An Experimental Study of Algorithms for Online Bipartite Matching

ALLAN BORODIN and CHRISTODOULOS KARAVASILIS, University of Toronto, Canada
DENIS PANKRATOV, Concordia University, Canada

We perform an experimental study of algorithms for online bipartite matching under the known i.i.d input model with integral types. In the last decade, there has been substantial effort in designing complex algorithms to improve worst-case approximation ratios. Our goal is to determine how these algorithms perform on more practical instances rather than worst-case instances. In particular, we are interested in whether the ranking of the algorithms by their worst-case performance is consistent with the ranking of the algorithms by their average-case/practical performance. We are also interested in whether preprocessing times and implementation difficulties that are introduced by these algorithms are justified in practice. To that end, we evaluate these algorithms on different random inputs as well as real-life instances obtained from publicly available repositories. We compare these algorithms against several simple greedy-style algorithms. Most of the complex algorithms in the literature are presented as being non-greedy (i.e., an algorithm can intentionally skip matching a node that has available neighbors) to simplify the analysis. Every such algorithm can be turned into a greedy one without hurting its worst-case performance. On our benchmarks, non-greedy versions of these algorithms perform much worse than their greedy versions. Greedy versions perform about as well as the simplest greedy algorithm by itself. This, together with our other findings, suggests that simplest greedy algorithms are competitive with the state-of-the-art worst-case algorithms for online bipartite matching on many average-case and practical input families. Greediness is by far the most important property of online algorithms for bipartite matching.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; Random graphs; • **Theory of computation → Online algorithms**;

Additional Key Words and Phrases: Bipartite graphs, bipartite matching, stochastic input models, greedy algorithms

ACM Reference format:
Allan Borodin, Christodoulos Karavasilis, and Denis Pankratov. 2020. An Experimental Study of Algorithms for Online Bipartite Matching. *J. Exp. Algorithmics* 25, 1, Article 1.4 (March 2020), 37 pages.
https://doi.org/10.1145/3379552

Part of this work was done while the first author was at the Toyota Technological Institute at Chicago, and the last author was a postdoc at the University of Toronto.
This research is supported by NSERC.
Authors' addresses: A. Borodin and C. Karavasilis, University of Toronto, 10 King's College Road, Toronto, ON, M5S 3G4, Canada; emails: {bor, ckar}@cs.toronto.edu; D. Pankratov, Concordia University, 1515 Ste-Catherine St. W., Montreal, QC, H3G 2W1, Canada; email: denis.pankratov@concordia.ca.

1084-6654/2020/03-ART1.4 $15.00
https://doi.org/10.1145/3379552

1 INTRODUCTION

One of the most active areas of theoretical computer science is the design and analysis of "efficient" approximation algorithms. Often the objective is to establish the best approximation ratio achieved by a polynomial time algorithm. Such analysis is often done in terms of adversarial worst-case inputs, or in the case of stochastic analysis, in terms of a worst-case i.i.d.[1] distributional setting. However, such analysis can be and is challenged as to whether or not these worst-case approximation bounds reflect results for more "realistic" settings. There are many reasons for the perceived and observed gap between theory and practice: asymptotic time bounds can hide large constant factors, typical inputs are not worst-case inputs, and simple algorithms are much easier to implement and are usually preferred (by practitioners) over more complex algorithms.

The most common approach to better understanding the gap between theory and practice is to perform experimental studies with respect to data that better reflects reality.[2] Following this approach, we wish to study relatively simple greedy and "greedy-like" algorithms for online bipartite matching in comparison with more complex non-greedy algorithms that have been designed for a known distribution stochastic setting. Since bipartite matching can be solved offline optimally and relatively efficiently, we are able to precisely compute the observed competitive ratios. Greedy offline algorithms for matching in general graphs have also been experimentally studied in the past [19].

A *matching* is a collection of vertex-disjoint edges in a graph. The *bipartite matching problem* asks to compute either exactly or approximately the cardinality of a maximum-size matching in a given bipartite graph. In addition, we typically want to find such a matching itself. In the *adversarial online setting*, one side of the bipartite graph is known in advance, while vertices from the other side arrive one by one. When an online vertex arrives, you learn the identity of this vertex together with identities of all its neighbors (all of them have to be on the other side of the partition). When an online vertex arrives, an algorithm can pick one of its available neighbors to match this vertex to. In the online setting, such a decision is final and cannot be changed in the future. Thus, an online algorithm makes decisions without seeing any future input items but knowing the past items. A typical application for maximum cardinality bipartite matching is the task of assigning jobs to workers. Offline nodes correspond to workers, while online nodes correspond to jobs that arrive online and have to be assigned (if possible) to some available appropriate worker. Not all jobs can be executed by all workers and the goal is to assign as many jobs as possible.

The assumption that an online algorithm does not know anything about the future input is quite pessimistic. In practical applications, instances of online maximum matching often need to be solved over and over again. It is natural to assume that these instances come from some input distribution. Thus, by observing past instances and collecting historical data we can estimate parameters of the input distribution. Armed with this historical data, an algorithm might perform better on future online instances, because it will have some statistical information about online input items. The *known i.i.d.* model describes a particular distribution family that has recently received a lot of attention in the bipartite matching community, in part because it is widely applicable in practice and can also be analyzed theoretically. The set of neighbors of an online vertex is called its *type*. From historical data one can derive the frequency of certain types of online nodes appearing in an input instance. This information is then aggregated into a data structure, called a

[1] i.i.d. = independent, identically distributed.

[2] This is not to say that the gap between theory and practice is restricted to experimental studies. Other approaches, such as smoothed analysis, as initiated in [29], and perturbation stable instances, as initiated in [4], have also been proposed. Thus far, these insightful analytical approaches have not yet been widely accepted. Arguably, experimental analysis remains the most common method for trying to understand the comparative performance of algorithms.

type graph. A type graph is a bipartite graph that on one side has nodes that appear as offline nodes in input instances. On the other side, the type graph contains nodes of distinct types together with a probability of each type occurring. Thus, for each type i we know the probability p_i of a node of this type arriving: $p_i \geq 0$ and $\sum_i p_i = 1$. With n offline nodes, the number of distinct types is 2^n, but typically not all types are present with positive probability. Oftentimes the set of types is very sparse, and even linear in n. Once we have this type graph information, we assume that an online bipartite *instance* graph G is generated according to the following randomized process. The set of offline nodes of G is the same as the set of offline nodes of the type graph, but each online node of G is sampled from the same distribution p independently at random.

In terms of the jobs and workers application, the known i.i.d. model corresponds to each job having a type which determines a set of workers that can work on that job. Based on past input instances, one can derive how frequently a job of a given type arrives. Then the input instance is assumed to be generated by simply sampling types of jobs independently from this distribution. Analyzing an algorithm with this extra information is equivalent to analyzing the algorithm in the known i.i.d. model.

Many other applications of online bipartite matching exist. For example, in the kidney exchange problem [28], the offline nodes correspond to kidney recipients and online nodes correspond to donors. The donors arrive online and edges represent compatibility constraints. Most bipartite matching problems have weighted variations that model real-world applications more realistically with internet advertising (e.g., Display Ads, AdWords) being a commonly studied case (see [23] for a detailed introduction). In the task of assigning jobs to workers, an edge weight could represent the profit of the assignment of the job to the corresponding worker. In this article, we study the unweighted problem, which models compatibility constraints and remains an important research topic for many applications.

In our study, we consider both synthetically generated type graphs as well as some type graphs based on real-world applications. Our experimental study indicates that simple greedy and greedy-like algorithms (that are unaware of the type graph) perform quite well in terms of the observed competitive ratio when compared to the significantly more complex algorithms designed to exploit the given known type graph. That is, while the provable worst-case approximation ratios (in expectation over the distribution) of these non-greedy algorithms are much better than what can be achieved by the simple greedy-like algorithms we consider, there is a good reason why practitioners might want to use simple greedy algorithms. The more complicated algorithms for known type graphs are stated as being non-greedy (in the sense that an online node is not necessarily matched whenever possible). However, we show that "greediness" can be easily achieved without loss of generality and, moreover, greediness is necessary for any algorithm to achieve good performance in practice.

The remainder of the article is organized as follows. In Section 2, we describe the set of algorithms under consideration. This includes two simple greedy algorithms (namely, a simple deterministic greedy algorithm and the randomized Ranking algorithm [17]), and five state-of-the-art algorithms for the known type graph model with integral types. Some of these algorithms have only been informally described in the literature and we provide a more detailed description when needed. We also consider a linear time two-pass "online" algorithm [11], which experimentally is almost a proxy for obtaining optimality. In Section 3, we discuss the datasets we use as well as the experimental setup. Section 4 provides the experimental results in terms of the observed competitive ratio. We also provide some timing results verifying that indeed the simple linear time greedy algorithms are significantly faster than the algorithms designed for known type graphs. Finally, in Sections 5 and 6, we summarize the experimental results drawing some overall conclusions from our experimental study.

2 PRELIMINARIES

We consider bipartite graphs $G = (L, R, E)$ with bi-partition (L, R). We shall often refer to the nodes in L as the left nodes, or the left-hand-side (LHS, for short) nodes, or the online nodes. Similarly, the nodes in R are referred to as the right nodes, the right-hand-side (RHS) nodes, or the offline nodes.

In the online version of bipartite matching, the right side is known to the algorithm in advance. The left-hand-side nodes are revealed one-by-one in a given order. When an online node is revealed, all its neighbors are revealed as well. After each arrival of an online node, the algorithm makes an irrevocable decision on which neighbor to match the current online node (if at all). Without loss of generality, we can also allow for the offline nodes to be associated with different capacities. A node having capacity c means it can be matched at most c times. This is achieved by generating c copies of that node before running the algorithm.

2.1 Definitions and Notation

Let M be a matching in a bipartite graph $G = (L, R, E)$. We say $\ell \in L$ *participates* in the matching M if there is $r \in R$ such that $\{\ell, r\} \in M$. We write $M(\ell)$ to denote such r. If ℓ does not participate in M, then we define $M(\ell) := \perp$. The same notions are defined for $r \in R$ symmetrically.

We shall measure the performance of an algorithm in one of two ways: in terms of the observed asymptotic approximation ratio, or in terms of the fraction of the matched offline nodes.

Definition 2.1. Let ALG be an online algorithm (possibly randomized) solving the bipartite matching problem over random graphs G_n parameterized by the input size $n = |R|$. We write $\mathrm{ALG}(G_n)$ to denote the expected size of the matching (random variable) that is constructed by running ALG on G_n. We write $\mathrm{OPT}(G_n)$ to denote the size of a maximum matching in G_n. *The asymptotic approximation ratio* of ALG with respect to G_n is defined as

$$\rho(\mathrm{ALG}, G_n) = \liminf_{n\to\infty} \frac{\mathbb{E}(\mathrm{ALG}(G_n))}{\mathbb{E}(\mathrm{OPT}(G_n))}.$$

The fraction of matched offline nodes of ALG with respect to G_n is defined as

$$\mu(\mathrm{ALG}, G_n) = \liminf_{n\to\infty} \frac{\mathbb{E}(\mathrm{ALG}(G_n))}{n}.$$

The expectations above are taken over the randomness of the algorithm and the randomness of the input.

2.2 Known I.I.D. Model and Integral Types

In the known i.i.d. model, one first chooses a *type graph* $G = (L, R, E)$ and a distribution $p : L \to [0, 1]$ on the LHS nodes. In this case, the nodes in L are also referred to as types. The type graph together with the distribution is given to the algorithm in advance. In the known i.i.d. model, an actual input instance $\hat{G} = (\hat{L}, R, \hat{E})$ is a random variable and is generated from G as follows. The right-hand side R is the same in G and $\hat{G}$, but the left-hand-side of $\hat{G}$ consists of m i.i.d. samples from p. Thus, say a given node $\hat{\ell} \in \hat{L}$ has type $\ell \in L$, then the neighbors of $\hat{\ell}$ in $\hat{G}$ are the same as the neighbors of ℓ in G. The graph $\hat{G}$ is presented to the algorithm in the vertex arrival model (the order of vertices is the same as the order in which they were generated). Note that a particular type ℓ can be absent altogether or can be repeated a number of times in $\hat{G}$. We refer to $\hat{G}$ as the *instance graph*. Note that the instance graph is fully specified by a pair (G, v) where G is a type graph and v is a vector of types, i.e., $v \in L^m$. When G is clear from the context, we will refer to v as an instance. The probability of seeing a particular vector v is given by $p(v) = \prod_{i=1}^{m} p(v_i)$.

Table 1. Algorithms with their Respective Provable Competitive Ratios

Algorithm	Worst-case analysis
BrubachEtAl	0.7299 [8]
JailletLu	0.7293 $(1 - 2/e^2)$ [15]
ManshadiEtAl	0.7025 [21]
BahmaniKapralov	0.6990 [3]
Ranking	0.6961 [20]
FeldmanEtAl	0.6702 $(\frac{1-2/e^2}{4/3-2/3e})$ [12]
Category-Advice	0.6321 $(1 - 1/e)$ [11]
3-Pass	0.6321 $(1 - 1/e))$ [6]
Greedy	0.6321 $(1 - 1/e)$ [17]
MinDegree	0.6321 $(1 - 1/e)$
KarpSipser	0.6321 $(1 - 1/e)$

A known i.i.d. problem is said to have *integral types* if the expected number of times a particular type occurs is integral. We will denote the number of times type ℓ occurs in an instance by the random variable Z_ℓ. Then the condition of integral types is equivalent to $\mathbb{E}(Z_\ell) = p(\ell)m \in \mathbb{Z}$. While the parameters $|L|$, $|R|$, and m can all be different, the most common setting is $m = |L|$. This assumption together with integral types implies that without loss of generality one can take p to be the uniform distribution on L (by duplicating types as necessary). An additional common assumption is that $|L| = |R|$. In that case, we talk about a single parameter $n = |L| = |R| = m$.

In our empirical evaluations, we only consider integral types, so when we say "known i.i.d. model" we mean the known i.i.d. model with integral types and uniform distribution, unless stated otherwise.

