New Iterative Algorithms for Weighted Matching

Alexander Idelberger* and Fredrik Manne**

Abstract

Matching is an important combinatorial problem with a number of practical applications. Even though there exist polynomial time solutions to most matching problems, there are settings where these are too slow. This has led to the development of several fast approximation algorithms that in practice compute matchings very close to the optimal.

The current paper introduces a new deterministic approximation algorithm named G^3, for weighted matching. The algorithm is based on two main ideas, the first is to compute heavy subgraphs of the original graph on which we can compute optimal matchings. The second idea is to repeatedly merge these matchings into new matchings of even higher weight than the original ones. Both of these steps are achieved using dynamic programming in linear or close to linear time.

We compare G^3 with the randomized algorithm GPA+ROMA which is the best known algorithm for this problem. Experiments on a large collection of graphs show that G^3 is substantially faster than GPA+ROMA while computing matchings of comparable weight.

1 Introduction

The matching problem is an archetypical combinatorial problem in computer science with a number of practical applications. Examples include areas such as community detection [1], graph partitioning [2, 3], and network alignment [4].

We consider the maximum weighted matching problem where one is given an undirected weighted graph $G = (V, E, \omega)$ with weights $\omega(e) > 0$ for all $e \in E$. The object is then to compute a set of non-adjacent edges such that the sum of the selected edges is as large as possible. The best known algorithm for solving this problem on a general graph has running time $O(n(m + n \log n))$ where $m = |V|$ and $n = |E|$ [5]. Still, this might be too slow for many applications, especially for large graphs. Therefore, there has been extensive research in developing fast approximation algorithms with linear or close to linear running time.

Such algorithms typically fall into one of two categories. The first is based on initially computing a subgraph of large weight on which an optimal matching can be found using dynamic programming. Examples of such subgraphs include only paths

This paper was presented at the NIK-2014 conference; see http://www.nik.no/.

*Institute for Theoretical Computer Science, University of Lübeck, D-23562 Lübeck, Germany. Email: idelberg@informatik.uni-luebeck.de

**Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Email: fredrikm@ii.uib.no

[6, 7], paths and cycles [8, 9], and trees [10]. The second category is based on finding short augmenting paths by which to improve the current matching [8, 11].

As this last type of algorithms can be applied to *any* matching, it is possible to start with an algorithm of the first type and then improve this using an algorithm of the second type. In fact, the current best fast approximation algorithm uses this method. It consist of first computing a matching through dynamic programming on a subgraph consisting of heavy paths and cycles (GPA) and then selecting random vertices on which to search for short augmenting paths (ROMA) [8]. Experiments show that in practice this combined algorithm produces matchings very close to the optimal solution.

In [9] an algorithm was developed by taking the union of two non-overlapping matchings and then performing dynamic programming on the resulting subgraph of paths and even length cycles. Using the standard Greedy algorithm to compute the two initial matchings gave a final matching of comparable quality to GPA. In this paper we expand further on this idea. We observe that the most time consuming part of GPA is the sorting of the edges. Once this has been done, it is possible to very efficiently compute a number of different matchings which can then be merged together to form new even heavier matchings. For this we utilize a number of existing matchings algorithms such as Greedy and GPA as well as new tree-based algorithms. The resulting matchings are then combined together to develop new deterministic algorithms. Experiments on a large number of different graphs show that our approach gives matchings of comparable quality as GPA+ROMA while using considerable less time.

The rest of the paper is organized as follows. Section 2 presents existing algorithms and methods. In Section 3 we elaborate further on the merging idea and also give several new tree-based matching algorithms, while results from experiments using different combination of the existing and the new algorithms are presented in Section 4. Finally, we conclude in Section 5.

2 Previous Work

In the following we present an overview of relevant algorithms for computing weighted matchings. It is well known that one can compute a maximum weight matching in linear time on graphs such as paths, cycles, and trees using dynamic programming. For trees one performs a post order traversal while computing the best matching for each vertex v when v is allowed to match with one of its descendant and the best matching when v is unmatched. For details see [8],[10].

The first algorithm to exploit dynamic programming for general matching was the Path Growing Algorithm (PGA') [6, 7] that repeatedly grows a path from an arbitrary unvisited vertex, each time following the heaviest edge to an unvisited vertex. Once a path cannot be extended further dynamic programming is used to find the optimal matching on the selected edges. The algorithm terminates when all vertices have been visited, and runs in $O(n+m)$ time.

