

Matching in Bipartite Graph Streams in a Small Number of Passes

(Extended Abstract)

Lasse Kliemann
lki@informatik.uni-kiel.de

Institut für Informatik
Christian-Albrechts-Universität zu Kiel
Christian-Albrechts-Platz 4
24098 Kiel, Germany

Permanent ID of this document:

dda51148-ac5b-4655-9c4f-e01f26511235

This version is dated 2011-05-01.

Bibliographic reference:

P.M. Pardalos and S. Rebennack (Eds.):
SEA 2011, LNCS 6630, pp. 254–266, 2011.
© Springer-Verlag Berlin Heidelberg 2011.

The original publication is available at www.springerlink.com.
http://dx.doi.org/10.1007/978-3-642-20662-7_22

Abstract

We consider the maximum-cardinality matching problem in bipartite graphs. The input graph $G = (V, E)$ is not available for random access, but only as a stream, and random-access memory is limited to storing $\Theta(n)$ edges at a time, $n = |V|$. The number of passes over the input stream required to achieve the desired approximation is an important measure. It was shown by Eggert et al. (2009, 2011) that a $1 + 1/k$ approximation can be computed in $O(k^5)$ passes, independently of the input size. In this work, we present a new algorithm with the same approximation guarantee of $1 + 1/k$, but show experimentally that it requires two orders of magnitude fewer passes. The proven bound on the number of passes is $O(kn)$. This bound depends on the input size, and so in principle is inferior to $O(k^5)$. But we emphasize that in experiments, we do not find any correlation between theoretical bounds and actual performance: for all algorithms the number of passes observed in experiments is far below the corresponding theoretical bound. The most interesting insight comes from an experimental comparison of the previous and the new algorithm: e.g., for $k = 9$, the new one never needed more than 94 passes, even for instances with up to 2×10^6 vertices, whereas the previous one went up to more than 32 000 passes. Our main new technique is aimed at making the most out of each pass: we maintain a complex structure, using trees, for building augmenting paths.

Keywords: bipartite graph matching, massive graphs, semi-streaming algorithms, approximation schemes.

1 Introduction

Streaming and Matching – Previous Work. Let $G = (V = A \cup B, E)$ be a bipartite, undirected graph with $n := |V|$ vertices and $|E|$ edges. We aim to find a matching of large cardinality in G , i.e., a large subset of the edges $M \subseteq E$ such that $e \cap e' = \emptyset$ for all $e, e' \in M$ with $e \neq e'$. A matching with maximum cardinality is called an *optimal matching* or a *maximum matching*. It is well known that this problem can be solved to optimality in polynomial time.

Assume now that the graph is stored in such a way that we cannot do the usual queries efficiently, like asking whether two vertices v and w are adjacent or requesting the neighborhood of a particular vertex. The only access to the graph is by doing a *pass*. A pass means that each edge is presented to us exactly once and in arbitrary order. In other words, the graph is stored on a device that only allows sequential access. There is also a fast random-access memory available,

but it is of size $\Theta(n \log n)$ bits, i.e., we can store $\Theta(n)$ edges at a time, not more. When $|E| = \Omega(n^2)$, then this is not enough to store the whole graph. This is called the *semi-streaming model* [8]. The “streaming” term refers to the way the graph is presented. The “semi” attribute refers to the amount of random access memory available; in the “streaming model”, memory is limited to $\Theta(\log n)$, which is too restrictive for many graph problems [4].

Clearly, we can realize a sort of “random access” to the graph on the basis of a streaming setting: to determine whether v and w are adjacent, do a pass and note whether $\{v, w\}$ occurs or not. We can also collect the neighborhood of a certain vertex in one pass. However, operating in such a manner can lead to a huge number of passes and hence be inefficient. The common requirement is that the number of passes is independent of the problem size, i.e., independent of n and $|E|$ (but it may depend on approximation parameters). Unfortunately, many algorithms are designed around adjacency and neighborhood queries, including very simple ones like BFS and DFS, which are used in many matching algorithms. Also minimum-degree heuristics (see, e.g., [6]) for matching require neighborhood information: after we have added $\{v, w\}$ to the matching, the degrees of all neighbors of v and w have to be decremented.

Many matching algorithms work by finding and eliminating augmenting paths. Repeatedly finding and eliminating augmenting paths will, in $O(n)$ steps, lead to an optimal matching. An augmenting path can be found by performing a variant of BFS starting at the free vertices (i.e., vertices not contained in any edge of the current matching) of one of the partitions of the bipartite graph. A more sophisticated approach finds a set of pairwise vertex-disjoint (shortly: disjoint) paths and then eliminates them all together. This is the basic idea for the algorithm by Hopcroft and Karp [5] running in $O(n^{2.5})$ time. Unfortunately, with neither approach we can guarantee an appropriate bound on the number of passes required, when in a streaming situation. McGregor [7] suggested resorting to approximation and using a blend of BFS and DFS in order to find sets of disjoint augmenting paths up to a certain length, depending on the approximation parameter. He presented a randomized approximation scheme for the matching problem in general graphs: given a parameter $k \in \mathbb{N}$, with a small error probability it finds a $1 + 1/k$ approximation¹ using a number of passes independent of the input size. However, the

¹If M is a matching and M^* is an optimal matching, then M is a $1 + \varepsilon$ approximation if $|M^*| \leq (1 + \varepsilon) |M|$. Sometimes, we specify approximation in percent: M being a ρ approximation with $\rho \in [0, 1]$ means that $|M| \geq \rho |M^*|$. For example, in our case, $k = 9$ means a 90% approximation.

