Engineering Algorithms for Approximate Weighted Matching*

Jens Maue[1] and Peter Sanders[2]

[1] ETH Zürich, Institute of Theoretical Computer Science, Universitätsstrasse 6, 8092 Zürich, Switzerland, jens.maue@inf.ethz.ch
[2] Universität Karlsruhe (TH), Fakultät für Informatik, Postfach 6980, 76128 Karlsruhe, Germany, sanders@ira.uka.de

Abstract. We present a systematic study of approximation algorithms for the maximum weight matching problem. This includes a new algorithm which provides the simple greedy method with a recent path heuristic. Surprisingly, this quite simple algorithm performs very well, both in terms of running time and solution quality, and, though some other methods have a better theoretical performance, it ranks among the best algorithms.

1 Introduction

Given a graph $G = (V, E)$ with $n := |V|$ nodes and $m := |E|$ edges, a subset of edges $M \subseteq E$ is called a *matching* if no two members of M share an endpoint. A node $u \in V$ is called *matched* if there is an edge $(u, v) \in M$; then, (u, v) is called a *matching* edge and v the *mate* of u. Otherwise, u is called *unmatched* or *free*. If G is weighted and $w\colon E \to \mathbb{R}^{\geq 0}$ denotes the associated weight function, the *weight* of M is defined by $w(M) := \sum_{e \in E} w(e)$, and M is said to be a *maximum weight matching* if there is no matching with larger weight.

The first polynomial time algorithm for the weighted matching problem was given by Edmonds [1] with a running time of $\mathcal{O}(n^2 m)$. This has been improved on repeatedly, and the fastest exact algorithm known so far has an asymptotic running time of $\mathcal{O}(n(m + n \log n))$ for general graphs [2]. Further improvements have been achieved for restricted problems and graph classes such as integer edge weights [3], planar graphs [4], or maximum cardinality matching [5, 6].

One of the most recent implementations of the maximum weighted matching problem is that of Mehlhorn and Schäfer [7], which is a variant of the algorithm of Galil, Micali, and Gabow [8]. A history of implementations can be found in [9]. Despite their polynomial running time, these algorithms are too slow for some practical applications on very large graphs or if matchings are computed repeatedly. (Some applications are mentioned in [10].) This motivates the use of approximation algorithms for the maximum weighted matching problem, with a lower—ideally linear—running time that yield very good results.

* Part of this work was done at Max-Planck-Institut für Informatik, Saarbrücken, Germany. Partially supported by DFG grants SA 933/1-2, SA 933/1-3.

Related Work. A well known folklore algorithm achieves a $\frac{1}{2}$-approximation by scanning the edges in descending order of their weights and greedily adding edges between free nodes to the matching [11]. Note that this yields a linear running time for integer edge weights. The first linear time $\frac{1}{2}$-approximation algorithm independent of integer sorting was given by Preis [12]. Later, Drake and Hougardy [13] presented a different $\frac{1}{2}$-approximation, called the Path Growing Algorithm (PGA), whose linear running time is easier to prove. They also developed a $(\frac{2}{3}-\epsilon)$-algorithm [10, 14] running in time $\mathcal{O}(\frac{n}{\epsilon})$. A simpler $(\frac{2}{3}-\epsilon)$-approximation algorithm running in time $\mathcal{O}(n \log \frac{1}{\epsilon})$ was later developed by Pettie and Sanders [15]. Several $\frac{1}{2}$-approximation algorithms were evaluated experimentally in [14]. This included a version of PGA improved by a path heuristic not affecting the linear running time; this version of PGA is called PGA$'$. This algorithm performed very well in the experiments, so the greedy algorithm with its super-linear running time was concluded to be a bad choice in most cases.

Our Contribution. As a seemingly trivial yet crucial contribution, we regard the measured solution qualities of the approximation algorithms in relation to the optimal solution. This not only demonstrates that these algorithms achieve solutions of a quality much better than their worst case guarantees suggest, it also makes comparing the algorithms with each other more meaningful. Although the solution quality of the algorithms differ by only a few percent, the *gap to optimality* can differ by a large factor. We also give the first experiments for real-world inputs from an application that is often cited as a main motivation for applying approximation algorithms for weighted matching.

Our most important algorithmic contribution is a rehabilitation of the greedy approach through a new algorithm called GPA, which is described in Sect. 2 in detail. Basically, GPA applies the earlier mentioned path heuristic used for PGA$'$ [14] to the greedy algorithm, which makes the previously outclassed greedy method jump ahead to one of the best practical algorithms. In particular, GPA is usually *faster* than the randomized $(\frac{2}{3}-\epsilon)$-approximation algorithm (RAMA) from [15] and often outperforms it in terms of solution quality. The overall winner is another new algorithm that first executes GPA and then applies a modified version of RAMA, called ROMA. This combination is as fast as ROMA alone since GPA accelerates the convergence of ROMA.

