Fast Approximate Optimum Matching on dynamic graphs in practice

Martin Grösbacher,

February 14, 2022

In this paper, we conduct and evaluate experiments for computing an approximate optimum matching on dynamic graph instances. The algorithmic approach we used is based on the Random Walker from [1] which tries to find augmenting paths to increase the matching size. Next to comparing our results, this paper also focuses on aspects that did not get as much attention in [1] like the impact of the number of repetitions of the Random Walker or the performance of different settings of the algorithm on specific graph instances as opposed to [1] where the average over many different graph instances were taken.

1 Introduction

A matching $\mathcal{M}$ of a graph $G = (V, E)$ is a subset of edges such that the vertices of each pair of elements of $\mathcal{M}$ are distinct. Applications sometimes require matchings with certain properties like being of maximal weight in weighted graphs, touching all vertices of the graph (i.e. perfect matching) or having maximal cardinality (i.e. optimum matchings). Finding such matchings in graphs is a well studied combinatorial problem with a wide variety of applications such as crossbar scheduling [2], the stable marriage problem [3] or when computing mail delivery routes [4]. Real world applications with connections to finding optimum matchings are found for example in two-processor scheduling [5], [6] or the pairwise kidney exchange problem [7]. Although computing matchings with these properties can be done in polynomial time, changes to the underlying graph in dynamic settings would still pose a computationally expensive task if the matching would have to be re-computed from scratch after every insertion or deletion of an edge. Hence, in the recent years, researchers have developed algorithms to maintain matchings in dynamic settings. Maintaining exact maximum matchings is bounded by an update time of $\Omega(\sqrt{(}m))$ [8], [9] and thus, research focuses mostly on maintaining approximate maximum matchings. Since there are still not many published results from the practical perspective and performance of these algorithms, this paper tries to complement [1] in its attempt to bridge the gap between theory and practice for one of these algorithmic approaches for the fully dynamic maximum matching problem. The algorithm used in this paper is based on a depth-bound Random Walker that is initialized after each dynamic operation and tries to find augmenting paths to increase the current matching size. After a short review of some

needed preliminaries, we will explain the core idea of the Random Walk based algorithm of [1] and then present and compare our experimental evaluation and results to theirs.

2 Preliminaries

Throughout this paper we only consider $G = (V, E)$ to be undirected, unweighted and without parallel edges or self-loops. We set $n = |V|$ and $m = |E|$. Δ denotes the highest degree that can be found in the current state of the graph. A matching $\mathcal{M} \subset E$ is a set of edges where no pair of edges shares a vertex. A matching is said to be *maximal*, if no edge of E can be added to $\mathcal{M}$ without violating the definition and a *maximum cardinality* or *optimum matching* $\mathcal{M}_{\text{opt}}$ is a maximal matching that contains the largest number of possible edges. An α-*approximate* maximum matching is a matching that contains at least $\frac{|\mathcal{M}_{\text{opt}}|}{\alpha}$ edges. We call a vertex *free* if it is not incident to an edge of $\mathcal{M}$ and *matched* otherwise. If a vertex u is matched, we call v with $(u, v) \in \mathcal{M}$ the *mate* of u. An augmenting path is a cycle-free path in G where the start and end vertex are free and all edges alternate between being edges of $\mathcal{M}$ or edges of $E \setminus \mathcal{M}$. Finding augmenting paths is crucial in maintaining optimum matchings because by simply reversing the matching status of all the edges of the path we can effectively increase the size of the matching by one without violating the matching definition.

3 The Random-Walk-based Algorithm

Conceptually, the Random-Walk-based algorithm works as follows: We start at a free vertex u and randomly choose a neighbour v of u. Our further proceeding is now based on the matching status of v: If v is free, then both u and v are free and we conclude that both vertices are not incident to any edge of $\mathcal{M}$. This means we can just match (u, v) and the Random Walk was successful. If v is matched, we unmatch $(v, \text{mate}(v))$, match (u, v) and continue the Random Walk from the mate(v). The Random Walks have a path length of $\frac{2}{\epsilon} - 1$ and in theory have to be repeated $\Delta^{\frac{2}{\epsilon}-1} \log(n)$ times to maintain a $(1 + \epsilon)$-approximation of the optimum matching as proven by [1]. However, by depth bounding the Random Walks we have to reverse all changes done to the matching status of the vertices and edges if no free vertex was found within the length of the path. In order to circumvent this problem and further improve the quality of the matching, [1] introduced the method of Δ-settling. With this option enabled, we scan $\frac{1}{\epsilon}$ neighbours of each vertex visited to try to find a free vertex instead of picking a random neighbour first. If a Walk would be unsuccessful because it exceeds the path length and the last vertex visited was not free, we scan through all of its neighbours first instead of undoing all changes. This requires an additional $O(\Delta)$ time per vertex visited but improves the matching results as we will show in the experimental evaluation section. Edge insertions on G are now handled as follows:

