HAL
archives-ouvertes.fr

Fully-dynamic Weighted Matching Approximation in Practice

Eugenio Angriman, Henning Meyerhenke, Christian Schulz, Bora Uçar

▶ **To cite this version:**

Eugenio Angriman, Henning Meyerhenke, Christian Schulz, Bora Uçar. Fully-dynamic Weighted Matching Approximation in Practice. 2021. hal-03210915

HAL Id: hal-03210915

https://hal.inria.fr/hal-03210915

Preprint submitted on 28 Apr 2021

Fully-dynamic Weighted Matching Approximation in Practice

Eugenio Angriman*   Henning Meyerhenke*   Christian Schulz†   Bora Uçar‡

Abstract

Finding large or heavy matchings in graphs is a ubiquitous combinatorial optimization problem. In this paper, we engineer the first non-trivial implementations for approximating the dynamic weighted matching problem. Our first algorithm is based on random walks/paths combined with dynamic programming. The second algorithm has been introduced by Stubbs and Williams without an implementation. Roughly speaking, their algorithm uses dynamic unweighted matching algorithms as a subroutine (within a multilevel approach); this allows us to use previous work on dynamic unweighted matching algorithms as a black box in order to obtain a fully-dynamic weighted matching algorithm. We empirically study the algorithms on an extensive set of dynamic instances and compare them with optimal weighted matchings. Our experiments show that the random walk algorithm typically fares much better than Stubbs/Williams (regarding the time/quality tradeoff), and its results are often not far from the optimum.

1 Introduction

A matching in a graph is a set of pairwise vertex-disjoint edges. Alternatively, a matching can be seen as a subgraph (restricted to its edges) with degree at most 1. A matching is *maximal* if no edges can be added to it without violating the matching property that no two matching edges share a common vertex. A matching of a graph G is *maximum*, in turn, if there exists no matching in G with higher cardinality. Computing (such) matchings in a graph is a ubiquitous combinatorial problem that appears in countless applications [28]. Two popular optimization problems in this context are (i) the maximum cardinality matching (MCM) problem, which seeks a matching with maximum cardinality, and (ii) the maximum weighted matching (MWM) problem, i.e., to find a matching (in a weighted graph) whose total edge weight is maximum. Micali and Vazirani [40] solve MCM in $\mathcal{O}(m\sqrt{n})$ time, Mucha and Sankowski [41] in $\mathcal{O}(n^\omega)$, where $\omega < 2.373$ is the matrix multiplication exponent [1], m is the number of edges, and n is the num-

ber of vertices. Concerning MWM, the best known algorithm is by Galil et al. [24], it takes $\mathcal{O}(mn \log n)$ time. For integral edge weights up to W, Gabow and Tarjan [23] proposed an $\mathcal{O}(m\sqrt{n}\log nW)$ time algorithm. Sankowski's algorithm [47], in turn, requires $\tilde{\mathcal{O}}(Wn^\omega)$ time, where $\tilde{\mathcal{O}}$ hides a polylogarithmic factor.

Real-world graphs occurring in numerous scientific and commercial applications are not only massive but can also change over time [39]. For example, edges can be inserted, deleted, or their weight can be updated – think of a road network where edge weights represent the traffic density, the constantly evolving web graph, or the virtual friendships between the users of a social network. Hence, recomputing a (weighted) matching from scratch each time the graph changes is expensive. With many frequent changes, it can even be prohibitive, even if we are using a polynomial-time algorithm. Further, when dealing with large graphs, computing an optimal matching can be too time-consuming and thus one often resorts to approximation to trade running time with solution quality. In recent years, several fully-dynamic algorithms for both exact and approximate MCM [3, 5, 6, 8–11, 14, 25, 27, 32, 33, 42, 43, 46, 48] and MWM [2, 26, 49] have been proposed. These algorithms exploit previously computed information about the matching to handle graph updates more efficiently than a static recomputation. Yet, limited effort has been invested to actually implement these algorithms in executable code and test their performance on real-world instances. Only recently, Henzinger et al. [30] engineered several fully-dynamic algorithms for MCM and investigated their practical performance on large real-world graphs. Similar results have not been produced yet for fully-dynamic MWM algorithms (neither exact nor approximate) and therefore their practical performance is still unknown.

Contributions. In this paper we focus on the fully-dynamic MWM problem and explore the gap between theoretical results and practical performance for two algorithms. The first one, inspired by Maue and Sanders [38], combines random walks and dynamic programming to compute augmenting paths in the graph; if the random walks are of appropriate length and repeated sufficiently often, the algorithm maintains a $(1 + \epsilon)$-approximation in $\mathcal{O}(\epsilon^{-1}\Delta^{2/\epsilon+3}\log n)$ time per

---

*Humboldt-Universität zu Berlin, Berlin, Germany
†Heidelberg University, Heidelberg, Germany
‡CNRS and LIP, ENS Lyon, France

update, where Δ is the maximum node degree observed throughout the algorithm. The second algorithm, introduced by Stubbs and Williams [49], uses existing fully-dynamic α-approximation algorithms for MCM as subroutines to maintain a $2\alpha(1 + \epsilon)$-approximation for MWM in fully-dynamic graphs. We provide the first implementations of the aforementioned algorithms, analyze their performance in systematic experiments on an extensive set of dynamic instances, and compare the quality of the computed matchings against the optimum. The best algorithm is very often less than 10% away from the optimum and at the same time very fast.

2 Preliminaries

Basic Concepts. Let $G = (V, E)$ be a simple *undirected graph* with edge weights $\omega : E \to \mathbb{R}_{>0}$. We extend ω to sets, i.e., $\omega(E') := \sum_{e \in E'} \omega(e)$. We set $n = |V|$, and $m = |E|$; $N(v) := \{u : \{v, u\} \in E\}$ denotes the *neighbors* of v. The (unweighted) degree of a vertex v is $d(v) := |N(v)|$. A matching $\mathcal{M} \subset E$ in a graph is a set of edges without common vertices. The *cardinality* or *size* of a matching is simply the cardinality of the edge subset $\mathcal{M}$. We call a matching *maximal* if there is no edge in E that can be added to $\mathcal{M}$. A *maximum cardinality matching* $\mathcal{M}_{\text{opt}}$ is a matching that contains the largest possible number of edges of all matchings. A *maximum weight matching* $\mathcal{M}'_{\text{opt}}$ is a matching that maximizes $\omega(\mathcal{M}'_{\text{opt}})$ among all possible matchings. An *α-approximate* maximum (weight) matching is a matching that has weight at least $\frac{\omega(\mathcal{M}_{\text{opt}})}{\alpha}$. A vertex is called *free* or *unmatched* if it is not incident to an edge of the matching. Otherwise, we call it *matched*. For a matched vertex u with $\{u, v\} \in \mathcal{M}$, we call vertex v the *mate* of u, which we denote as $\text{mate}(u) = v$. For an unmatched vertex u, we define $\text{mate}(u) = \bot$. An *augmenting path* is defined as a cycle-free path in the graph G that starts and ends on a *free* vertex and where edges from $\mathcal{M}$ alternate with edges from $E \backslash \mathcal{M}$. The *trivial augmenting path* is a single edge with both endpoints free. Throughout this paper, we call such an edge a *free* edge. If we take an augmenting path and resolve it by matching every unmatched edge and unmatching every matched edge, we increase the cardinality of the matching by one. In the maximum cardinality case, any matching without *augmenting paths* is a maximum matching [7] and any matching with no augmenting paths of length at most $2k-3$ is a $(k/(k-1))$-approximate maximum matching [31]. A *weight-augmenting path* $\mathcal{P}$ with respect to a matching $\mathcal{M}$ is an alternating path whose free edges are heavier than its edges in M: $w(\mathcal{M} \oplus \mathcal{P}) > w(\mathcal{M})$, where $\oplus$ denotes the symmetric difference. A weight-augmenting path with k edges outside $\mathcal{M}$ is called *weight-augmenting k-*

path. Note that a weight-augmenting k-path can have $2k - 1, 2k$, or $2k + 1$ edges, whereas in the unweighted case an augmenting path with k edges outside of a given matching has exactly $2k - 1$ edges.

Our focus in this paper are *fully-dynamic graphs*, where the number of vertices is fixed, but edges can be added and removed.[1] All the algorithms evaluated can handle edge insertions as well as edge deletions. In the following, Δ denotes the maximum degree that can be found in any state of the dynamic graph.

Related Work. Dynamic algorithms are a widely researched topic. We refer the reader to the recent survey by Hanauer et al. [29] for most material related to theoretical and practical dynamic algorithms. Many of the numerous matching applications require matchings with certain properties, like maximal (no edge can be added to $\mathcal{M}$ without violating the matching property) or maximum cardinality matchings. Edmonds's blossom algorithm [22] computes a maximum cardinality matching in a static graph in time $\mathcal{O}(mn^2)$. This result was later improved to $\mathcal{O}(m\sqrt{n})$ by Micali and Vazirani [40]. Some recent algorithms use simple data reductions rules [34] or shrink-trees instead of blossoms [21] to speed up computations in static graphs. In practice, these algorithms can still be time-consuming for many applications involving large graphs. Hence, several practical approximation algorithms with nearly-linear running time exist such as the local max algorithm [13], the path growing algorithm [19], global paths [38], and suitor [37]. As this paper focuses on dynamic graphs, we refer the reader to the quite extensive related work section of [21] for more recent static matching algorithms.