The Global Path Algorithm (GPA) is an extension of PGA' that also first builds an intermediate graph H. However, here the edges are considered by decreasing weight, and an edge is added to H as long as H still consists of paths and even length cycles. Following this, a maximum weight matching is computed on H using dynamic programming as described above. The running time of GPA is dominated by the sorting of the edges which is $O(m \log n)$. We note that Greedy, PGA', and

GPA all have approximation ratios of 0.5.

In [9] it was shown that, similar to GPA, one can compute an intermediate graph consisting of paths and even length cycles by taking the union of any two existing matchings. A scheme was proposed and tested where one computes two non-overlapping matchings using the regular Greedy algorithm followed by dynamic programming to compute the final matching as in GPA. Experiments showed that this procedure gave matchings on par with those of GPA.

As the matching resulting from a dynamic programming step might not be maximal on G, it is possible to post process any remaining edge with two unmatched endpoints. One possibility is to add any such eligible remaining edge to the matching when it is inspected. If edges are considered by decreasing weight this is equivalent to using Greedy. Another option, which was suggested in [8], is to run the original algorithm on the remaining edges again. This can either be done a fixed number of times, or as long as there are candidate edges.

RAMA and ROMA are two randomized algorithms [8, 11] where the basic idea is to repeatedly pick a vertex v and then to find the best augmenting path of a fixed length centered in v. Whenever such a path is found it is used to augment the current matching. The algorithms differs in that RAMA chooses the vertex v from a uniform distribution while ROMA iterates multiple times through all vertices in a random order. It is suggested to restrict any augmenting path to at most two unmatched edges.

Experiments performed in [8] showed that ROMA is most suitable as an algorithm for improving on a given matching. In combination with GPA, the results were particularly good. The number of iterations through all vertices was at most eight giving an overall time complexity of $O(m+n)$.

We note that a recent algorithm for weighted matching gives a $(1-\epsilon)$ approximation in $O(m\epsilon^{-1}\log\epsilon^{-1})$ time [12]. Although the execution time is linear for fixed ϵ, no experimental results have been presented. We also note that the algorithm appears to be somewhat non-trivial, thus its practical impact is not yet clear.

3 New Algorithms

In the following we describe two ideas for constructing new deterministic algorithms for efficiently computing heavy matchings. The first is to use merging repeatedly on matchings computed by different algorithms, while the second idea is to use more advanced algorithms to compute larger and heavier overlay graphs, on which we can still compute an optimal matching in linear time using dynamic programming. We first present how merging can be used.

Repeated Merging

The most expensive part of algorithms such as Greedy and GPA is the sorting of the edge list taking $O(m\log n)$ time in the general case. Once this has been done, the remainder of the algorithm traverses the edge list only once in linear time. Thus the marginal cost for computing the matching is low once the sorting has been done. Note that this is true even if a predefined subset of the edges is to be excluded from consideration in the new matching.

We now consider how this observation can be used together with the merging operation. In [9] a variant of Greedy was used to compute the first matching.

This algorithm is then repeated a second time, but now excluding any edges that were included in the first matching. Finally, the two non-overlapping matchings are merged using dynamic programming. We can view this process as if we are *augmenting* the first matching M_1 with a second matching M_2 computed on the edges of $G' = (V, E \setminus M_1, \omega)$. We denote this operation as $M_1 \oslash M_2$. It is clear that $(M_1 \oslash M_2) = (M_2 \oslash M_1)$ is not true in general.

If two matchings computed with different algorithms are to be merged, then it might not be necessary to require that the matchings are non-overlapping. This could be advantageous if the algorithms tend to behave differently; the resulting matching will in any case be at least as heavy as both stand alone algorithms. We denote the merging operation of two *independent* matchings M_1 and M_2 as $M_1 \oplus M_2$. As the outcome of $M_1 \oplus M_2$ is an optimal matching on the edges of M_1 and M_2 it follows that $(M_1 \oplus M_2) = (M_2 \oplus M_1)$ and that $(M_1 \oplus M_1) = M_1$.