Outline. The tested approximation algorithms are described in Sect. 2. Section 3 introduces the graph instances, on which the tests are performed as presented in Sect. 4. The main results are summarized in Sect. 5.

2 Algorithms

Greedy. The well-known algorithm shown in Fig. 1 follows a simple greedy strategy [11]: repeatedly, the currently heaviest non-matching edge with free endpoints is added to the matching until no edges are left.

```
GRDY(G = (V, E), w : E → ℝ^≥0)
1  M := ∅
2  while E ≠ ∅ do
3     let e be the edge with biggest weight in E
4     add e to M
5     remove e and all edges adjacent to its endpoints from E
6  return M
```

Fig. 1. The greedy algorithm for approximate weighted matchings.

The greedy algorithm runs in time $\mathcal{O}(m + \text{sort}(m))$, where $\text{sort}(m)$ denotes the time for sorting m items. This yields a linear running time for integer edge weights and $\mathcal{O}(m \log n)$ for comparison-based sorting. Let $M^\star$ be a maximum weight matching and M a matching found by the greedy algorithm. Every time an edge e is added to M, at most two edges $e_1, e_2 \in M^\star$ are removed from the graph. Since both $w(e) \geq w(e_1)$ and $w(e) \geq w(e_2)$, the greedy matching M satisfies $2w(M) \geq w(M^\star)$, so the performance ratio is $\frac{1}{2}$.

```
PGA′(G = (V, E), w : E → ℝ^≥0)
1  M := ∅
2  while E ≠ ∅ do
3     P := ⟨⟩
4     arbitrarily choose v ∈ V with deg(v) > 0
5     while deg(v) > 0 do
6        let e = (v, u) be the heaviest edge adjacent to v
7        append e to P
8        remove v and its adjacent edges from G
9        v := u
10    M := M ∪ MaxWeightMatching(P)
11 extend M to a maximal matching
12 return M
```

Fig. 2. The improved Path Growing Algorithm PGA′.

PGA′. The improved version of the Path Growing Algorithm by Drake and Hougardy is referred to by PGA′ [14]. The original algorithm (PGA) [13] without improvements first grows a set of node disjoint paths one after another. Each path is built starting from an arbitrary free node, repeatedly extending it along the heaviest edge $e = (u, v)$ adjacent to its current endpoint u with a free opposite endpoint v and deleting all edges adjacent to u. PGA starts a new path if the current one cannot be extended and finishes when no edges are left. While growing the paths, selected edges are alternately added to two different matchings M_1 and M_2, and the heavier one is finally returned.

The algorithm processes each edge at most once and thus has a linear running time of $\mathcal{O}(m)$. The algorithm yields an approximation ratio of $\frac{1}{2}$, which can be shown by assigning every edge to the end node from which it was deleted during a run of the algorithm. Then, for every edge e of a maximum weight matching M, there is an edge $e' \in M_1 \cup M_2$ with $w(e') \geq w(e)$ which is adjacent to the node that e is assigned to. Therefore, $w(M_1 \cup M_2) \geq w(M)$ and the heavier set of M_1 and M_2 has a weight of at least $\frac{1}{2}w(M)$.

The improved version (PGA′) of this algorithm is shown in Fig. 2. Instead of alternately adding the edges of a path to the two matchings, PGA′ calculates an optimal matching for every path. This process does not affect the asymptotic running time since computing a maximum weight matching of a path by dynamic programming requires a linear time in the length of the path, which is described at the end of this section. Furthermore, the contribution of a path to the final matching can only increase in weight compared to the original algorithm, so the approximation ratio of $\frac{1}{2}$ is not impaired. As the second improvement, the computed matching is extended to a maximal matching at the end of PGA′, which is done by just scanning all edges once without increasing the running time of $\mathcal{O}(m)$.

```
GPA(G = (V, E), w : E → ℝ^{≥0})
1 M := ∅
2 E' := ∅
3 for each edge e ∈ E in descending order of their weight do
4    if e is applicable then add e to E'
5 for each path or cycle P in E' do
6    M' := MaxWeightMatching(P)
7    M := M ∪ M'
8 return M
```

Fig. 3. The Global Paths Algorithm GPA.

GPA. Our Global Paths Algorithm (GPA) shown in Fig. 3 presents a new approximation method by integrating the greedy algorithm and PGA′. GPA generates a maximal weight set of paths and even length cycles and then calculates a maximum weight matching for each of them by dynamic programming. These paths initially contains no edges and hence represent n trivial paths—isolated nodes. The set is then extended by successively adding applicable edges in descending order of their weight. An edge is called *applicable* if it connects two endpoints of different paths or the two endpoints of an odd length path. An edge is not applicable if it closes an odd length cycle, or if it is incident to an inner node of an existing path. Once all edges are scanned, a maximum weight matching is calculated for each path and cycle.