Edge insertion. Upon inserting an edge (u, v), we differ between three cases:

1. both u and v are free: We simply match (u, v)
2. both u and v are matched: We do nothing
3. only one of u and v is matched: W.l.o.g. let this be u. We unmatch $u, \text{mate}(u)$,

match (u, v) and start a random walk from mate(u) as described above. If the walk is unsuccessful we undo all changes.

Deletions of edges can be handled similarly by doing nothing if the (u, v) was free and scanning the neighbours of both u and v if the delete edge (u, v) was matched, trying to find free neighbours to directly match them again. If no such neighbour can be found, Random Walks can be started from u and/or v respectively. However since the focus of this paper is the comparison of the results of the matching sizes on graph instances with different parameters of the Random Walker, our experiments use random insertions of edges only and compare the final result to the optimum matching obtained by the algorithm of Micali and Vazirani, analogously to the first type of experiments of [1].

4 Experimental Setting

Implementation and System. The algorithm described in the previous section has been implemented in C++ and compiled using g++ 11.2. Experiments were conducted on one core of a machine using an AMD Ryzen 7 processor with 16GB of RAM. The graph data structure is built as follows: Each vertex maintains a vector of edges incident to it and each edge knows its two end points. When searching a random neighbour for u we can simply choose a random index of the edge vector of u and via this edge we can also retrieve the other end point. Since accessing elements via index in a vector in C++ is constant, the neighbour search operation is constant.
Methodology. For every graph instance and different settings (choice of epsilon, enabling/disabling Δ-settling, Random Walk repetitions) we performed ten repetitions. We measure both the average time and the quality of the result, i.e. the size of the matching. We compare the matching size to the optimum matching obtained by running the algorithm of Micali and Vazirani [10] on the same graph instances. An implementation of this algorithm is available at Github and with slight adaptations was made compatible with our input graphs[1]. We used this as a benchmark for all our experiments. For each graph, we let the algorithm run for $\epsilon = 1, 0.5, 0.25, 0125$ respectively and for each ϵ we perform the repetitions with and without *Delta*-Settling. Similarly to [1] we start by invoking the Random Walker only once for every insertion but then also test multiple invocations, described in detail for each graph separately in the next section.
Graph Instances. Table 1 summarizes the main properties of our example graphs. These are static graphs obtained from Stanfords large network dataset collection[2]. Directed graph instances were made undirected by mirroring all edges and removing self-loops. Dynamicity is now simulated by shuffling the edges randomly before inserting them or undoing insertions, however the same permutation of insertions/deletions is then used for all the experiments to have a meaningful comparison. The initial shuffle is there only to generalize the results as if the input graphs and their edge insertions are done in order, the results can be very different due to the nature of a graph.

[1] https://github.com/ggawryal/MV-matching
[2] https://snap.stanford.edu/data/

Graph	n	m	D	avg. deg	$\mathcal{M}_{\text{opt}}$
WikiTalk	2394385	5021410	9	2.097	56063
as-skitter	1696415	11095298	25	6, 54	513304
p2p-Gnutella31	62586	147892	11	2, 36	15693

Table 1: graph instances and their properties. $n = |V|$, $m = |E|$, D =diameter, avg. deg refers to the average vertex degree and $\mathcal{M}_{\text{opt}}$ to the size of the optimum matching computed by the algorithm of Micali and Vazirani.