In the dynamic setting, the maximum matching problem has been prominently studied ensuring α-approximate guarantees. A major exception is the algorithm by van den Brand et al. [50], which maintains the *exact* size of a maximum matching in $\mathcal{O}(n^{1.407})$ update time. One can trivially maintain a maximal (2-approximate) matching in $\mathcal{O}(n)$ update time by resolving all trivial augmenting paths of length one. Ivković and Llyod [32] designed the first fully-dynamic algorithm to improve this bound to $\mathcal{O}((n + m)^{\sqrt{2}/2})$ update time. Later, Onak and Rubinfeld [43] presented a randomized algorithm for maintaining an $\mathcal{O}(1)$-approximate matching in a dynamic graph that takes $\mathcal{O}(\log^2 n)$ expected amortized time for each edge update. This result led to a flurry of results in this area. Baswana, Gupta and Sen [6] improved the approximation ratio of [43] from $\mathcal{O}(1)$ to 2 and the amortized

---

[1]The sole focus on dynamic edges is no major limitation: one could simply insert enough singleton vertices into the original graph to mimic vertex insertions. Only new edges inserted over time would make the new vertices eligible as matching partners.

update time to $\mathcal{O}(\log n)$. Further, Solomon [48] improved the update time of [6] from amortized $\mathcal{O}(\log n)$ to *constant*. However, the first deterministic data structure improving [32] was given by Bhattacharya et al. [10]; it maintains a $(3 + \epsilon)$ approximate matching in $\tilde{O}(\min(\sqrt{n}, m^{1/3}/\epsilon^2))$ amortized update time, which was further improved to $(2 + \epsilon)$ requiring $\mathcal{O}(\log n/\epsilon^2)$ update time by Bhattacharya et al. [11]. Recently, Bhattacharya et al. [9] achieved the first $\mathcal{O}(1)$ amortized update time for a deterministic algorithm but for a weaker approximation guarantee of $\mathcal{O}(1)$. For worst-case bounds, the best results are by (i) Gupta and Peng [26] requiring $\mathcal{O}(\sqrt{m}/\epsilon)$ update time for a $(1 + \epsilon)$-approximation, (ii) Neiman and Solomon [42] requiring $\mathcal{O}(\sqrt{m})$ update time for a $(3/2)$-approximation, and (iii) Bernstein and Stein [8] requiring $\mathcal{O}(m^{1/4}/\epsilon^{2.5})$ update time for a $(3/2 + \epsilon)$-approximation. Recently, Charikar and Solomon [14] as well as Arar et al. [3] (using [12]) presented independently the first algorithms requiring $\mathcal{O}(poly \log n)$ worst-case update time while maintaining a $(2 + \epsilon)$-approximation. Recently, Grandoni et al. [25] gave an incremental matching algorithm that achieves a $(1+\epsilon)$-approximate matching in constant deterministic amortized time. Barenboim and Maimon [5] present an algorithm that has $\tilde{\mathcal{O}}(\sqrt{n})$ update time for graphs with constant neighborhood independence. Kashyop and Narayanaswamy [33] give a conditional lower bound for the update time, which is sublinear in the number of edges for two subclasses of fully-dynamic algorithms, namely lax and eager algorithms.

Despite this variety of different algorithms, to the best of our knowledge, very limited efforts have been made so far to engineer these dynamic algorithms and to evaluate them on real-world instances. Henzinger et al. [30] have started to evaluate algorithms for the dynamic maximum cardinality matching problem in practice. To this end, the authors engineer several dynamic maximal matching algorithms as well as an algorithm that is able to maintain the maximum matching. The algorithms implemented in their work are Baswana, Gupta and Sen [6], which performs edge updates in $\mathcal{O}(\sqrt{n})$ time and maintains a 2-approximate maximum matching, the algorithm of Neiman and Solomon [42], which takes $\mathcal{O}(\sqrt{m})$ time to maintain a $(3/2)$-approximate maximum matching, as well as two *novel* dynamic algorithms: a random walk-based algorithm as well as a dynamic algorithm that searches for augmenting paths using a (depth bounded) blossom algorithm. Experiments indicate that maintaining optimum matchings can be done much more efficiently than the naive algorithm that recomputes maximum matchings from scratch (more than an order of magnitude faster). Second, all inexact dynamic algorithms that

have been considered in that work are able to maintain near-optimum matchings in practice while being multiple orders of magnitudes faster than the naive optimum dynamic algorithm. The study concludes that in practice an extended random walk-based algorithms should be the method of choice.

For the *weighted* dynamic matching problem, Anand et al. [2] propose an algorithm that can maintain an 4.911-approximate dynamic maximum weight matching that runs in amortized $\mathcal{O}(\log n \log D)$ time, where D is the ratio between the highest and the lowest edge weight. Gupta and Peng [27] maintain a $(1 + \epsilon)$-approximation under edge insertions/deletions that runs in time $\mathcal{O}(\sqrt{m}\epsilon^{-2-\mathcal{O}(1/\epsilon)} \log N)$ time per update. Their result is based on several ingredients: (i) rerunning a static algorithm from time to time, (ii) a trimming routine that trims the graph to a smaller equivalent graph whenever possible and (iii) in the weighted case, a partition of the edges (according to their weights) into geometrically shrinking intervals.

Stubbs and Williams [49] present metatheorems for dynamic weighted matching. They reduce the dynamic maximum weight matching problem to the dynamic maximum cardinality matching problem in which the graph is unweighted. The authors prove that using this reduction, if there is an α-approximation for maximum cardinality matching with update time T, then there is also a $2\alpha(1 + \epsilon)$-approximation for maximum weight matching with update time $\mathcal{O}(\frac{T}{\epsilon^2} \log^2 C)$. Their basic idea is an extension/improvement of the algorithm of Crouch and Stubbs [15] who tackled the problem in the streaming model. We go into more detail in Section 4. None of these algorithms have been implemented.

3 Algorithm DynMWMRandom: Random Walks + Dynamic Programming

Random walks have already been successfully employed for dynamic maximum cardinality matching by Henzinger et al. [30]; however, in their current form they cannot be used for dynamic maximum weight matching. The main idea of our first dynamic algorithm for the weighted dynamic matching problem is to find random augmenting paths in the graph. To this end, our algorithm uses a random walk process in the graph and then uses dynamic programming on the computed random path $\mathcal{P}$ to compute the best possible matching on $\mathcal{P}$.[2] We continue by explaining these concepts in our context and how we handle edge insertions and deletions.

---

[2]One could rather speak solely of "random paths" instead of "random walks". We decided to keep the "walk" term, because the underlying process is that of a (cycle-free) random walk.

3.1 Random Walks For Augmenting Paths.
Our first dynamic algorithm is called DynMWMRan-
dom and constructs a random (cycle-free) path $\mathcal{P}$ as fol-
lows: initially, all nodes are set to be eligible. Whenever
an edge is added to the path $\mathcal{P}$, we set the endpoints to
be not eligible. The random walk starts at an arbitrary
eligible vertex u. If u is free, the random walker tries
to randomly choose an eligible neighbor w of u. If suc-
cessful, the corresponding edge e is added to the path $\mathcal{P}$
and the walk is continued at w (otherwise the random
walk stops and returns $\mathcal{P}$). Moreover, u is set to not el-
igible. On the other hand, if u is matched and mate(u)
is eligible, we add the edge $\{u, \text{mate}(u)\}$ to $\mathcal{P}$, set u
to not eligible and continue the walk at $v := \text{mate}(u)$.
There the algorithm tries to randomly choose an eligible
neighbor w of v. If successful, the random walker adds
the corresponding edge to the path, sets v to not eligible
and continues the walk at w. If at any point during the
algorithm execution, there is no adjacent eligible vertex,
then the algorithm stops and returns the path $\mathcal{P}$. Note
that, by construction, no vertex on the path is adjacent
to a matched edge that is not on the path. We stop the
algorithm after $\mathcal{O}(\frac{1}{\epsilon})$ steps for a given ϵ.

The algorithm tries to find a random eligible neigh-
bor by sampling a neighbor u uniformly at random. If
u is eligible, then we are done. If u is not eligible, we re-
peat the sampling step. We limit, however, the number
of unsuccessful repetitions to a constant. If our algo-
rithm did not find an eligible vertex, we stop and return
the current path $\mathcal{P}$. Overall, this ensures that picking a
random neighbor can be done in constant time. Hence,
the time to find a random path is $\mathcal{O}(\frac{1}{\epsilon})$. Here, we as-
sume that the eligible state of a node is stored in an
array of size n that is used for many successive random
walks. This array can be reset after the random walk
has been done in $\mathcal{O}(\frac{1}{\epsilon})$ by setting all nodes on the path
to be eligible again. The length of the random walk is a
natural parameter of the algorithm that we will inves-
tigate in the experimental evaluation.

After the path has been computed, we run a dy-
namic program on the path to compute the optimum
weighted matching on the path in time $\mathcal{O}(\frac{1}{\epsilon})$. If the
weight of the optimum matching on $\mathcal{P}$ is larger than
the weight of the current matching on the path, we re-
place the current matching on $\mathcal{P}$. Note that, due to
the way the path has been constructed, this yields a
feasible matching on the overall graph.

Dynamic Programming on Paths. It is well-
known that the weighted matching problem can be
solved to optimality on paths using dynamic program-
ming [38]. In order to be self-contained, we outline
briefly how this can be done. The description of Algo-
rithm 1 in Appendix A follows Maue and Sanders [38].