dependence on the approximation parameter k is rather strong, namely $\Omega(k)^{\Omega(k)}$, even when restricting to the bipartite case. In [3], Eggert, Kliemann, and Srivastav gave a deterministic algorithm for the bipartite case requiring only $O(k^8)$ passes. Recently, Eggert, Kliemann, Munstermann, and Srivastav improved that to $O(k^5)$ passes [2]. The basic idea of all those approaches is to grow multiple disjoint alternating paths at the same time. In [2], as an edge e goes by in the stream, it is attempted to use it to extend any of the alternating paths constructed so far, provided certain conditions are met, either forming a longer alternating path or completing it to an augmenting one. Backtracking is used to “revive” paths that currently fail to grow any further. A scheme called *position limiting* is used to limit the ways the edges may be appended to the alternating paths; details will be explained later. Position limiting is important to establish the desired bound on the number of passes, but at the same time it is so designed that it does not get in the way of achieving the required approximation.

Main Results. We present a new approximation scheme for maximum matching in bipartite graph streams. The algorithm has a guaranteed approximation of $1 + 1/k$ (Thm. 5.2) and requires at most $O(kn)$ passes (Thm. 5.3). In principle, this theoretical bound is inferior to the $O(k^{O(1)})$ -type bounds known for the previous algorithm, since it depends on the input size via n ; but for smaller n this bound is better (e.g., for the $O(k^5)$ version, $n \leq 3 \times 10^6$ is “small” enough). A main contribution of this work is that experimentally our new algorithm outperforms the previously known one by two orders of magnitude, while already the latter stays far below its theoretical $O(k^5)$ bound. For example for $k = 9$, the previous algorithm exhibits a number of passes up to about 32 000 per instance. This is far below the theoretical $O(k^{O(1)})$ -type bound, which is at least 14×10^6 . However, the breakthrough lies in the pass requirement that we observe for our *new* algorithm: for $k = 9$ and up to $n = 2\,000\,000$ it never required more than 94 passes, with an appropriate choice of further parameters even no more than 65 passes. In our opinion, this work crosses the borderline between theory and practice towards really practically efficient streaming algorithms.

2 Our Algorithmic Innovation

Tree Structure. The basic challenge for streaming algorithms is: how to make the most out of a pass? When the task is to find augmenting paths, one usually

maintains some structure incorporating alternating paths. Then the question more precisely reads: what kind of structure provides a high number of *extension points* where we can append new edges to it or which help to restructure it, as new information becomes available? In [2, 3] we took a path-based approach. We grew multiple alternating paths in parallel, and the end of each path provided an extension point. In this new work, we present a tree-based approach. Trees are rooted at free vertices and each path starting at the root is alternating. *All vertices with an even distance to the root act as extension points.* This requires some further considerations: we have to adapt the position limiting scheme and the stopping criterion. We prove the same approximation guarantee for the tree-based algorithm (Thm. 5.2) as we have for the path-based one. Regarding the number of passes, we only give a bound depending (linearly) on the size of the input; this is due to the new position limiting scheme. There is a simple work-around for this potential drawback: we can let the path-based algorithm run in parallel along the tree-based one, feeding both algorithms with the same edges as they go by in the stream. As soon as one of the algorithms terminates, we have a guaranteed approximation. This combined algorithm inherits the bound on the number of passes from the path-based one, which is independent of the input size. We did not implement this combination; given the experimental results this did not appear necessary.

Parameters. We make an extension that applies to both the path-based and tree-based algorithm. Our approximation technique is based on considering augmenting paths only up to a certain length, and to terminate when only a certain number of those remains. The length and the termination criterion both depend on the approximation parameter k . We introduce an additional parameter γ that is used to control a trade-off between path length and termination criterion. For the path-based algorithm, γ influences the worst-case bound on the number of passes: it ranges between $O(k^7)$ and $O(k^5)$. Experiments show that for some instance classes, the $O(k^7)$ version requires substantially fewer passes than the $O(k^5)$ one. This accentuates the importance of not only considering theoretical bounds. However, it should be noted that the path-based algorithm is by far outperformed by the tree-based one, independently of γ .

In addition to γ , another parameter $s \geq 1$, which we call the *stretch*, is introduced. It is related to the allowable length of the constructed augmenting paths and offers another trade-off control. For the path-based algorithm, only $s = 1$ makes sense, but larger values are meaningful for the tree-based one.

The following two sections, 3 and 4, describe the theoretical foundation of our approximation technique and explain the path-based algorithm from [2]. They also introduce the γ and s parameters. In Sect. 5, we present the new tree-based algorithm and in Sect. 6 the experimental setup and detailed discussion of results.

3 Approximation Technique

A *DAP algorithm* is one that finds, given a matching M , a set of disjoint augmenting paths. For $\lambda \in \mathbb{N}$, we call a path a λ *path* if it is of length at most $2\lambda + 1$; the length of a path being the number of its edges. For $\lambda_1, \lambda_2 \in \mathbb{N}$, $\lambda_1 \leq \lambda_2$, a set \mathcal{Y} of paths is called a (λ_1, λ_2) *DAP set* if:

1. All paths in \mathcal{Y} are augmenting λ_2 paths.
2. Any two paths in \mathcal{Y} are vertex-disjoint.
3. We cannot add another augmenting λ_1 path to \mathcal{Y} without violating condition 2.