5 Experimental Results

In this section we show our experimental results. The figure graphs show the average of the ten repetitions of the respective setting. For figures $1-3$, the x-axis shows the different choices of ϵ and the y-axis shows the size of the resulting matchings. Also, for these figures, the top of the y-axis represents the size of the optimum matching of the respective graph computed with [10]. "Base" refers to the algorithm being run with only one invocation of the Random Walker per edge insertion and no *Delta*-Settling. "Repetitions" means we invoked the Random Walker at least $\frac{2}{\epsilon} - 1$ times which is still not even close to guaranteeing the theoretical bound of a $(1 + \epsilon)$ approximation as proven by [1]. The number of repetitions actually required would be $\Delta^{\frac{2}{\epsilon}-1}$ which for large graphs quickly became computationally infeasible. However, not only do our results confirm the results of [1] in the fact that a single repetition of the Random Walker already delivers results well within the desired approximation in all conducted experiments, they also indicate that the number of repetitions does make at least some small difference in the quality of the result. "Δ-Settling" means we enabled Δ-Settling, however only performed one invocation of the Random Walker per edge insertion in this case.

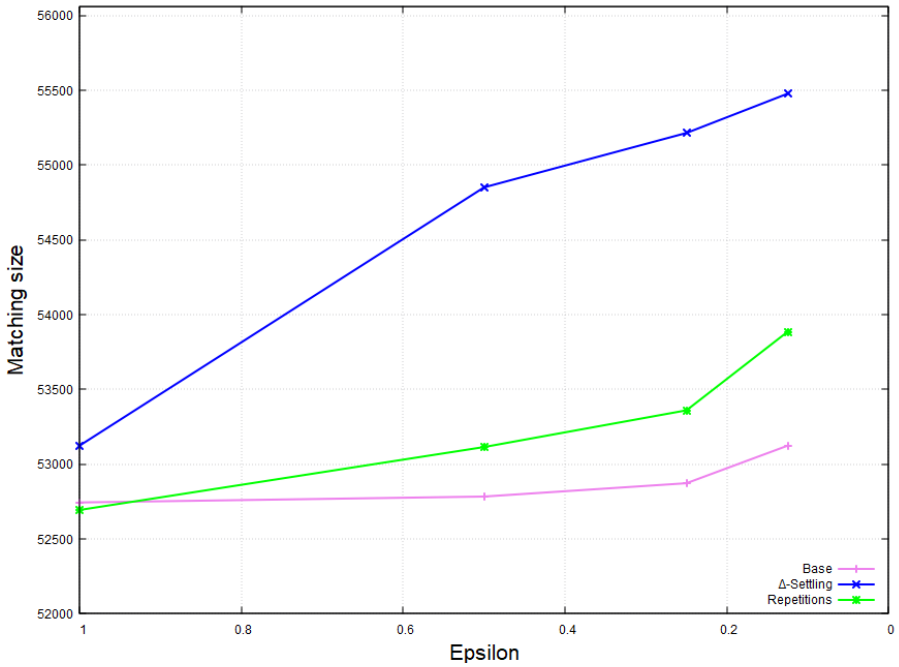

Figure 1: Matching size results on the large WikiTalk graph of small diameter $D = 9$

