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. ACDA 2021 - SIAM Conference on Applied and Computational Discrete Algorithms, SIAM, Jul 2021, Virtual, France. 10.1137/1.9781611976830.4 . hal-03210915

HAL Id: hal-03210915
https://inria.hal.science/hal-03210915
Submitted on 28 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Fully-dynamic Weighted Matching Approximation in Practice

Eugenio Angriman* Henning Meyerhenke* Christian Schulz† Bora Učar†

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 \( \mathcal{O}(n^ω) \), where \( ω < 2.373 \) is the matrix multiplication exponent [4], \( m \) is the number of edges, and \( n \) is the number 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 \( \tilde{O}(Wn^ω) \) time, where \( \tilde{O} \) 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 \( \mathcal{O}(\epsilon^{-1}Δ^2/\epsilon^3 + \log n) \) time per

---

*Humboldt-Universität zu Berlin, Berlin, Germany
†Heidelberg University, Heidelberg, Germany
‡CNRS and LIP, ENS Lyon, France
update, where $\Delta$ is the maximum node degree observed throughout the algorithm. The second algorithm, introduced by Stubbs and Williams [40], uses existing fully-dynamic $\alpha$-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 $\omega : E \to \mathbb{R}_{\geq 0}$. We extend $\omega$ 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 $M \subset E$ 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 $M$. We call a matching maximal if there is no edge in $E$ that can be added to $M$. A maximum cardinality matching $M_{\text{opt}}$ is a matching that contains the largest possible number of edges of all matchings. A maximum weight matching $M'_{\text{opt}}$ is a matching that maximizes $\omega(M'_{\text{opt}})$ among all possible matchings. An $\alpha$-approximate maximum (weight) matching is a matching that has weight at least $\frac{\omega(M_{\text{opt}})}{\alpha}$. 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 $\{ u, v \} \in M$, we call vertex $v$ the mate of $u$, which we denote as $\text{mate}(u) = v$. For an unmatched vertex $u$, we define $\text{mate}(u) = \perp$. 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 $M$ alternate with edges from $E \setminus M$. 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 unmatched 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 $P$ with respect to a matching $M$ is an alternating path whose free edges are heavier than its edges in $M$: $w(M \oplus P) > w(M)$, where $\oplus$ denotes the symmetric difference. A weight-augmenting path with $k$ edges outside $M$ 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. All the algorithms evaluated can handle edge insertions as well as edge deletions. In the following, $\Delta$ 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 $M$ 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 $O(mn^2)$. This result was later improved to $O(mn\sqrt{n})$ 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 $\alpha$-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 $O(n^{1.407})$ update time. One can trivially maintain a maximal (2-approximate) matching in $O(n)$ update time by resolving all trivial augmenting paths of length one. Ivković and Llyod [42] designed the first fully-dynamic algorithm to improve this bound to $O((n + m)^{3/2})$ update time. Later, Onak and Rubinfeld [43] presented a randomized algorithm for maintaining an $O(1)$-approximate matching in a dynamic graph that takes $O(\log^2 n)$ 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 $O(1)$ to 2 and the amortized...
update time to $O(\log n)$. Further, Solomon\cite{18} improved the update time of \cite{6} from amortized $O(\log n)$ to constant. However, the first deterministic data structure improving \cite{32} was given by Bhattacharya et al.\cite{10}: it maintains a $(3 + \epsilon)$ approximate matching in $O(\min(\sqrt{n}, m^{1/3}/\epsilon^2))$ amortized update time, which was further improved to $(2 + \epsilon)$ requiring $O(\log n/\epsilon^2)$ update time by Bhattacharya et al.\cite{11}. Recently, Bhattacharya et al.\cite{9} achieved the first $O(1)$ amortized update time for a deterministic algorithm but for a weaker approximation guarantee of $O(1)$. For worst-case bounds, the best results are by (i) Gupta and Peng\cite{26} requiring $O(\sqrt{m}/\epsilon)$ update time for a $(1 + \epsilon)$-approximation, (ii) Neiman and Solomon\cite{42} requiring $O(\sqrt{m})$ update time for a $(3/2)$-approximation, and (iii) Bernstein and Stein\cite{8} requiring $O(m^{1/4}/\epsilon^{2.5})$ update time for a $(3/2 + \epsilon)$-approximation. Recently, Charikar and Solomon\cite{14} as well as Arar et al.\cite{3} (using \cite{12}) presented independently the first algorithms requiring $O(\text{poly} \log n)$ worst-case update time while maintaining a $(2 + \epsilon)$-approximation. Recently, Grandoni et al.\cite{25} gave an incremental matching algorithm that achieves a $(1 + \epsilon)$-approximate matching in constant deterministic amortized time. Barenboim and Maimon\cite{5} present an algorithm that has $O(\sqrt{n})$ update time for graphs with constant neighborhood independence. Kashyap and Narayananswamy\cite{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.\cite{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\cite{6}, which performs edge updates in $O(\sqrt{n})$ time and maintains a 2-approximate maximum matching, the algorithm of Neiman and Solomon\cite{42}, which takes $O(\sqrt{m})$ 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 optimal 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.\cite{2} propose an algorithm that can maintain an $4.911$-approximate dynamic maximum weight matching that runs in amortized $O(\log n \log D)$ time, where $D$ is the ratio between the highest and the lowest edge weight. Gupta and Peng\cite{27} maintain a $(1 + \epsilon)$-approximation under edge insertions/deletions that runs in time $O(\sqrt{m}\epsilon^{-2 - O(1/\epsilon)} \log N)$ time per update. Their result is based on several ingredients: (i) re-running 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\cite{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 $\alpha$-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 $O(T\epsilon^{-2} \log^2 C)$. Their basic idea is an extension/improvement of the algorithm of Crouch and Stubbs\cite{15} who tackled the problem in the streaming model. We go into more detail in Section\cite{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.\cite{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 $P$ to compute the best possible matching on $P$.\footnote{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.}