2.3 Algorithms

In this section, we describe all algorithms that are included in our experimental study:

(1) SimpleGreedy.
(2) Ranking due to Karp et al. [17].
(3) FeldmanEtAl due to Feldman et al. [12].
(4) BahmaniKapralov due to Bahmani and Kapralov [3].
(5) ManshadiEtAl due to Manshadi et al. [21].
(6) JailletLu due to Jaillet and Lu [15].
(7) BrubachEtAl due to Brubach et al. [8].
(8) Category-Advice due to Dürr et al. [11].
(9) 3-Pass due to Borodin et al. [6].
(10) MinDegree.
(11) KarpSipser due to Karp and Sipser [16].
(12) Offline optimal algorithm that runs Edmonds-Karp flow algorithm on the canonical flow network associated with a bipartite graph. Sometimes, we initialize the algorithm by a solution computed by one of the other algorithms.

The provable competitive ratios of these algorithms are shown in Table 1. We begin by presenting several algorithms that work in the online adversarial setting. This is followed by the description of algorithms that work in the known i.i.d. setting, and other algorithms that do not fit into online

or known i.i.d. settings. Observe that algorithms that are designed for the online adversarial setting also work in the known i.i.d. setting—they just ignore the side information, i.e., the type graph. It is worth noting that preprocessing is the running time bottleneck of algorithms using the type graph and the matching phase is executed in linear time for all algorithms.

2.3.1 Algorithms for Online Adversarial Setting. We start with a helper subroutine, which we call GreedyWithPermutation. This online algorithm accepts the RHS R and a permutation π of R. The rank of $r \in R$, denoted by $\mathrm{rk}_\pi(r)$, is the position of r when in the arrangement of R according to π. The GreedyWithPermutation algorithm matches each online node with an available neighbor of smallest rank (if there is at least one available neighbor). The pseudocode is presented in Algorithm 1.

ALGORITHM 1: A helper algorithm.

procedure GreedyWithPermutation($G = (L, R, E), \pi : R \to R$)
  for all $\ell \in L$ **do**
    When ℓ arrives, let $N(\ell)$ be the set of unmatched neighbors of ℓ.
    if $N(\ell) \neq \emptyset$ **then**
      Match ℓ with $\arg\min\{\mathrm{rk}_\pi(r) \mid r \in N(\ell)\}$.

SimpleGreedy. Next, we describe the simplest online algorithm—SimpleGreedy. The SimpleGreedy algorithm is obtained by fixing a permutation π on the RHS and applying GreedyWithPermutation. While π could be any fixed permutation (not depending on the type graph), for concreteness, we define it to be the following. The RHS nodes are labeled with strings over some alphabet. We define π_{alphabet} to be the ordering of the RHS nodes alphabetically according to their labels. Thus, formally SimpleGreedy(G)=GreedyWithPermutation(G, π_{alphabet}). A tight analysis of the performance of Greedy in the random order and known i.i.d. models can be found in [14].

Remark 2.1. A word of caution with regard to the terminology: SimpleGreedy should not be confused with an arbitrary *greedy* algorithm. When we say that an algorithm is greedy, we mean that it has the following property: whenever a given online node has at least one unmatched neighbor, this online node is guaranteed to be matched. This property alone is not sufficient to specify the algorithm, since the algorithm also needs to break ties when several unmatched neighbors are available. SimpleGreedy is a very specific greedy algorithm, which breaks ties according to an alphabetical order. It turns out that *any algorithm* for online bipartite matching can be turned into a greedy one without hurting its approximation ratio. In particular, without loss of generality, an optimal algorithm is greedy. Thus, the whole area of designing good online algorithms for bipartite matching revolves around designing better and better tie-breaking rules. We discuss this in more detail below when we talk about more advanced algorithms for the known i.i.d model.

Ranking. The next algorithm is Ranking due to Karp et al. [17]. Unlike the previous algorithms, Ranking is randomized. Let S_R denote the set of all permutations of the RHS R. Ranking samples π uniformly at random from S_R prior to seeing any online nodes. This is followed by running GreedyWithPermutation with π as the input permutation—see Algorithm 2. The original paper contained a bug in the proof of the algorithm's performance and alternative proofs were later published [10, 14].

2.3.2 Algorithms for Known I.I.D. Setting. We start with a special subroutine. Consider a bipartite graph of maximum degree 2, that is, a set of paths and cycles. Such a graph can be decomposed

ALGORITHM 2: A randomized algorithm due to Karp et al. [17]

procedure RANKING($G = (L, R, E)$)
Sample a permutation $\pi : R \to R$ uniformly at random.
Run GREEDYWITHPERMUTATION(G, π).

into two matchings, which we will call blue and red. Strictly speaking, the blue subgraph returned by the subroutine is not always a matching; sometimes it is a matching plus some extra edges. However, the blue subgraph always satisfies the property that there is at most one edge incident on each LHS node, i.e., the blue subgraph is a "matching on the left." For simplicity and slightly abusing notation, we shall sometimes refer to both blue and red subgraphs as matchings. However, for clarity, we can say that the blue edges form a "semi-matching." When we actually run the Feldman et al. algorithm on an i.i.d. instance, the blue edges become a matching as determined by the assignment of the online node. We present a particular decomposition in Algorithm 3, which we call BLUEREDDECOMPOSITION and which is due to Feldman et al. [12]. This decomposition is used in several algorithms that we consider later.

ALGORITHM 3: Blue red decomposition due to Feldman et al. [12]. Applies to bipartite graphs of maximum degree 2.

procedure BLUEREDDECOMPOSITION($G = (L, R, E)$)
Color edges of the cycles alternating blue and red.
Color edges of the odd-length paths alternating blue and red, with more blue than red.
For the even-length paths that start and end with nodes in R, alternate blue and red.
For the even-length paths that start and end with nodes in L, color the first two edges blue, then alternate red, blue, red, blue, and so on.
return (semi-matching formed by blue edges, matching formed by red edges).

FELDMANETAL. The first algorithm to ever beat the $1 - 1/e$ barrier of the online adversarial model in the known i.i.d. model is due to Feldman et al. [12]. The algorithm has a preprocessing stage and the online stage. In the preprocessing stage, the algorithm solves the following modification of the standard network flow problem for biparite matching: add two new nodes s and t, add directed edges each from s to r for each $r \in R$, and add directed edges from ℓ to t for each $\ell \in L$, orient the rest of the edges in G from RHS to LHS (these edges will be called the graph edges). Each outgoing edge from s, as well as each incoming edge into t, has capacity 2. The rest of the edges have capacities 1. We denote this flow network by $\widetilde{G}$. The algorithm of Feldman et al. finds an integral optimal solution to this network flow problem. The subgraph induced by the graph edges with positive flow on them has maximum degree 2. The last step of the preprocessing stage is to apply BLUEREDDECOMPOSITION to this subgraph to obtain a blue semi-matching M_b and a red matching M_r. In the online stage, the algorithm receives online nodes in the i.i.d. fashion and matches them as follows: if a node of type i arrives for the first time, the algorithm tries to match it to $M_b(i)$. If $M_b(i) = \perp$ or $M_b(i)$ has been previously matched, the algorithm leaves the current node unmatched. If a node of type i arrives for the second time, the algorithm tries to match it to $M_r(i)$. Otherwise, a node of type i is left unmatched. See Algorithm 4 for the pseudocode.

BAHMANIKAPRALOV. Bahmani and Kapralov [3] observed that the performance of the Feldman et al. algorithm can be improved by modifying the preprocessing stage. Recall, that G refers to the type graph, $\widetilde{G}$ to the associated flow network in FELDMANETAL, and f is an integral max flow in $\widetilde{G}$. Consider a subset A of L and define A^z to be those vertices in A such that the amount of

ALGORITHM 4: The known i.i.d. algorithm of Feldman et al. [12].

procedure FeldmanEtAl($G = (L, R, E)$ – type graph)

▷ Preprocessing stage:

Set up flow network $\widetilde{G} = (\widetilde{V}, \widetilde{E})$, where
$\widetilde{V} = L \cup R \cup \{s, t\}$
$\widetilde{E} = \{(s, r) \mid r \in R\} \cup \{(\ell, t) \mid \ell \in L\} \cup \{(r, \ell) \mid \{r, \ell\} \in E\}$.
Set up capacities $cap(s, r) = 2, cap(\ell, t) = 2$ for $\ell \in L, r \in R$ and $cap(r, \ell) = 1$ for $(r, \ell) \in \widetilde{E}$.
Solve the flow network to obtain a maximum integral flow f.
Let G' denote the bipartite subgraph induced by edges $\{r, \ell\}$ such that $f(r, \ell) = 1$.
Set $(M_b, M_r) =$ BlueRedDecomposition(G').

▷ Online stage:

for all arriving online nodes u **do**
  Let ℓ denote the type of u.
  if it is the first arrival of type ℓ and $M_b(\ell) \neq \perp$ and $M_b(\ell)$ is unmatched **then**
    Match u to $M_b(\ell)$.
  if it is the second arrival of type ℓ and $M_b(\ell) \neq \perp$ and $M_b(\ell)$ is unmatched **then**
    Match u to $M_r(\ell)$.

flow through them in f is z for $z \in \{0, 1, 2\}$. In other words, no flow goes through vertices in A^0, one unit of flow goes through each vertex in A^1, and two units of flow go through each vertex in A^2. The main insight of Bahmani and Kapralov is that the more balanced the flow is the better, i.e., we want A^1 to be as large as possible. They give a procedure that redirects some of the flow from A^2 into A^0 without affecting the optimality of the flow. The procedure actually works on two sets $A \subseteq L$ and $B \subseteq R$ and can be done to balance the flow either on the left or on the right. We first describe the procedure and then show which sets to apply it to in order to improve on the algorithm of Feldman et al.

Let us first define the procedure to balance the left side (the right side can be handled similarly). The algorithm sets up a completely new flow network $\widehat{G}$ as follows: the vertex set of the network consists of $A \cup B$ together with two new vertices s_A and t_A. We add an edge (s_A, a) of unit capacity for each $a \in A^2$ and an edge (a, t_A) of unit capacity for each $a \in A^0$. For each edge (b, a) such that $a \in A, b \in B$, and $f(b, a) = 1$, we add an edge (a, b) to the flow network of unit capacity (note that this essentially reverses the edges with positive flow in $\widetilde{G}$). For each (b, a) in $\widetilde{G}$ with $f(b, a) = 0$ (in $\widetilde{G}$), we add an edge (b, a) to $\widehat{G}$ of unit capacity (note that this essentially preserves the graph edges in $\widetilde{G}$ that do not carry any flow). Let f_A denote an integral maximum flow in the newly constructed flow network. If we use the convention that $f(a, b) = -f(b, a)$, then by adding f_A to f on edges (b, a) and fixing the flow on edges (a, t) accordingly, we essentially "undo" some flow going into A^2 nodes and replace it with a flow going into A^0 nodes in $\widetilde{G}$.

Perhaps, this is best illustrated with a small example. Consider $K_{4,2}$ type graph, where $L = \{\ell_1, \ell_2, \ell_3, \ell_4\}$ and $R = \{r_1, r_2\}$. One possible max flow that the Feldman et al. algorithm finds for the corresponding network is to send two units of flow through r_1 and into ℓ_1, ℓ_2 and to send two units of flow through r_2 and into ℓ_1, ℓ_2. Call this flow f. Consider $A = L$ and $B = R$. Then $A^2 = \{\ell_1, \ell_2\}$ and $A^0 = \{\ell_3, \ell_4\}$. Solving the new flow problem corresponding to the balancing procedure, we find that we can send one unit of flow through ℓ_1 to r_1 and to ℓ_3 and another unit of flow through ℓ_2 to r_2 and to ℓ_4. Thus, this new flow can be used to augment f: it undoes one unit of flow from r_1 to ℓ_1 and replaces it with one unit of flow from r_1 to ℓ_3, and it undoes one unit of flow from r_2 to ℓ_2 and replaces it with one unit of flow from r_2 to ℓ_4. This results in a new flow

being completely balanced on the LHS, i.e., $A^1 = L$. The two balancing procedures are described in Algorithms 5 and 6.

ALGORITHM 5: The balancing procedure on the LHS due to Bahmani and Kapralov [3] that takes as input type graph G, a maximum integral flow for the flow network from Feldman et al. and two sets $A \subseteq L$ and $B \subseteq R$. Returns a flow f_A in the new flow network that can be used to define a more balanced f.

procedure BALANCELEFT($G = (L, R, E), A, B, f$)
Set up flow network $\widehat{G}_A = (\widehat{V}_A, \widehat{E}_A)$, where
$\widehat{V}_A = A \cup B \cup \{s_A, t_A\}$
$\widehat{E}_A = \{(s_A, a) \mid a \in A^2\} \cup \{(a, t_A) \mid a \in A^0\} \cup \{(a, b) \mid f(b, a) = 1\} \cup \{(b, a) \mid f(b, a) = 0\}$.
Set capacities of all edges in $\widehat{E}_A$ to 1.
Solve the flow network to obtain a maximum integral flow f_A.
return f_A.

ALGORITHM 6: The balancing procedure on the RHS due to Bahmani and Kapralov [3] that takes as input type graph G, a maximum integral flow for the flow network from Feldman et al., and two sets $A \subseteq L$ and $B \subseteq R$. Returns a flow f_A in the new flow network that can be used to define a more balanced f.

procedure BALANCERIGHT($G = (L, R, E), A, B, f$)
Set up flow network $\widehat{G}_B = (\widehat{V}_B, \widehat{E}_B)$, where
$\widehat{V}_B = A \cup B \cup \{s_B, t_B\}$
$\widehat{E}_B = \{(s_B, b) \mid b \in B^0\} \cup \{(b, t_B) \mid b \in B^2\} \cup \{(a, b) \mid f(b, a) = 1\} \cup \{(b, a) \mid f(b, a) = 0\}$.
Set capacities of all edges in $\widehat{E}_B$ to 1.
Solve the flow network to obtain a maximum integral flow f_B.
return f_B.