The test whether an edge is applicable can be done in constant time, and growing the set of paths and cycles needs a linear number of such tests. Calculating a maximum weight matching for a given path needs linear time in the length of the path using the dynamic programming approach already mentioned above and described below. Thus, GPA needs a linear amount of time after the edges have been sorted, so GPA has a running time of $\mathcal{O}(\operatorname{sort}(m) + m)$, which is $\mathcal{O}(m \log n)$ in general.

Let $M^\star$ be an optimum matching and $P \subset E$ the set of edges that are added to some path or cycle in the first round of GPA. For every $e \in M^\star \setminus P$, e was not applicable at the time considered by GPA, so there are two edges $e_1, e_2 \in P \setminus M^\star$ adjacent to e with both $w(e_1) \geq w(e)$ and $w(e_2) \geq w(e)$. Conversely, every member of P is adjacent to at most two edges of $M^\star$, so there is an injective mapping $f\colon M^\star \to P$ with $w(f(e)) \geq w(e)$, which implies $w(P) \geq w(M^\star)$. Moreover, the maximum weight matching of a path has a weight of at least half the weight of the path—which also holds for even length cycles. Therefore, any matching M computed in the first round of GPA satisfies $w(M) \geq \frac{1}{2} w(P) \geq \frac{1}{2} w(M^\star)$, so GPA has a performace ratio of $\frac{1}{2}$. This ratio is tight as Fig. 4 shows: in the example graph GPA finds a matching with a weight of not more than $\frac{m}{4}(c + \epsilon) = \frac{1}{2} w_{\text{opt}} + \epsilon'$.

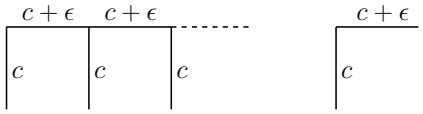

Fig. 4. This graph with a maximum weight matching of weight $w_{\text{opt}} = \frac{m}{2} c$ shows that the approximation ratio of $\frac{1}{2}$ is tight for the greedy algorithm, PGA′, and GPA.

A second round of GPA can be run on the set of remaining edges with two unmatched endpoints after GPA finishes. We even allow up to three rounds, but the algorithm almost never runs the third round in the experiments presented in Sect. 4 since no applicable edges are left. Running GPA a second time on the result produced by itself presents an alternative to the postprocessing used in PGA′, which extends the computed matching to a maximal matching by simply collecting edges with two free endpoints. The postprocessing of PGA′ could clearly be applied to GPA too.

RAMA. A path P is *alternating* if it consists of edges drawn alternately from M and $E \setminus M$. An alternating path P is called an *augmentation* if the symmetric difference $M \oplus P = (M \setminus P) \cup (P \setminus M)$ is a matching too and $w(M \setminus P) > w(P \setminus M)$. The *gain* of an alternating path P is defined by $g(P) = w(P \setminus M) - w(P \cap M)$. A *$k$-augmentation* is an augmentation with at most k non-matching edges, and a 2-augmentation P is called *centered* at v if all edges of $P \setminus M$ are incident to either v or its mate. Finally, a maximum-gain 2-augmentation centered at v is denoted by $\operatorname{aug}(v)$.

```
RAMA(G = (V, E), w: E → ℝ^≥0, int k)
1 M := ∅ (or initialise M with any matching)
2 for i := 1 to k do
3    randomly choose v ∈ V
4    M := M ⊕ aug(v)
5 return M
```

Fig. 5. The Random Augmentation Matching Algorithm RAMA.

Figure 5 shows a randomized algorithm by Pettie and Sanders [15]. It repeatedly chooses a random node and augments the current matching with the highest-gain 2-augmentation centered at that node. The way how to find this highest-gain 2-augmentation—according to which the algorithm is implemented—is described in [15]. All this is iterated k times, and by putting $k := \frac{1}{3}\, n \log \frac{1}{\epsilon}$ the algorithm has an expected running time of $O(m \log \frac{1}{\epsilon})$ and an expected performance ratio of $\frac{2}{3} - \epsilon$ [15].

```
ROMA(G = (V, E), w: E → ℝ^≥0, int ℓ)
1 M := ∅ (or initialise M with any matching)
2 for i := 1 to ℓ do
3    for each node v ∈ V in random order do
4       M := M ⊕ aug(v)
5 return M
```

Fig. 6. The Random Order Augmentation Matching Algorithm ROMA.

ROMA. The new Random Order Augmentation Matching Algorithm (ROMA) shown in Fig. 6 presents a variant of RAMA from above. The algorithm operates in phases whose number is denoted by ℓ (in Fig. 6, lines 3–4 together form one phase): in every phase, the algorithm successively selects all nodes in random order, and the current matching is repeatedly augmented with the highest-gain 2-augmentation centered at the node currently selected. As described in [15], the time required to find the highest-gain 2-augmentation $\mathrm{aug}(v)$ centered at v is $\mathcal{O}(\deg v + \deg(\mathrm{mate}(v)))$, so the time averaged over all nodes required for one phase of n iterations is $\mathcal{O}(m)$. By setting the number of phases properly, running time and performance ratio of ROMA correspond to those of RAMA.