We continue by explaining these concepts in our context and how we handle edge insertions and deletions.
3.1 Random Walks For Augmenting Paths.

Our first dynamic algorithm is called DYNWMRANDOM and constructs a random (cycle-free) path \( P \) as follows: initially, all nodes are set to be eligible. Whenever an edge is added to the path \( P \), 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 successful, the corresponding edge \( e \) is added to the path \( P \) and the walk is continued at \( w \) (otherwise the random walk stops and returns \( P \)). Moreover, \( u \) is set to not eligible. On the other hand, if \( u \) is matched and mate(\( u \)) is eligible, we add the edge \{\( u \), mate(\( u \))\} to \( P \), 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 \( P \). 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 \( O(\frac{1}{\epsilon}) \) steps for a given \( \epsilon \).

The algorithm tries to find a random eligible neighbor by sampling a neighbor \( u \) uniformly at random. If \( u \) is eligible, then we are done. If \( u \) is not eligible, we repeat the sampling step. We limit, however, the number of unsuccessful repetitions to a constant. If our algorithm did not find an eligible vertex, we stop and return the current path \( P \). Overall, this ensures that picking a random neighbor can be done in constant time. Hence, the time to find a random path is \( O(\frac{1}{\epsilon}) \). Here, we assume 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 \( O(\frac{1}{\epsilon}) \) 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 investigate in the experimental evaluation.

After the path has been computed, we run a dynamic program on the path to compute the optimum weighted matching on the path in time \( O(\frac{1}{\epsilon}) \). If the weight of the optimum matching on \( P \) is larger than the weight of the current matching on the path, we replace the current matching on \( P \). 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 programming [38]. In order to be self-contained, we outline briefly how this can be done. The description of Algorithm 1 in Appendix A follows Maue and Sanders [38].

Given the path \( P = (e_1, \ldots, e_k) \), 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 optimum matching of the subproblem \((e_1, \ldots, e_i)\) 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 \((e_1, \ldots, e_{i-1})\). If this is larger than the weight of the optimum subsolution for the path \((e_1, \ldots, e_{i-1})\), then the edge \( e_i \) is used in an optimum solution for \((e_1, \ldots, e_i)\); otherwise it is not.

**Optimizations.** The random walk can be repeated 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 \( \ell \) walks, where \( \ell \) is a tuning parameter. However, repeating the algorithm too often can be time-consuming. Hence, we optionally use a heuristic to break early if we already performed a couple of unsuccessful random walks: we stop searching for augmenting paths if we made \( \beta \) 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 insertions \( \{u, v\} \) 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 \( P \). 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 endpoints is matched, say \( u \), we add the edges \{\( u, \text{mate}(u) \)\} and \{\( u, v \)\} to the path \( P \) 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 \( \text{mate}(v) \). Note that it is necessary to include matched edges, since the optimum weight matching on the path that we are constructing with the random 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 program 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 \( P \) that are created by the two random walks.
3.2 Analysis. The algorithm described above, called \texttt{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 \(w(\mathcal{M}) \geq \frac{1}{\epsilon} \cdot w(\mathcal{M}^*)\), where \(\mathcal{M}^*\) is a matching with the maximum weight.

