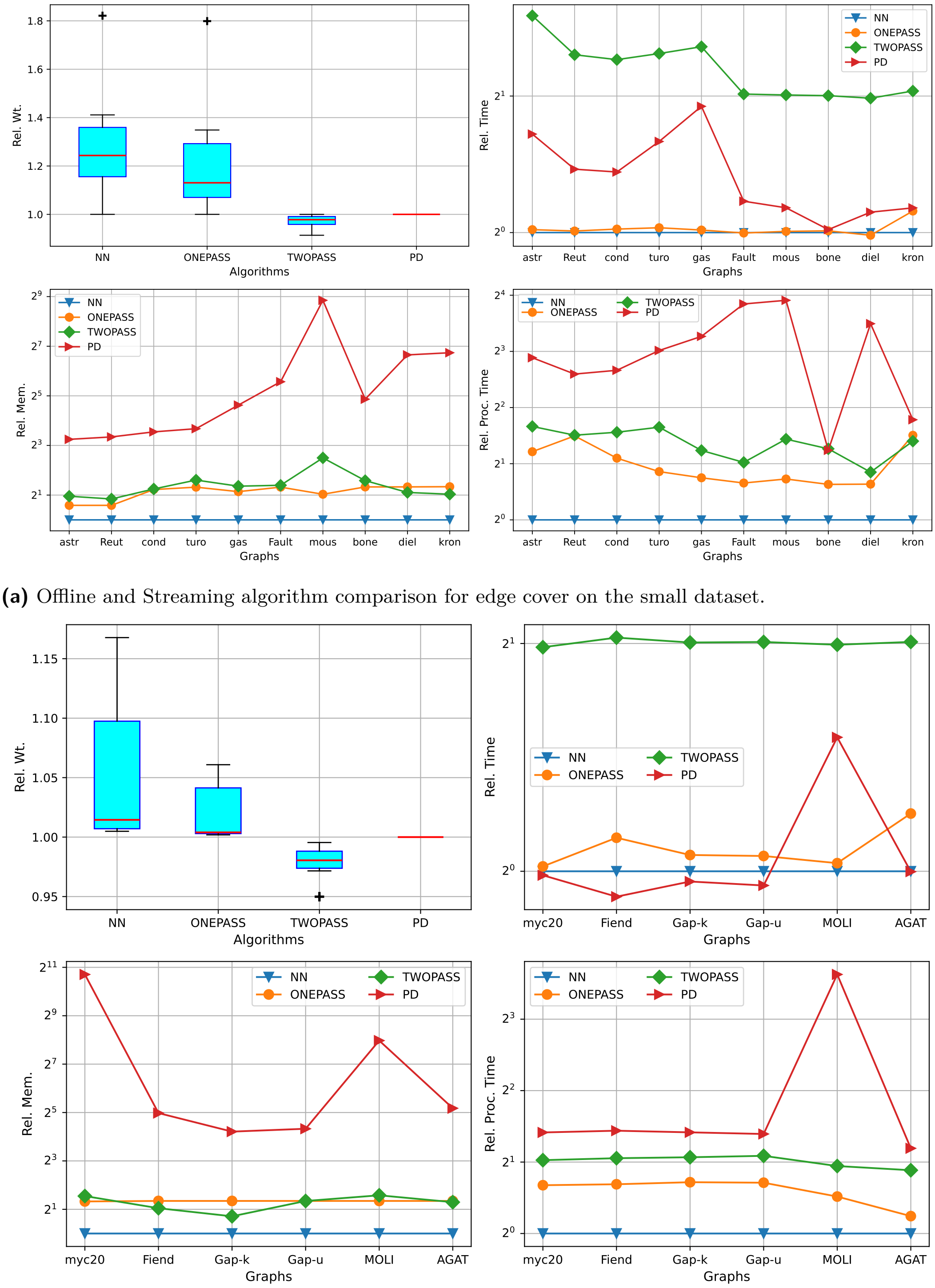
The best streaming algorithms for matching and edge cover compute weights close to the offline algorithms, but as expected, use one to two factors of ten less memory to solve the problems. In a machine learning application, we show that the offline algorithm can run out of memory where the streaming algorithm does not. The streaming algorithms are generally also faster than the offline algorithms. For the MWC, we describe a two-pass $\frac{3}{2} + \varepsilon$-algorithm, and an open question is whether there exists a single pass algorithm with the same guarantee.



(a) Offline and Streaming algorithm comparison for matching on the small dataset.

(b) Offline and Streaming algorithm comparison for matching on the large dataset. The GPA+ROMA algorithm is run on 128 threads to compute solution faster.