Given the path $\mathcal{P} = \langle e_1, \ldots, e_k \rangle$, the main idea of the
dynamic programming approach is to scan the edges in
the given order. Whenever it scans the next edge e_i,
there are two cases: either that edge is used in an op-
timum matching of the subproblem $\langle e_1, \ldots, e_i \rangle$ or not.
To figure this out, the algorithm takes the weight of the
current edge e_i and adds the weight of the optimum
subsolution for the path $\langle e_1, \ldots, e_{i-2} \rangle$. If this is larger
than the weight of the optimum subsolution for the path
$\langle e_1, \ldots, e_{i-1} \rangle$, then the edge e_i is used in an optimum
solution for $\langle e_1, \ldots, e_i \rangle$; otherwise it is not.

Optimizations. The random walk can be re-
peated multiple times, even if it a weight-augmenting
path found. This is because, even if the random walker
found an improvement, it can be possible that there
is another augmenting path left in the graph. In our
experiments, we use ℓ walks, where ℓ is a tuning param-
eter. However, repeating the algorithm too often can
be time-consuming. Hence, we optionally use a heuris-
tic to break early if we already performed a couple of
unsuccessful random walks: we stop searching for aug-
menting paths if we made β consecutive random walks
that were all unsuccessful. In our experiments, we use
$\beta = 5$; similar choices of parameters should work just
as well. We call this the *"stop early" heuristic*. We now
explain how we perform edge insertions and deletions
using the augmented random walks introduced above.

Edge Insertion. Our algorithm handles edge in-
sertions $\{u, v\}$ as follows: if both u and v are free, we
start a random walk at either u or v (randomly chosen)
and make sure that the new edge is always included in
the path $\mathcal{P}$. That means if we start at u, then we add
the edge $\{u, v\}$ to the path and start the random walk
described above at v (and vice versa). If one of the end-
points is matched, say u, we add the edges $\{u, \text{mate}(u)\}$
and $\{u, v\}$ to the path $\mathcal{P}$ and continue the walk at v.
If both endpoints are matched, we add $\{u, \text{mate}(u)\}$,
$\{u, v\}$ and $\{v, \text{mate}(v)\}$ to the path and continue the
random walk at mate(v). Note that it is necessary to in-
clude matched edges, since the optimum weight match-
ing on the path that we are constructing with the ran-
dom walk may match the newly inserted edge. Hence,
if we did not added matched edges incident to u and
v, we may end up with an infeasible matching. After
the full path is constructed, we run the dynamic pro-
gram as described above and augment the matching if
we found a better one.

Edge Deletion. In case of the deletion of an
edge $\{u, v\}$, we start two subsequent random walks as
described above, one at u and one at v. Note that if u or
v are matched, then the corresponding matched edges
will be part of the respective paths $\mathcal{P}$ that are created
by the two random walks.

3.2 Analysis. The algorithm described above, called DYNMWMRANDOM, can maintain a $(1 + \epsilon)$-approximation if the random walks are of appropriate length and repeated sufficiently often. To this end, we first show a relation between the existence of a weight-augmenting k-path and the quality of a matching. For the proofs, see Appendix B.

PROPOSITION 3.1. *Let $\mathcal{M}$ be a matching and $k \geq 1$ be the smallest number such that there is a weight-augmenting k-path with respect to $\mathcal{M}$. Then $w(\mathcal{M}) \geq \frac{k-1}{k} \cdot w(\mathcal{M}^*)$, where $\mathcal{M}^*$ is a matching with the maximum weight.*

THEOREM 3.1. *Algorithm DYNMWMRANDOM maintains a $(1 + \epsilon)$-approximate maximum weight matching with high probability if the length of each path is $\lceil 2/\epsilon + 3 \rceil$ and the walks are repeated $\lceil \Delta^{2/\epsilon+3} \log n \rceil$ times.*

4 Algorithm Family DynMWMLevel based on Results by Crouch, Stubbs and Williams

The algorithm by Stubbs and Williams [49] can use dynamic *unweighted* matching algorithms as a black box to obtain a weighted dynamic matching algorithm. We outline the details of the meta-algorithm here, as this is one of the algorithms that we implemented. Consider a dynamic graph that has integer weights in $[L, N]$. Moreover, let $C = N/L$. The basic idea of the algorithm, due to Crouch and Stubbs [15], is to maintain $\mathcal{O}(\epsilon^{-1} \log C)$ levels. Each of the levels only contains a subset of the overall set of edges. More precisely, the i-th level contains all edges of weight at least $(1 + \epsilon)^i$ for $i \in \{\lfloor \log_{1+\epsilon} L \rfloor, \lfloor \log_{1+\epsilon} L \rfloor + 1, \ldots, \lfloor \log_{1+\epsilon} N \rfloor\}$. The algorithm then maintains unweighted matchings on each level by employing a dynamic maximum cardinality matching algorithm for each level when an update occurs. The output matching is computed by including all edges from the matching in the highest level, and then by adding matching edges from lower levels in descending order, as long as they are not adjacent to an edge that has been already added. Stubbs and Williams [49] extend the result by Crouch and Stubbs [15] and show how the greedy merge of the matchings can be updated efficiently.

THEOREM 4.1. (STUBBS AND WILLIAMS [49]) *If the matchings on every level are α-approximate maximum cardinality matchings, then the output matching of the algorithm is a $2\alpha(1 + \epsilon)$-approximate maximum weight matching. Moreover, the algorithm can be implemented such that the update time is $\mathcal{O}(t(m, n)(\log_{1+\epsilon} N/L)^2)$, where $t(m, n)$ is the update time of the dynamic cardinality matching algorithm used.*

In our implementation, we use the recent dynamic maximum cardinality matching algorithms due to Henzinger et al. [30] that fared best in their experiments. The first algorithm is a random walk-based algorithm, the second algorithm is a (depth-bounded) blossom algorithm – leading to the dynamic counterparts DYNMWM-LEVELR and DYNMWMLEVELB, respectively. The latter also has the option to maintain optimum maximum cardinality matchings. Both are described next to be self-contained. We refer to [30] for more details.

Random Walk-based MCM. In the update routines of the algorithm (*insert* and *delete*), the algorithm uses a random walk that works as follows: The algorithms starts at a free vertex u and chooses a random neighbor w of u. If the chosen neighbor is free, the edge $\{u, w\}$ is matched and the random walk stops. If w is matched, then we unmatch the edge $\{w, \text{mate}(w)\}$ and match $\{u, w\}$ instead. Note that $u \neq \text{mate}(w)$ since u is free in the beginning and therefore $\text{mate}(u) = \bot$, but $\text{mate}(\text{mate}(w)) = w$ and $w \neq \bot$. Afterwards, the previous mate of w is free. Hence, the random walk is continued at this vertex. The random walk performs $\mathcal{O}(\frac{1}{\epsilon})$ steps. If unsuccessful, the algorithm can perform multiple repetitions to find an augmenting path. Moreover, the most successful version of the algorithm uses Δ-*settling*, in which the algorithm tries to settle visited vertices by scanning through their neighbors. The running time of the Δ-settling random walk is then $\mathcal{O}(\Delta/\epsilon)$.

The algorithm handles edge insertions $\{u, v\}$ as follows: if both endpoints are free, the edge is matched and the algorithm returns. The algorithm does nothing if both endpoints are matched. If only one of the endpoints is matched, say u, then the algorithm unmatches u, sets $w := \text{mate}(u)$, and matches $\{u, v\}$. Then a random walk as described above is started from w to find an augmenting path (of length at most $\mathcal{O}(1/\epsilon)$). If the random walk is unsuccessful, all changes are undone. Deleting a matched edge $\{u, v\}$ leaves the two endpoints u and v free. Then a random walk as described above is started from u if it is free and from v if v is free.

LEMMA 4.1. (HENZINGER ET AL. [30]) *The random walk based algorithm maintains a $(1 + \epsilon)$-approximate maximum matching if the length of the walk is at least $2/\epsilon - 1$ and the walks are repeated $\Delta^{2/\epsilon-1} \log n$ times.*

COROLLARY 4.1. *If the algorithm by Stubbs and Williams uses the random walk-based MCM algorithm on each level s.t. the length of the walks is $2/\epsilon - 1$ and the walks are repeated $\Delta^{2/\epsilon-1} \log n$ times, the resulting DYNMWMLEVELR algorithm is $2(1 + \epsilon)^2$-approximate with update time $\mathcal{O}(\epsilon^{-1} \Delta^{2/\epsilon-1} \log n (\log_{1+\epsilon} N/L)^2)$.*

Blossom-based MCM. Roughly speaking, the algorithm runs a modified BFS that is bounded in

Table 1: Basic properties of the benchmark set of static graphs obtained from [4, 16, 36].