Now, let (S, T) be the min cut in the flow network $\widetilde{G}$ obtained in a standard way: S is defined to be the set of nodes reachable from s in the residual network defined by max flow f. Define $S_L = S \cap L, S_R = S \cap R, T_L = T \cap L, T_R = T \cap R$. The Bahmani and Kapralov algorithm computes $f_L = \text{BALANCELEFT}(G, T_L, T_R, f)$ and $f_R = \text{BALANCERIGHT}(G, S_L, S_R, f)$. It then creates a subgraph of G consisting of those edges $\{r, \ell\}$ that have $f(r, \ell) + f_L(r, \ell) + f_R(r, \ell) > 0$ (using the convention $f(u, v) = -f(v, u)$). The rest is exactly as in Feldman et al.—use BLUEREDDECOMPOSITION on this subgraph and use the resulting blue and red matchings in the online stage in the same way as Feldman et al. See Algorithm 7 for the pseudocode.

MANSHADIETAL. The next algorithm is due to Manshadi et al. [21] and it is based on the idea of a fractional optimal solution. Fix an algorithm for obtaining an offline optimal solution (e.g., Edmonds-Karp). Consider all possible instances arising out of the given type graph $G = (L, R, E)$. Recall, that an instance can be described as a vector of types $v \in L^n$. We assume without loss of generality that the expected number of arrivals of nodes of a given type ℓ is bounded above by 1. The matching M given by the optimal algorithm can be viewed as an indicator vector of length $|E|$ indexed by edge names. This indicator vector specifies for each position $\{\ell, r\}$ whether $\{\ell, r\}$ is in M or not. Abusing the notation, we denote this indicator vector by $\text{OPT}(v)$. An optimal fractional solution is given by the expected value of this indicator vector, i.e., $f_{\text{OPT}} = \sum_{v \in L^n} p(v)\, \text{OPT}(v)$.

ALGORITHM 7: The known i.i.d. algorithm of Bahmani and Kapralov [3].

procedure BahmaniKapralov($G = (L, R, E)$ – type graph)

▷ Preprocessing stage:

Compute f as in Algorithm 4.
Compute the canonical (S, T) cut from f.
Set f_L = BalanceLeft(G, T_L, T_R, f).
Set f_R = BalanceRight(G, S_L, S_R, f).
Let G' be induced by edges $\{r, \ell\}$ such that $f(r, \ell) + f_L(r, \ell) + f_R(r, \ell) > 0$ (using the convention $f(u, v) = -f(v, u)$).
Set (M_b, M_r) = BlueRedDecomposition(G').

▷ Online stage:

Same as in Algorithm 4.

Observe that $f_{\text{OPT}} \in [0, 1]^E$ and for each edge $\{\ell, r\} \in E$, we have $f_{\text{OPT}}(\{\ell, r\})$ = the probability that edge $\{\ell, r\}$ appears in an optimal matching.

Let $W_{\{\ell,r\}}$ denote the random variable indicating the event that $\{\ell, r\}$ appears in an optimal matching. Let Z_ℓ denote the number of online nodes generated of type ℓ. For each ℓ, we have $\sum_{r:\{\ell,r\}\in E} W_{\{\ell,r\}} \le Z_\ell$. By taking the expectation of both sides, we have $\sum_{r:\{\ell,r\}\in E} f_{\text{OPT}}(\{\ell, r\}) \le 1$ (using the assumption described above). For a given type ℓ let $r_1, \ldots, r_k$ be its neighbors in G ordered such that $f_{\text{OPT}}(\{\ell, r_1\}) \ge f_{\text{OPT}}(\{\ell, r_2\}) \ge \cdots \ge f_{\text{OPT}}(\{\ell, r_k\})$. Add a dummy node r_{k+1} and define $f_{\text{OPT}}(\{\ell, r_{k+1}\}) = 1 - \sum_{i=1}^{k} f_{\text{OPT}}(\{\ell, r_i\})$. The dummy node simulates the event that ℓ is not matched in an optimal solution—for the purpose of the algorithm, the dummy node is always considered to be matched before the online stage.

Now, $f_{\text{OPT}}(\{\ell, \cdot\})$ defines a probability mass function (PMF) on the neighbors of ℓ. The algorithm of Manshadi et al. samples two random neighbors from this distribution during the online stage in the following correlated fashion. Partition the interval $[0, 1]$ into $k + 1$ consecutive non-overlapping intervals I_p where the length of I_p is $f_{\text{OPT}}(\{\ell, r_p\})$. We denote this partition by $\mathcal{I}_\ell$. Also, partition the interval $[0, 1]$ into $k + 1$ consecutive non-overlapping intervals J_p where the length of J_p is $f_{\text{OPT}}(\{\ell, r_{p+1}\})$ if $p \le k$ and the length of J_{k+1} is $f_{\text{OPT}}(\{\ell, r_1\})$. We denote this partition by $\mathcal{J}_\ell$. In order to sample from the PMF on the neighbors of ℓ, one could sample a uniform random number between 0 and 1 and output the neighbor of ℓ corresponding to the interval to which the number belongs. If we do this procedure independently for I intervals and J intervals, we get two independent samples. Instead, Manshadi et al. do the correlated sampling—a single number is sampled between 0 and 1. Let I_p and J_q be the intervals in which this number falls. The two neighbors returned by the procedure are the two neighbors of ℓ corresponding to I_r and J_q. The partitioning of $[0, 1]$ into I intervals and J intervals was chosen so that there is as little overlap between intervals corresponding to the same neighbor as possible.

When an online node of type ℓ arrives, the algorithm of Manshadi et al. performs a correlated sampling from $\mathcal{I}_\ell$ and $\mathcal{J}_\ell$ as described above. Let $r_{\ell,1}$ and $r_{\ell,2}$ denote the two samples returned by the correlated sampling procedure. The algorithm tries to match the online node first to $r_{\ell,1}$. If $r_{\ell,1}$ was matched previously, the algorithm tries to match the online node to $r_{\ell,2}$. If $r_{\ell,2}$ was matched previously, the algorithm gives up on matching the online node. See Algorithm 8 for the pseudocode. There is an outstanding issue of how to compute f_{OPT} in practice. This is a difficult problem, and rather than computing it exactly, Manshadi et al. suggest approximating it by the Monte Carlo method—sample a number of instances, solve them optimally, and record the fraction of times each edge appears in an optimal offline solution. This is what we do in our implementation, as well (see Section 2.5).

ALGORITHM 8: The known i.i.d. algorithm of Manshadi et al. [21].

procedure ManshadiEtAl($G = (L, R, E)$ – type graph)

▷ Preprocessing stage:

Compute a fractional optimal matching f_{OPT}.
For each ℓ construct the two partitions $\mathcal{I}_\ell$ and $\mathcal{J}_\ell$.

▷ Online stage:

for all arriving online nodes u **do**
Let ℓ denote the type of u.
Let $r_{\ell,1}$ and $r_{\ell,2}$ be the two neighbors of ℓ returned by the correlated sampling procedure performed on $\mathcal{I}_\ell$ and $\mathcal{J}_\ell$ as described in the text.
if $r_{\ell,1}$ is unmatched **then**
Match u to $r_{\ell,1}$.
else if $r_{\ell,2}$ is unmatched **then**
Match u to $r_{\ell,2}$.
else
Leave u unmatched.

JailletLu. Jaillet and Lu [15] introduced a template of algorithms called Random Lists Algorithms, RLA for short, for online bipartite matching under the known i.i.d. input model. For type ℓ, define Ω_ℓ to be the set of all possible ordered (sub)lists of neighbors of ℓ in the type graph. In the preprocessing stage, an RLA constructs a distribution D_ℓ on Ω_ℓ for each $\ell \in L$. In the online stage, when a node of type ℓ arrives, the RLA samples a list of neighbors from D_ℓ and matches the online node to the first available neighbor according to that list. If there are no available neighbors in that list, the online node is left unmatched. The pseudocode for this template appears in Algorithm 9. In order to get an actual algorithm out of this template, one has to specify how D_ℓ are constructed in the preprocessing step.

ALGORITHM 9: Random Lists Algorithm template due to Jaillet and Lu [15].

procedure RLA($G = (L, R, E)$ – type graph)

▷ Preprocessing stage:

For each $\ell \in L$ construct a distribution D_ℓ on Ω_ℓ.

▷ Online stage:

for all arriving online nodes u **do**
Let ℓ denote the type of u.
Sample a list of neighbors of ℓ from Ω_ℓ according to D_ℓ.
if all neighbors in the list are matched **then**
Leave u unmatched.
else
Match u to the first available neighbor in the list.

Jaillet and Lu [15] also gave an actual algorithm based on this template, which we refer to as JailletLu. Jaillet and Lu consider the following LP:

$$\begin{array}{lll} \text{maximize} & \sum_{\ell \in L, r \in R} f_{\ell,r} & \\ \text{subject to} & \sum_{\ell:\{\ell,r\}\in E} f_{\ell,r} \le 1 & r \in R, \\ & \sum_{r:\{\ell,r\}\in E} f_{\ell,r} \le 1 & \ell \in L, \\ & f_{\ell,r} \in [0, 2/3] & \ell \in L, r \in R, \{\ell, r\} \in E. \end{array} \tag{1}$$

A vertex solution f^* to this LP has the property that $f^*_{\ell,r} \in \{0, 1/3, 2/3\}$ for all $\ell \in L, r \in R$. Restrict the neighbors of ℓ to only those r that have $f^*_{\ell,r} > 0$. There can be at most three neighbors, since for such r we have $f^*_{\ell,r} \geq 1/3$. If $\sum_{r:\{\ell,r\}\in E} f^*_{\ell,r} < 1$, then add a dummy node d_ℓ and define $f^*_{\ell,d_\ell} = 1 - \sum_{r:\{\ell,r\}\in E} f^*_{\ell,r}$. Even after adding dummy nodes, each ℓ has at most three neighbors. Jaillet and Lu define D_ℓ such that it is supported only on lists of these restricted neighborhoods. More specifically, if ℓ has a single neighbor, then D_ℓ assigns unit weight to the list consisting of that neighbor; if ℓ has two neighbors r_1, r_2, then D_ℓ assigns probability f^*_{ℓ,r_1} to the list $\langle r_1, r_2\rangle$ and probability f^*_{ℓ,r_2} to the list $\langle r_2, r_1\rangle$; if ℓ has three neighbors r_1, r_2, r_3, then D_ℓ assigns probability $1/6$ to each permutation of r_1, r_2, r_3. After that, JailletLu runs RLA with these distributions.

BrubachEtAl. Next we describe the state-of-the-art[3] algorithm for the known i.i.d. input model with integral arrival rates due to Brubach et al. [8]. This algorithm is (predictably) the most difficult to explain and implement. It is a RLA-style algorithm. The preprocessing stage consists of five steps:

(1) solve a special LP,
(2) round the solution,
(3) apply the first modification to the rounded solution,
(4) apply the second modification to the modified solution from the second step, and
(5) define distributions D_ℓ on Ω_ℓ for each $\ell \in L$.

Next, we describe each of these steps in detail. **Step 1**—Brubach et al. define and solve the following LP:

$$\begin{array}{lll} \text{maximize} & \sum_{\ell\in L, r\in R} f_{\ell,r} & \\ \text{subject to} & \sum_{\ell:\{\ell,r\}\in E} f_{\ell,r} \leq 1 & r \in R, \\ & \sum_{r:\{\ell,r\}\in E} f_{\ell,r} \leq 1 & \ell \in L, \\ & 0 \leq f_{\ell,r} \leq 1 - \frac{1}{e} & \ell \in L, r \in R, \{\ell, r\} \in E, \\ & f_{\ell_1,r} + f_{\ell_2,r} \leq 1 - \frac{1}{e^2} & \ell_1, \ell_2 \in L, r \in R, \{\ell_1, r\}, \{\ell_2, r\} \in E, \end{array} \tag{2}$$

The idea behind LP (2) is to introduce extra constraints to bring the optimal value of the objective down closer to the fractional optimal solution, while maintaining feasibility of the fractional optimal solution. Let f^* denote an optimal solution to (2). **Step 2** is to apply the rounding procedure of Gandhi et al. [13] to $3f^*$, i.e., f^* multiplicatively scaled by 3. This results in an integral vector $\widetilde{f}$ such that $\widetilde{f}_{\ell,r} \in \{0, 1, 2, 3\}$. Then, Brubach et al. scale the rounded solution back down and set $h := \widetilde{f}/3$. For completeness, we describe the rounding procedure here. Say an edge in our bipartite graph is fractional if $f^*_{\ell,r} \notin \mathbb{Z}$. While there are fractional edges remaining, repeat the following. Find either a cycle or a maximal path consisting only of fractional edges. Let P denote this cycle/path, and partition it into two matchings M_1 and M_2. Define

$$\alpha = \min\left\{\gamma > 0 \mid (\exists (i,j) \in M_1 : f^*_{i,j} + \gamma = \lceil f^*_{i,j}\rceil) \wedge (\exists (i,j) \in M_2 : f^*_{i,j} - \gamma = \lfloor f^*_{i,j}\rfloor)\right\},$$

$$\beta = \min\left\{\gamma > 0 \mid (\exists (i,j) \in M_1 : f^*_{i,j} - \gamma = \lfloor f^*_{i,j}\rfloor) \wedge (\exists (i,j) \in M_2 : f^*_{i,j} + \gamma = \lceil f^*_{i,j}\rceil)\right\}.$$

With probability $\beta/(\alpha+\beta)$ round $f^*_{i,j}$ to $f^*_{i,j} + \alpha$ for all $\{i,j\} \in M_1$ and to $f^*_{i,j} - \alpha$ for all $\{i,j\} \in M_2$. With complementary probability, round $f^*_{i,j}$ to $f^*_{i,j} - \beta$ for all $\{i,j\} \in M_1$ and to $f^*_{i,j} + \beta$ for all $\{i,j\} \in M_2$

Step 3—the first modification to h. Restrict the original type graph to a subgraph of edges $\{\ell, r\}$ such that $h_{\ell,r} > 0$. This graph is sparse—each online node can have at most three neighbors. In Step 3, the goal is to break certain 4-cycles—see Figure 1 for details. Formally, this procedure is

[3]The state-of-the-art is in terms of the best provable competitive ratio over worst-case type graphs.