We call $s := \frac{\lambda_2}{\lambda_1}$ the *stretch*, since it specifies how far paths may stretch beyond λ_1 . Given $\delta \in [0, 1]$, a DAP algorithm is called a $(\lambda_1, \lambda_2, \delta)$ *DAP approximation algorithm* if it always delivers a result \mathcal{A} of disjoint augmenting λ_2 paths such that there exists a (λ_1, λ_2) DAP set \mathcal{Y} so that $|\mathcal{Y}| \leq |\mathcal{A}| + \delta |M|$. Let $\delta_{\text{inn}}, \delta_{\text{out}} \in [0, 1]$ and DAP be a $(\lambda_1, \lambda_2, \delta_{\text{inn}})$ DAP approximation algorithm. All our algorithms utilize the loop shown in Algorithm 1. When this loop terminates, clearly there exists a (λ_1, λ_2) DAP set \mathcal{Y} with $|\mathcal{Y}| \leq |\mathcal{A}| + \delta_{\text{inn}} |M| \leq \delta_{\text{out}} |M| + \delta_{\text{inn}} |M| = (\delta_{\text{inn}} + \delta_{\text{out}}) |M|$, where M denotes the matching before the last augmentation. Let $k \in \mathbb{N}$ and $k \leq \lambda_1 \leq \lambda_2$ and

$$\delta(\lambda_1, \lambda_2) := \frac{\lambda_1 - k + 1}{2k\lambda_1(\lambda_2 + 2)} > 0 . \quad (1)$$

Algorithm 1: Outer Loop

- 1 $M :=$ any inclusion-maximal matching;
 - 2 **repeat**
 - 3 $c := |M|$;
 - 4 $\mathcal{A} := \text{DAP}(M)$;
 - 5 augment M using \mathcal{A} ;
 - 6 **until** $|\mathcal{A}| \leq \delta_{\text{out}} c$;
-

Following the pattern of [2, Lem. 4.1 and 4.2] we can prove:

3.1 Lemma. *Let M be an inclusion-maximal matching. Let \mathcal{Y} be a (λ_1, λ_2) DAP set such that $|\mathcal{Y}| \leq 2\delta |M|$ with $\delta = \delta(\lambda_1, \lambda_2)$. Then M is a $1 + 1/k$ approximation.*

The lemma yields the $1 + 1/k$ approximation guarantee for Algorithm 1 when $\delta_{\text{inn}} = \delta_{\text{out}} = \delta(\lambda_1, \lambda_2)$. What are desirable values for λ_1 and λ_2 ? The DAP approximation algorithms presented in later sections (the path-based and the tree-based one) can work with any allowable setting for λ_1 and λ_2 , so we have some freedom of choice. We assume that constructing longer paths is more expensive, so we would like to have those values small and in particular $\lambda_1 = \lambda_2$. (We will later encounter situations where it is conceivable that higher λ_2 is beneficial.) On the other hand, we would like to have δ large in order to terminate quickly. The function $\lambda \mapsto \delta(\lambda, \lambda)$ climbs until $\lambda = k - 1 + \sqrt{k^2 - 1} \leq 2k - 1$ and falls after that. Since we only use integral values for λ_1 , the largest value to consider is $\lambda_1 = \lambda_2 = 2k - 1$. The smallest one is $\lambda_1 = \lambda_2 = k$. We parameterize the range in between by defining

$$\lambda(\gamma) := \lceil k(1 + \gamma) \rceil - 1 \quad \text{for each } \gamma \in [1/k, 1] . \quad (2)$$

Consider the setting $\lambda_1 := \lambda_2 := \lambda(\gamma)$ and $\delta_{\text{inn}} := \delta_{\text{out}} := \delta(\lambda_1, \lambda_2)$. Then increasing γ increases path length, but also increases δ_{inn} and δ_{out} , which means that we are content with a less good approximation from the DAP algorithm and also relax the stopping condition of the outer loop. So γ controls a trade-off between path length and stopping criterion.

4 Path-based DAP Approximation

We briefly describe how we find a $(\lambda_1, \lambda_2, \delta_{\text{inn}})$ DAP approximation with $\lambda_1 = \lambda_2$ in [2]; please consult that text for details. Fix an inclusion-maximal matching M . A vertex v is called *covered* if $v \in e \in M$ for some e , and *free* otherwise. We call an edge $e \in E$ a *matching edge* if $e \in M$, and *free* otherwise. Denote $\text{free}(A)$ and $\text{free}(B)$ the free vertices of partitions A and B , respectively. If $v \in V$ is not free, denote its *mate* by M_v , i.e., the unique vertex so that $\{v, M_v\} \in M$. We construct disjoint alternating paths starting at vertices of $\text{free}(A)$, the *constructed paths*, and we index them by their starting vertices: $(P(\alpha))_{\alpha \in \text{free}(A)}$. When we find augmenting paths, they are stored in a set \mathcal{A} and their vertices marked as *used*; a vertex not being used is called *remaining*. Denote $\text{remain}(X)$ the remaining vertices in a set $X \subseteq V$.