The randomized algorithms can be initialized with the empty matching or any (non-empty) matching found by one of the algorithms GRDY, PGA′, or GPA. Several combinations are tested in Sect. 4. A similar approach is followed in [14], where a maximal set of node disjoint 2-augmentations is calculated in linear time to improve a previously computed matching.

```
MaxWeightMatching(P = ⟨e_1, ..., e_k⟩)
 1 W[0] := 0;   W[1] := w(e_1)
 2 M[0] := ∅;   M[1] := {e_1}
 3 for i := 2 to k do
 4   if w(e_i) + W[i − 2] > W[i − 1] then
 5     W[i] := w(e_i) + W[i − 2]
 6     M[i] := M[i − 2] ∪ {e_i}
 7   else
 8     W[i] := W[i − 1]
 9     M[i] := M[i − 1]
10 return M[k]
```

Fig. 7. Obtaining a maximum weight matching for a path by dynamic programming.

Maximum Weighted Matching for Paths and Cycles. Some of the algorithms described above compute maximum weighted matchings for paths or cycles, which is done using the dynamic programming algorithm shown in Fig. 7. In the description, $P := \langle e_1, \ldots, e_k \rangle$ denotes a solution of the subproblem for the path $\langle e_1, \ldots, e_i \rangle$, and $W[i]$ denotes the corresponding weight.

The algorithm has a running time which is linear in the length of the input path. In order to compute the maximum weight matching of a cycle C, let e_1 and e_2 be any two consecutive edges of C, let P_1, P_2 be the paths obtained by removing e_1, e_2 from C respectively, and let M_1 and M_2 denote the maximum weight matching of P_1 and P_2 respectively. Then, both M_1 and M_2 are valid matchings for C, and the one of larger weight must be a maximum weight matching for C since not both e_1 and e_2 can be member of a maximum weight matching of C. Thus, maximum weight matchings for cycles can be found in linear time of their length too.

3 Test Instances

Three families of (synthetic) instances taken from [7] are presented below, followed by a description of the real-world graphs. We used the C++ library LEDA 4.5 [16] for some steps of the generation process as described below.

Delaunay Instances are created by randomly choosing n points in the unit square and computing their Delaunay triangulation using LEDA. The numbers of points n are chosen to be $n := 2^x$ with $x \in \{10, \ldots, 18\}$, giving nine different Delaunay graphs in total. Their edge weights are obtained by scaling the Euclidean distances in the unit square to integers in the range between 0 and 2^{31-x}.

Complete Geometric Instances are generated each by choosing n random points in an $(n \times n)$-square. n is set to $n := 2^x$, now with $x \in \{6, \ldots, 12\}$, yielding seven instances. The edge weights correspond to the Euclidean distances between their endpoints, rounded off to integer values.

Random Instances are generated with the LEDA implementation for random simple undirected graphs. We generate 64 graphs by choosing eight different values for n and eight different values of m for each n: for every $n := 2^x$, $x \in \{10, \ldots, 17\}$, eight random graphs are created by putting $m := 2^y\, n$, $y \in \{1, \ldots, 8\}$. The edges are assigned random integer weights between 0 and 2^{31-x}.

Real-World Instances. Some of the best known practical graph partitioning algorithms reduce the size of their input graph by successively contracting them in the followin way: starting with unit edge weights, an approximate weighted matching is computed in each contraction for the current graph using PGA′ [14]. Then, every matching edge is contracted into a single node. Parallel edges resulting from a contraction are replaced by a single edge whose weight is the sum of the weights of its constituent edges. The initial 34 graphs are taken from [17], for each of which eight contractions are successively applied, yielding 272 instances with integer edge weights.

4 Experimental Results

The algorithms described in Sect. 2 are run on the instances presented above and compared by their solution quality and running times. The RAMA and ROMA algorithms are further tested in combination with initial matchings obtained by the other algorithms. The algorithms were implemented using the data structure 'graph' and the sorting algorithms of the C++-library LEDA-4.5 [16] and compiled with the GNU C++-compiler g++-3.2.

All tested algorithms actually perform much better than their worst case bounds suggest, and even the worst approximate solution in the tests has a relative error of less than 10 %. Still, the measured error, i.e. the gap to optimality, differs by a large factor between the best and worst algorithm, and whether some algorithm performs better than another depends on the graph family. In this section, the ratio between the number of nodes and edges of a graph will be denoted by α, i.e. $\alpha := \frac{m}{n}$.

The presented running times should not be over-interpreted since the algorithms could probably be implemented more efficiently. In particular, using fast integer sorting would probably accelerate GPA considerably. Still, we believe that the plotted running times are meaningful to some degree since in a sense all the implementations are "of the same quality" and they all use LEDA.