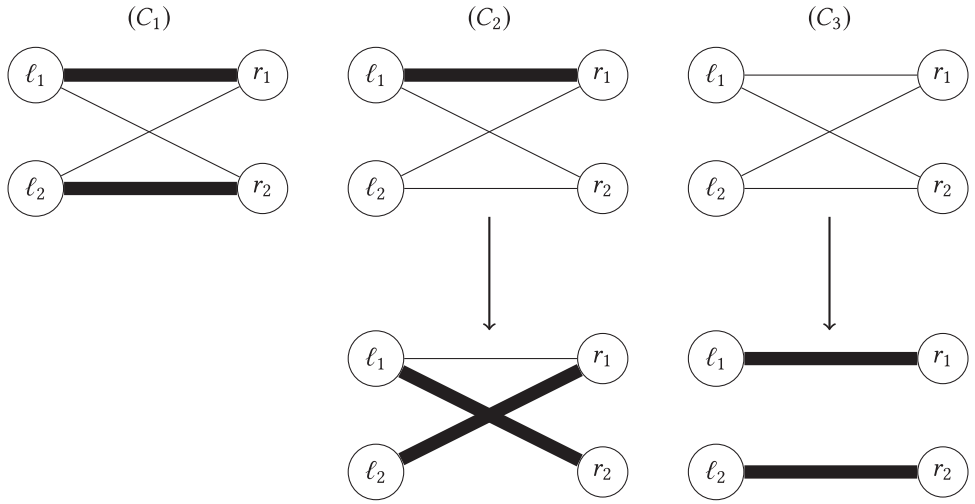

Fig. 1. Three possible cycles induced by h. The thin edges correspond to $h_{\ell,r} = 1/3$ and thick edges correspond to $h_{\ell,r} = 2/3$.

done by breaking all (C_2)-type cycles first. Then, if there is a (C_3)-type cycle, break it. Return to trying to break (C_2) cycles. This way you always try to break (C_2) cycles first. This continues until all (C_2) and (C_3) cycles are broken.

Step 4—the second modification to h. We call the result of this modification h'. This modification is presented in Figure 2. In that figure, the numbers next to an offline node r indicate the total value of h at that node, i.e., $\sum_\ell h(\ell, r)$. Thin edges correspond to $h_{\ell,r} = 1/3$ and thick edges correspond to $h_{\ell,r} = 2/3$. The number above the edge corresponds to the newly assigned h'. For example, a thin edge with value 0.15 above it means that $h_{\ell,r} = 1/3$ and after modification we have $h'(\ell, r) = 0.15$. Any edges not covered by one of the cases in the figure retain their old value of h.

Lastly, in **Step 5**, the distributions on lists are defined as follows. If ℓ has 1 or 0 neighbors in the sparse graph based on h', then the distribution is fully supported on either the single-element list or the empty list, respectively. If ℓ has two neighbors, say, r_1 and r_2, then the distribution is supported on two lists (r_1, r_2) and (r_2, r_1) with the probability of (r_1, r_2) being proportional to $h'(\ell, r_1)$. If the neighborhood of ℓ consists of three vertices, say, r_1, r_2, r_3, then the distribution is supported on all possible permutations of (r_1, r_2, r_3), such that the probability that the list is (r_i, r_j, r_k) is proportional to $\frac{h'(\ell,r_i)h'(\ell,r_j)}{h'(\ell,r_j)+h'(\ell,r_k)}$. Algorithm 10 summarizes this procedure.

2.3.3 Algorithms for Other Settings. **Category-Advice.** Dürr et al. [11] suggested a greedy-like algorithm that performs a second pass over the input called Category-Advice. The Category-Advice algorithm belongs to the class of category algorithms that were introduced in the work of Dürr et al. These algorithms are neither online nor known i.i.d. They can be viewed as conceptually simple offline algorithms, or online algorithms with advice (see [7]), or as defining their own computational model.

A category algorithm starts with a permutation σ of the offline nodes (e.g., given adversarially, or by an alphabetical order of names of the offline nodes). Instead of running GreedyWithPermutation directly with σ, the algorithm starts by computing a category function $c : R \to \mathbb{Z}$. The algorithm updates σ to σ_c as follows: σ_c is the unique permutation satisfying that for all $v_1, v_2 \in R$, we have $\sigma_c(v_1) < \sigma_c(v_2)$ if and only if $c(v_1) < c(v_2)$ or $(c(v_1) = c(v_2)$ and $\sigma(v_1) < \sigma(v_2))$. Then, GreedyWithPermutation is performed with σ_c as the permutation of the offline nodes. In other

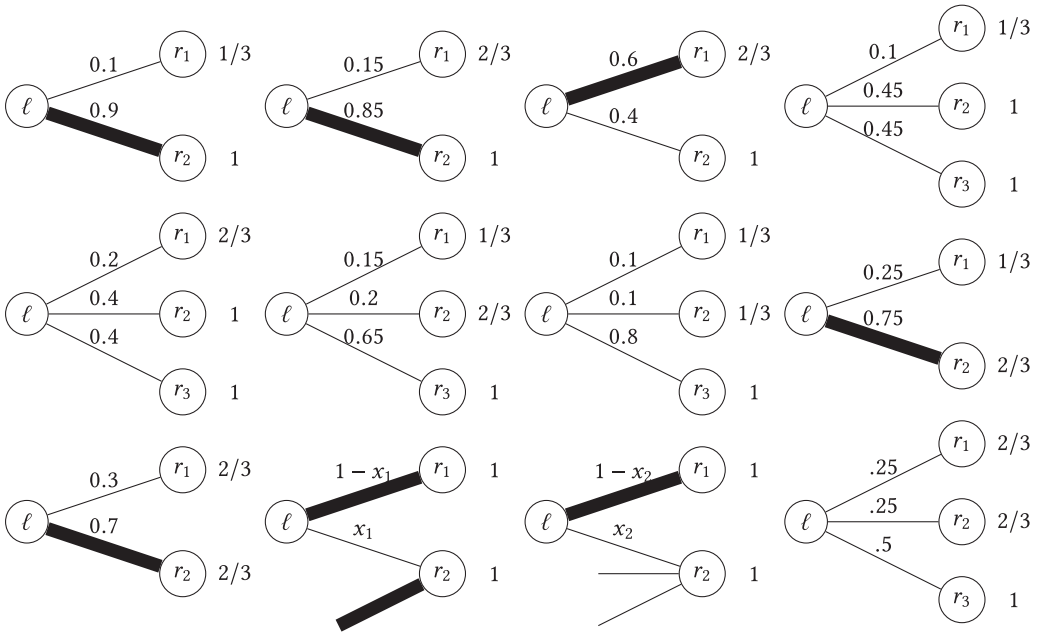

Fig. 2. Three possible cycles induced by h. The thin edges correspond to $h_{\ell,r} = 1/3$ and thick edges correspond to $h_{\ell,r} = 2/3$. Numbers above edges correspond to new values of h'. The numbers next to r nodes correspond to total values of h. The two magic numbers are $x_1 = 0.2744$ and $x_2 = 0.15877$.

ALGORITHM 10: The known i.i.d. algorithm due to Brubach et al. [8].

procedure BrubachEtAl($G = (L, R, E)$ – type graph)

▷ Preprocessing stage:

Solve LP (2). Let f^* denote an optimal solution. (**Step 1**)

Scale f^* multiplicatively to $3f^*$ and apply the rounding procedure of Gandhi et al. [13]. (**Step 2**)

Set h to be the scaled down (multiplicatively by 1/3) rounded solution.

Apply the two modification steps to get h'. (**Steps 3 and 4**)

Define the distributions on (sub)lists of neighbors. (**Step 5**)

▷ Online stage:

Run RLA with the above distribution.

words, a category algorithm partitions the offline nodes into $|\mathrm{Im}(c)|$ categories and specifies the ranking of the categories; the ranking within the category is induced by the initial permutation σ.

The Category-Advice algorithm starts with σ, and in the first pass, runs GreedyWithPermutation with σ. Let M be the matching obtained in the first pass. The category function $c : R \to [2]$ is defined as follows: $c(v) = 1$ if v does not participate in M and $c(v) = 2$ otherwise. In the second pass, the Category-Advice algorithm runs GreedyWithPermutation with σ_c. The output of the second run of GreedyWithPermutation is declared as the output of the Category-Advice algorithm. In other words, in the second pass the algorithm gives preference to those vertices that were **not** matched in the first pass. Algorithm 11 shows the pseudocode.

3-Pass. The algorithm of Dürr et al. was extended to multiple passes in [6]. In this article, we shall only consider the generalization of the algorithm to three passes, which we call 3-Pass. In the first

ALGORITHM 11: The CATEGORY-ADVICE algorithm of Dürr et al. [11].

procedure CATEGORY-ADVICE($G = (L, R, E), \sigma : R \to R$)
Set $M =$ GREEDYWITHPERMUTATION(G, σ).
Define $c : R \to [2]$ by $c(v) = 1$ if $M(v) = \perp$ and $c(v) = 2$ otherwise.
Define σ_c as stated in the main text.
Return GREEDYWITHPERMUTATION(G, σ_c).

two passes, the algorithm behaves the same way as CATEGORY-ADVICE. The generalization is quite natural: in the third pass, the algorithm prefers to match an incoming node to an offline node that was not matched in the first or second pass. If there is no such node available, then 3-PASS prefers to match an incoming node to an offline node that was not matched in the first pass. If there is no such offline node, then 3-PASS matches an incoming node to the first (according to the original fixed ordering) available offline node.

MINDEGREE. A commonly used heuristic for matching in general graphs is picking the node with the minimum degree, the idea being that this lowers the probability of the remaining nodes becoming unmatchable. In the offline setting, the minimum degree heuristic can be utilized to decide not only which node to match next, but also which neighbor it should be matched to, resulting in a one-sided and a two-sided version of the MinDegree algorithm, respectively. This distinction, along with the tie-breaking strategy used, gives rise to a family of MinDegree algorithms, some of which have also been studied in the context of bipartite graphs [18]. In our study, we consider a natural online variation of the MINDEGREE algorithm where the offline degrees, as they are being formed online,[4] guide the matches. Since the arriving online node is always the one to be matched next, ours can be considered as a one-sided algorithm in this regard. We break ties in a fixed predetermined order.

ALGORITHM 12: ONLINE MINDEGREE

procedure MINDEGREE($G = (L, R, E)$)
Initialize the degree of every offline node to 0.
for all $\ell \in L$ **do**
When ℓ arrives, increment the degrees of its neighbors by one.
Let $N(\ell)$ be the set of unmatched neighbors of ℓ of minimum degree.
if $N(\ell) \neq \emptyset$ **then**
Match ℓ with $\arg\min\{\text{rk}_{\pi_{alphabet}}(r) \mid r \in N(\ell)\}$.

KARPSIPSER. Similar to the MINDEGREE algorithm, Karp and Sipser [16] considered an algorithm that picks a node of degree 1, if it exists, otherwise it chooses a node randomly. This is another commonly studied strategy and was shown to be quite effective for offline bipartite matching [18]. As with MINDEGREE, we consider an online variation of this greedy algorithm that looks at the offline degrees.

2.4 Conversion to Greedy

As mentioned in Remark 2.1, all of the complicated known i.i.d. algorithms from the previous section are presented in the corresponding papers as non-greedy to simplify the analysis. For example, suppose that u is an online node of type ℓ. Moreover, assume that it is the third arrival of type ℓ

[4] Each new online node increases the degree of its neighbors by one.

ALGORITHM 13: Online KarpSipser

procedure KarpSipser($G = (L, R, E)$)
  Initialize the degree of every offline node to 0.
  for all $\ell \in L$ **do**
    When ℓ arrives, increment the degrees of its neighbors by one.
    Let $N(\ell)$ be the set of unmatched neighbors of ℓ of degree one.
    Let $N'(\ell)$ be the set of unmatched neighbors of ℓ.
    if $N(\ell) \neq \emptyset$ **then**
      Match ℓ with $\arg\min\{\mathrm{rk}_{\pi_{alphabet}}(r) \mid r \in N(\ell)\}$.
    else if $N'(\ell) \neq \emptyset$ **then**
      Sample a permutation $\pi : R \to R$ uniformly at random.
      Match ℓ with $\arg\min\{\mathrm{rk}_{\pi}(r) \mid r \in N'(\ell)\}$.

and consider the behavior of FeldmanEtAl. Regardless of how many neighbors of u are available, FeldmanEtAl is not going to match u since FeldmanEtAl only attempts to match first and second arrivals of a given type. A greedy algorithm would match u if it had at least one available neighbor. Similar considerations hold for the rest of the algorithms in that section. Thus, vanilla versions of these algorithms immediately forgo a constant fraction of possible matches in order to simplify the analysis and optimize for the worst-case. Clearly, there are type graphs (e.g., the complete type graph), on which any greedy algorithm would be able to find a perfect matching. On such graphs, the complicated algorithms would be vastly outperformed by any greedy algorithm.

Fortunately, as stated in Pena and Borodin [26] and sketched in Borodin et al. [6], there is a simple idea to turn all of these algorithms into greedy ones while preserving their worst-case guarantees. The idea is just to run a greedy algorithm, and if there are several available neighbors, break ties by using the suggestions of the non-greedy algorithm. More precisely, we are going to convert a non-greedy algorithm ALG to a greedy algorithm ALG′. At all times, ALG′ will simulate ALG and know to which node $r_i \in R$ (if any) ALG would match each online node $\ell_i \in L$. Consider the first time that a non-greedy algorithm ALG is about to leave an online node $\ell_i \in L$ unmatched even though there is an available neighbor. Instead, our greedy algorithm ALG′ will match ℓ_i with an arbitrary available neighbor $r' \in R$. Now ALG′ continues on simulating ALG until ALG tries to match some later ℓ_j with the r' used to match ℓ_j. If there is still an available r'' to match ℓ_j, then that match is made. And we continue in this manner always making a match according to ALG when possible and otherwise making an arbitrary match if one is still available. If at any time ALG wanted to match some later ℓ_j to some r that has been used by ALG′, ℓ_j will go unmatched. But in this case, the addition of matching ℓ_i offsets the loss of not matching ℓ_j. Moreover, this modification is easy to implement and does not seem to have a significant effect on the runtime. To apply this conversion to an adaptive[5] algorithm, we would keep a second copy of the graph for the non-greedy algorithm to operate on so that the performance guarantee is maintained despite our greedy choices. In our experiments, we report the performance of both greedy and non-greedy versions of known i.i.d. algorithms.