Suppose $P(\alpha) = (\alpha, e_1, b_1, m_1, a_1, \dots, m_t, a_t)$ is a path with free vertex $\alpha \in \text{free}(A)$, vertices $a_1, \dots, a_t \in A$ and $b_1, \dots, b_t \in B$, free edges $e_1, \dots, e_t \in E$ and matching edges $m_1, \dots, m_t \in M$. Then we say that matching edge m_i has *position* i , $i \in [t]$. Each matching edge m has a *position limit* $\ell(m)$, initialized to $\ell(m) := \lambda_1 + 1$. We perform *position limiting*, i.e., a matching edge m will only be inserted into a constructed path if its new position is strictly smaller than its position limit. When a matching edge is inserted, its position limit is decremented to its position in the constructed path.

After each pass, we backtrack conditionally: each constructed path that was not modified during that preceding pass has its last two edges removed. When the number of constructed paths of positive length falls on or below $\delta_{\text{inn}} |M|$, we terminate and deliver all augmenting paths found so far. Position limiting is important for proving the bound $2\lambda_1\delta_{\text{inn}}^{-1} + 1$ on the number of passes of this DAP algorithm [2, Lem. 7.1]. By the stopping criterion of the outer loop, it is invoked at most $\delta_{\text{out}}^{-1} + 1$ times [2, Thm. 7.2]. Hence, with (2), we have the following bound on the number of passes conducted in total: $(\delta_{\text{out}}^{-1} + 1)(2\lambda_1\delta_{\text{inn}}^{-1} + 1) = O(\gamma^{-2}k^5)$. Let us specify γ by $\tilde{\gamma} \in [0, 1]$ via the relation $\gamma = k^{-\tilde{\gamma}}$. Then for $\tilde{\gamma} = 0$ the bound is $O(k^5)$, for $\tilde{\gamma} = 1/2$ it is $O(k^6)$, and for $\tilde{\gamma} = 1$ it is $O(k^7)$. We compare these three values for $\tilde{\gamma}$ in experiments.

5 Tree-based DAP Approximation

An *alternating tree* is a pair consisting of a tree T that is a subgraph of G , and a vertex $r \in V(T)$, called its *root*, so that each path from r to any other vertex of T is an alternating path. For $v \in V(T)$ the subtree induced by all vertices reachable from r via v is called the *subtree below* v and denoted $T[v]$. An *alternating forest* consists of one or more alternating trees being pairwise vertex-disjoint. Our tree-based DAP algorithm maintains an alternating forest with trees indexed by vertices. We write $T(v) = (V(v), E(v))$ for the tree indexed with $v \in V$; we ensure that it is either empty or rooted at v . The *forest* \mathcal{F} is $\mathcal{F} = \{T(v); v \in \text{remain}(V)\}$. We only deal with trees from \mathcal{F} . We call a tree *properly rooted* if its root is a free vertex. A properly rooted tree $T(\alpha)$ together with an edge $\{a, \beta\}$ with β being free and $a \in V(T)$ at an even distance from α , yield an augmenting path.

We initialize by setting $T(\alpha) := (\{\alpha\}, \emptyset)$ for each $\alpha \in \text{free}(A)$ and $T(v) := (\emptyset, \emptyset)$ for each $v \in V \setminus \text{free}(A)$. So we have empty trees and one-vertex trees with a free vertex of A . Position limits are initialized $\ell(m) := \lambda_1 + 1$ for each $m \in M$ as

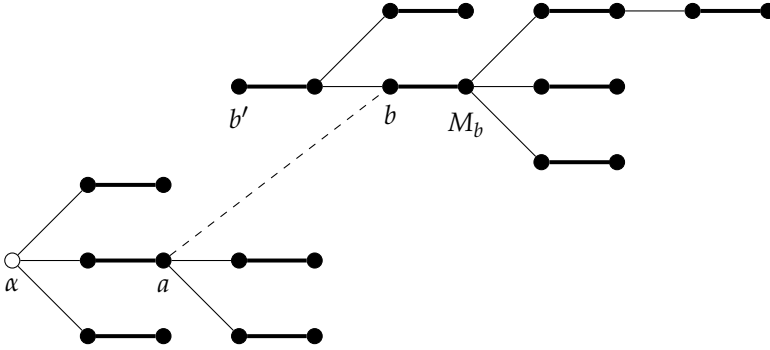


Figure 1. A properly rooted tree $T(\alpha)$ and a non-properly rooted tree $T(b')$. Free vertices are drawn non-filled, covered vertices are drawn filled. Free edges are drawn normal and matching edges are drawn thick. Matching edges in $T(b')$ have position limit $\lambda_1 + 1$ (due to position limit release, explained later on page 10). When the dashed edge $\{a, b\}$ comes along in the stream, part of $T(b')$ is migrated to $T(\alpha)$ as shown in Fig. 2 on the next page.

usual. If $(\alpha, e_1, b_1, m_1, a_1, \dots, m_t, a_t)$ is a path in the properly rooted tree $T(\alpha)$, then we say that matching edge m_i , $i \in [t]$, has *position* i . Results (i.e., the augmenting paths found) will be stored into a set \mathcal{A} , that is initialized to $\mathcal{A} := \emptyset$.