4.1 Solution Quality and Running Times

Figures 8 and 9 show the results for synthetic instances. For the random graphs Fig. 8 contains two diagrams for each $\alpha \in \{4, 16, 64, 256\}$. The x-axis shows the number of nodes n. The y-axis shows the difference of an algorithm's approximate solution to the optimum in percent (left column) and the corresponding running times (right) respectively. The randomized algorithms RAMA and ROMA perform up to $8n$ augmentation steps. ROMA cancels if its current matching is *saturated*, i.e. if one phase of n iterations has not further improved it.

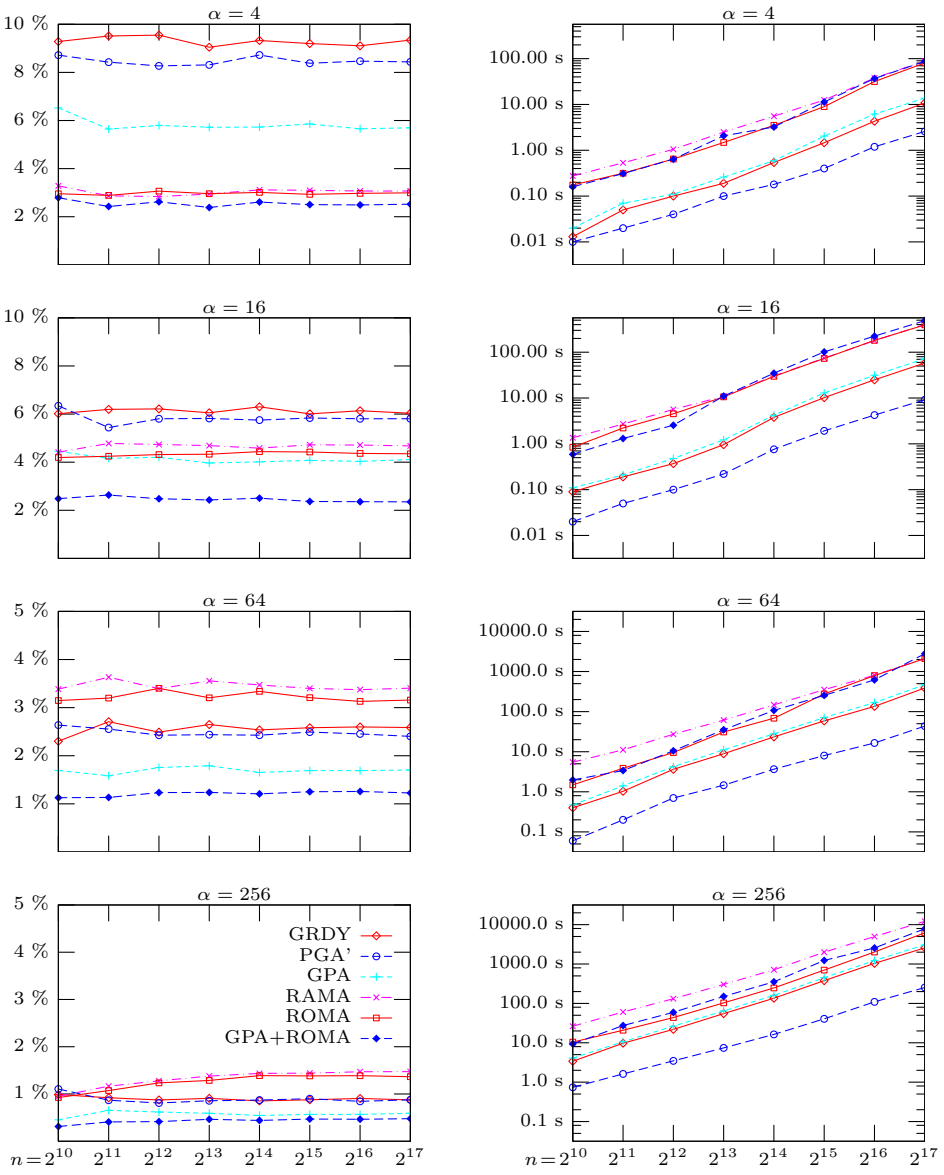

Fig. 8. Gap to optimality (left) and running times (right) for random graphs with $n = 2^{10}, \ldots, 2^{17}$ and $\alpha \in \{4, 16, 64, 256\}$. The key applies to all plots.