Figure 2 Comparison of matching algorithms in the small and large dataset. $\varepsilon = 0.001$ for the PS algorithm. Weight baseline: GPA+ROMA, Runtime and memory baseline: FB. For runtime and memory results, the y-axis scale is logarithmic.



(a) Offline and Streaming algorithm comparison for edge cover on the small dataset.

(b) Offline and Streaming algorithm comparison for edge-cover on the large dataset.

Figure 3 Comparison of edge cover algorithms in the small and large datasets. $\varepsilon = 0.001$ for the TwoPass algorithm. Weight baseline: PD, Runtime, and memory baseline: NN. For runtime and memory results, the y-axis scale is logarithmic.

References

1 Eugenio Angriman, Michal Boron, and Henning Meyerhenke. A batch-dynamic Suitor algorithm for approximating maximum weighted matching. *ACM J. Exp. Algorithmics*, 27:1.6:1–1.6:41, 2022. doi:10.1145/3529228.

2 Eugenio Angriman, Henning Meyerhenke, Christian Schulz, and Bora Uçar. Fully-dynamic weighted matching approximation in practice. In *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms(ACDA)*, pages 32–44. SIAM, 2021. doi:10.1137/1.9781611976830.4.

3 David Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13(4):475–493, 1983. doi:10.1002/net.3230130404.

4 Michael Barlow, Christian Konrad, and Charana Nandasena. Streaming set cover in practice. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 181–192. SIAM, 2021.

5 Andre Berge. A parallel version of the random order augmentation matching algorithm. Master's thesis, University of Bergen, 2020.

6 Michael S. Crouch and Daniel M. Stubbs. Improved streaming algorithms for weighted matching, via unweighted matching. In *Proceedings of Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM)*, volume 28 of *LIPIcs*, pages 96–104. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.APPROX-RANDOM.2014.96.

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

8 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.

9 Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1):1:1–1:23, 2014. doi:10.1145/2529989.

10 David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. STINGER: High performance data structure for streaming graphs. In *Proceedings of IEEE Conference on High Performance Extreme Computing (HPEC)*, pages 1–5. IEEE, 2012. doi:10.1109/HPEC.2012.6408680.

11 Yuval Emek and Adi Rosén. Semi-streaming set cover. *ACM Trans. Algorithms*, 13(1):6:1–6:22, 2016. doi:10.1145/2957322.

12 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. doi:10.1137/100801901.

13 Kamal Eyubov, Marcelo Fonseca Faraj, and Christian Schulz. FREIGHT: Fast streaming hypergraph partitioning. In *Proceedings of the 21st International Symposium on Experimental Algorithms (SEA)*, volume 265 of *LIPIcs*, pages 15:1–15:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.SEA.2023.15, doi:10.4230/LIPICS.SEA.2023.15.

14 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. doi:10.1016/j.tcs.2005.09.013.

15 SM Ferdous, Arif Khan, and Alex Pothen. Parallel algorithms through approximation: b-edge cover. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–33. IEEE, 2018.

16 SM Ferdous, Alex Pothen, and Arif Khan. New approximation algorithms for minimum weighted edge cover. In *Proceedings of the Eighth SIAM Workshop on Combinatorial Scientific Computing (CSC)*, pages 97–108. SIAM, 2018. doi:10.1137/1.9781611975215.10.

17 Buddhima Gamlath, Sagar Kale, Slobodan Mitrovic, and Ola Svensson. Weighted matchings via unweighted augmentations. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 491–500. ACM, 2019. doi:10.1145/3293611.3331603.

18 Mohsen Ghaffari and David Wajc. Simplified and space-optimal semi-streaming $(2+\varepsilon)$-approximate matching. In *Proceedings of the 2nd Symposium on Simplicity in Algorithms (SOSA)*, volume 69 of *OASIcs*, pages 13:1–13:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/OASIcs.SOSA.2019.13.

19 Kathrin Hanauer, Monika Henzinger, Stefan Schmid, and Jonathan Trummer. Fast and heavy disjoint weighted matchings for demand-aware datacenter topologies. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, pages 1649–1658. IEEE, 2022. doi:10.1109/INFOCOM48880.2022.9796921.

20 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms - A quick reference guide. *ACM J. Exp. Algorithmics*, 27:1.11:1–1.11:45, 2022. doi:10.1145/3555806.

21 Monika Henzinger, Shahbaz Khan, Richard Paul, and Christian Schulz. Dynamic matching algorithms in practice. In *Proceedings of the 28th Annual European Symposium on Algorithms (ESA)*, volume 173 of *LIPIcs*, pages 58:1–58:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ESA.2020.58.