graph	n	m	graph	n	m
144	144 649	1 074 393	eu-2005	862 664	16 138 468
3elt	4 720	13 722	fe_4elt2	11 143	32 818
4elt	15 606	45 878	fe_body	45 087	163 734
598a	110 971	741 934	fe_ocean	143 437	409 593
add20	2 395	7 462	fe_pwt	36 519	144 794
add32	4 960	9 462	fe_rotor	99 617	662 431
amazon-2008	735 323	3 523 472	fe_sphere	16 386	49 152
as-22july06	22 963	48 436	fe_tooth	78 136	452 591
as-skitter	554 930	5 797 663	finan512	74 752	261 120
auto	448 695	3 314 611	in-2004	1 382 908	13 591 473
bcsstk29	13 992	302 748	loc-brightkite_edges	56 739	212 945
bcsstk30	28 924	1 007 284	loc-gowalla_edges	196 591	950 327
bcsstk31	35 588	572 914	m14b	214 765	1 679 018
bcsstk32	44 609	985 046	memplus	17 758	54 196
bcsstk33	8 738	291 583	p2p-Gnutella04	6 405	29 215
brack2	62 631	366 559	PGPgiantcompo	10 680	24 316
citationCiteseer	268 495	1 156 647	rgg_n_2_15_s0	32 768	160 240
cnr-2000	325 557	2 738 969	soc-Slashdot0902	28 550	379 445
coAuthorsCiteseer	227 320	814 134	t60k	60 005	89 440
coAuthorsDBLP	299 067	977 676	uk	4 824	6 837
coPapersCiteseer	434 102	16 036 720	vibrobox	12 328	165 250
coPapersDBLP	540 486	15 245 729	wave	156 317	1 059 331
crack	10 240	30 380	web-Google	356 648	2 093 324
cs4	22 499	43 858	whitaker3	9 800	28 989
cti	16 840	48 232	wiki-Talk	232 314	1 458 806
data	2 851	15 093	wing	62 032	121 544
email-EuAll	16 805	60 260	wing_nodal	10 937	75 488
enron	69 244	254 449	wordassociation-2011	10 617	63 788

depth by $2/\epsilon - 1$ to find augmenting paths from a free node. This has a theoretically faster running time of $\mathcal{O}(\Delta^{2/\epsilon-1})$ per free node and still guarantees a $(1 + \epsilon)$ approximation. The algorithm handles insertions as follows. Let $\{u, v\}$ be the inserted edge. If u and v are free, then the edge is matched directly and the algorithm stops. Otherwise, an augmenting path search is started from u if u is free and from v if v is free. If both u and v are not free, then a breadth-first search from u is started to find a free node reachable via an alternating path. From this node an augmenting path search is started. If the algorithm does not handle this case, i.e., if both u and v are not free, it does nothing and is called *unsafe*. If the algorithm handles the this case and the augmenting path search is not depth-bounded, then it maintains a maximum matching. The algorithm handles deletions as follows: let $\{u, v\}$ be the deleted edge. After the deletion, an augmenting path search is started from any free endpoint u or v. The algorithm uses two optimizations: *lazy augmenting path search*, in which an augmenting path search from u and v is only started if at least x edges have been inserted or deleted since the last augmenting path search from u or v or if no augmenting path search has been started so

far. The second optimization limits the search depth of the augmenting path search to $2/\epsilon - 1$. This ensures that there is no augmenting path of length at most $2/\epsilon - 1$, thus yielding a deterministic $(1+\epsilon)$-approximate matching algorithm (if the algorithm is run with the *safe* option).

COROLLARY 4.2. *If the algorithm by Stubbs and Williams uses the blossom-based MCM algorithm on each level s.t. the depth is bounded by $2/\epsilon - 1$, then the resulting DYNMWMLEVELB algorithm is $2(1+\epsilon)^2$-approximate with update time $\mathcal{O}(\Delta^{2/\epsilon-1}(\log_{1+\epsilon} N/L)^2)$.*

5 Experiments

Implementation and Compute System. We implemented the algorithms described above in C++ compiled with g++-9.3.0 using the optimization flag -O3. For the dynamic MCM algorithms, we use the implementation by Henzinger et al. [30]. All codes are sequential. We intend to release our implementation after paper acceptance.

Our algorithms use the following dynamic graph data structure. For each node u, we maintain a vector L_u of adjacent nodes, and a hash table $\mathcal{H}_u$ that maps

a neighbor v of u to its position in L_u. This data structure allows for expected constant time insertion and deletion as well as a constant time operation to select a random neighbor of u. The deletion operation on $\{u, v\}$ is implemented as follows: get the position of v in L_u via a lookup in $\mathcal{H}_u(v)$. Swap the element in L_u with the last element w in the vector and update the position of w in $\mathcal{H}_u$. Finally, pop the last element (now v) from L_u and delete its entry from $\mathcal{H}_u$.

Our experiments are conducted on one core of a machine with a AMD EPYC 7702P 64-core CPU (2 GHz) and 1 TB of RAM. In order to be able to compare our results with optimum weight matchings, we use the LEMON library [17], which provides algorithms for static weighted maximum matching.

Methodology. By default we perform ten repetitions per instance. We measure the total time taken to compute all edge insertions and deletions and generally use the *geometric mean* when averaging over different instances in order to give every instance a comparable influence on the final result. In order to compare different algorithms, we use *performance profiles* [18]. These plots relate the matching weight of all algorithms to the optimum matching weight. More precisely, the y-axis shows $\#\{objective \geq \tau \times OPT\}/\#\text{instances}$, where *objective* corresponds to the result of an algorithm on an instance and OPT refers to the optimum result. The parameter $\tau \leq 1$ in this equation is plotted on the x-axis. For each algorithm, this yields a non-decreasing, piecewise constant function.

Instances. We evaluate our algorithms on graphs collected from various resources [4, 16, 29, 35, 36, 45], which are also used in related studies [30]. Table 1 summarizes the main properties of the benchmark set. Our benchmark set includes a number of graphs from numeric simulations as well as complex networks. These include static graphs as well as real dynamic graphs. As our algorithms only handle undirected simple graphs, we ignore possible edge directions in the input graphs, and we remove self-loops and parallel edges. We perform *two different types* of experiments. First, we use the algorithms using insertions only, i.e., we start with an empty graph and insert all edges of the static graph in a random order. We do this with all graphs from Table 1. Second, we use real dynamic instances from Table 2. Most of these instances, however, only feature insertions (exceptions: `dewiki` and `wiki-simple-en`). Hence, we perform additional experiments with fully-dynamic graphs from these inputs, by undoing x percent of the update operations performed last. As the instances are unweighted, we generated weights for edges uniformly at random in $[1, 100]$.

Table 2: Basic properties of the benchmark set of dynamic graphs with number of update operations $\mathcal{U}$. Most of the graphs only feature insertions. The only two exceptions are marked with a *. All of these graphs have been obtained from the KONECT graph database [45].

graph	n	$\mathcal{U}$
amazon-ratings	2 146 058	5 838 041
citeulike_ui	731 770	2 411 819
dewiki*	2 166 670	86 337 879
dnc-temporalGraph	2 030	39 264
facebook-wosn-wall	46 953	876 993
flickr-growth	2 302 926	33 140 017
haggle	275	28 244
lastfm_band	174 078	19 150 868
lkml-reply	63 400	1 096 440
movielens10m	69 879	10 000 054
munmun_digg	30 399	87 627
proper_loans	89 270	3 394 979
sociopatterns-infections	411	17 298
stackexchange-overflow	545 197	1 301 942
topology	34 762	171 403
wikipedia-growth	1 870 710	39 953 145
wiki_simple_en*	100 313	1 627 472
youtube-u-growth	3 223 590	9 375 374

5.1 DynMWMRandom.

First, we look at DYNMWMRANDOM (in various configurations) on insertions-only instances. We start with the effect of the number of walks done after each update and the impact of the "stop early" heuristic. In Fig 1 (left), we look at different numbers of walks when (not) using the "stop early" heuristic (if used, ϵ is set to 10^{-3}). Geometric mean values for running time and achieved matching weight are in Table 3. As expected, increasing the number of walks increases the weight of the computed matchings (with and without "stop early" heuristic). With the "stop early" heuristic (in our experiments $\beta = 5$), increasing the number of walks more than ten does not significantly improve the solutions. When disabling the "stop early" heuristic, solutions improve again, even for the same number of 20 walks. The strongest configuration in this setting uses 100 walks with the "stop early" heuristic being enabled. The configuration is as close as 4% to the optimum on more than 95% of the instances. Note that the number of walks done here is much less than what Lemma 4.1 suggests to achieve a $(1 + \epsilon)$-approximation. Moreover, using more walks as well as not using the "stop early" heuristic also comes at the cost of additional running time. For example, for $\epsilon = 0.001$ and 20 walks, using the "stop early" heuristic saves roughly a factor of three in running time. We conclude that for a fixed ϵ, the
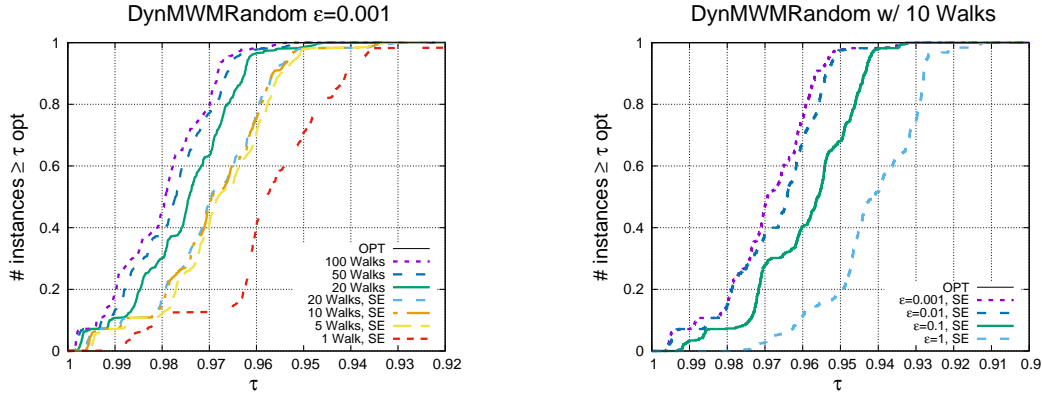
Figure 1: Left: Number of random walks and random walks with "stop early" heuristic (SE) for fixed epsilon. Right: Various values of ϵ for fixed number of walks.