2.5 Notes on Implementation

We used the adjacency list representation of graphs for all of the above algorithms. Compared to adjacency matrix representation, this allowed for significant speedup on sparse graphs.

[5] An algorithm whose order of preference over the offline nodes changes dynamically as new nodes arrive and the matching is formed.

The max flow problems in FeldmanEtAl and BahmaniKapralov are solved via straightforward implementations of the Edmonds-Karp max flow algorithm. The same algorithm is used to obtain an optimal maximum matching.

For ManshadiEtAl, we estimate a fractional optimal solution by running the Edmonds-Karp algorithm initialized with a greedy solution (for speed) on 100 samples generated for a given type graph.

The linear program (1) in JailletLu can be formulated as a max flow problem with integral capacities. This is done by rescaling constraints by a multiple of 3, and constructing the following flow network. Add a source s and a sink t, connect s to each $r \in R$ via edges of capacity 3, connect each $\ell \in L$ to t via edges of capacity 3, orient edges of G from R to L, and assign capacity 2 to them. In our implementation, we use Edmonds-Karp to solve this max flow problem via an integral flow. Then f^* can be obtained by scaling the max flow by a multiple of 1/3.

We solve the linear program (2) in BrubachEtAl using the simplex method in the GNU Linear Programming Kit (GLPK) [1]. The actual code is freely available at [2].

3 EXPERIMENTAL SETUP

All our experiments were performed on a personal laptop with an Intel Core i5-7300HQ processor clocked at 2.5 GHz. The laptop had 8 GB 2400 MHz DDR4 of RAM and 256 GB M.2 SSD. The laptop was running Windows 10 64-bit Home edition. All algorithms under consideration were coded in C++ and compiled with Microsoft Visual Studio Community 2017 version 15.5.7. The code was compiled for the 64-bit target architecture with an optimization flag O2. The implementation is single-threaded, so all algorithm runs were performed on a single core.

In the rest of this section, we describe our benchmarks for online bipartite matching algorithms under the known i.i.d. input model with integral types. Our benchmarks can naturally be split into three categories; namely, parameterized families of graphs, stand-alone graphs, and bipartite graphs derived from real-world graphs (which we will call "real-world graphs" for short). Graphs in these categories refer to *type graphs* with the understanding that *instance graphs* corresponding to a particular type graph from the benchmark will be obtained by sampling n online nodes uniformly at random from all possible types, i.i.d.

Families of graphs are obtained by either a random or a deterministic process that has a natural parameter. For example, this parameter could be a proxy for edge density of a graph. For families of graphs, we will be interested in the performance of the algorithms as a function of the given parameter.

We call a graph stand-alone if it is obtained either by a random or a deterministic process, but there are no associated parameters. For example, worst-case graphs for online algorithms. Although all graphs can be parameterized by the size of the graph, we are interested in the asymptotic behavior of the algorithms on large graphs, so we typically take stand-alone graphs of largest size that can be solved in reasonable time by all algorithms under consideration. Note that stand-alone graphs are not necessarily fixed and can still be the result of a random process.

Stand-alone graphs FewG, ManyG, Rope, Hexa, Zipf are taken from Cherkassky et al. [9], where these graphs were used to measure the performance of various offline algorithms for bipartite matching. Our implementation of the generating procedures for these graphs does not perfectly match the code accompanying the paper [9], because their code is designed for more general families of graphs. Instead, our implementation follows the descriptions in the paper [9] itself, where parameters are often fixed to certain values that simplify the generating process.

We also consider a number of graphs that are publicly available from online repositories.

Most of our synthetically generated instances are bipartite. A few of our synthetically generated instances, as well as all real-world instances are non-bipartite. In case of a non-bipartite graph, we

use one of the following two ways of creating a bipartite graph out of a non-bipartite graph. Let $G = (V, E)$ be a given graph that is not necessarily bipartite. The first way of creating a bipartite graph out of G is the standard technique of the *the duplicating method*. The idea is to duplicate the vertex set V. Let $L = V$ be the first copy of V and $R = V$ be the second copy. Put an edge between $\ell \in L$ and $r \in R$ if and only if $\{\ell, r\} \in E$. We call the second way of creating a bipartite graph out of G *the random balanced partition method*. In this method, we partition V randomly into two blocks L and R, such that $|L| = \lfloor |V|/2 \rfloor$ and $|R| = \lceil |V|/2 \rceil$. We keep only those edges that connect two vertices from different partitions. Solving the matching problem on a graph obtained from the random balanced partition method applied to social network graphs has a natural interpretation. This corresponds to dividing the whole population into two groups and pairing up as many "friends" ("co-authors," "co-stars," etc.) from the two groups as possible.

3.1 Families of Graphs

Erdős-Rényi Graphs. A graph of this family is denoted by $G_{n,n,p}$. We have that $|L| = |R| = n$ and for each $\ell \in L$ and $r \in R$ an edge $\{\ell, r\}$ is included in G with probability p independently. We consider p to be of the form c/n and c is the parameter defining this family of graphs.

Random Regular on the Left (Right) Graphs. We say that a graph G is d-regular on the left (right) if the degree of every vertex in L (in R) is the same and equal to d. To generate a random graph that is d-regular on the left, for each ℓ we sample a uniformly random subset of d vertices from R and declare them to be neighbors of ℓ. The samples for different ℓ are independent. The procedure to generate d-regular graphs on the right is analogous. These families of graphs are parameterized by d.

Molloy-Reed. Molloy and Reed [24] gave a procedure to generate a graph with a given degree distribution p. We describe the procedure for non-bipartite graphs. To generate a graph on n nodes, for each node u, sample its degree from p. Initially, degree d of u corresponds to d non-paired ends of edges. The idea is to choose randomly two such ends of edges and connect them together—this forms an edge and decreases the number of non-paired edges by one for each of the two participating vertices. While there are vertices with non-paired edges, pick two such vertices at random and pair up one end of an edge from the first vertex with one end of an edge from the second vertex. There are a couple of problems with this procedure as stated. First of all, if the sum of all degrees is odd, this procedure will leave one end of an edge non-paired. This is fixed by modifying the first step—after sampling degrees of vertices and before pairing up any ends of edges. While the total degree is odd, pick a random vertex and resample its degree. The second problem is that this procedure does not necessarily generate a *simple graph*—i.e., there might be self-loops and duplicate (parallel) edges. To address this issue, when pairing up edges, we perform 100 random samples of pairs of vertices to try and find ends of edges that do not result in self-loops or parallel edges. If all of these trials fail, then we add the self-loop or the parallel edge of the last trial. At the end of the procedure we obtain the graph by removing all self-loops and parallel edges.

While the Erdős-Rényi model is natural, it does not seem to model many real-life scenarios, such as social networks. It has long been observed that degree distributions of many social networks (e.g., Facebook, Twitter, movie actor databases, researcher co-authorship databases) are not binomial, but rather seem to have heavy tails. Thus, they are more accurately modeled by power-law distributions. Newman et al. [25] describe a particular family of distributions that combined with the Molloy-Reed procedure results in a fairly accurate model of many social networks. This family of distributions is called a power-law distribution with exponential cutoff. This distribution has two parameters: τ, which is called the exponent, and κ, which is called the cutoff. The idea is that for small values of d, the probability of a node having degree d should be modeled by $x^{-\tau}$ (the

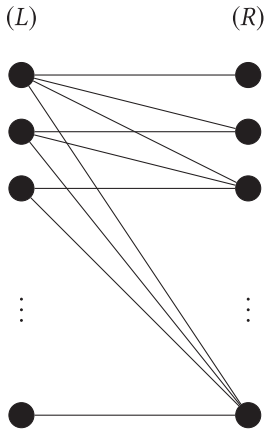

Fig. 3. Upper Triangular graph.

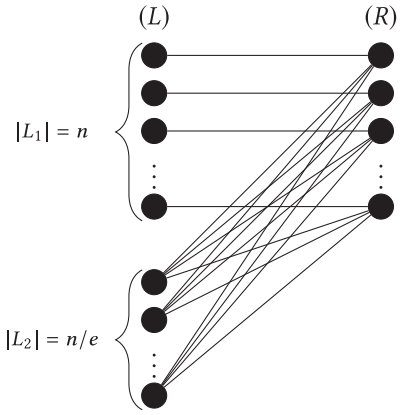

Fig. 4. Manshadi-Hard graph.

power-law part), but for $d > \kappa$ the probability should be dropping off exponentially (the exponential cutoff part). Formally, it is defined as follows. Let p_d denote the probability of our random variable having value d; then we have

$$p_d = \begin{cases} 0 & \text{if } d = 0 \\ cx^{-\tau} e^{-d/\kappa} & \text{if } d > 0, \end{cases}$$

where c is the normalizing constant. The Molloy-Reed procedure on a power-law distribution with exponential cutoff, followed by the random balanced partition method, defines a family of type graphs that is parameterized by τ and κ.

Preferential Attachment Bigraphs. We also consider the following natural modification of the preferential attachment model that immediately produces bipartite graphs without having to use the random balanced partition method. We refer to this model as the *preferential attachment bigraph* model. To generate a bigraph in this model, start with n offline nodes R and introduce online nodes L one at a time. The model has a single parameter c which is the average degree of an online node. When a new online node $i \in L$ arrives, sample $Z_i \sim \text{Bin}(n, c/n)$ to decide on a number of its offline neighbors. Let d_j denote the current degree of an offline node $j \in R$. Define a probability distribution μ on offline nodes such that $\mu(j) = \frac{1+d_j}{n+\sum_{t \in R} d_t}$. Sample RHS nodes from μ i.i.d. repeatedly until Z_i *unique* offline nodes are generated. These offline nodes define the neighborhood of the current online node i. Update the d_j and continue.

3.2 Stand-Alone Graphs

Upper-Triangular (UT). Figure 3. This graph is the fixed graph defined by an upper-triangular adjacency matrix with the columns representing online nodes that arrive from right to left. This is known to be the worst-case example for Ranking in the adversarial online model [17].

Manshadi-Hard (MH). Figure 4. In [21], Manshadi et al. present a type graph for which no online algorithm can achieve an expected competitive ratio better than $1 - \frac{1}{e^2} \approx 0.86$. Let $G(L, R, E)$ be the type graph where $L = L_1 \cup L_2$, $|L_1| = |R| = n$, and $L_2 = n/e$. There is a perfect matching between the vertices of L_1 and R and a complete bipartite graph between L_2 and R. Each type has arrival rate 1 and there are $|L| = n(1 + 1/e)$ online i.i.d. draws.

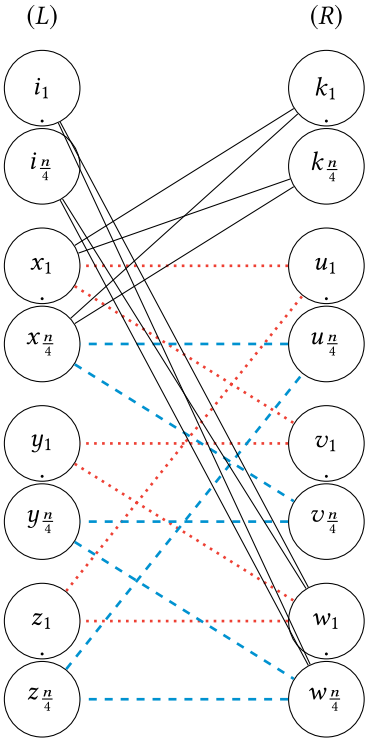

Fig. 5. Feldman-Hard graph. Depicted with red-dotted and blue-dashed edges are the cycles $(u_1, x_1, v_1, y_1, w_1, z_1, u_1)$ and $(u_{n/4}, x_{n/4}, v_{n/4}, y_{n/4}, w_{n/4}, z_{n/4}, u_{n/4})$, respectively. The remaining edges form two complete bipartite graphs.

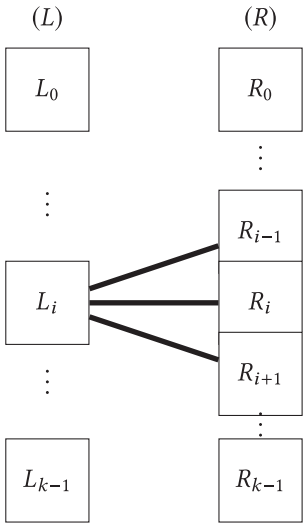

Fig. 6. Graph FewG and ManyG. For $i = 0, \ldots, k-1$, vertices in L_i are assigned random neighbors from groups R_{i-1} to R_{i+1}.

Feldman-Hard (FH). Figure 5. Feldman et al. [12] present a family of graphs that is the worst case for their algorithm, proving that their analysis of the competitive ratio of their algorithm is tight. R is partitioned into four blocks: K, U, V, and W, each of size $n/4$. Similarly, L is partitioned into four blocks: I, X, Y, and Z, each of size $n/4$. We use a lower-case letter to refer to an element in the given block, e.g., elements of U are denoted by u_i, where $i \in [n/4]$. The edge set consists of a 6-cycle $(u_i, x_i, v_i, y_i, w_i, z_i, u_i)$ for $i \in [1, \frac{n}{4}]$, a complete bipartite between K and X, and a complete bipartite graph between I and W.

FewG and ManyG. Figure 6. To construct these bipartite graphs, the vertices in L are randomly permuted and then L and R are partitioned into k groups of equal size. Each vertex of the i-th group of L is assigned Y random neighbors from the $(i-1)$-th through $(i+1)$-th group of R (with wrap around). To be consistent with previous literature ([9]), Y is set to be binomially distributed with $\mathbb{E}(Y) = 5$, and we consider the two cases of $k = 32$ **(FewG)** and $k = 256$ **(ManyG)**.

rope. Figure 7. For this graph, vertices in L and R are grouped into $t = n/d$ groups of size d, denoted $L_0 \ldots L_{t-1}$ and $R_0 \ldots R_{t-1}$. Block i on one side is connected to block $i+1$ on the other side, for $i = 0 \ldots t-2$; block L_{t-1} is connected to block R_{t-1}. Thus, the graph is a "rope" that zigzags between the two sides of the graph, first up and then down. Consecutive pairs of blocks along

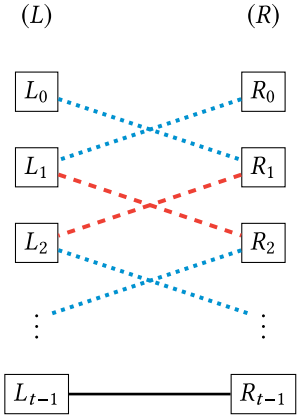

Fig. 7. Graph Rope. Blue dotted edges correspond to perfect matchings and red dashed edges correspond to random bipartite graphs. Edge (L_{t-1}, R_{t-1}) can be either dotted or dashed.