The merge operation can be applied iteratively, thus letting the resulting matching be the input in a new merge step. This is particularly convenient if new matchings can be computed efficiently once the edge list is sorted. In this way it is possible to specify the computation of the final matching as a fully parenthesized expression using the basic matchings as literals and $\oslash$ and $\oplus$ as binary operators. For this purpose one could define that any edge used in the final matching of the left expression when using the $\oslash$ operator cannot be used in any matching in the right expression. To simplify things we will always use a single basic matching as the right expression in this case. An example could be $((M_1 \oplus M_2) \oslash M_3)$. Here M_1 and M_2 are computed independently and merged before M_3 is computed while excluding any edge used in $M_1 \oplus M_2$.

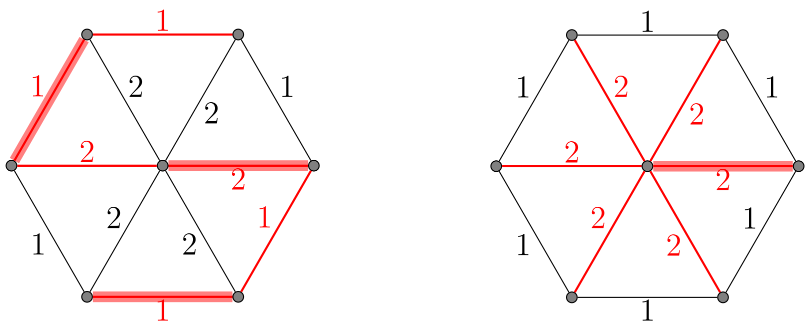
To analyze the overall running time of such a scheme assume that the edges are sorted initially and that the algorithm uses k unique basic matchings, where the time to compute the ith matching is $f(i)$. Then the overall running time is $O(m \log n + (k-1)m + \Sigma_{i=1}^{k} f(i))$ where the second term is due to the dynamic programming when combining two matchings. Now if $f(i) = O(m)$ as is the case for algorithms such as Greedy, GPA, and PGA', then this reduces to $O(m \log n + km)$ and if k is also a constant, this reduces further to $O(m \log n)$.

In Section 4 we investigate how one can combine algorithms to obtain heavy matchings efficiently.

New Tree Algorithms

We next consider how one can compute extended overlay structures where it is still possible to compute an optimal matching in linear time using dynamic programming. To do so we start by considering trees. Using trees as an intermediate graph structure was first discussed in [10] but this only considered a local heuristic to build the trees. As we want to compute an overlay graph of maximum weight we instead compute a maximum weight spanning forest. Once this spanning forest has been computed one can compute the optimal matching on each tree component in linear time using dynamic programming. We label this algorithm as the Global Tree Algorithm (GTA). Using Kruskal's algorithm with a fast union-find data structure, GTA has time complexity $O(n + m \cdot \alpha(m))$ where α is the inverse Ackermann function.

We next show how GTA can be extended by including more edges in the intermediate graph while still maintaining the linear running time of the dynamic programming step. We do this in two steps. First, we allow for cycles in GTA as



(a) Intermediate graph and matching generated by GPA

(b) Intermediate graph and matching generated by GTA

Figure 1: GPA can perform better than GTA. The intermediate graph is given by thin red edges (—) while the matching is given by the thick red edges (▬).

well. Thus, when an edge changes a path into an even length cycle, this edge is added into the intermediate graph. Once a cycle has been created it is not inspected further. Clearly this does not change the time complexity of the ensuing dynamic programming. The second extension, GTA*, also allows an even cycle to be included into the tree as a leaf. If a cycle has already been built, an edge can be added to the intermediate graph if it connects this cycle with a tree. The cycle then becomes a leaf in the tree. After this, the leaf is again not inspected any further. We note that this algorithm does not handle all possible cases of creating maximal greedy intermediate graphs. For example, it is possible to create a cycle as a leaf by connecting two suitable arms of the same tree. But this is hard to check in constant time. Therefore, this and other similar cases are ignored.

To see that the dynamic programming can still be done in linear time consider a cycle $C = v_0, v_1, \ldots, v_k, v_0$ connected to a tree through the vertex v_0. We then need to compute two values for C. First, the regular optimal matching for C. If this does not leave v_0 unmatched then we need to compute the best matching where v_0 is unmatched. This is achieved by computing the best matching on the path $v_1, v_2, \ldots, v_k$. In this way when we start the dynamic programming on the tree, v can be treated as a regular internal node where we have two values, the optimal value when v is not matched with any of its descendants and the best value when it is allowed to match with a descendant.