Table 3: Top: Geometric mean running times and weight of the final matching for fixed $\epsilon = 0.001$ and varying number of walks. Bottom: Geometric mean running times and weight of final matching for fixed number of walks (10) and varying ϵ.

# walks	geo. mean t [s]	geo. mean $\omega(\mathcal{M})$
with SEHeuristic		
1	4.0	1572285.92
5	16.4	+1.15%
10	20.5	+1.28%
20	21.8	+1.31%
without SEHeuristic		
20	56.4	+1.91%
50	128.6	+2.24%
100	241.6	+2.44%

ϵ	geo. mean t [s]	geo. mean $\omega(\mathcal{M})$
with SEHeuristic		
1	2.4	1547129.56
10^{-1}	5.9	+1.84%
10^{-2}	13.7	+2.71%
10^{-3}	20.5	+2.93%

number of walks and the "stop early" heuristic yield a favorable speed quality trade-off.

Next, we fix the use of the "stop early" heuristic and the number of walks per update to ten. Figure 1 (right) shows the performance compared to the optimum solution value and Table 3 gives geometric mean values for running time and matching weight again. When using $\epsilon = 1$, the algorithm computes in all cases at least a matching that has 91.2% of the optimum weighted matching weight. As soon as we decrease ϵ, solutions improve noticeably. For the case $\epsilon = 10^{-1}$, solutions improve by 1.84% on average. The algorithm always computes a matching which has at least 93.3% of the optimum weight matching. However, the parameter ϵ

seems to be exhaustive in the sense that decreasing it from 10^{-2} to 10^{-3} does not significantly improve solutions anymore. Also note that the running time grows much slowlier than $1/\epsilon$. This is due to the "stop early" heuristic. Overall, we conclude here that both the number of walks and the parameter ϵ are very important to maintain heavy matchings.

On the other hand, it is important to mention that in the unweighted case the quality of the algorithms based on random walks is not very sensitive to both parameters, ϵ and the number of walks [30]. That is why Henzinger et al. [30] use only one random walk and vary only ϵ in their final MCM algorithm. The different behavior comes from two observations: first, in the unweighted case many subpaths of an augmenting path are often interchangeable. In the weighted case, this may not be the case and hence it can be hard to find some of the remaining augmenting paths in later stages of the algorithms. Moreover, the algorithms differ in the way the random walkers work. Our weighted algorithm builds a path and then solves a dynamic program on it (hence, the length of a random walker is limited by n). In the unweighted case, a random walker does changes to the graph on the fly and undoes changes in the end if it was unsuccessful. Hence, it can in principle run much longer than n steps.

In terms of running time, we can give a rough estimate of a speedup that would be obtained when running the optimal algorithm after each update. Note that the comparison is somewhat unfair since we compare an optimal algorithm with a heuristic algorithm and since we did not run the optimum algorithm after each update step due to excessive time necessary to run this experiment. On the largest instance of the set, eu-2005, the time per update of the slowest dynamic algorithm in these experiments ($\epsilon = 0.001$, 100 walks, no stop heuristic) is 0.00227s on average. On the other hand, the optimum algorithm needs 79.8998s to compute the solution on the final graph. This is more than four or-
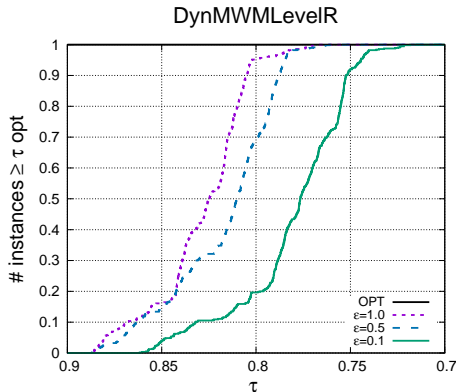
Figure 2: Performance profile of the meta-algorithm DYNMWMLEVELR instantiated with random-walk based algorithms as MCM algorithms for different values of ϵ.

ders of maginitude difference. On the other hand, the fastest algorithms in this experiment are at least one additional order of magnitude faster.

5.2 DynMWMLevel. We now evaluate the performance of our implementation DYNMWMLEVEL of the meta-algorithm by Crouch, Stubbs, and Williams. First, we use $\alpha = (1+\epsilon)$, so the algorithms under investigation have approximation guarantee of $2(1 + \epsilon)^2$. Note that decreasing ϵ increases both, the number of levels in the algorithm necessary as well as the work performed by the unweighted matching algorithms on each level of the hierarchy. The results for the meta-algorithm are roughly the same when using the unweighted random walk algorithm compared to using the unweighted blossom-based algorithm for the same value of ϵ on each level. For example, the difference between DYNMWM-LEVELR and DYNMWMLEVELB is smaller than 0.5% on average for $\epsilon = 0.1$ (with random walks having the slight advantage). Moreover, both instantiations have roughly the same running time on average (less than 1.6% difference on average). Hence, we focus our presentation on DYNMWMLEVELR. Figure 2 shows three different configurations of the algorithm using different values of $\epsilon \in \{1, 0.5, 0.1\}$. We do not run smaller values of ϵ as the algorithm creates a lot of levels for small ϵ and thus needs a large amount of memory. This is due to the large amount of levels created for small values of ϵ. To give a rough estimate, on the smallest graph from the collection, add20, DYNMWMLEVELR with $\epsilon = 0.1$ needs more than a factor 40 more memory than DYN-MWMRANDOM.

First, note that the results are significantly worse than the results achieved by DYNMWMRANDOM: DYNMWMLEVELR computes at least a weighted matching that has 71.8% of the optimum weight matching for $\epsilon = 0.1$. Note that this is very well within the

factor 2 that theory suggests. However, for larger values of ϵ, the algorithm computes even better results. For $\epsilon = 0.5$ the algorithm computes a weighted matching of at least 76.1% of the optimum weight matching, and for $\epsilon = 1$, the algorithm computes 76.8% of the optimum weight matching. We believe that this somewhat surprising behavior is due to two reasons. First, if one only increases the work done by the MCM algorithms on each level, then the cardinality of the matchings on each level increases as expected. However, the weight of the computed matching on each level does not necessarily increase. Hence, the level-based meta-algorithm has a too strong preference on the cardinality as it ignores the weights on each level. Moreover, when increasing the number of levels, the priority of heavy edges increases, as there are only very few of them at the highest levels in the hierarchy. Hence, the algorithm becomes closer to a greedy algorithm that picks heavy edges first.

In terms of running time, DYNMWMLEVELR uses 1.53s, 2.66s, and 14.61s on average for $\epsilon = 1$, 0.5, and 0.1, respectively. Moreover, the best and fastest configuration in this category uses $\epsilon = 1$, computes 11% smaller matchings than the augmented random walk for $\epsilon = 1$ (recall that the number of steps of the walk is $2/\epsilon + 3$)) and the geometric mean running time of the random walk is 0.61s. We conclude here that DYNMWMRANDOM is better and faster than DYNMWMLEVEL. Moreover, augmented random walks need significantly less memory.

5.3 Real-World Dynamic Instances. We now switch to the real-world dynamic instances. Note that most of these instances are insertion-only (with two exceptions); however, the insertions are ordered in the way they appear in the real world. Hence, we perform additional experiments with fully-dynamic graphs from these inputs, by undoing x percent of the update operations performed last (call them $\mathcal{U}_x$). More precisely, we perform the operations in $\mathcal{U}_x$ in reverse order. If an edge operation was an insertion in $\mathcal{U}_x$, we perform a delete operation and if it was a delete operation, we insert it. However, when reinserting an edge, we use a new random weight. In our experiments, we perform 0%, 10% and 25% undo operations. As before, we compute the update on the graph after each removal/insertion. Table 4 summarizes the results of the experiment and Figure 3 compares the final quality of the dynamic algorithms with the optimum on the final graph after all updates have been performed. For our experiments with DYNMWMRANDOM, we use the configurations $\epsilon = 10^{-3}$ with 10 walks, $\epsilon = 1$ with 10 walks, and $\epsilon = 1$ with 1 walk. In any case, for these algorithms the "stop early" heuristic is enabled. For the
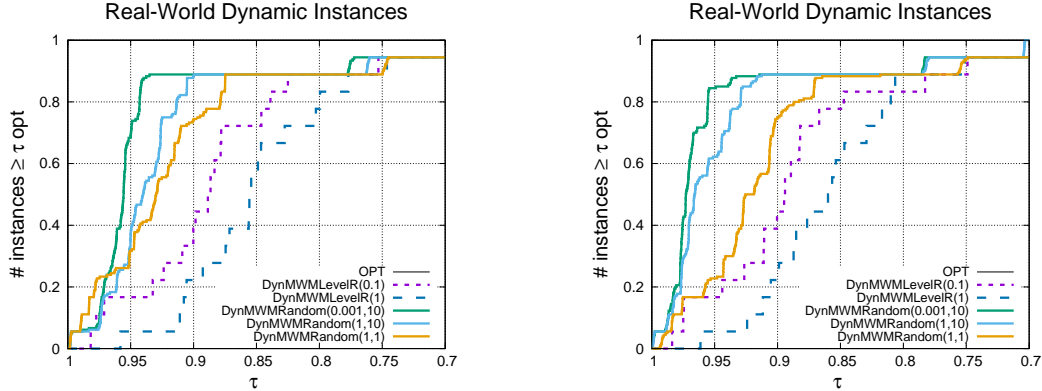
Figure 3: Comparison of different algorithm configuration on real-world dynamic instances without (left) and with (right) 25% undo operations.