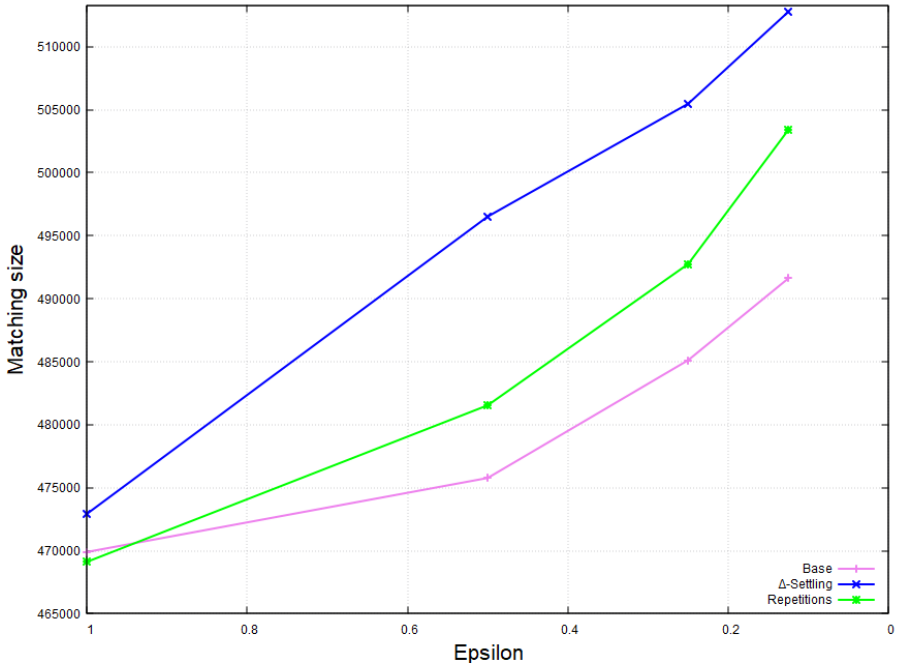

Figure 2: Matching size results on the large as-skitter graph of higher diameter $D = 25$.

For the WikiTalk graph in figure 1 for example, we see that the "Repetitions" graph starts off at around the same quality as the "Base" graph. This is not surprising as when $\epsilon = 1$ then $\frac{2}{\epsilon} - 1 = 1$. However with smaller ϵ this starts to improve the result up to slightly over 1% at the smallest $\epsilon = 0.125$. For the as-skitter graph in figure 2, this difference was

even greater (up to 2% for $\epsilon = 0.125$). We believe this is due to the fact that the as-skitter graph is of significantly larger diameter than the WikiTalk graph which increases the probability of finding longer augmenting paths. This is also reinforced by the differences in the optimum matching size for these two graphs. Enabling *Delta*-Settling generally significantly improved all the matching results. While the effect is not quite as significant for $\epsilon = 1$, the difference quickly becomes larger with smaller ϵ. This is due to the fact that if $\epsilon = 1$, then the maximum path depth $\frac{2}{\epsilon} - 1 = 1$ and thus we do not gain the main advantage of *Delta*-Settling which would scan $\frac{1}{\epsilon}$ neighbours of each visited vertex during the Random Walk. However we do scan all the neighbours of the last vertex visited which already shows improved matching results. The real advantage of *Delta*-Settling however is shown with smaller ϵ. In Figures 1 and 2, enabling *Delta*-Settling with $\epsilon = 0.125$ brought us within 1% of the optimum matching size of the graphs computed by [10] even though the Random Walker is only invoked once per edge insertion.

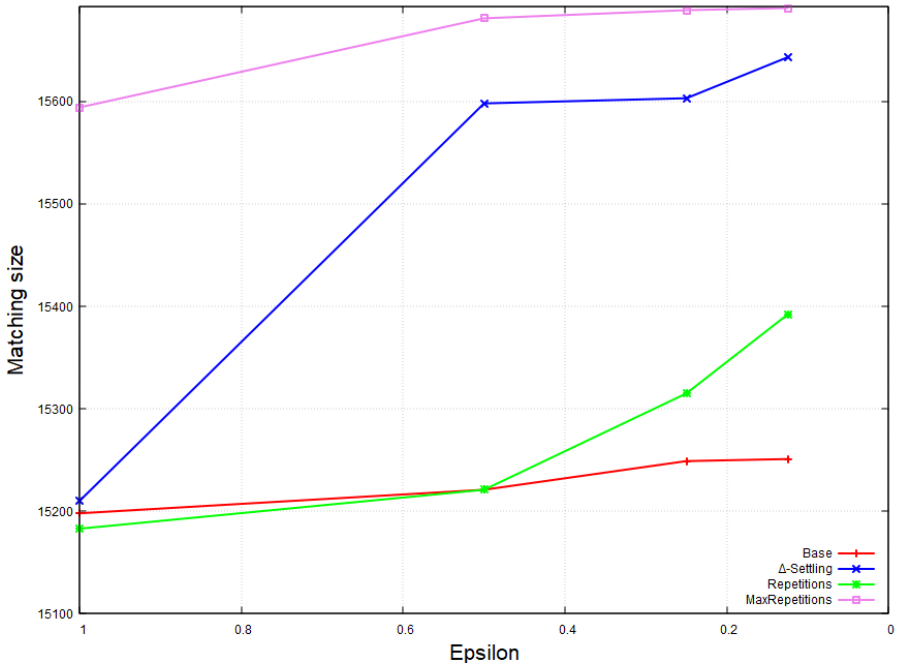

Figure 3: Matching size results on the smaller p2p-gnutella31 graph.