Even if the intermediate graph in GTA is always greater than in GPA, the resulting matching prior post-processing may not always be greater. Figure 1 illustrates this. A resort is to split GTA* into two phases. In the first phase, only a maximal degree of two is allowed in the overlay graph and the restriction to even cycles is kept. Thereby, we get the exact same structure as from GPA. In the second phase the restriction of degree two is dropped, the paths are extended to trees, and the cycles can again be integrated into the trees as leaves. This guarantees that the generated matching is always at least as heavy as the matching resulting from GPA. We label this approach as GTA^{2*}.

As to the approximation factor of these algorithms we note that the example in Figure 1 can easily be modified to show that GTA gives an arbitrarily low approximation factor. To do this we first increase the weight of each edge on the perimeter to $2 - \epsilon$. This will not influence the final matchings, but the weight of the solution given by GPA is now $6 - 2\epsilon$ while GTA still gives a solution of weight 2.

This result can be generalized by replacing an edge (x,y) on the perimeter with a new vertex z and two new edges (x,z) and (z,y) each of weight $2-\epsilon$. We also add a new edge from the center vertex to z of weight 2. For this instance the weight of the solution given by GTA will remain fixed at 2, while the weight of the optimal solution will be proportional to the number of new edges inserted. This result also applies to GTA*.

Since GTA^{2*} gives a solution of weight no less than that given by GPA, it is clear that GTA^{2*} does have an approximation factor of 0.5. Finally we note that the matching resulting from a merge operation is always of weight at least as large as the maximum of the two initial matchings. Thus any approximation guarantee that applies to any of the initial matchings also applies to the final matching.

4 Experiments

In the following we describe experiments performed to test the algorithms and to explore the possibilities of combinations of different algorithms in terms of running time and quality of the matchings.

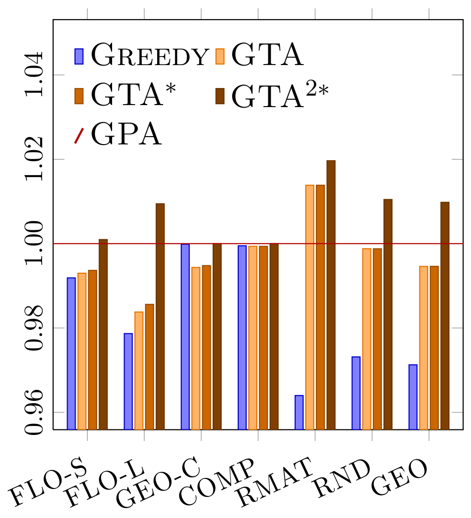
The tests were run on a GNU Linux machine with 128 GB memory using one 2.00 GHz Intel Xeon E7-4850 processor. All algorithms were implemented in C and compiled with `gcc` using the `-O3` flag. Arrays are used as the basic data structure. The sorting is performed by the default `qsort` routine in C.

Different data sets are used as test graphs. As real world instances, we use matrices from the Florida Sparse Matrix Collection [13]. We consider two sets, one with 121 smaller matrices, also used in [9, 14], and one set of larger matrices, similar to [9]. We label these as FLO-S and FLO-L, respectively. Additionally, we have data sets for random graphs (RND), random geometric graphs (GEO), complete random graphs (COMP), and complete geometric random graphs (GEO-C). Also, we consider graphs generated by the RMAT algorithm [15] (RMAT). All these data sets are similar to the ones used in [9].

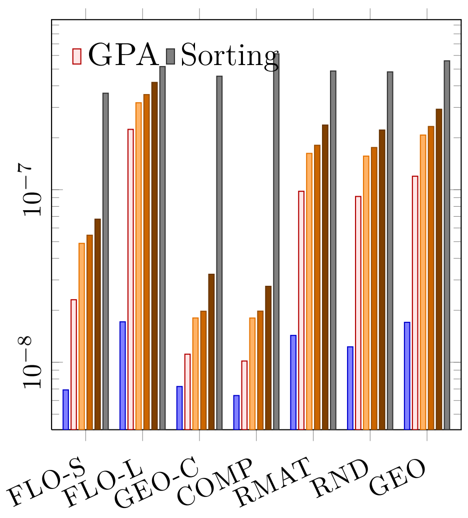
Throughout the experiments the differences between the quality of the solutions is in most cases small. Still, as described in [8], the relative difference between a particular solution and the optimal solution, known as the *gap to optimality*, can still vary considerably between the algorithms. In [8] it was shown that GPA+ROMA gave a gap to optimality that was in many cases more than 50% lower than that of only using GPA.