Table 4: Geometric mean results for real-world dynamic instances with and without undo operations depending on ϵ and walks per update (w). Smaller is better.

Algorithm	$\overline{t[s]}$	$\overline{OPT/\omega(\mathcal{M})}$
without undo ops		
Random($\epsilon = 10^{-3}$, w=10)	11.2	7.3%
Random($\epsilon = 1$, w=10)	3.9	9.0%
Random($\epsilon = 1$, w=1)	1.0	10.0%
LevelR($\epsilon = 10^{-1}$)	17.3	14.0%
LevelR($\epsilon = 1$)	1.7	18.7%
with 10% undo ops		
Random($\epsilon = 10^{-3}$, w=10)	14.3	6.2%
Random($\epsilon = 1$, w=10)	5.0	7.2%
Random($\epsilon = 1$, w=1)	1.2	10.7%
LevelR($\epsilon = 10^{-1}$)	18.6	13.5%
LevelR($\epsilon = 1$)	1.9	17.9%
with 25% undo ops		
Random($\epsilon = 10^{-3}$, w=10)	18.1	6.0%
Random($\epsilon = 1$, w=10)	6.3	7.0%
Random($\epsilon = 1$, w=1)	1.5	10.9%
LevelR($\epsilon = 10^{-1}$)	20.5	13.8%
LevelR($\epsilon = 1$)	2.1	17.5%

DynMWMLevelR algorithm, we use $\epsilon = 10^{-1}$ and $\epsilon = 1$. Overall, the results confirm the results of the previous section. For example, DynMWMRandom ($\epsilon = 0.001$, 10 walks per update), computes matchings with 93.5% weight of the optimum weight matching on more than 89% of the instances. The fastest configuration DynMWMRandom ($\epsilon = 1$, 1 walk per update), is also already very good. It computes 90% of the optimum weight matching on 89% of the instances. On the other hand, DynMWMLevelR computes 82.5% and 77.7% of the optimum weight matching on 88.9% of the instances for $\epsilon = 0.1$ and $\epsilon = 1$, respectively.

Moreover, the experiments indicate that DynMWMLevelR is always outperformed in terms of quality and speed by a configuration of DynMWMRandom. This is true with and without running undo operations. For example, without running undo operations, DynMWMLevelR ($\epsilon = 10^{-1}$) computes solutions that are 14% smaller than the optimum matching weight on average while taking 17.3s running time. On the other hand, already DynMWMRandom ($\epsilon = 1$, 1 walk per update) computes weighted matchings that are 10% worse than the optimum solution while needing only 1s running time on average. The same comparison can be done when undo operations are performed, i.e., DynMWMLevelR ($\epsilon = 10^{-1}$) is always outperformed by DynMWMRandom ($\epsilon = 1$, 1 walk).

When considering undo operations, the quality of all algorithms is stable, i.e., there is no reduction in quality observable over the case without undo operations. Note that in contrast the quality in terms of distance to OPT often improves when undo operations are performed. This is due to the fact that the algorithms perform additional work when the undo operations are called. Hence, the results may find additional augmenting paths that had not been found when the algorithm was at the original state of the graph.

6 Conclusions and Future Work

Our experiments show that fully-dynamic approximation algorithms for the MWM problem are viable alternatives compared to an exact approach. Of the algorithms we implemented, DynMWMRandom fares best overall and should be the method of choice whenever exact solutions are unnecessary. After all, the solutions are often close to the optimum. Future work could focus on finding augmenting paths of length 3 efficiently, e.g. as suggested in [42] for the unweighted case.

References

[1] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. *CoRR*, abs/2010.05846, 2020. URL: `https://arxiv.org/abs/2010.05846`, arXiv:2010.05846.

[2] Abhash Anand, Surender Baswana, Manoj Gupta, and Sandeep Sen. Maintaining approximate maximum weighted matching in fully dynamic graphs. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, LIPIcs, pages 257–266. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. `doi:10.4230/LIPIcs.FSTTCS.2012.257`.

[3] Moab Arar, Shiri Chechik, Sarel Cohen, Cliff Stein, and David Wajc. Dynamic matching: Reducing integral algorithms to approximately-maximal fractional algorithms. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.ICALP.2018.7`.

[4] D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2018. `doi:10.1007/978-1-4939-7131-2\_23`.

[5] Leonid Barenboim and Tzalik Maimon. Fully dynamic graph algorithms inspired by distributed computing: Deterministic maximal matching and edge coloring in sublinear update-time. *ACM J. Exp. Algorithmics*, 24(1):1.14:1–1.14:24, 2019. `doi:10.1145/3338529`.

[6] Surender Baswana, Manoj Gupta, and Sandeep Sen. Fully dynamic maximal matching in $O(\log n)$ update time. *SIAM J. Comput.*, 44(1):88–113, 2015. `doi:10.1137/16M1106158`.

[7] Claude Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences*, 43(9):842–844, 1957. URL: `http://www.pnas.org/content/43/9/842`, arXiv:http://www.pnas.org/content/43/9/842.full.pdf, `doi:10.1073/pnas.43.9.842`.

[8] Aaron Bernstein and Cliff Stein. Faster fully dynamic matchings with small approximation ratios. In *Proceedings of the 27th Symposium on Discrete Algorithms SODA*, pages 692–711. SIAM, 2016. `doi:10.1137/1.9781611974331.ch50`.

[9] Sayan Bhattacharya, Deeparnab Chakrabarty, and Monika Henzinger. Deterministic fully dynamic approximate vertex cover and fractional matching in O(1) amortized update time. In *19th Intl. Conf. on Integer Programming and Combinatorial Optimization IPCO*, volume 10328 of *LNCS*, pages 86–98. Springer, 2017. `doi:10.1007/978-3-319-59250-3\_8`.

[10] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. *SIAM J. Comput.*, 47(3):859–887, 2018. `doi:10.1137/140998925`.

[11] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. New deterministic approximation algorithms for fully dynamic matching. In *Proceedings of the 48th Annual Symposium on Theory of Computing*, pages 398–411. ACM, 2016. `doi:10.1145/2897518.2897568`.

[12] Sayan Bhattacharya, Monika Henzinger, and Danupon Nanongkai. Fully dynamic approximate maximum matching and minimum vertex cover in $O(\log^3 n)$ worst case update time. In Philip N. Klein, editor, *Proc. of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms SODA*, pages 470–489. SIAM, 2017. `doi:10.1137/1.9781611974782.30`.

[13] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In *Euro-Par 2013*, volume 8097 of *LNCS*, pages 659–670. Springer, 2013. `doi:10.1007/978-3-642-40047-6\_66`.

[14] Moses Charikar and Shay Solomon. Fully dynamic almost-maximal matching: Breaking the polynomial worst-case time barrier. In *45th International Colloquium on Automata, Languages, and Programming ICALP*, volume 107 of *LIPIcs*, pages 33:1–33:14, 2018. `doi:10.4230/LIPIcs.ICALP.2018.33`.

[15] Michael Crouch and Daniel S. Stubbs. Improved streaming algorithms for weighted matching, via unweighted matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014*, volume 28 of *LIPIcs*, pages 96–104, 2014. `doi:10.4230/LIPIcs.APPROX-RANDOM.2014.96`.

[16] T. Davis. The University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices`, 2008. URL: `http://www.cise.ufl.edu/research/sparse/matrices/`.

[17] Balázs Dezső, Alpár Jüttner, and Péter Kovács. LEMON – an Open Source C++ Graph Template Library. *Electronic Notes in Theoretical Computer Science*, 264(5):23 – 45, 2011. URL: `http://www.sciencedirect.com/science/article/pii/S1571066111000740`, `doi:https://doi.org/10.1016/j.entcs.2011.06.003`.

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

[19] D. Drake and S. Hougardy. A Simple Approximation Algorithm for the Weighted Matching Problem. *Information Processing Letters*, 85(4):211–213, 2003. `doi:10.1016/S0020-0190(02)00393-9`.

[20] Doratha E. Drake and Stefan Hougardy. Linear time local improvements for weighted matchings in graphs. In *Experimental and Efficient Algorithms (WEA)*, volume 2647 of *Lecture Notes in Computer Science*, pages 107–119, Berlin, Heidelberg, 2003. Springer. `doi:10.1007/3-540-44867-5\_9`.

[21] Andre Droschinsky, Petra Mutzel, and Erik Thordsen. Shrinking trees not blossoms: A recursive maximum

matching approach. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2020*, pages 146–160. SIAM, 2020. `doi:10.1137/1.9781611976007.12`.

[22] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

[23] Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for general graph-matching problems. *J. ACM*, 38(4):815–853, 1991. `doi:10.1145/115234.115366`.

[24] Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, 1986. `doi:10.1145/6462.6502`.

[25] Fabrizio Grandoni, Stefano Leonardi, Piotr Sankowski, Chris Schwiegelshohn, and Shay Solomon. $(1 + \epsilon)$-approximate incremental matching in constant deterministic amortized time. In *Proceedings of the 20th Symposium on Discrete Algorithms*, pages 1886–1898. SIAM, 2019. `doi:10.1137/1.9781611975482.114`.

[26] Manoj Gupta and Richard Peng. Fully dynamic (1+ e)-approximate matchings. In *54th Symposium on Foundations of Computer Science, FOCS*, pages 548–557. IEEE Computer Society, 2013. `doi:10.1109/FOCS.2013.65`.