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

4 Algorithm Family \texttt{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 \(O(\epsilon^{-1} \log C)\) 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 \(i \in \{\log_{1+\epsilon} L, \log_{1+\epsilon} L + 1, \ldots, \log_{1+\epsilon} N\}\). 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 already been 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.** \textsc{(Stubbs and Williams [49])} If the matchings on every level are \(\alpha\)-approximate maximum cardinality matchings, then the output matching of the algorithm is an \(2\alpha(1 + \epsilon)\)-approximate maximum weight matching. Moreover, the algorithm can be implemented such that the update time is \(O(t(m, n)(\log_2, N/L)^2)\), 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 \texttt{DynMWMLevelR} and \texttt{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 (\textit{insert} and \textit{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 \(\{w, \mate(w)\}\) and match \(\{u, w\}\) instead. Note that \(u \neq \mate(w)\) since \(u\) is free in the beginning and therefore \(\mate(u) = \bot\), but \(\mate(\mate(w)) = w\) 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 \(O(\Delta/\epsilon)\) steps. If unsuccessful, the algorithm can perform multiple repetitions to find an augmenting path. Moreover, the most successful version of the algorithm uses \(\Delta\)-settling, in which the algorithm tries to settle visited vertices by scanning through their neighbors. The running time of the \(\Delta\)-settling random walk is then \(O(\Delta/\epsilon)\).

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 \(w := \mate(u)\), and matches \(\{u, v\}\). Then a random walk as described above is started from \(w\) to find an augmenting path (of length at most \(O(1/\epsilon)\)). 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.** \textsc{(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 \texttt{DynMWMLevelR} algorithm is \(2(1 + \epsilon)^2\)-approximate with update time \(O(\epsilon^{-1} \Delta^{2/\epsilon - 1} \log n (\log_2, N/L)^2)\).

**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].

<table>
<thead>
<tr>
<th>Graph</th>
<th>n</th>
<th>m</th>
<th>Graph</th>
<th>n</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>144</td>
<td>144</td>
<td>1074</td>
<td>eu-2005</td>
<td>862</td>
<td>16138</td>
</tr>
<tr>
<td>3elt</td>
<td>4720</td>
<td>13722</td>
<td>fe_4elt2</td>
<td>11143</td>
<td>32181</td>
</tr>
<tr>
<td>4elt</td>
<td>15606</td>
<td>45878</td>
<td>fe_body</td>
<td>45087</td>
<td>163734</td>
</tr>
<tr>
<td>598a</td>
<td>110971</td>
<td>741934</td>
<td>fe_ocean</td>
<td>143437</td>
<td>409593</td>
</tr>
<tr>
<td>add20</td>
<td>2395</td>
<td>7462</td>
<td>fe_pwt</td>
<td>36519</td>
<td>144794</td>
</tr>
<tr>
<td>add32</td>
<td>4960</td>
<td>9462</td>
<td>feRotor</td>
<td>99617</td>
<td>662431</td>
</tr>
<tr>
<td>amazon-2008</td>
<td>735323</td>
<td>3523472</td>
<td>fe_sphere</td>
<td>16386</td>
<td>49152</td>
</tr>
<tr>
<td>as-22july06</td>
<td>22963</td>
<td>48436</td>
<td>fe_tooth</td>
<td>78136</td>
<td>452591</td>
</tr>
<tr>
<td>as-skitter</td>
<td>554930</td>
<td>5797663</td>
<td>finman512</td>
<td>74752</td>
<td>261120</td>
</tr>
<tr>
<td>auto</td>
<td>448695</td>
<td>3314611</td>
<td>in-2004</td>
<td>1382908</td>
<td>1359473</td>
</tr>
<tr>
<td>bcstk29</td>
<td>13992</td>
<td>302748</td>
<td>loc-brightkite_edges</td>
<td>56739</td>
<td>212945</td>
</tr>
<tr>
<td>bcstk30</td>
<td>28924</td>
<td>1007284</td>
<td>loc-gowalla_edges</td>
<td>196591</td>
<td>950327</td>
</tr>
<tr>
<td>bcstk31</td>
<td>35588</td>
<td>572914</td>
<td>m14b</td>
<td>214765</td>
<td>1679018</td>
</tr>
<tr>
<td>bcstk32</td>
<td>44609</td>
<td>985046</td>
<td>memplus</td>
<td>17758</td>
<td>54196</td>
</tr>
<tr>
<td>bcstk33</td>
<td>8738</td>
<td>291583</td>
<td>p2p-Gnutella04</td>
<td>6405</td>
<td>29215</td>
</tr>
<tr>
<td>brack2</td>
<td>62631</td>
<td>366559</td>
<td>PGPgiantcompo</td>
<td>10680</td>
<td>24316</td>
</tr>
<tr>
<td>citationCiteseer</td>
<td>268495</td>
<td>1156647</td>
<td>rgg_n_2_15_s0</td>
<td>32768</td>
<td>160240</td>
</tr>
<tr>
<td>coAuthorsCiteseer</td>
<td>227320</td>
<td>814134</td>
<td>t60k</td>
<td>60005</td>
<td>89440</td>
</tr>
<tr>
<td>coAuthorsDBLP</td>
<td>299067</td>
<td>977676</td>
<td>uk</td>
<td>4824</td>
<td>6837</td>
</tr>
<tr>
<td>coPapersDBLP</td>
<td>434102</td>
<td>16036720</td>
<td>vibrobox</td>
<td>12328</td>
<td>165250</td>
</tr>
<tr>
<td>crack</td>
<td>10240</td>
<td>30380</td>
<td>web-Google</td>
<td>356648</td>
<td>2093324</td>
</tr>
<tr>
<td>cs4</td>
<td>22499</td>
<td>43858</td>
<td>whitaker3</td>
<td>9800</td>
<td>28989</td>
</tr>
<tr>
<td>cti</td>
<td>16840</td>
<td>48232</td>
<td>wiki-Talk</td>
<td>232314</td>
<td>1458806</td>
</tr>
<tr>
<td>data</td>
<td>2851</td>
<td>15093</td>
<td>wing</td>
<td>62032</td>
<td>121544</td>
</tr>
<tr>
<td>email-EuAll</td>
<td>16805</td>
<td>60260</td>
<td>wing_nodal</td>
<td>10937</td>
<td>75488</td>
</tr>
<tr>
<td>enron</td>
<td>69244</td>
<td>254449</td>
<td>wordassociation-2011</td>
<td>10617</td>
<td>63788</td>
</tr>
</tbody>
</table>