It is far from trivial to implement an optimal weighted matching algorithm. The only publicly available algorithm for this problem that we were able to to find was the one in the LEMON package [16]. But as our own solutions in several cases were better than those given by LEMON, we instead report performance relative to GPA. We note that it is straightforward to check if a given matching is correct or not, thus we are confident that our own solutions are legal matchings.

We first compare the performance of five different basic algorithms, namely Greedy, GPA, GTA, GTA*, and GTA^{2*}. Except for Greedy, all algorithms use a greedy post processing. Figures 2a and 2b show the relative weight of the matchings relative to GPA and average running times respectively. This is done separately for each data set. Note that the time for sorting the edges is given separately in Fig. 2b. Thus to get the total time of each algorithm one has to add in this time. On average GTA^{2*} gives the heaviest matchings for all sets of graphs among the tested algorithms, while GTA, GTA*, and Greedy yields worse matchings than



(a) Relative weight of the matching with respect to GPA ($\omega(M)/\omega(M_{GPA})$).



(b) Running time per edge in seconds (logarithmic scale).

Figure 2: Comparison of the basic algorithms with regard to results of the matching and running time per edge for different data sets.

$\frac{\omega(M_1 \oplus M_2)}{\omega(M_{GPA})} - 1$ in %		M_2: $\emptyset$	GPA	GTA	GTA*	GTA^{2*}
M_1	Greedy	-0.8127	0.2452	0.3235	0.3274	0.3121
	GPA	0		0.4083	0.4082	0.1041
	GTA	-0.7010			-0.6203	0.4460
	GTA*	-0.6346				0.4460

Table 1: Performance of $M_1 \oplus M_2$ compared to GPA on FLO-S

GPA or GTA^{2*} for most of the graph sets. An exception is RMAT for which both GTA and GTA* give better results than GPA. On both complete graph sets, all algorithms give very similar results.

Regarding running time, sorting is the most time consuming step. Algorithms using a tree structure are more time consuming than GPA, which only uses paths and cycles. Greedy is, as expected, the fastest among the tested algorithms.

We next evaluate different post processing methods. In addition to Greedy, we consider reevaluation of the remaining edges by the same algorithm either a constant number of times or as long as there are remaining edges. This showed that the advantage of running the same algorithm multiple times was negligible compared to Greedy. Still, post processing should not be skipped completely and we therefore use Greedy for post processing onwards.

We now look at how we can use the merge operation to develop more elaborate algorithms. We start this by considering the merging of only two matchings. For this purpose tables 1 and 2 shows the results for computing $M_1 \oplus M_2$ and $M_1 \oslash M_2$ respectively for all possible combinations of algorithms on the FLO-S matrices. The results are averaged for each graph class and given relative to the weight of the solution from GPA.

The tests on merging independent matchings show that the quality of the matching depends on the distinction of the combined algorithms. If they are likely to generate

$\frac{\omega(M_1 \oslash M_2)}{\omega(M_{GPA})} - 1$ in %		M_2 for $G' = (V, E \setminus M_1, \omega)$					
		$\emptyset$	Greedy	GPA	GTA	GTA*	GTA^{2*}
	Greedy	-0.8127	-0.1726	-0.0035	-0.0204	-0.0198	0.0427
	GPA	0	0.0020	0.2351	0.2753	0.2762	0.2707
M_1	GTA	-0.7010	0.0727	0.1224	-0.3257	-0.2932	0.1541
	GTA*	-0.6346	0.0914	0.1388	-0.2752	-0.2654	0.1710
	GTA^{2*}	0.0963	0.1050	0.2906	0.3296	0.3298	0.3119

Table 2: Performance of $M_1 \oslash M_2$ compared to GPA on FLO-S.

very similar matchings, e. g. GPA and GTA^{2*}, or GTA and GTA*, the combination gives only a small improvement. On the other hand, if the concept of the algorithms is more different, for example Greedy and GTA, then merging gives a larger improvement. When it comes to computing an augmenting matching, the picture is less clear, but on average the improvement of augmenting matchings computed by GTA^{2*} are slightly heavier. On the other hand, a run of GTA^{2*} take 2–5 times the running time of GPA.