[27] Manoj Gupta and Richard Peng. Fully Dynamic $(1 + \epsilon)$-Approximate Matchings. In *54th Annual IEEE Symposium on Foundations of Computer Science*, pages 548–557. IEEE Computer Society, 2013. `doi:10.1109/FOCS.2013.65`.

[28] Mahantesh Halappanavar. *Algorithms for Vertex-Weighted Matching in Graphs*. PhD thesis, Old Dominion University, May 2009.

[29] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms. *CoRR*, abs/2102.11169, 2021. URL: `https://dyngraphlab.github.io/`, `arXiv:2102.11169`.

[30] Monika Henzinger, Shahbaz Khan, Richard Paul, and Christian Schulz. Dynamic Matching Algorithms in Practice. In *28th Annual European Symposium on Algorithms*, volume 173 of *LIPIcs*, pages 58:1–58:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.ESA.2020.58`.

[31] J. E. Hopcroft and R. M. Karp. A $n^{5/2}$ algorithm for maximum matchings in bipartite. In *12th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 122–125, 1971. `doi:10.1109/SWAT.1971.1`.

[32] Zoran Ivkovic and Errol L. Lloyd. Fully dynamic maintenance of vertex cover. In *19th International Workshop Graph-Theoretic Concepts in Computer Science*, volume 790 of *LNCS*, pages 99–111, 1993. `doi:10.1007/3-540-57899-4\_44`.

[33] Manas Jyoti Kashyop and N.S. Narayanaswamy. Lazy or eager dynamic matching may not be fast. *Information Processing Letters*, 162:105982, 2020. URL: `http://www.sciencedirect.com/science/article/pii/S0020019020300697`, `doi:https://doi.org/10.1016/j.ipl.2020.105982`.

[34] Viatcheslav Korenwein, André Nichterlein, Rolf Niedermeier, and Philipp Zschoche. Data Reduction for Maximum Matching on Real-World Graphs: Theory and Experiments. In *26th European Symposium on Algorithms ESA*, volume 112 of *LIPIcs*, pages 53:1–53:13, 2018. `doi:10.4230/LIPIcs.ESA.2018.53`.

[35] Jérôme Kunegis. KONECT: the koblenz network collection. In *22nd World Wide Web Conference, WWW '13*, pages 1343–1350, 2013. `doi:10.1145/2487788.2488173`.

[36] J. Lescovec. Stanford Network Analysis Package (SNAP). `http://snap.stanford.edu/index.html`.

[37] 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`.

[38] J. Maue and P. Sanders. Engineering Algorithms for Approximate Weighted Matching. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *LNCS*, pages 242–255. Springer, 2007. URL: `http://dx.doi.org/10.1007/978-3-540-72845-0_19`.

[39] Aranyak Mehta, Amin Saberi, Umesh V. Vazirani, and Vijay V. Vazirani. Adwords and generalized on-line matching. In *46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 264–273. IEEE Computer Society, 2005. `doi:10.1109/SFCS.2005.12`.

[40] Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In *21st Symposium on Foundations of Computer Science*, pages 17–27. IEEE Computer Society, 1980. `doi:10.1109/SFCS.1980.12`.

[41] Marcin Mucha and Piotr Sankowski. Maximum matchings in planar graphs via gaussian elimination. *Algorithmica*, 45(1):3–20, 2006. `doi:10.1007/s00453-005-1187-5`.

[42] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. *ACM Trans. Algorithms*, 12(1):7:1–7:15, 2016. `doi:10.1145/2700206`.

[43] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. In *STOC*, pages 457–464, 2010. `doi:10.1145/1806689.1806753`.

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

[45] Julia Preusse, Jérôme Kunegis, Matthias Thimm, Thomas Gottron, and Steffen Staab. Structural dynamics of knowledge networks. In *Proc. Int. Conf. on Weblogs and Social Media*, 2013. URL: `http://www.aaai.org/ocs/index.php/ICWSM/ICWSM13/paper/view/6076`.

[46] Piotr Sankowski. Faster dynamic matchings and vertex connectivity. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 118–126. SIAM, 2007. URL: `http://dl.acm.org/`

citation.cfm?id=1283383.1283397.

[47] Piotr Sankowski. Maximum weight bipartite matching in matrix multiplication time. *Theor. Comput. Sci.*, 410(44):4480–4488, 2009. `doi:10.1016/j.tcs.2009.07.028`.

[48] Shay Solomon. Fully dynamic maximal matching in constant update time. In *57th Symposium on Foundations of Computer Science FOCS*, pages 325–334, 2016. `doi:10.1109/FOCS.2016.43`.

[49] Daniel Stubbs and Virginia Vassilevska Williams. Metatheorems for dynamic weighted matching. In *8th Innovations in Theoretical Computer Science Conference*, volume 67 of *LIPIcs*, pages 58:1–58:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.ITCS.2017.58`.

[50] Jan van den Brand, Danupon Nanongkai, and Thatchaphol Saranurak. Dynamic matrix inverse: Improved algorithms and matching conditional lower bounds. In *60th IEEE Annual Symposium on Foundations of Computer Science*, pages 456–480. IEEE Computer Society, 2019. `doi:10.1109/FOCS.2019.00036`.

A Pseudocode of Dynamic Programming for MWM on Paths

Algorithm 1 MWM($\mathcal{P} = \langle e_1, \ldots, e_k \rangle$) [38]

1: $W[0] := 0; W[1] := \omega(e_1)$
2: $M[0] := \emptyset; M[1] := \{e_1\}$
3: **for** $i := 2$ **to** k **do**
4:      **if** $\omega(e_i) + W[i-2] > W[i-1]$ **then**
5:          $W[i] := \omega(e_i) + W[i-2]$
6:          $M[i] := M[i-2] \cup \{e_i\}$
7:      **else**
8:          $W[i] := W[i-1]$
9:          $M[i] := M[i-1]$
10: **return** $M[k]$

B Omitted Proofs

B.1 Proof of Proposition 3.1

A *weight-augmenting path* $\mathcal{P}$ with respect to a matching $\mathcal{M}$ is an alternating path with respect to $\mathcal{M}$, and the matching $\mathcal{M} \oplus \mathcal{P}$ has a larger weight than $\mathcal{M}$; in other words $w(\mathcal{M} \oplus \mathcal{P}) > w(\mathcal{M})$. A weight-augmenting path with k edges outside $\mathcal{M}$ is called *weight-augmenting k-path*. A weight-augmenting k-path thus can have $2k-1, 2k$, or $2k+1$ edges, whereas in the unweighted case an augmenting path with k edges outside of a given matching should have exactly $2k-1$ edges. The following proposition writes the approximation guarantee of a matching in terms of weight-augmenting k-paths.

PROPOSITION B.1. *Let $\mathcal{M}$ be a matching and $k \geq 1$ be the smallest number such that there is a weight-augmenting k-path with respect to $\mathcal{M}$. Then $w(\mathcal{M}) \geq \frac{k-1}{k} \cdot w(\mathcal{M}^*)$, where $\mathcal{M}^*$ is a matching with the maximum weight.*

Before proving the proposition, we note that its result is known. Drake and Hougardy [20] show a version of the proposition for weight-augmenting 3-paths. Pothen et al. [44, Lemma 3.1] generalize the proof given by Drake and Hougardy and show the result given in Proposition B.1. Our proof is somewhat different from the previous one, is closer to the way we make us of the result in the random walk based algorithm, and is therefore given for completeness.

Proof. [Proposition B.1] Consider the symmetric difference $\mathcal{M} \oplus \mathcal{M}^*$, which can have four different type of components displayed in Figure 4. In the figure, the edges labeled with m_i belong to $\mathcal{M}$ and are shown with the solid lines. The edges labeled with o_j belong to $\mathcal{M}^*$ and are shown with the dotted lines. There are other edges of $\mathcal{M}$ and $\mathcal{M}^*$ which are not shown. In the leftmost component, there is one more edge in $\mathcal{M}$ than in $\mathcal{M}^*$; in the second and third one, there are equal number of edges in $\mathcal{M}$ and $\mathcal{M}^*$; in the rightmost one, there is one more edge in $\mathcal{M}^*$ than in $\mathcal{M}$. In any of these components, $\ell > k$, if there is to be any weight-augmenting paths.

We now show that in a component C with at least k edges from O, we have $w(\mathcal{M} \cap C) \geq \frac{k-1}{k} w(\mathcal{M}^* \cap C)$. This will show that in all components of $\mathcal{M} \oplus \mathcal{M}^*$, we have the same guarantee of $\frac{k-1}{k}$ and hence the result.