Trees grow over time, and there may also emerge non-properly rooted trees. When a free edge $\{a, b\}$ between two remaining vertices goes by in the stream with b being covered, the algorithm checks whether to extend any of the trees. Conditions are: the tree has to be properly rooted, say $T(\alpha)$, it must contain a , and $i < \ell(\{b, M_b\})$, where i is the position that the matching edge $\{b, M_b\}$ would take in $T(\alpha)$. If all those conditions are met, an *extension step* occurs: the two edges $\{a, b\}$ and $\{b, M_b\}$ are added to $T(\alpha)$, and, if $\{b, M_b\}$ is already part of a tree $T(b')$, then $T(b')[b]$ is removed from $T(b')$ and connected to $T(\alpha)$ via $\{a, b\}$. The tree $T(b')$ is not required to be properly rooted, but it may be. Bipartiteness ensures that $M_b \in V(T(b')[b])$. Position limits for all inserted or migrated edges are updated to reflect their new positions. Figures 1 and 2 show an example.

When a free edge $\{a, \beta\}$ with $a, \beta \in \text{remain}(V)$ goes by in the stream with β being *free*, then we check whether we can build an augmenting path. If there is a properly rooted tree $T(\alpha)$ with $a \in V(\alpha)$, the path P in $T(\alpha)$ from α to β is augmenting. In that case, a *completion step* occurs: we store P into the result set \mathcal{A} , and mark all vertices on P as used. Also, we adjust our forest as follows. For

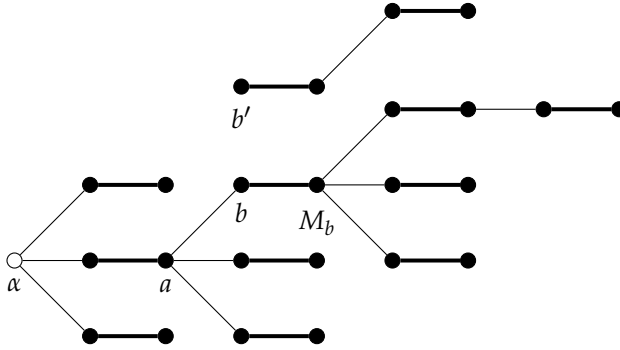


Figure 2. The subtree $T(b')[b]$ was migrated to $T(\alpha)$. The migrated edges have new position limits, e.g., $\ell(\{b, M_b\}) = 2$. There are only 3 edges left in tree $T(b')$.

each $a \in V(P) \cap A$ and each of its neighbors in $T(\alpha)$ and not in P , i.e., for each $b \in N_{T(\alpha)}(a) \setminus V(P)$, we set $T(b) := T(\alpha)[b]$. In other words, we “cut” P out of $T(\alpha)$ and make each of the resulting subtrees that “fall off” a new tree of its own. None of those is properly rooted, and also they are rooted at vertices of partition B , not A as the properly rooted ones. However, they – or parts of them – can subsequently be connected to remaining properly rooted trees by an extension step as described above and shown in Fig. 1 and 2. One last and crucial feature of the completion step is *position limit release*: we release position limits to $\lambda_1 + 1$ on edges of the new (non-properly rooted) trees. This is important for the proof of the approximation guarantee in Lem. 5.1. We do not explicitly backtrack; yet, position limit release can be considered a form of backtracking.

Position limit release requires further considerations. In an extension step, although position limits at first are not higher than $\lambda_1 + 1$, edges can be included in a tree at positions beyond λ_1 . Assume $m = \{b, M_b\}$ is inserted at position $i < \lambda_1$ into a properly rooted tree $T(\alpha)$ and subsequently, more edges are inserted behind m . Then an augmenting path is found in $T(\alpha)$ not incorporating m , hence the position limit of m is released. Later m can be inserted at a position j with $i < j \leq \lambda_1$ in another properly rooted tree $T(\alpha')$. When m carries a sufficiently deep subtree with it, then $T(\alpha')$ could grow beyond λ_1 , even though $j \leq \lambda_1$. Here, the second length parameter λ_2 comes into play. When the migrated subtree is too deep, we trim its branches just so that it can be migrated without making the destination tree reach beyond λ_2 . The trimmed-off branches become non-properly

rooted trees of their own. We control a trade-off this way: higher λ_2 means fewer trimming and hence fewer destruction of previously built structure. But higher λ_2 reduces $\delta(\lambda_1, \lambda_2)$ and so may prolong termination. Choosing $\lambda_2 := \lambda_1$ is possible.

After each pass, it is checked whether it is time to terminate and return the result \mathcal{A} . We terminate when any of the following two conditions is met:

- (T1) During the last pass, no extension or completion occurred. In other words, the forest did not change. (It then would not change during further passes.)
- (T2) The number of properly rooted trees (which is also the number of remaining free vertices of A) is on or below $\delta_{\text{inn}} |M|$.

5.1 Lemma. *The algorithm described in this section is a $(\lambda_1, \lambda_2, \delta_{\text{inn}})$ DAP approximation algorithm.*

Sketch. When the algorithm terminates via condition (T2), it could have, by carrying on, found at most $\delta_{\text{inn}} |M|$ additional augmenting paths. We show that when we restrict to termination condition (T1), we have a $(\lambda_1, \lambda_2, 0)$ DAP approximation algorithm. To this end, it only remains to show that we cannot add an augmenting λ_1 path to \mathcal{A} without hitting at least one of the paths already included. Suppose there is an augmenting path $(\alpha, e_1, b_1, m_1, a_1, e_2, b_2, m_2, a_2, \dots, a_t, e_{t+1}, \beta)$ with $t \leq \lambda_1$, $\alpha \in \text{free}(A)$ and $\beta \in \text{free}(B)$ that is disjoint to all paths in \mathcal{A} . We can show by induction that when the algorithm terminates, then a_t is in a properly rooted tree T ; this part of the proof depends crucially on position limit release. This claim helps establishing a contradiction: first, by the stopping criterion, a_t was there for the whole pass, since an extension step would have made termination impossible. But then the algorithm would have pulled out an augmenting path from T when $e_{t+1} = \{a_t, \beta\}$ came along in the stream during the last pass and so it would not have been allowed to terminate. \square