22 Monika Rauch Henzinger, Prabhakar Raghavan, and Sridhar Rajagopalan. Computing on data streams. In James M. Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, May 20-22, 1998*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 107–118. DIMACS/AMS, 1998. doi:10.1090/dimacs/050/05.

23 Dawei Huang and Seth Pettie. Approximate generalized matching: f-matchings and f-edge covers. *Algorithmica*, 84(7):1952–1992, 2022. doi:10.1007/s00453-022-00949-5.

24 Tony Jebara, Jun Wang, and Shih-Fu Chang. Graph construction and b-matching for semi-supervised learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, pages 441–448. ACM, 2009. doi:10.1145/1553374.1553432.

25 Arif Khan, Krzysztof Choromanski, Alex Pothen, S. M. Ferdous, Mahantesh Halappanavar, and Antonino Tumeo. Adaptive anonymization of data using b-edge cover. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, pages 59:1–59:11. IEEE / ACM, 2018. URL: http://dl.acm.org/citation.cfm?id=3291735.

26 Arif Khan and Alex Pothen. A new 3/2-approximation algorithm for the b-edge cover problem. In *Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing (CSC)*, pages 52–61. SIAM, 2016. doi:10.1137/1.9781611974690.CH6.

27 Arif Khan, Alex Pothen, Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Fredrik Manne, Mahantesh Halappanavar, and Pradeep Dubey. Efficient approximation algorithms for weighted b-matching. *SIAM J. Sci. Comput.*, 38(5):S593–S619, 2016. doi:10.1137/15M1026304.

28 Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *Soc. Netw. Anal. Min.*, 8(1):61:1–61:29, 2018. doi:10.1007/s13278-018-0537-7.

29 Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van, Sören Auer, et al. DBpedia–A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic web*, 6(2):167–195, 2015.

30 Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150. The Association for Computer Linguistics, 2011. URL: https://aclanthology.org/P11-1015/.

31 Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In *Proceedings of the 6th International Workshop of Experimental Algorithms (WEA)*, volume 4525, page 242. Springer, 2007. `doi:10.1007/978-3-540-72845-0\_19`.

32 Andrew McGregor. Finding graph matchings in data streams. In *Proceedings of the 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, volume 3624 of *LNCS*, pages 170–181. Springer, 2005. `doi:10.1007/11538462\_15`.

33 Andrew McGregor. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, 2014. `doi:10.1145/2627692.2627694`.

34 S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005. URL: `http://dx.doi.org/10.1561/0400000002`, `doi:10.1561/0400000002`.

35 Ami Paz and Gregory Schwartzman. A $(2+\varepsilon)$-approximation for maximum weight matching in the semi-streaming model. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2153–2161. SIAM, 2017. `doi:10.1137/1.9781611974782.140`.

36 Seth Pettie and Peter Sanders. A simpler linear time $2/3$-ε approximation for maximum weight matching. *Inf. Process. Lett.*, 91(6):271–276, 2004. `doi:10.1016/J.IPL.2004.05.007`.

37 Yoann Pigné, Antoine Dutot, Frédéric Guinand, and Damien Olivier. Graphstream: A tool for bridging the gap between complex systems and dynamic graphs. *CoRR*, abs/0803.2093, 2008. URL: `http://arxiv.org/abs/0803.2093`, `arXiv:0803.2093`.

38 Alex Pothen, S. M. Ferdous, and Fredrik Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numer.*, 28:541–633, 2019. `doi:10.1017/S0962492919000035`.

39 Robert Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Proocedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STAC)*, volume 1563 of *Lecture Notes in Computer Science*, pages 259–269. Springer, 1999. `doi:10.1007/3-540-49116-3\_24`.

40 David Tench, Evan West, Victor Zhang, Michael A Bender, Abiyaz Chowdhury, J Ahmed Dellas, Martin Farach-Colton, Tyler Seip, and Kenny Zhang. GraphZeppelin: Storage-friendly sketching for connected components on dynamic graph streams. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 325–339. ACM, 2022. `doi:10.1145/3514221.3526146`.

41 Mariano Zelke. Weighted matching in the semi-streaming model. *Algorithmica*, 62(1-2):1–20, 2012. `doi:10.1007/s00453-010-9438-5`.

42 Xiang Zhang, Junbo Jake Zhao, and Yann LeCun. Character-level convolutional networks for text classification. In *Proceedings of Annual Conference on Neural Information Processing Systems*, pages 649–657, 2015. URL: `https://proceedings.neurips.cc/paper/2015/hash/250cf8b51c773f3f8dc8b4be867a9a02-Abstract.html`.