Depth by $2/ε − 1$ to find augmenting paths from a free node. This has a theoretically faster running time of $O(Δ^{2/ε−1})$ per free node and still guarantees a $(1 + ε)$-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 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 $2/ε − 1$. This ensures that there is no augmenting path of length at most $2/ε − 1$, thus yielding a deterministic $(1+ε)$-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 $2/ε − 1$, then the resulting DYNMWMLEVELB algorithm is $2(1+ε)^2$-approximate with update time $O(Δ^{2/ε−1}(\log 1+ε N/L)^2)$.

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 $H_u$ that maps...
a neighbor \( v \) of \( u \) to its position in \( L_v \). 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 \( H_u \). Swap the element in \( L_u \) with the last element \( w \) in the vector and update the position of \( w \) in \( H_u \). Finally, pop the last element (now \( v \)) from \( L_u \) and delete its entry from \( H_u \).

Our experiments are conducted on one core of a machine with an 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 \([15]\). These plots relate the matching weight of all algorithms to the optimum matching weight. More precisely, the \( y \)-axis shows \( \#\{\text{objective} \geq \tau \times \text{OPT}\}/\#\text{instances} \), where \text{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 2 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 \( \mathcal{U} \).

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

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, \( \epsilon \) is set to \( 10^{-5} \)). 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 \( \epsilon \), the
Figure 1: Left: Number of random walks and random walks with “stop early” heuristic (SE) for fixed epsilon. Right: Various values of $\epsilon$ 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 $\epsilon$.

<table>
<thead>
<tr>
<th># walks</th>
<th>geo. mean t [s]</th>
<th>geo. mean $\omega(M)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.0</td>
<td>1572285.92</td>
</tr>
<tr>
<td>5</td>
<td>16.4</td>
<td>+1.15%</td>
</tr>
<tr>
<td>10</td>
<td>20.5</td>
<td>+1.28%</td>
</tr>
<tr>
<td>20</td>
<td>21.8</td>
<td>+1.31%</td>
</tr>
<tr>
<td></td>
<td>with SEHeuristic</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>56.4</td>
<td>+1.91%</td>
</tr>
<tr>
<td>50</td>
<td>128.6</td>
<td>+2.24%</td>
</tr>
<tr>
<td>100</td>
<td>241.6</td>
<td>+2.44%</td>
</tr>
<tr>
<td></td>
<td>without SEHeuristic</td>
<td></td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>geo. mean t [s]</td>
<td>geo. mean $\omega(M)$</td>
</tr>
<tr>
<td>1</td>
<td>2.4</td>
<td>1547129.56</td>
</tr>
<tr>
<td>$10^{-1}$</td>
<td>5.9</td>
<td>+1.84%</td>
</tr>
<tr>
<td>$10^{-2}$</td>
<td>13.7</td>
<td>+2.71%</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>20.5</td>
<td>+2.93%</td>
</tr>
<tr>
<td></td>
<td>with SEHeuristic</td>
<td></td>
</tr>
</tbody>
</table>