Following from Lem. 3.1 and 5.1 we have:

5.2 Theorem. *Algorithm 1 with the tree-based DAP approximation algorithm described in this section gives a $1 + 1/k$ approximation, if $\delta_{\text{inn}} = \delta_{\text{out}} = \delta(\lambda_1, \lambda_2)$.*

As for the number of passes, at this time we only have a bound depending on the problem size. The main hindrance for an independent bound is position limit release, as can be seen by a comparison with the techniques in [2, Lem. 7.1].

5.3 Theorem. *Algorithm 1 with the tree-based DAP approximation algorithm described in this section requires at most $\frac{\lambda_1 n}{4} + 1 \leq \frac{kn}{2} + 1$ passes.*

Table 1. Results for path-based (indicated by “p” in the left column) and tree-based (“t”) algorithms. Parameters are $\lambda_1 = \lambda(\gamma)$ as per (2), $\gamma = k^{-\tilde{\gamma}}$, and stretch $s = \frac{\lambda_2}{\lambda_1}$. Numbers state the maximum and rounded mean number of passes, respectively, that were observed for the different choices of parameters and instance classes. We have relatively small instances: $n = 40\,000, 41\,000, \dots, 50\,000$. Density is limited by $D_{\max} = 1/10$. Number of edges ranges up to about $|E| = 62 \times 10^6$.

| | $\tilde{\gamma}$ | s | maximum | | | | | mean | | | | |
|---|------------------|-----|---------|--------|-------|-------|-------|------|------|-------|-----|------|
| | | | rand | degm | hilo | rbg | rope | rand | degm | hilo | rbg | rope |
| p | 0 | 1 | 11 886 | 14 180 | 7 032 | 4 723 | 2 689 | 107 | 145 | 3 337 | 257 | 378 |
| p | 1/2 | 1 | 7 817 | 31 491 | 7 971 | 4 383 | 3 843 | 80 | 127 | 2 071 | 500 | 541 |
| p | 1 | 1 | 7 121 | 32 844 | 9 106 | 5 687 | 5 126 | 74 | 166 | 2 033 | 844 | 790 |
| t | 0 | 1 | 6 | 9 | 75 | 41 | 79 | 3 | 3 | 51 | 5 | 22 |
| t | 0 | 2 | 6 | 9 | 74 | 52 | 94 | 3 | 3 | 51 | 5 | 26 |
| t | 1/2 | 1 | 6 | 9 | 59 | 37 | 63 | 3 | 3 | 38 | 5 | 20 |
| t | 1/2 | 2 | 6 | 9 | 59 | 44 | 70 | 3 | 3 | 38 | 5 | 22 |
| t | 1 | 1 | 6 | 9 | 54 | 38 | 61 | 3 | 3 | 35 | 5 | 20 |
| t | 1 | 2 | 6 | 9 | 55 | 40 | 67 | 3 | 3 | 36 | 6 | 21 |

Sketch. Consider an invocation of the DAP algorithm for a matching M . For each $m \in M$ denote $\text{dist}(m)$ the minimum position of m over all alternating paths that start at a free vertex of A and use only remaining vertices (regardless whether the path can be realized by the algorithm as part of a tree). If there is no such path, we put $\text{dist}(m) := \infty$. All matching edges m that are eligible to be inserted into a (properly rooted) tree of our forest have $\text{dist}(m) \leq \lambda_1$. It can be seen easily by induction that while no further augmenting path is found, after at most λ_1 passes, for each m with $\text{dist}(m) \leq \lambda_1$, we have $\ell(m) = \text{dist}(m)$. Thus, after at most λ_1 passes, a new augmenting path is found or the algorithm terminates due to lack of action, i.e., by termination condition (T1). Since the initial matching is already a 50% approximation, the former can happen at most $\frac{|M^*|}{2} \leq \frac{n}{4}$ times over all executions of the DAP algorithm, with M^* denoting an optimal matching. \square

6 Experiments

Setup. We use randomly generated instances with various structure:

rand: Random bipartite graph; each edge occurs with probability $p \in [0, 1]$.

Table 2. Using trees and a good setting determined in previous series, namely $\tilde{\gamma} = 1$ and $s = 1$. We treat larger instances: $n = 100\,000, 200\,000, \dots, 1\,000\,000$. Density is limited by $D_{\max} = 1/10$. Development for growing n is shown. Number of edges ranges up to about $|E| = 1 \times 10^9$, which takes about 7.5 GiB of space.