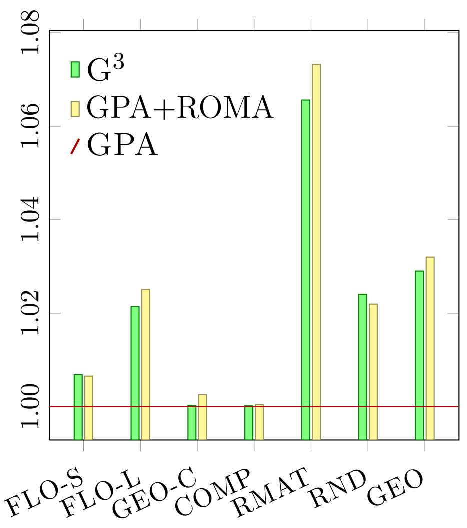
When exploring more involved combinations of algorithms, we have to look at combinations using more than two executions of a matching algorithm. But even if we restrict ourselves to at most three executions, the number of possibilities gets so large that it is hardly feasible to test every combination. Thus we base our algorithms on the observations made for the basic combinations.

Our intention is then to build a reasonable combination which includes independent matchings of Greedy, and one of GTA or GTA*, and one of GPA or GTA^{2*}. To investigate if the order we combine the algorithms influences the final result we computed $((M_1 \oplus M_2) \oplus M_3)$ and $(M_1 \oplus (M_2 \oplus M_3))$ for every way of picking one algorithm from each of $M_1 = \{\text{Greedy}\}$, $M_2 = \{\text{GPA}, \text{GTA}^{2*}\}$, and $M_3 = \{\text{GTA}, \text{GTA}^*\}$. The results ranged from 0.5201% up to 0.5585% improvement over GPA. Thus all combinations were better than the best solution when using only two matchings. Still, the differences in quality for different combination orders and for different algorithms was small. Among the possible choices the use of GTA^{2*} over GPA gave the most significant improvement.

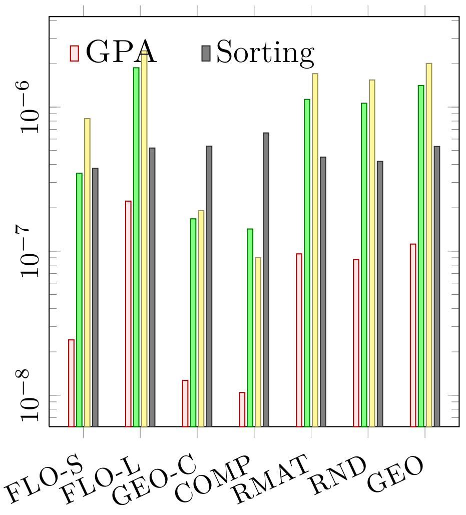
We next consider how to build a combination of independent and augmenting matchings such that we get the best matching but also maintain a reasonable running time of the total algorithm. As independent matchings we use Greedy, GTA^{2*}, and GTA* as the combination of these performed well. Again, by adding augmenting matchings the number of possible combinations grows fast. Therefore, we use only augmenting matchings generated by GTA^{2*} and consider two basic possibilities: First, to compute and apply one augmenting matching after the three independent matchings are combined, while the second possibility is to compute and apply an augmenting matching for each independent matching.

When not applying any augmentations, the average relative performance compared to GPA on FLO-S graphs was 0.5585% higher, this increased to 0.6994% when applying all four augmentations. However, as we must balance the execution time with the obtained quality, we chose to only use augmentation for each of the initial matchings giving a relative improvement of 0.6838% compared to GPA. Thus the final algorithm, labeled G^3, can be described as follows:

$$(M_{\text{Greedy}} \oslash M_{\text{GTA}^{2*}}) \oplus ((M_{\text{GTA}^*} \oslash M_{\text{GTA}^{2*}}) \oplus (M_{\text{GTA}^{2*}} \oslash M_{\text{GTA}^{2*}}))$$



(a) Relative weight of the matching with respect to GPA ($\omega(M)/\omega(M_{GPA})$).



(b) Running time per edge in seconds (logarithmic scale).