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 weight matching weight. As soon as we decrease $\epsilon$, 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 $\epsilon$ 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 slower than $1/\epsilon$. This is due to the “stop early” heuristic. Overall, we conclude here that both the number of walks and the parameter $\epsilon$ 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, $\epsilon$ and the number of walks [30]. That is why Henzinger et al. [30] use only one random walk and vary only $\epsilon$ 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-
We now evaluate the fastest algorithms in this experiment are at least one\(\epsilon\) values of\(\tau\) based algorithms as MCM algorithms for different values of\(\tau\). We compute at least a weighted matching for \(\epsilon = 0.1\). Note that this is very well within the
factor 2 that theory suggests. However, for larger values of \(\epsilon\), 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.

### 5.2 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 \(\epsilon\) 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 \(\epsilon\) on each level. For example, the difference between DYNMWMLEVELR and DYNMWMLEVEL.B 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 \(\epsilon\) as the algorithm creates a lot of levels for small \(\epsilon\) and thus needs a large amount of memory. This is due to the large amount of levels created for small values of \(\epsilon\). 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 DYNMWMRANDOM.

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

### 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 \(U_x\)). More precisely, we perform the operations in \(U_x\) in reverse order. If an edge operation was an insertion in \(U_x\), 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 $\epsilon$ and walks per update ($w$). Smaller is better.

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

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


A Pseudocode of Dynamic Programming for MWM on Paths

Algorithm 1 MWM($P = \langle e_1, \ldots, e_k \rangle$) \[38\]
\begin{algorithmic}
1: \textbf{if} $\omega(e_i) + W[i-1] > W[i-1]$ \textbf{then}
2: \quad $W[i] := \omega(e_i) + W[i-2]$
3: \textbf{else}
4: \quad $W[i] := W[i-1]$
5: \textbf{end if}
6: \textbf{end for}
7: \textbf{return} $M[k]$
\end{algorithmic}

B Omitted Proofs

B.1 Proof of Proposition 3.1

A weight-augmenting path $P$ with respect to a matching $\mathcal{M}$ is an alternating path with respect to $\mathcal{M}$, and the matching $\mathcal{M} \oplus P$ has a larger weight than $\mathcal{M}$; in other words $w(\mathcal{M} \oplus P) > w(\mathcal{M})$. 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 use 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

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

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

$$m_1 + m_2 + m_3 \geq o_1 + o_2 + o_3$$

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, $m_i$ for $2 \leq j \leq \ell - k + 1$ is included in $k$ inequalities. Once we have included the last edge $m_\ell$ 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:

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

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

$$\vdots$$

$$m_{\ell - 1} + m_\ell \geq o_{\ell - 1},$$
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:

- $m_3$ 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 $m_i$.

- $m_\ell$ 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

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

and 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
\[
w(M \cap C) \geq \frac{k - 1}{k} w(M^* \cap C)
\]
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 \( M \) to obtain a matching \( M' \) and
a component \( C' \).

Claim B.1. Any weight-augmenting \( k \)-path for \( M' \) in \( C' \) is also a weight-augmenting \( k \)-path for \( M \) 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 \( M \).
\( \square \)

Based on the claim, the component \( C' \) is thus of the
first form with \( \ell - 1 \) edges outside \( M \). As the weights
in \( C \) are preserved in \( C' \), we have the desired result
\[
w(M \cap C) \geq \frac{k - 1}{k} w(M^* \cap C)
\]
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 \( M \oplus M^* \) with at least \( k \) edges from \( M^* \) we have
\[
w(M \cap C) \geq \frac{k - 1}{k} w(M^* \cap C),
\]
and there is no weight-augmenting \( j \)-path with \( j < k \),
the claim is proved. \( \square \)

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 \((1 + \epsilon)\)-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 \( \geq (\frac{1}{\epsilon})^{2/\epsilon + 3} \); after all, one possibility is that the random walker makes the
"correct" decision at every vertex of the path. The probability that \( \lambda \) random walks of length \( 2/\epsilon + 3 \) do not find
an augmenting path of length \( 2/\epsilon + 3 \) is \( \leq (1 - \frac{1}{2\sqrt{\lambda / \epsilon}})^\lambda \leq e^{-\frac{1}{2\sqrt{\lambda / \epsilon}}} \). Thus, for \( \lambda \geq \Delta^{2/\epsilon + 3} \log n \), the probability
is \( \leq 1/n \). \( \square \)