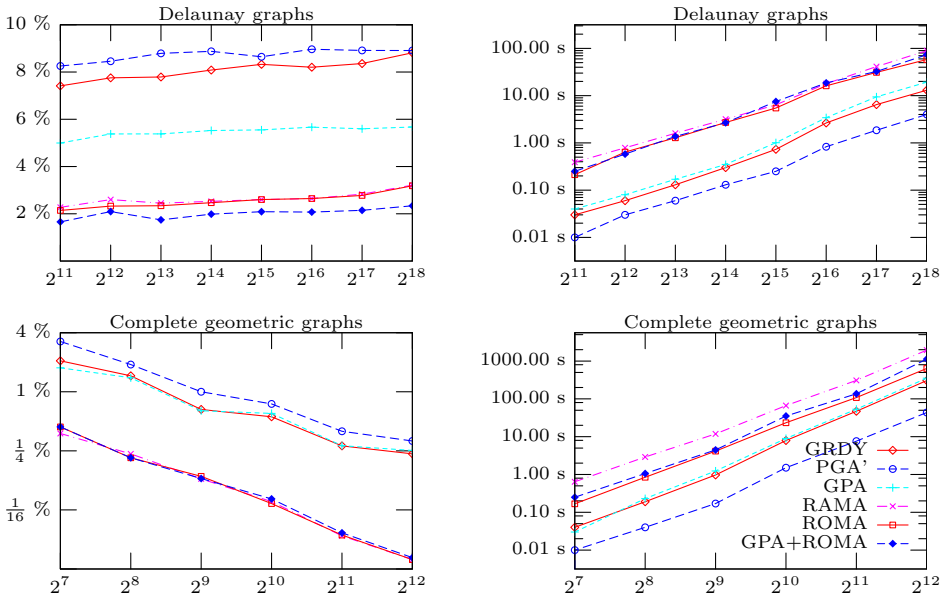

Fig. 9. Gap to optimality (left) and running times (right) for Delaunay (top) and complete geometric (bottom) graphs. The key applies to all plots.

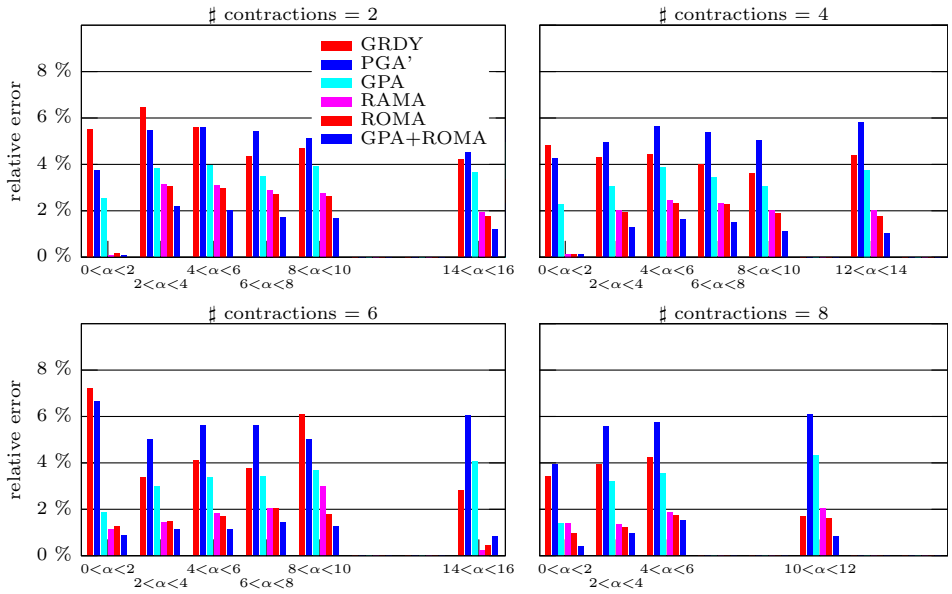

Fig. 10. Gap to optimality for real-world graphs. Each plot corresponds to a class of graphs created by the same number of contractions and shows the average solution quality, where graphs with a similar value of α are combined.

Figure 8 shows that for random graphs of any value of α, GPA performs significantly better than the greedy algorithm and PGA$'$. The randomized algorithms RAMA and ROMA produce approximations even better than GPA for $\alpha < 16$, but this advantage decreases with increasing α: their solution quality is almost equal for $\alpha = 16$, whereas RAMA and ROMA are outperformed by GPA for graphs with $\alpha > 16$, and also by the greedy algorithm and PGA$'$ for higher values of α. Moreover, GPA followed by post-processing through ROMA performs best for any value of α. The linear time algorithm PGA$'$ runs fastest in practice. Interestingly, the superlinear time algorithms greedy and GPA are *faster* than the linear time algorithms RAMA and ROMA. The reason seems to be that the constant factors involved in sorting are much smaller than the constant factors in RAMA and ROMA, which involve multiple passes over the graph involving a massive number of random accesses to the graph data structure that can cause cache faults. ROMA is generally faster than RAMA since it usually finishes earlier due to saturation. Furthermore, calculating an initial matching with GPA causes an even earlier saturation, which makes up for the additional preprocessing time.

The tested Delaunay instances have values $\alpha \approx 3$, and the solution qualities and running times are similar to those for random graphs with similar values of α (see Fig. 9). Also for the complete geometric instances, for which $\alpha = \frac{n-1}{2}$, the algorithms perform just as expected from random graphs of corresponding α. For both instance families the main difference to random graphs is that the greedy algorithm performs better than PGA$'$, but GPA combined with ROMA still produces the best results.

Figure 10 shows the solution quality for the real-world graphs, each plot corresponding to a set of graphs derived by the same number of contractions. The x-axis shows the graphs' values of α grouped by similar values. The y-axis shows the deviation from the optimal solution. The results for real-world graphs basically support what has been concluded for synthetic graphs. In particular, GPA followed by ROMA performs best in almost all cases.