| n | maximum | | | | | mean | | | | |
|-----------|---------|------|------|-----|------|------|------|------|-----|------|
| | rand | degm | hilo | rbg | rope | rand | degm | hilo | rbg | rope |
| 100 000 | 3 | 8 | 53 | 30 | 62 | 2.5 | 3.2 | 35.0 | 5.1 | 19.8 |
| 200 000 | 3 | 7 | 56 | 31 | 63 | 2.5 | 2.8 | 37.6 | 4.7 | 19.1 |
| 300 000 | 3 | 7 | 55 | 29 | 64 | 2.5 | 2.9 | 38.6 | 3.9 | 18.2 |
| 400 000 | 3 | 8 | 56 | 33 | 63 | 2.5 | 2.9 | 36.3 | 5.3 | 15.6 |
| 500 000 | 3 | 7 | 58 | 34 | 64 | 2.5 | 3.0 | 36.7 | 4.4 | 19.4 |
| 600 000 | 3 | 9 | 58 | 30 | 64 | 2.5 | 3.5 | 38.4 | 3.3 | 18.1 |
| 700 000 | 6 | 9 | 56 | 35 | 62 | 2.5 | 3.6 | 37.4 | 3.9 | 18.5 |
| 800 000 | 3 | 8 | 58 | 31 | 63 | 2.5 | 3.5 | 37.9 | 3.1 | 16.2 |
| 900 000 | 7 | 8 | 61 | 32 | 62 | 2.6 | 3.3 | 37.0 | 3.7 | 14.5 |
| 1 000 000 | 6 | 9 | 60 | 34 | 65 | 2.5 | 3.1 | 33.4 | 4.6 | 18.2 |

degm: The degrees in one partition are a linear function of the vertex index. A parameter $p \in [0, 1]$ is used to scale degrees.

hilo, rbg, rope: Vertices in both partitions are divided into l groups of equal size and connected based on that according to different schemes. For details on hilo and rbg (the latter also known as fewg or manyg), we refer to [1, 6, 9]. In rope [1, 9], the construction results in a layered graph, where the odd layers are perfect matchings, and the even layers are random bipartite graphs with parameter $p \in [0, 1]$. Such a graph has a unique perfect matching.

Instances are kept completely in RAM, so the streaming situation is only simulated. For Tab. 2, we impose a hard limit of 1×10^9 edges, meaning about 7.5 GiB (each vertex is stored as a 32 bit unsigned integer). A series is specified by a density limit and a set of values for n . For each n of a series and for each class, we generate 256 instances on n vertices. For hilo, rbg, and rope, parameter l is chosen randomly from the set of divisors of $|A| = n/2$. For all classes, a parameter controlling the (expected) number of edges (e.g., p for rand) is being moved through a range such that we start with very few (expected) edges and go up to (or close to) the maximum number of edges possible, given the hard limit, the limit D_{\max} on the density (allowing some overstepping due to randomness), and any limit resulting from structural properties (e.g., number of groups l). This way we

Table 3. The same algorithm and parameters as in Tab. 2, but larger number of vertices and lower density limit, namely $D_{\max} = 1 \times 10^{-4}$. Development for growing n is shown. Number of edges ranges up to about $|E| = 100 \times 10^6$.

| n | maximum | | | | | mean | | | | |
|-----------|---------|------|------|-----|------|------|------|------|-----|------|
| | rand | degm | hilo | rbg | rope | rand | degm | hilo | rbg | rope |
| 1 000 000 | 48 | 43 | 62 | 41 | 48 | 5.5 | 8.8 | 29.6 | 4.5 | 17.7 |
| 1 100 000 | 47 | 49 | 60 | 42 | 50 | 5.1 | 8.6 | 29.6 | 4.4 | 17.0 |
| 1 200 000 | 45 | 51 | 60 | 33 | 52 | 4.4 | 8.4 | 29.0 | 5.2 | 14.5 |
| 1 300 000 | 31 | 30 | 59 | 41 | 47 | 3.9 | 8.7 | 30.0 | 3.9 | 15.5 |
| 1 400 000 | 32 | 35 | 61 | 35 | 51 | 4.5 | 7.6 | 28.5 | 4.6 | 14.9 |
| 1 500 000 | 28 | 29 | 57 | 33 | 51 | 3.9 | 8.5 | 28.7 | 4.5 | 15.5 |
| 1 600 000 | 25 | 27 | 58 | 34 | 52 | 4.1 | 6.9 | 26.7 | 4.5 | 15.9 |
| 1 700 000 | 28 | 42 | 60 | 35 | 52 | 3.6 | 7.7 | 28.8 | 4.6 | 16.2 |
| 1 800 000 | 31 | 29 | 60 | 35 | 54 | 4.1 | 6.8 | 28.1 | 3.2 | 15.2 |
| 1 900 000 | 23 | 26 | 56 | 34 | 50 | 3.2 | 6.3 | 27.7 | 4.6 | 14.0 |
| 2 000 000 | 32 | 21 | 60 | 35 | 49 | 3.4 | 6.4 | 28.9 | 4.5 | 15.7 |

produce instances of different densities. For rand and degm, we use 16 different densities and generate 16 instances each. For hilo, rbg, and rope, we use 64 random choices of l and for each 4 different densities. This amounts to 256 instances per n and class. In total, we treated more than 60 000 instances. After an instance is generated, its edges are brought into random order. Then each algorithm is run on it once, and then again with partitions A and B swapped. During one run of an algorithm, the order of edges in the stream is kept fix.

Results. Result details are given in Tables 1, 2, and 3. We only consider numbers of passes and completely neglect running times. The approximation parameter is fixed to $k = 9$, which means a guaranteed 90% approximation. We limit the graph density $D = \frac{|E|}{|A||B|}$ to $1/10$ or lower, since preliminary tests had shown that usually the tree-based algorithm already exhibits its worst number of passes there. The density limit saves computation time, since each single pass takes fewer time when there are fewer edges. We give maximum and mean numbers of passes, while we focus the discussion on the maximum.