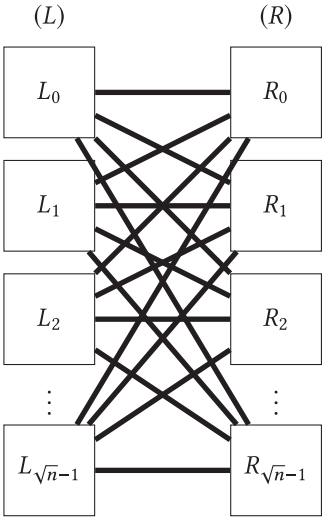

Fig. 8. Graph Hexa. Connecting groups form a complete bipartite graph. Each edge depicts a random hexagon between the corresponding groups.

the rope are connected alternately by perfect matchings and random bipartite graphs of average degree $d-1$, beginning and ending with perfect matchings. As in [9], we fix $d = 6$.

HEXA. Figure 8. In these graphs, each side is partitioned into $\sqrt{n}$ blocks of size $\sqrt{n}$ each. A random bipartite hexagon is added between block i on one side and block j on the other side for all $i, j \in [\sqrt{n}]$. This results in the average degree of each vertex being 6. The random hexagon is generated by the following procedure. Pick three random nodes on each side (inside the corresponding blocks), say, ℓ_1, ℓ_2, ℓ_3 and r_1, r_2, r_3. Sample two random permutations $\pi, \sigma : [3] \to [3]$. Add the following cycle to the graph $(\ell_{\pi(1)}, r_{\sigma(1)}, \ell_{\pi(2)}, r_{\sigma(2)}, \ell_{\pi(3)}, r_{\sigma(3)}, \ell_{\pi(1)})$.

ZIPF. In these bipartite graphs, we have $|L| = |R| = n$ and an edge between nodes $\ell_i \in L$ and $r_j \in R$ exists with probability roughly proportional to $1/ij$. More precisely, the probability is $\Pr(\ell_i \sim r_j) = \min(\frac{n \cdot d}{\log^2 n} \cdot \frac{1}{i \cdot j}, 1)$ with $d = 6$ and $i, j \in [n]$. This results in graphs that are denser around vertices with smaller indices.

3.3 Real-World Data

To perform experiments on real datasets, we used some publicly available graphs from the Network Data Repository [27]. In the experiments, we used both the duplicating method and the random balanced partition method of bipartite transformations.

The **socfb** datasets are social frienship networks extracted from Facebook. Nodes are users and edges represent friendship ties. The **bio-CE** datasets correspond to biological datasets representing links by similar phylogenetic profiles and gene neighborhoods of bacterial and archaeal orthologs. We also used two **econ** datasets that model US economic transactions in 1972

Table 2. Real-World Graph Statistics

Dataset	Application Domain	Nodes	Edges	Max Degree	Avg. Degree
SOCFB-CALTECH36	social networks	769	16.7k	248	43
SOCFB-REED98	social networks	962	18.8k	313	39
BIO-CE-GN	biological networks	2.2k	53.7k	242	48
BIO-CE-PG	biological networks	1.9k	47.8k	913	51
ECON-MBEAFLW	economic networks	492	49.5k	679	201
ECON-BEAUSE	economic networks	507	44.2k	766	174

by connecting commodities to commodities and industries. These datasets, along with various properties of the corresponding graphs, can be found in the following links[6]:

—http://networkrepository.com/socfb-Caltech36.php
—http://networkrepository.com/socfb-Reed98.php
—http://networkrepository.com/bio-CE-GN.php
—http://networkrepository.com/bio-CE-PG.php
—http://networkrepository.com/econ-mbeaflw.php
—http://networkrepository.com/econ-beause.php

Table 2 summarizes some real-world graph statistics.

4 EXPERIMENTAL RESULTS

In this section, we present results of our experiments and provide some comments about the experiments. However, we leave the main discussion about performance of algorithms and lessons learned from the experiments to Section 5. With as many algorithms and as many graphs as we consider in this article, it is difficult to present all of the data in a completely satisfying way. We settled on the following presentation formats. For families of graphs, we plot the performance of an algorithm as a time series with an independent variable being the parameter corresponding to the family of graphs and the dependent variable being the achieved competitive ratio. The time series allows us to identify regimes of parameters that are easy and that are hard for most algorithms. We list the performance of algorithms in those regimes sorted according to their competitive ratios. When we plot the results for these regimes, we also graphically indicate *sample standard deviations* by *horizontal line segments*: the length of each segment is 2 standard deviations and the segment is centered around the sample mean. We treat stand-alone and real-world instances differently. We collect the performance of all algorithms on all stand-alone instances in one table, and on real-world instances in two tables (one for the random bipartition conversion method and one for the duplicating method). We also use the following notation: we add a letter "g" in brackets following an algorithm's name to indicate the greedy version of the algorithm, e.g., FELDMANETAL(G). The algorithm's name by itself (e.g., FELDMANETAL) refers to a non-greedy version of the algorithm. We have also tested for statistical significance of the results using t-tests and even when the performance of two algorithms differs by 0.01, the p-values are less than 1%, allowing us to compare them with confidence. The rest of this section is organized as follows. We describe the results for families of graphs in Subsections 4.1, 4.2, 4.3, and 4.4. We present our results for stand-alone graphs in Subsection 4.5 and real-world instances in Subsection 4.6. Finally, we finish with a small discussion of running times in Subsection 4.7.

[6]Accessed 2018-05-25.

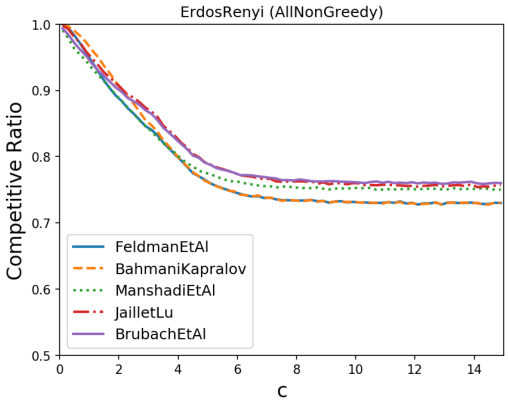

Fig. 9. Performance of all non-greedy algorithms on Erdős-Rényi family of graphs.

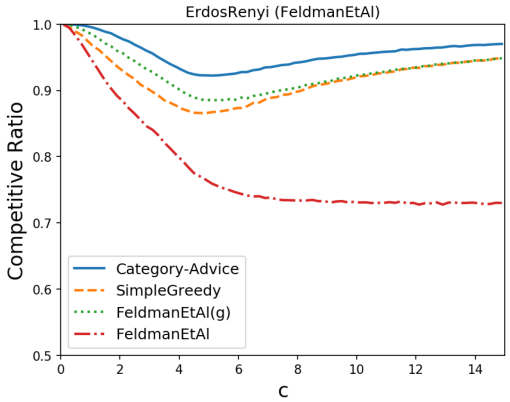

Fig. 10. Performance of FeldmanEtAl algorithm on Erdős-Rényi family of graphs.

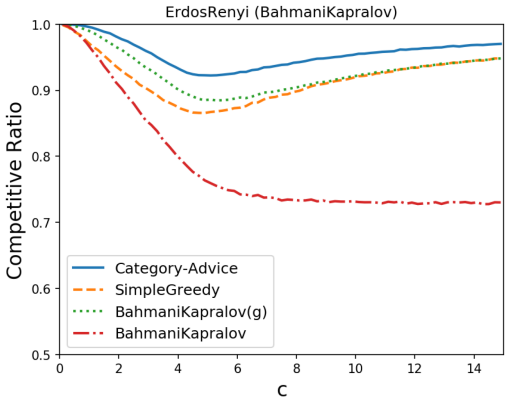

Fig. 11. Performance of BahmaniKapralov algorithm on Erdős-Rényi family of graphs.

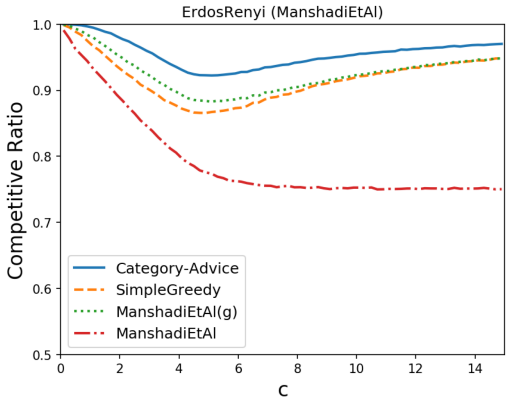

Fig. 12. Performance of ManshadiEtAl algorithm on Erdős-Rényi family of graphs.

4.1 Erdős-Rényi Experiments

The experiments in this section were performed with Erdős-Rényi type graphs where the number of nodes was fixed to be 1,000 on each side, and the parameter c varied from 0.1 to 14.9 with a step of 0.2. For each value of c, 100 type graphs were generated. The reported competitive ratios of algorithms are (ratios of) the average values over these 100 trials. In Figure 9 you can see the time series of performance of all non-greedy algorithms in this experiment. Each non-greedy algorithm is compared with greedy algorithms (including its own counterpart) in Figures 10, 11, 12, 13, and 14. We did not plot Ranking, since its behavior in this experiment was analogous to that of SimpleGreedy. Observe that from Figures 9 and, for example, 10, one can infer all other figures. We only show other figures here for completeness, and in the future experiments we shall omit them. Looking at the figures, we observe that there are essentially three regimes of c that are of interest in this experiment: (1) small c, i.e., a sparse type graph, regime; (2) "hard" values of c, where the relative order of algorithms changes, and performance of greedy algorithms experiences a dip; and (3) asymptotic, i.e., steady-state, value of c, where the performance guarantees of various non-greedy algorithms stabilizes. In order to "zoom-in" and see what happens in each of these regimes,

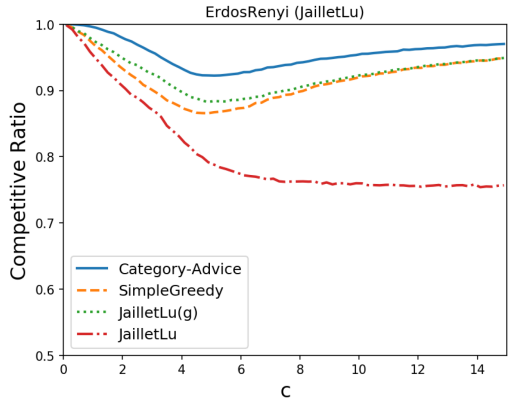

Fig. 13. Performance of JailletLu algorithm on Erdős-Rényi family of graphs.

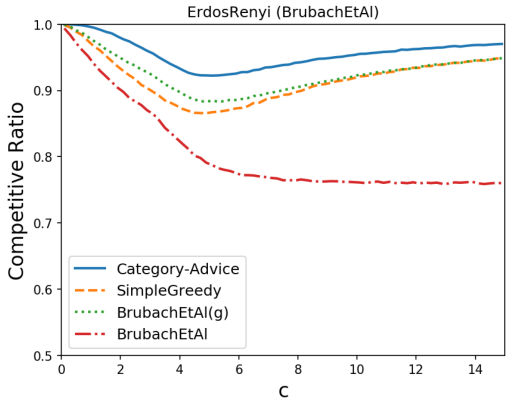

Fig. 14. Performance of BrubachEtAl algorithm on Erdős-Rényi family of graphs.

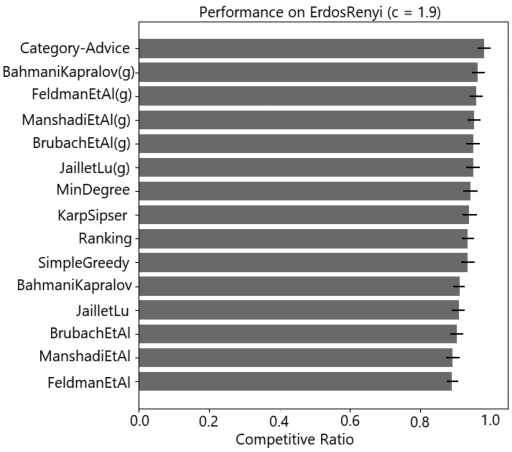

Fig. 15. Performance of all algorithms on Erdős-Rényi graph with $c = 1.9$.

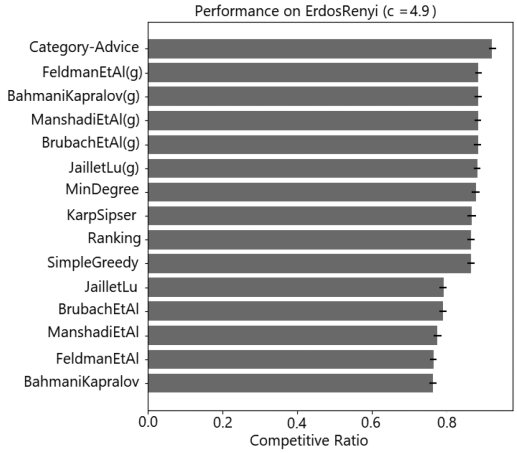

Fig. 16. Performance of all algorithms on Erdős-Rényi graph with $c = 4.9$.

we plotted competitive ratios of algorithms in decreasing order (top to bottom) for $c = 1.9$ (regime (1)), $c = 4.9$ (regime (2)), and $c = 14.9$ (regime(3)) in Figures 15, 16, and 17, respectively.