Let us start with the leftmost component of Figure 4. We will use m_i and o_j for the weight of the corresponding edges. We have

$$m_1 + m_2 \geq o_1$$

otherwise, there will be a weight-augmenting 1-path. We can continue with

$$m_1 + m_2 + m_3 \geq o_1 + o_2$$

with the same reasoning until

$$m_1 + m_2 + \cdots + m_k \geq o_1 + o_2 + \cdots + o_{k-1} \,,$$

where we have k edges of $\mathcal{M}$ on the left-hand side of the equation and $k-1$ edges of $\mathcal{M}^*$ on the right-hand side of the equation. At this point, we have included m_1 and m_2 in $k-1$ inequalities, m_3 in $k-2$ inequalities, and in general m_j for $3 \leq j \leq k$ in $k-j+1$ inequalities. Then we keep the same number of edges for another $\ell - k$ steps and obtain the inequalities

$$m_2 + m_3 + \cdots + m_{k+1} \geq o_2 + o_3 + \cdots + o_k \,,$$
$$m_3 + m_4 + \cdots + m_{k+2} \geq o_3 + o_4 + \cdots + o_{k+1} \,,$$
$$\vdots$$
$$m_{\ell-k+1} + \cdots + m_\ell \geq o_{\ell-k+1} + \cdots + o_{\ell-1} \,,$$

as none of these $(k-1)$-paths are weight-augmenting. In these inequalities, m_j for $2 \leq j \leq \ell-k+1$ appears as the first term once, and it is included in the equalities starting with m_i for all $j-k+1 \leq i < j$. Up to this point, m_j for $2 \leq j \leq \ell-k+1$ is included in k inequalities. Once we have included the last edge m_ℓ of $\mathcal{M}$ in an inequality, we continue with writing down new inequalities by removing one edge at a time from the left and the right of the last inequality, starting from the edge with the smallest index:

$$m_{\ell-k+2} + \cdots + m_\ell \geq o_{\ell-k+2} + \cdots + o_{\ell-1}$$
$$m_{\ell-k+3} + \cdots + m_\ell \geq o_{\ell-k+3} + \cdots + o_{\ell-1}$$
$$\vdots$$
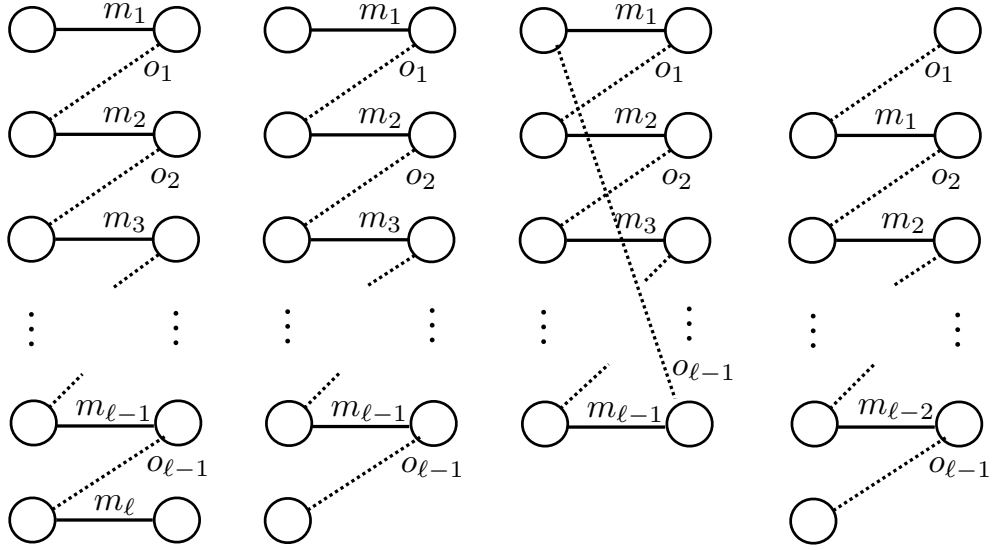$$m_{\ell-1} + m_\ell \geq o_{\ell-1} \,,$$

Figure 4: Four different type of components of $\mathcal{M} \oplus \mathcal{M}^*$. The edges labeled with m_i belong to $\mathcal{M}$ and are shown with the solid lines. The edges labeled with o_j belong to $\mathcal{M}^*$ and are shown with the dotted lines. There are other edges of $\mathcal{M}$ and $\mathcal{M}^*$ which are not shown. If k is the smallest number for which is a weight-augmenting k-path, then $\ell > k$.

for otherwise there would be a weight-augmenting j-path with a $j < k$. We see that the last k edges of $\mathcal{M}$ are symmetrical to the first k edges of $\mathcal{M}$, with respect to the inequalities we include. We have $m_\ell + m_{\ell-1} \geq o_{\ell-1}$, as otherwise there would be a weight-augmenting 1-path; then we have $m_\ell + m_{\ell-1} + m_{\ell-2} \geq o_{\ell-2} + o_{\ell-1}$ so on so forth. We also observe that each o_j appears in $k-1$ inequalities. The first inequality containing o_j has m_{j+1} as the last term, and the last inequality containing o_j has m_j as the first term, as to include o_j in $\mathcal{M}$, one has to remove m_j and m_{j+1} from $\mathcal{M}$. All together there are $k-1$ inequalities containing m_j and m_{j+1} together.

We collect the observations made so far below:

- m_1 appears in $(k-1)$ inequalities, as the first one is $m_1 + m_2 \geq o_1$, the last one is $m_1 + m_2 + \cdots + m_k \geq o_1 + o_2 + \cdots + o_{k-1}$, and each time we add one more m_i.

- m_ℓ appears in $(k-1)$ inequalities, as the first one is $m_{\ell-k+1} + \cdots + m_\ell \geq o_{\ell-k+1} + \cdots + o_{\ell-1}$; the last one is $m_{\ell-1} + m_\ell \geq o_{\ell-1}$, and each time we remove one m_i;

- all other m_i appear in k inequalities. For $i \leq k$, m_i appears $k - i + 1$ times for inequalities whose first term is m_1, then $i - 1$ more times until m_{i+1} becomes the first term. For $i > k$, m_i appears in all inequalities with m_j as the first term where $i - k + 1 \leq j \leq i$;

- each o_j appears in $k - 1$ inequalities.

Under the light of these observations, by adding the listed equalities side-by-side we obtain

$$(k-1)m_1 + (k-1)m_\ell + \sum_{i=2}^{\ell-1} km_i \geq \sum_{j=1}^{\ell-1}(k-1)o_j \ ,$$

therefore

$$k\sum_{i=1}^{\ell} m_i \geq (k-1)\sum_{j=1}^{\ell-1} o_j \ ,$$

hence

$$w(\mathcal{M} \cap C) \geq \frac{k-1}{k} w(\mathcal{M}^* \cap C)$$

in this component.

Let us continue with showing the bound for the cycles with $2(\ell-1)$ edges as shown in the third component of Figure 4. Let us take the part of the component with m_1 to m_k, thus containing also o_1 to o_{k-1}. We have

$$m_1 + m_2 + \cdots + m_k \geq o_1 + o_2 + \cdots + o_{k-1} \ ,$$

otherwise there would be a weight-augmenting $(k-1)$-path. We shift both sides of the inequality by one edge at a time and obtain $\ell - 2$ more inequalities of the form

$$m_i + m_{i+1} + \cdots + m_{i+k-1} \geq o_i + \cdots + o_{i+k-2} \ ,$$

where an index $p > \ell - 1$ is converted to $p - \ell + 1$ and starts from 1. This way we collect $\ell - 1$ inequalities, where each m_i appears in total k times, and each o_j appears in total $k-1$ times. Adding the $\ell - 1$ inequalities side-by-side gives the desired result

$$w(\mathcal{M} \cap C) \geq \frac{k-1}{k} w(\mathcal{M}^* \cap C)$$

in a cyclic component C with $2(\ell - 1)$ edges where $\ell > k$.

We now look at a component C of the second type shown in Figure 4. Let us add an artificial vertex v, an edge of zero weight from v to the left vertex of $o_{\ell-1}$, and add that edge to $\mathcal{M}$ to obtain a matching $\mathcal{M}'$ and a component C'.

CLAIM B.1. *Any weight-augmenting k-path for $\mathcal{M}'$ in C' is also a weight-augmenting k-path for $\mathcal{M}$ in C.*

Proof of the claim: If a weight-augmenting path does not contain the new edge we are done. If it contains the new edge, then simply dropping that edge is still weight augmenting with the same number of edges outside $\mathcal{M}$. □

Based on the claim, the component C' is thus of the first form with $\ell - 1$ edges outside $\mathcal{M}$. As the weights in C are preserved in C', we have the desired result

$$w(\mathcal{M} \cap C) \geq \frac{k-1}{k} w(\mathcal{M}^* \cap C)$$

for any component C of the second type shown in Figure 4.

For the components of the remaining form, the right-most one in Figure 4, one can show the desired property by adding an edge m_0 of zero weight and converting the case the second one (and hence to the first one).

Since we have shown that in any component C of $\mathcal{M} \oplus \mathcal{M}^*$ with at least k edges from $\mathcal{M}^*$ we have $w(\mathcal{M} \cap C) \geq \frac{k-1}{k} w(\mathcal{M}^* \cap C)$, and there is no weight-augmenting j-path with $j < k$, the claim is proved. □

B.2 Proof of Theorem 3.1

Proof. The proof is adapted from the proof contained in [30] for the dynamic unweighted matching case. Assume that no weight-augmenting k-path (i.e., no augmenting path of length $\leq 2/\epsilon + 3$) exists. Then, according to Proposition 3.1, the matching is a $(\frac{1/\epsilon+1}{1/\epsilon}) = (1 + \epsilon)$-approximate maximum matching. To see this, rewrite the length of the path to $2(1/\epsilon + 1) + 1$ and set $k = 1/\epsilon + 1$ in Proposition 3.1. If there is such a path, then the probability of finding it is $\geq (\frac{1}{\Delta})^{2/\epsilon+3}$: after all, one possibility is that the random walker makes the "correct" decision at every vertex of the path. The probability that λ random walks of length $2/\epsilon+3$ do not find an augmenting path of length $2/\epsilon+3$ is $\leq (1 - \frac{1}{\Delta^{2/\epsilon+3}})^{\lambda} \leq e^{-\frac{1}{\Delta^{2/\epsilon+3}} \cdot \lambda}$. Thus, for $\lambda \geq \Delta^{2/\epsilon+3} \log n$, the probability is $\leq 1/n$. □