4.2 Convergence of Randomized Algorithms

Figure 11 shows how RAMA and ROMA converge for the random instances. The left column of Fig. 11 contains one diagram for each $\alpha \in \{4, 16, 64, 256\}$, with the x-axis showing the number of iterations and the y-axis showing the relative error (average value over several runs) after this number of steps.

Even with an initially empty matching, i.e. without preprocessing, the randomized algorithms quickly converge on an almost saturated matching with a small error and do not show much improvement after about $4n$ steps for any α. The bigger the value of α, the better the first approximation after n steps is. Used as stand-alone methods, ROMA performs better than RAMA again. Furthermore, starting with an initial matching—obtained by any of the greedy algorithm, PGA$'$, or GPA—yields a significant improvement. GPA clearly achieves the best initialization, which can be improved up to n or $2n$ iterations of ROMA

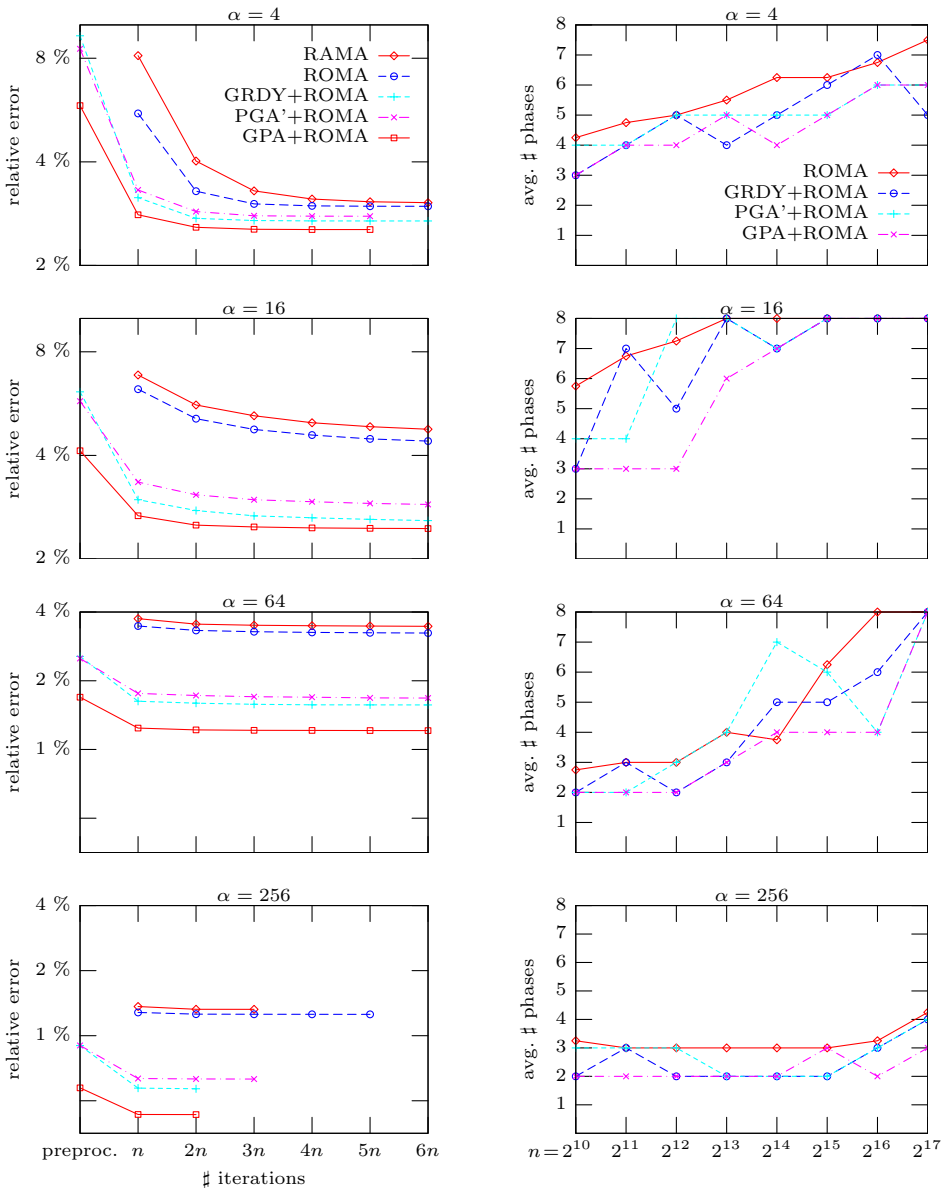

Fig. 11. Convergence (left) and average number of phases until saturation (right) of RAMA and ROMA for the random instances with $\alpha \in \{4, 16, 64, 256\}$.

depending on the instance, and the solution quality cannot be achieved with a different preprocessing method using more iterations.