4.2 Random Left-Regular Experiments

The experiments in this subsection are based on type graphs with 1,000 nodes on each side, where left-hand-side nodes are of degree d each. As before, results are averaged over 100 i.i.d. trials. We present time series of all non-greedy algorithms in Figure 18, and we present FeldmanEtAl versus greedy algorithms in Figure 19. Figures comparing other non-greedy algorithms with greedy algorithms are omitted, since they are very similar to Figure 19, as discussed at the beginning of Subsection 4.1. We identify three regimes of d that correspond to (1) sparse case ($d = 2$), (2) difficult case ($d = 5$), and (3) asymptotic case ($d = 30$). This is very similar to what we did in Subsection 4.1. The competitive ratios of different algorithms under these regimes are plotted in Figures 20, 21, and 22.

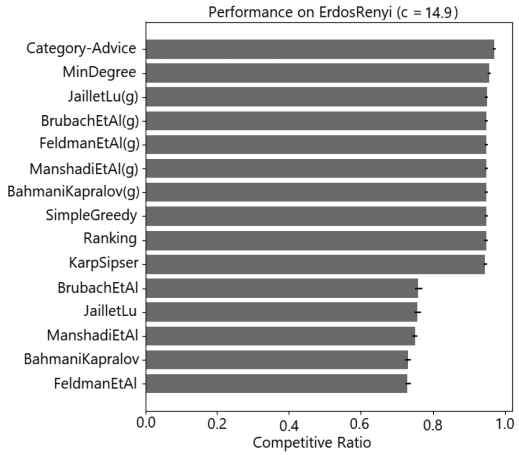

Fig. 17. Performance of all algorithms on Erdős-Rényi graph with $c = 14.9$.

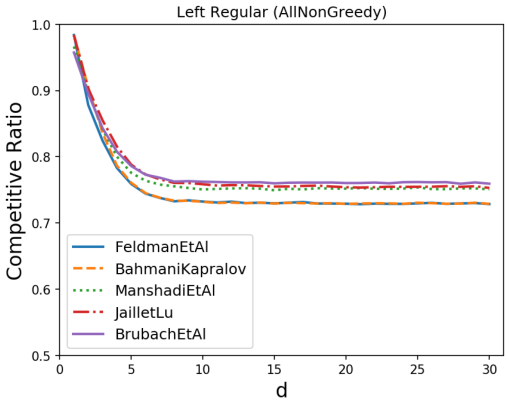

Fig. 18. Performance of all non-greedy algorithms on Left-Regular family of graphs.

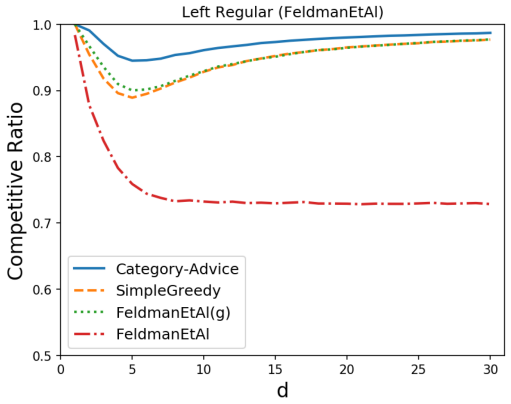

Fig. 19. Performance of FeldmanEtAl algorithm on Left-Regular family of graphs.

4.3 Molloy-Reed Experiments

The Molloy-Reed family of graphs has two parameters: τ and κ. Thus, we generated a whole grid of results. More specifically, for each value of τ from 0.5 to 4.0 with a step of 0.1 and for each value of κ from 1 to 96 with a step of 5, we generated 100 Molloy-Reed graphs with those values of τ and κ and averaged competitive ratios of algorithms over these 100 runs. Since plotting three-dimensional time series is awkward, we present τ- and κ-slices of the resulting grid for values of τ and κ that exhibit more interesting behavior. We show what competitive ratios of non-greedy algorithms look like as a function of τ when κ is fixed to 96 in Figure 23, and as a function of κ when τ is fixed to 0.5 in Figure 24. Time series comparing the non-greedy version of BahmaniKapralov with greedy algorithms for the respective scenarios are shown in Figures 25 and 26. As in other subsections, comparisons of other non-greedy algorithms with greedy algorithms look very similar, so we omit them. For $\tau = 0.5$, we identify two regimes: difficult regime for greedy algorithms, where $\kappa = 11$; and a steady-state regime, where $\kappa = 41$. We "zoom in" to show competitive ratios of algorithms for these two regimes in Figures 27 and 28. Similarly, the two regimes for $\kappa = 96$ are when $\tau = 1.0$ and when $\tau = 2.0$. Those are depicted in Figures 29 and 30.

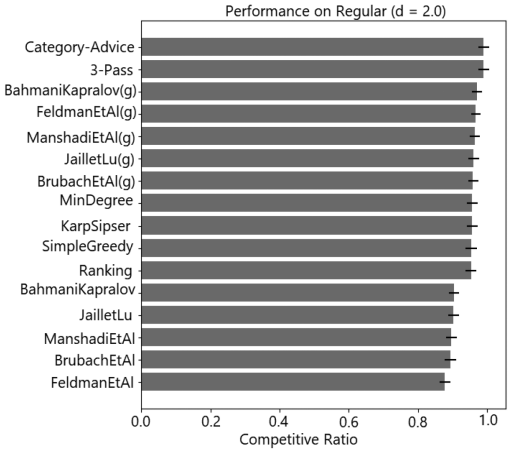

Fig. 20. Performance of all algorithms on Left-Regular graph with $d = 2$.

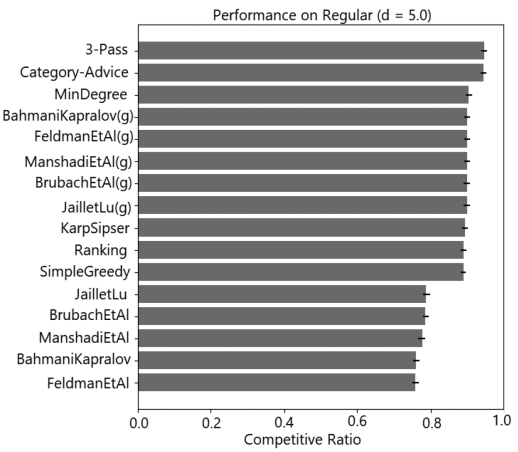

Fig. 21. Performance of all algorithms on Left-Regular graph with $d = 5$.

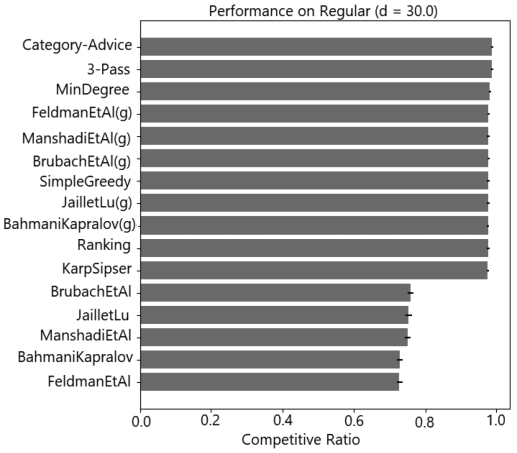

Fig. 22. Performance of all algorithms on Left-Regular graph with $d = 30$.

4.4 Preferential Attachment Bigraph Experiments

The experiments in this subsection are based on type graphs with 1,000 nodes on each side, where types are generated via the preferential attachment method with a single parameter c. The values of c range from 0.1 to 14.9 with a step of 0.1. All results are averaged over 100 i.i.d. trials. We present time series of all non-greedy algorithms in Figure 31, and we present BahmaniKapralov versus greedy algorithms in Figure 32. Figures comparing other non-greedy algorithms with greedy algorithms are omitted, since they can be inferred from the given two figures, as discussed at the beginning of Subsection 4.1. We identify three regimes of c that correspond to (1) sparse case ($c = 2.1$), (2) intermediate case ($c = 8.1$), and (3) asymptotic case ($c = 14.9$); this is similar to Subsection 4.1. The competitive ratios of different algorithms are plotted in Figures 33, 34, and 35.

4.5 Stand-Alone Graphs

In this subsection, we collect competitive ratios of all algorithms considered in this article on all stand-alone graphs, as discussed in Section 3. Since some of the algorithms and graph

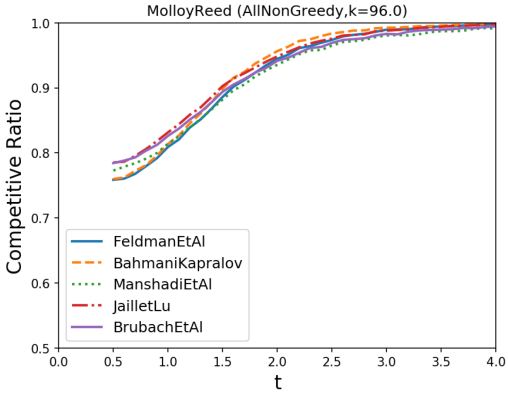

Fig. 23. Performance (as a function of τ) of all non-greedy algorithms on the Molloy-Reed family of graphs with $\kappa = 96$.

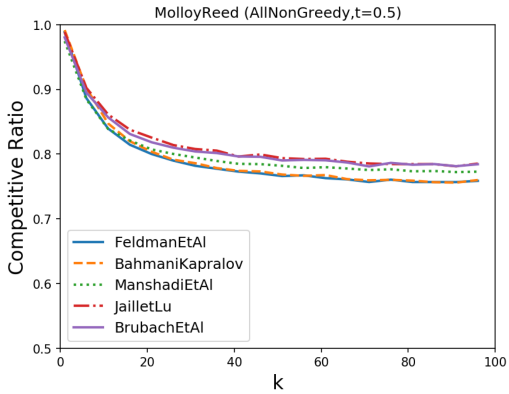

Fig. 24. Performance (as a function of κ) of all non-greedy algorithms on the Molloy-Reed family of graphs with $\tau = 0.5$.

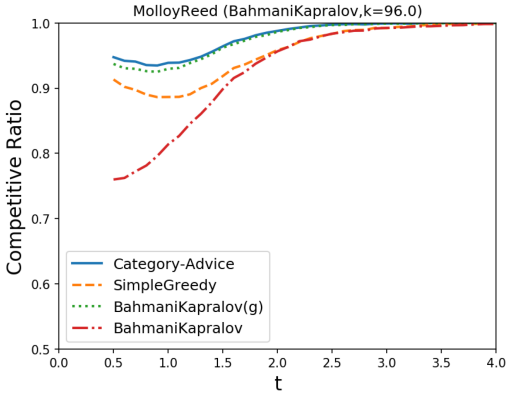

Fig. 25. Performance (as a function of τ) of the BahmaniKapralov algorithm on the Molloy-Reed family of graphs with $\kappa = 96$.

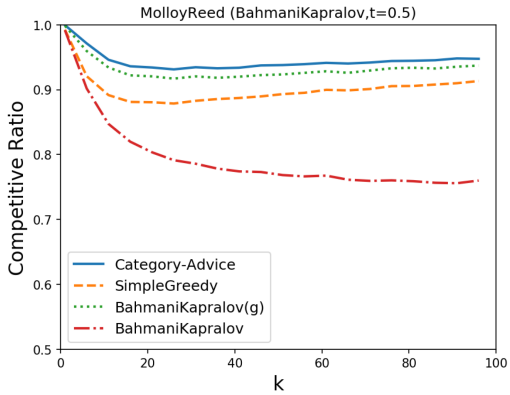

Fig. 26. Performance (as a function of κ) of the BahmaniKapralov algorithm on the Molloy-Reed family of graphs with $\tau = 0.5$.

constructions are randomized, we show results that are averaged over 100 trials. See Table 3 for the summary.

4.6 Real-World Instances

In this subsection, we collect competitive ratios of our algorithms on graphs based on real-world applications, as discussed in Section 3. That is, since these application graphs are not bipartite, we consider two methods of converting them into a bipartite graph: the random bipartition method, and the duplicating method. Since some of the algorithms and the random bipartition conversion method are randomized, we show results that are averaged over 100 trials. See Table 4 for the summary of results for the random bipartition method, and Table 5 for the duplicating method.

4.7 Running Times

In this section, we describe our experimental findings about running times of the algorithms under consideration. We have not dedicated a lot of time to optimizing runtimes of individual algorithms or to finding state-of-the-art libraries for LP solving, for example. Our aim is not to provide an

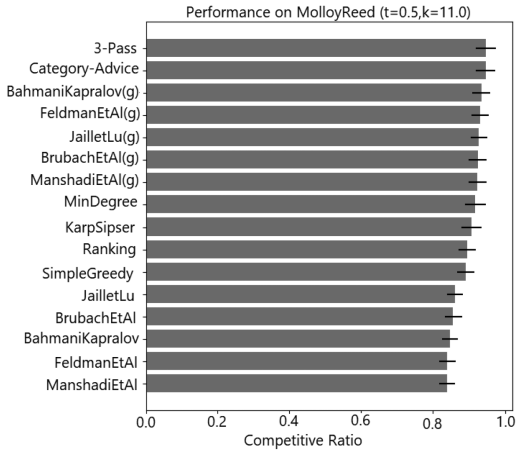

Fig. 27. Performance of all algorithms on the Molloy-Reed graph with $\tau = 0.5, \kappa = 11$.

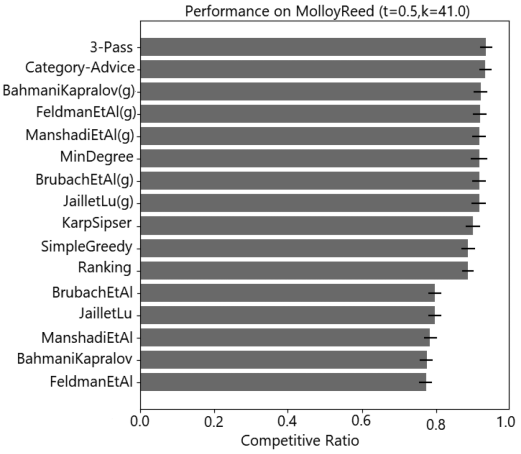

Fig. 28. Performance of all algorithms on the Molloy-Reed graph with $\tau = 0.5, \kappa = 41$.