Figure 3: Comparison of G^3 to GPA+ROMA with regard to results of the matching and running time per edge for different data sets.

As discussed in Section 3 it is clear that the asymptotic running time of G^3 is $O(m \log n)$ due to the sorting of the edge list.

We next compare G^3 with the combined algorithm of GPA+ROMA. Similar to [8] the post processing of GPA is done by two runs of GPA on the remaining edges. Figures 3a and 3b show the average quality of the matching and running time respectively for each graph class.

The results show that in terms of quality the two algorithms are very similar, although GPA+ROMA is on average the best. G^3 gives an improvement over GPA of up to 6.5% with an average of 2.1%. The corresponding numbers for GPA+ROMA are 7.3% and 2.3%. When excluding the time spent on sorting, the running time of G^3 is between 8 and 15 times higher than that of GPA with an average of 12.3. The running time of GPA+ROMA is between 8 and 34 times higher than GPA with an average of 17.5. On average GPA+ROMA uses 42% more time than G^3. The relative low time for GPA+ROMA on the complete graphs is because ROMA will stop as soon as it cannot improve the matching any further.

5 Conclusion

We have presented a framework for constructing fast matching approximation algorithms and heuristics. Through careful experiments we designed our final algorithm G^3 which we showed to run faster than the best randomized approximation algorithm while giving solutions of comparable quality.

In [9] it was shown how Greedy and the merge operation could be parallelized. Thus a natural question is to ask if the same is possible for GPA or any of the tree algorithms. It would also be of interest to determine if it is possible to improve the approximation factor beyond $\frac{1}{2}$ by using the merge operation. We suspect that this is not the case.

References

[1] E. J. Riedy, H. Meyerhenke, D. Ediger, and D. A. Bader, “Parallel community detection for massive graphs,” in *Graph Partitioning and Graph Clustering*, 2012, pp. 207–222.

[2] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar, “Multithreaded clustering for multi-level hypergraph partitioning,” in *IPDPS*, 2012, pp. 848–859.

[3] M. Holtgrewe, P. Sanders, and C. Schulz, “Engineering a scalable high quality graph partitioner,” in *IPDPS*, 2010, pp. 1–12.

[4] A. M. Khan, D. F. Gleich, A. Pothen, and M. Halappanavar, “A multithreaded algorithm for network alignment via approximate matching,” in *SC*, 2012, p. 64.

[5] H. N. Gabow, “Data structures for weighted matching and nearest common ancestors with linking,” in *SODA'90*. SIAM, 1990, pp. 434–443.

[6] D. E. Drake and S. Hougardy, “A simple approximation algorithm for the weighted matching problem,” *Inf. Proc. Let.*, vol. 85, pp. 211–213, 2003.

[7] ——, “Linear time local improvements for weighted matchings in graphs,” in *WEA'03*, vol. 2647. LNCS, Springer, 2003, pp. 107–119.

[8] J. Maue and P. Sanders, “Engineering algorithms for approximate weighted matching,” in *WEA'07*. LNCS, Springer, 2007, pp. 242–255.

[9] F. Manne and M. Halappanavar, “New effective multithreaded matching algorithms,” To appear in the proceedings of IPDPS 2014, available at http://www.ii.uib.no/~fredrikm/ipdps2014.pdf, 2014.

[10] M. Birn, “Engineering Fast Parallel Matching Algorithms,” diploma thesis, Karlsruhe Institute of Technology, 2012.

[11] S. Pettie and P. Sanders, “A simpler linear time $2/3 - \epsilon$ approximation for maximum weight matching,” *Inf. Process. Lett.*, vol. 91, no. 6, pp. 271–276, 2004.

[12] R. Duan and S. Pettie, “Linear-time approximation for maximum weight matching,” *J. ACM*, vol. 61, no. 1, pp. 1:1–1:23, Jan. 2014.

[13] T. A. Davis and Y. Hu, “The University of Florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[14] M. Birn, V. Osipov, P. Sanders, C. Schulz, and N. Sitchinava, “Efficient parallel and external matching,” in *EuroPar'13*, vol. 8097. LNCS, Springer, 2013, pp. 659–670.

[15] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-mat: A recursive model for graph mining,” Proc. SIAM Intl. Conf. on Data Mining, 2004.

[16] “Lemon graph library,” http://lemon.cs.elte.hu, 2013.