Figure 3 shows four graphs. In addition to the ones in the previous figures we have a graph called "MaxRepetitions". This graph refers to the algorithm used with $\Delta\frac{2}{\epsilon} - 1$ repetitions of the Random Walker. Even for the small p2p-gnutella31 graph, the actual theoretical bound of $\Delta^{\frac{2}{\epsilon}-1}$ was computationally infeasible for $\epsilon < 1$. Repeating the Random Walker this often shows significantly improved results and for small ϵ got up to 0.02% close to the optimum matching. For the other graphs in the figure, similar tendencies could be observed as for the large graph instances of figure 1 and 2.

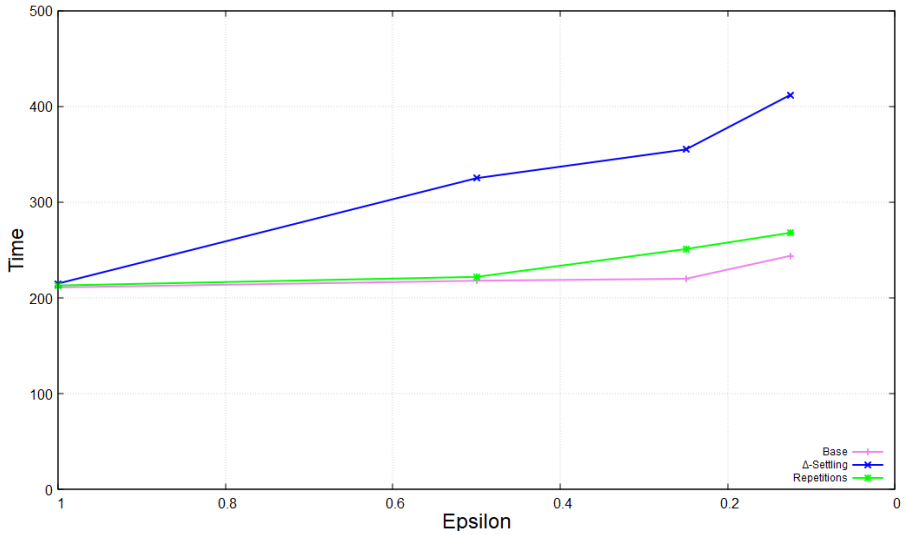

Figure 4: Time results for the runs with different settings on the as-skitter graph. The y-axis shows the time spent in seconds.

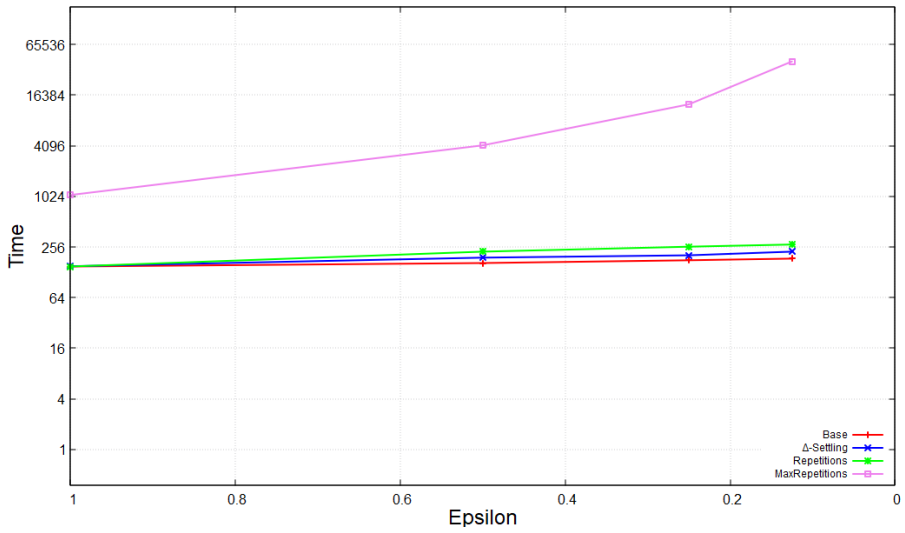

Figure 5: Time results for the runs with different settings on the p2p-gnutella31 graph. The logarithmically scaled y-axis shows the time spent in milliseconds.