Table 1 compares the path-based and tree-based algorithms. The tree-based outperforms the path-based by a large margin, in the worst and the average case. For no setting of $\tilde{\gamma}$ or s , the tree-based algorithm needed more than 94 passes,

whereas the path-based one ranges up to more than 32 000 passes. This also means that the main improvement stems from using trees instead of paths, and not from the trade-off parameters. For the path-based algorithm, there are considerable differences for different values of $\tilde{\gamma}$. There is no best setting for $\tilde{\gamma}$, but it depends on the instance class: compare in particular the `rand` and `rbg` or `rope` classes. For the `hilo` class, the maximum and the mean number of passes move in opposite directions when changing $\tilde{\gamma}$. For the tree-based algorithm, $\tilde{\gamma} = 1$ and $s = 1$ is a good setting, with maximum number of passes not exceeding 61. For the tree-based algorithm, the highest number of passes is attained for $\tilde{\gamma} = 0$, namely 79 for $s = 1$ and 94 for $s = 2$.

Tables 2 and 3 consider the tree-based algorithm for larger instances. In particular they address the concern that the maximum number of passes could grow with the number n of vertices. There is a slight upward tendency for some classes, e.g., for `hilo` in Tab. 2. The maximum number of passes in the range $n = 100\,000$ to $n = 1\,000\,000$ is attained for each n by `rope` in Tab. 2. The smallest value is 62 and the largest 65. From this perspective, the increase is below 5% while the number of vertices grows by factor 10. For even larger n and smaller density, Tab. 3, the maximum is 62. Additionally, we conducted two series with density limits 1×10^{-3} and 1×10^{-4} , respectively, and up to $n = 1 \times 10^6$. The maximum observed in those series was 60.

7 Conclusion and Future Work

The tree-based algorithm outperforms the path-based one by a large margin. Its theoretical pass bound depends on n , but experiments give rise to the hypothesis that there in fact is at most a minor dependence. Future work will put this hypothesis to a test, in particular we will consider larger instances.

As an algorithmic addition, one could use multiple matchings in parallel. This follows the same guiding question: how to make the most out of each pass? We initialize a number of matchings in a randomized manner. Then we work on all of those at once: when an edge comes along in the stream, it might not be usable to extend all of the forests, but maybe some of them.

Concerning running time, we have preliminary results that in certain cases, our tree-based algorithm outperforms random-access algorithms – even if the instance completely fits into random-access memory. This may be due to streaming algorithms making better use of memory caches. This will be addressed in detail

in future work.

Acknowledgments

I thank Ole Kliemann for helpful discussions, supporting the experiments, and for profiling and improving our C+ implementation. I thank Anand Srivastav and the anonymous referees for helpful comments on my manuscript. I thank the Rechenzentrum at Universität Kiel for many months of computation time. I thank the Deutsche Forschungsgemeinschaft (DFG) for financial support through Priority Program 1307, *Algorithm Engineering*, Grant Sr7/12-2.

References

- [1] Boris V. Cherkassky, Andrew V. Goldberg, and Paul Martin. Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. *ACM Journal of Experimental Algorithms*, 3, 1998. Available from: <http://www.jea.acm.org/1998/CherkasskyAugment/>.
- [2] Sebastian Eggert, Lasse Kliemann, Peter Munstermann, and Anand Srivastav. Bipartite matching in the semi-streaming model. Technical Report 1101, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, 2011. Document ID: 519a88bb-5f5a-409d-8293-13cd80a66b36. Available from: http://www.informatik.uni-kiel.de/~lki/tr_1101.pdf.
- [3] Sebastian Eggert, Lasse Kliemann, and Anand Srivastav. Bipartite graph matchings in the semi-streaming model. In *Proceedings of the 17th Annual European Symposium on Algorithms, Copenhagen, Denmark, September 2009 (ESA 2009)*, pages 492–503, 2009. Presented also at the *MADALGO Workshop on Massive Data Algorithmics*, June 2009, Århus, Denmark. Available from: http://www.informatik.uni-kiel.de/~discopt/person/asr/publication/streaming_esa_09.pdf, doi: 10.1007/978-3-642-04128-0_44.
- [4] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming, Turku, Finland, July 2004 (ICALP 2004)*, pages 531–543, 2004. doi:10.1007/b99859.

-
- [5] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. doi:10.1137/0202019.
- [6] Johannes Langguth, Fredrik Manne, and Peter Sanders. Heuristic initialization for bipartite matching problems. *ACM Journal of Experimental Algorithmics*, 15:1.3:1.1–1.3:1.22, 2010. Available from: <http://doi.acm.org/10.1145/1712655.1712656>.
- [7] Andrew McGregor. Finding graph matchings in data streams. In *Proceedings of the International Workshop on Approximation Algorithms for Combinatorial Optimization Problems and Randomization and Computation, Berkeley, CA, USA, August 2005 (APPROX RANDOM 2005)*, pages 170–181, 2005. doi:10.1007/11538462_15.
- [8] S. Muthu Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):67 pages, 2005. Available from: <http://algo.research.googlepages.com/eight.ps>.
- [9] João C. Setubal. Sequential and parallel experimental results with bipartite matching algorithms. Technical Report IC-96-09, Institute of Computing, University of Campinas, Brazil, 1996. Available from: <http://www.dcc.unicamp.br/ic-tr-ftp/1996/96-09.ps.gz>.