The plotted lines are cut off if no improvement is achieved anymore. Since the errors are averages over several graphs and runs, this only happens if the matching is saturated in ALL sample runs for this number of iterations. However, saturation is achieved earlier on the average, and the right column of Fig. 11 describes after how many phases ROMA achieves saturation on the average.

5 Conclusion

The simple greedy algorithm, PGA′, and GPA all produce solutions of a quality much better than their lower bound guarantees. Applying the path heuristic from PGA′ to the greedy algorithm yields a very high improvement: GPA produces very good solutions at a reasonable running time, so sorting-based matching algorithms remain interesting and must not be neglected.

Moreover, the randomized algorithms also perform very well. As stand-alone methods they are superior in terms of solution quality for graphs of small ratios of $\frac{m}{n}$, and they are highly suitable to post-process existing matchings. The variant ROMA improves RAMA, and its combination with GPA shows the best experimental results among all methods tried.

Future Work. For more meaningful running time comparisons, it would be interesting to compare tuned versions of the best algorithms presented here (GPA, ROMA, and optimal). We see a considerable potential for optimization: GPA does not need complicated graph data structures, and the LEDA function `sort_edges` we currently use, which seems to be quite slow, could be replaced by integer sorting. ROMA (and RAMA) could use static graph data structures optimized for the required operation mix. More interestingly, these algorithm could also be changed in a way that only non-saturated nodes are considered as centers for augmenting paths. The main task here is how to maintain the set of candidate centers efficiently.

Since parallel processing is becoming ubiquitous, we could also consider parallelization. In GPA we can parallelize sorting and independent dynamic programming problems. We can consider possible augmentations for ROMA in parallel, but we have to be careful when actually performing an augmentation.

Another interesting question is whether the running time of exact weighted matching algorithms can be improved by calculating initial matchings with the presented approximation algorithms.

Acknowledgements. We would like to thank Seth Pettie for interesting discussions about the subject. We would also like to thank the reviewers for their detailed comments.

References

1. Edmonds, J.: Maximum matching and a polyhedron with $0,1$-vertices. J. Res. Nat. Bur. Standards 69B (1965) 125–130
2. Gabow, H.N.: Data structures for weighted matching and nearest common ancestors with linking. In: Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-90), SIAM (1990) 434–443
3. Gabow, H.N., Tarjan, R.E.: Faster scaling algorithms for general graph matching problems. Journal of the ACM **38**(4) (1991) 815–853
4. Lipton, R.J., Tarjan, R.E.: Applications of a planar separator theorem. SIAM Journal on Computing **9**(3) (1980) 615–627
5. Micali, S., Vazirani, V.V.: An $\mathcal{O}(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In: Proceedings of the 21st Annual Symposium on Foundations of Computer Science (FOCS-80), IEEE (1980) 17–27
6. Vazirani, V.V.: A theory of alternating paths and blossoms for proving correctness of the $\mathcal{O}(\sqrt{V}E)$ general graph maximum matching algorithm. Combinatorica **14**(1) (1994) 71–109
7. Mehlhorn, K., Schäfer, G.: Implementation of $O(nm \log n)$ weighted matchings in general graphs: The power of data structures. ACM Journal of Experimental Algorithms **7** (2002)
8. Galil, Z., Micali, S., Gabow, H.N.: An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. SIAM Journal on Computing **15**(1) (1986) 120–130
9. Cook, W., Rohe, A.: Computing minimum-weight perfect matchings. INFORMS Journal on Computing **11**(2) (1999) 138–148
10. Drake Vinkemeier, D.E., Hougardy, S.: A linear-time approximation algorithm for weighted matchings in graphs. ACM Trans. Algorithms **1**(1) (2005) 107–122
11. Avis, D.: A survey of heuristics for the weighted matching problem. Networks **13**(4) (1983) 475–493
12. Preis, R.: Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In: Proc. of the 16th Annual Symp. on Theoretical Aspects of Computer Science (STACS-99). Volume 1563 of LNCS., Springer (1999) 259–269
13. Drake, D.E., Hougardy, S.: A simple approximation algorithm for the weighted matching problem. Information Processing Letters **85**(4) (2003) 211–213
14. Drake, D.E., Hougardy, S.: Linear time local improvements for weighted matchings in graphs. In: Proceedings of the 2nd International Workshop on Experimental and Efficient Algorithms (WEA-03). Volume 2647 of LNCS., Springer (2003) 107–119
15. Pettie, S., Sanders, P.: A simpler linear time $2/3 - \epsilon$ approximation for maximum weight matching. Information Processing Letters **91**(6) (2004) 271–276
16. Mehlhorn, K., Näher, S.: LEDA - A Platform for Combinatorial and Geometric Computing. Cambridge University Press, Cambridge, UK (1999)
17. Soper, A.J., Walshaw, C., Cross, M.: A combined evolutionary search and multilevel optimisation approach to graph-partitioning. J. Global Optimization **29**(2) (2004) 225–241 `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`.