Lastly, we show the differences in terms of time spent for the execution of the algorithm on a large graph (figure 4) and a small graph (figure 5) respectively. In the large as-skitter graph, repeating the Random Walker up to $\frac{2}{\epsilon} - 1$ times has a negligible impact on the run time as long as epsilon does not get too small. Enabling Δ-Settling does have an impact on the run time and in relation for $\epsilon = 0.125$ it increases the run time roughly by a factor of 2. However in terms of absolute numbers this is still well within practical usage so long as one does not deal with extremely large graphs. In the smaller p2p-gnutella31 graph, computations for all three graphs were quite similar as the algorithm on such small graphs in general can be executed in milliseconds for each different setting. However when using

a larger number of repetitions such as $\Delta\frac{2}{\epsilon} - 1$, the computation time quickly rises up to almost a minute for $\epsilon = 0.125$ which in relation is an increase in time by a factor of roughly 25. This yields that such a large number of repetitions in practice will not be applicable except when dealing with very small input graphs.

6 Conclusion

In this paper, we showed the practical usage of a Random-Walk-based algorithm to maintain a $(1+\epsilon)$-approximate maximum matching on different input graphs. The style of the implementation as well as the experimental setup were inspired by [1] and complemented by analyzing the impact of the number of repetitions of the Random Walker as well as analyzing the behaviour of the algorithm on specific graphs with certain properties (i.e. graphs of small and large diameter, number of vertices or edges). For smaller graphs like p2p-gnutella31, a large number of repetitions of the Random Walker will yield the best results while still not consuming too much computational time as long as ϵ does not become too small. For small ϵ, the matchings computed approach over 99.9% of the size of the optimum matching. For larger graphs of small diameter like WikiTalk, a large number of repetitions of the Random Walker quickly becomes computationally infeasible. Therefore, we recommend using a single repetition of the Random Walker with Δ-Settling enabled for these type of graphs. For large graphs with large diameter like as-skitter, we would again recommend a single Random Walk repetition per edge insertion with Δ-Settling enabled. Although Δ-Settling does increase the run time for these graphs, this increase is negligible so long as not dealing with very large graphs or running the algorithm with very small ϵ. Δ-Settling does increase the quality of the matching significantly and should be used when possible. This work can still be easily extended by conducting similar experiments when including fully dynamic real world graphs featuring edge insertions and deletions or on large and very dense real world graphs. Unfortunately, such graphs are hardly available in practice so it might be more practical to generate them artificially. Another idea would be to try to parallelize the Random Walks by use of threads which seems an intuitive implementation option for such a concept.

References

[1] M. Henzinger, S. Khan, R. Paul, and C. Schulz, "Dynamic matching algorithms in practice," 04 2020.

[2] L. Gong, L. Liu, S. Yang, J. Xu, Y. Xie, and X. Wang, "Serenade: A parallel randomized algorithm suite for crossbar scheduling in input-queued switches," *ArXiv*, vol. abs/1710.07234, 2017.

[3] D. Gale and L. S. Shapley, "College admissions and the stability of marriage," *The American Mathematical Monthly*, vol. 69, no. 1, pp. 9–15, 1962.

[4] J. Edmonds and E. Johnson, "Matching, euler tours and the chinese postman," *Mathematical Programming*, vol. 5, pp. 88–124, 12 1973.

[5] T. Fuju and N. Ninomiya, "Optimal sequence of two equivalent processors," *Siam Journal on Applied Mathematics - SIAMAM*, 01 1971.

[6] H. N. Gabow, "An almost-linear algorithm for two-processor scheduling," *J. ACM*, vol. 29, no. 3, p. 766–780, jul 1982. [Online]. Available: https://doi.org/10.1145/322326.322335

[7] A. Roth, S. Tayfun, and U. Unver, "Pairwise kidney exchange," 09 2004.

[8] A. Abboud and V. Williams, "Popular conjectures imply strong lower bounds for dynamic problems," *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, 02 2014.

[9] T. Kopelowitz, S. Pettie, and E. Porat, "Higher lower bounds from the 3sum conjecture," in *SODA*, 2016.

[10] S. Micali and V. Vazirani, "An o(sqrt(|v|) |e|) algorithm for finding maximum matching in general graphs," 10 1980, pp. 17–27.