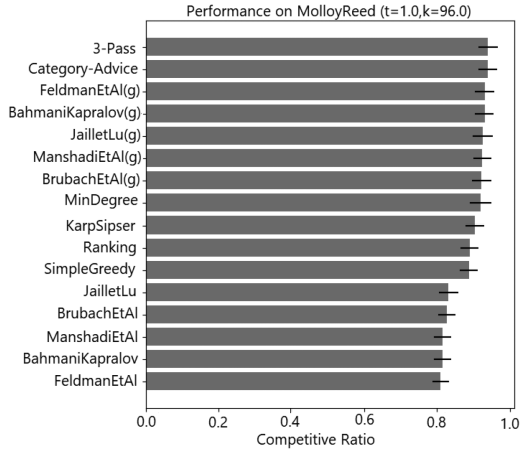

Fig. 29. Performance of all algorithms on the Molloy-Reed graph with $\tau = 1.0, \kappa = 96$.

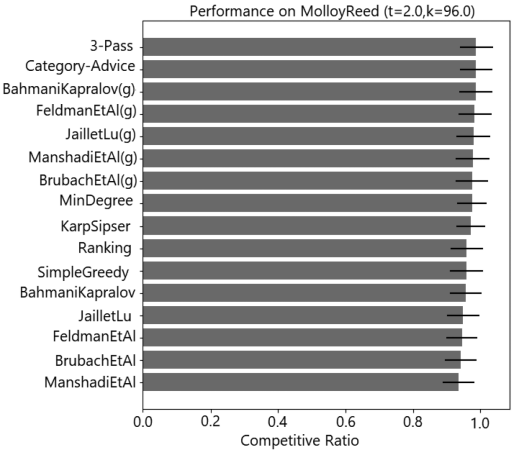

Fig. 30. Performance of all algorithms on the Molloy-Reed graph with $\tau = 2.0, \kappa = 96$.

extensive scalability study, but rather to see if complicated algorithms with an off-the-shelf implementation incur prohibitive runtime costs. The main difference between complicated and simple algorithms is that the complicated algorithms have a non-trivial preprocessing component. Theoretically, we know that pre-processing adds a significant overhead to the running time and in certain cases it asymptotically dominates the online matching phase of the algorithm (see the discussion of BrubachEtAl at the end of this section). The goal of this section is to see if this is supported by our experiments, and, indeed, it is. We observe how running times scale with the number of edges. For that purpose, we present running times from random left-regular graph experiments. The edge density for the other dataset families in Section 3.1 is also controlled by a single parameter and the runtimes for experiments for these datasets behave exactly the same way. Thus, the conclusions we derive for the random left-regular graph experiments are quite general and are expected to carry over to other scenarios. In the random left-regular graph experiment, the number of online nodes is fixed to be 1,000, the same as the number of offline nodes.

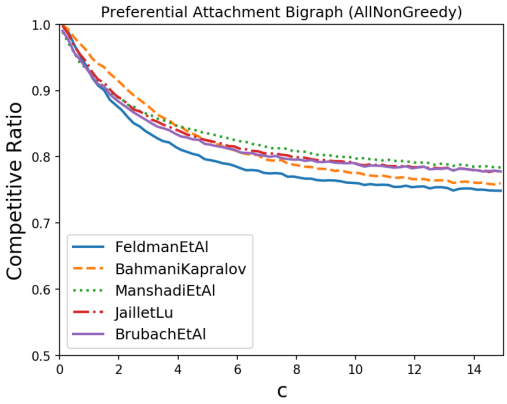

Fig. 31. Performance of all non-greedy algorithms on the Preferential Attachment Bigraph family of graphs.

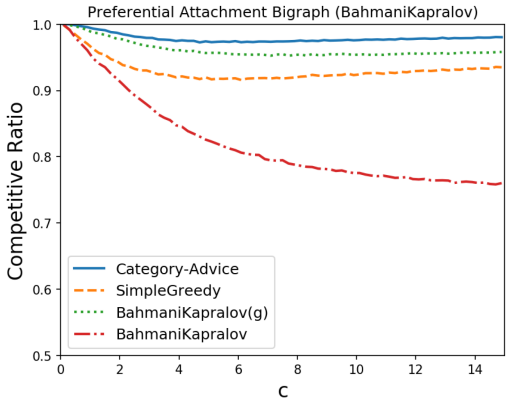

Fig. 32. Performance of the BahmaniKapralov algorithm on the Preferential Attachment Bigraph family of graphs.

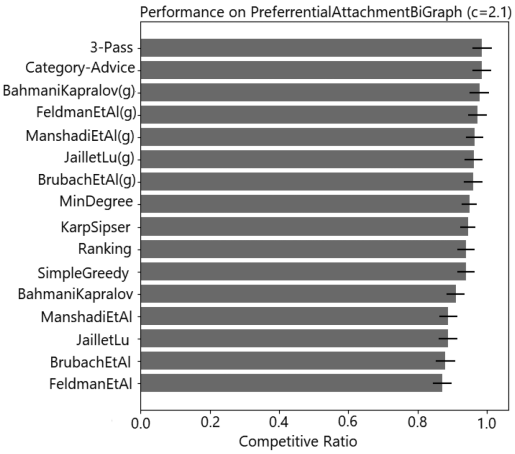

Fig. 33. Performance of all algorithms on the Preferential Attachment Bigraph graph with $c = 2.1$.

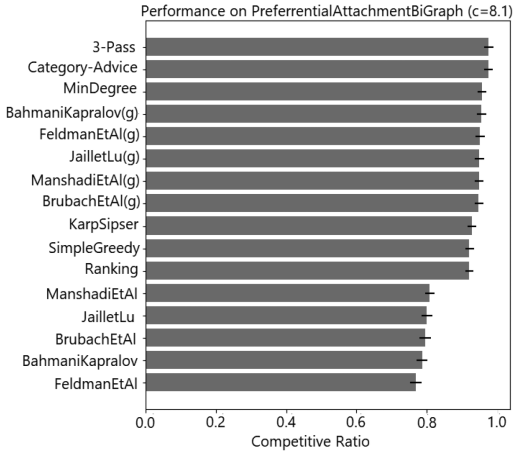

Fig. 34. Performance of all algorithms on the Preferential Attachment Bigraph graph with $c = 8.1$.

The neighborhood of each online node is decided by selecting a subset of neighbors of size d uniformly at random. Thus, as d increases, the total number of edges increases. Figure 36 shows how the running times of the various algorithms scale with d.

For ease of presentation, we omit greedy versions of complicated algorithms (i.e., FeldmanEtAl, BahmaniKapralov, ManshadiEtAl, JailletLu, BrubachEtAl). In our experiments, greedy versions had the same runtimes as their non-greedy versions, because turning an algorithm into a greedy one has virtually no overhead. Runtimes of complicated algorithms are dominated by their preprocessing stages. Therefore, the more complicated is the preprocessing stage, the slower is the algorithm. Greedy-like algorithms have either no or minimal preprocessing, e.g., SimpleGreedy and Category-Advice, and thus are the fastest algorithms, as expected. BahmaniKapralov is essentially FeldmanEtAl with some additional preprocessing steps, thus the two algorithms behave similarly with BahmaniKapralov being slightly slower. JailletLu is slower still, but not by much. The behavior of ManshadiEtAl might seem mysterious at first as the running time

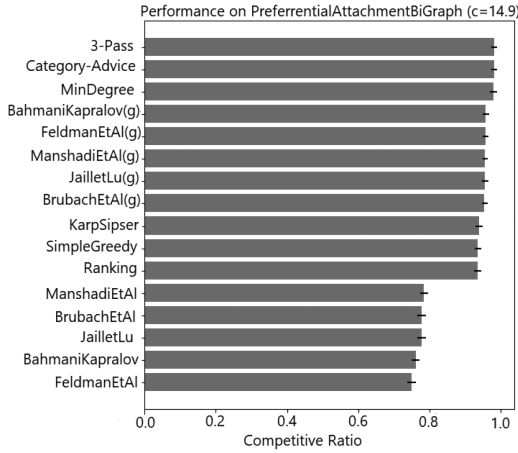

Fig. 35. Performance of all algorithms on the Preferential Attachment Bigraph graph with $c = 14.9$.

Table 3. Performance of Algorithms on Stand-alone Graphs

Algorithm	UT	MH	FH	FewG	ManyG	Rope	Hexa	Zipf
FeldmanEtAl	0.76	0.76	0.67	0.77	0.80	0.92	0.75	0.86
FeldmanEtAl(g)	0.90	0.87	0.88	0.89	0.92	0.99	0.89	0.96
BahmaniKapralov	0.76	0.76	0.80	0.77	0.80	0.93	0.76	0.93
BahmaniKapralov(g)	0.90	0.87	0.93	0.89	0.92	0.99	0.89	0.98
ManshadiEtAl	0.77	0.79	0.84	0.78	0.80	0.91	0.77	0.89
ManshadiEtAl(g)	0.89	0.87	0.96	0.89	0.92	0.99	0.89	0.96
JailletLu	0.78	0.80	0.78	0.79	0.82	0.93	0.78	0.87
JailletLu(g)	0.90	0.87	0.87	0.89	0.92	0.99	0.89	0.94
BrubachEtAl	0.78	0.81	0.78	0.79	0.82	0.94	0.78	0.87
BrubachEtAl(g)	0.91	0.87	0.92	0.89	0.92	0.99	0.89	0.95
MinDegree	0.98	0.87	0.91	0.89	0.92	0.99	0.89	0.92
KarpSipser	0.82	0.87	0.92	0.88	0.92	0.99	0.87	0.91
SimpleGreedy	0.66	0.87	0.91	0.86	0.90	0.99	0.86	0.87
Ranking	0.92	0.87	0.95	0.87	0.91	0.99	0.87	0.93
Category-Advice	0.76	0.95	0.99	0.92	0.95	1.00	0.92	0.97
3-Pass	0.77	0.95	0.99	0.92	0.95	1.00	0.92	0.97

increases sharply until $d = 4$ and then suddenly drops almost matching the greedy runtime. To explain this behavior, we recall how the preprocessing step of ManshadiEtAl works. It samples 100 graphs from the distribution specified by the type graph and solves each of the samples optimally. Thus, the runtime depends not only on the density of the edges, but also on how easy it is to solve a sample optimally. The runtime plot suggests that the case $d = 4$ is the hardest to solve optimally (among integral d). Clearly, as d increases, it becomes easier and easier to find a perfect matching in the samples. This leads to a faster runtime when $d > 4$. Of course, we applied a simplification where we fixed the number of samples used by ManshadiEtAl in the preprocessing stage to be 100. In practice, one would have to adjust the number of samples with the density of the graph with denser type graphs requiring far more samples. Thus, one would expect the runtime of

Table 4. Performance of Algorithms on Real-Life Instances Transformed into Bipartite Instances via the Random-Bipartition Method

Algorithm	Caltech36	Reed98	CE-GN	CE-PG	beause	mbeaflw
FeldmanEtAl	0.78	0.78	0.78	0.81	0.76	0.74
FeldmanEtAl(g)	0.91	0.91	0.94	0.96	0.94	0.95
BahmaniKapralov	0.80	0.81	0.84	0.89	0.80	0.76
BahmaniKapralov(g)	0.92	0.92	0.96	0.98	0.96	0.96
ManshadiEtAl	0.81	0.81	0.84	0.87	0.81	0.79
ManshadiEtAl(g)	0.91	0.91	0.96	0.97	0.96	0.96
JailletLu	0.81	0.81	0.80	0.82	0.78	0.77
JailletLu(g)	0.91	0.91	0.94	0.96	0.95	0.96
BrubachEtAl	0.81	0.81	0.81	0.83	0.79	0.77
BrubachEtAl(g)	0.91	0.91	0.94	0.95	0.94	0.95
MinDegree	0.88	0.88	0.94	0.95	0.95	0.97
KarpSipser	0.84	0.84	0.91	0.94	0.90	0.92
SimpleGreedy	0.87	0.87	0.93	0.94	0.94	0.95
Ranking	0.86	0.87	0.93	0.94	0.94	0.95
Category-Advice	0.92	0.93	0.97	0.98	0.97	0.97
3-Pass	0.92	0.93	0.97	0.98	0.97	0.97

Table 5. Performance of Algorithms on Real-Life Instances Transformed into Bipartite Instances via the Duplicating Method

Algorithm	Caltech36	Reed98	CE-GN	CE-PG	beause	mbeaflw
FeldmanEtAl	0.77	0.77	0.77	0.80	0.74	0.73
FeldmanEtAl(g)	0.90	0.90	0.95	0.95	0.94	0.97
BahmaniKapralov	0.78	0.78	0.82	0.88	0.76	0.75
BahmaniKapralov(g)	0.91	0.91	0.97	0.98	0.95	0.97
ManshadiEtAl	0.79	0.78	0.84	0.86	0.78	0.77
ManshadiEtAl(g)	0.90	0.90	0.96	0.97	0.95	0.96
JailletLu	0.79	0.79	0.79	0.81	0.77	0.76
JailletLu(g)	0.90	0.90	0.95	0.96	0.95	0.97
BrubachEtAl	0.80	0.79	0.80	0.82	0.77	0.76
BrubachEtAl(g)	0.91	0.91	0.94	0.95	0.95	0.97
MinDegree	0.88	0.87	0.94	0.95	0.95	0.98
KarpSipser	0.82	0.82	0.91	0.93	0.91	0.94
SimpleGreedy	0.72	0.72	0.95	0.95	0.91	0.94
Ranking	0.86	0.86	0.93	0.94	0.94	0.97
Category-Advice	0.82	0.83	0.98	0.99	0.96	0.97
3-Pass	0.83	0.84	0.98	0.99	0.96	0.97

ManshadiEtAl to scale much worse than what is suggested by our figure. So far, our plot suggests that the runtimes of simple greedy-like algorithms scale linearly with the number of edges, while the runtimes of other more complicated algorithms scale like small polynomials. When it comes to the last complicated algorithm, BrubachEtAl, the runtime scales exponentially with d. The runtime of BrubachEtAl is dominated by the part of the preprocessing stage that corresponds to

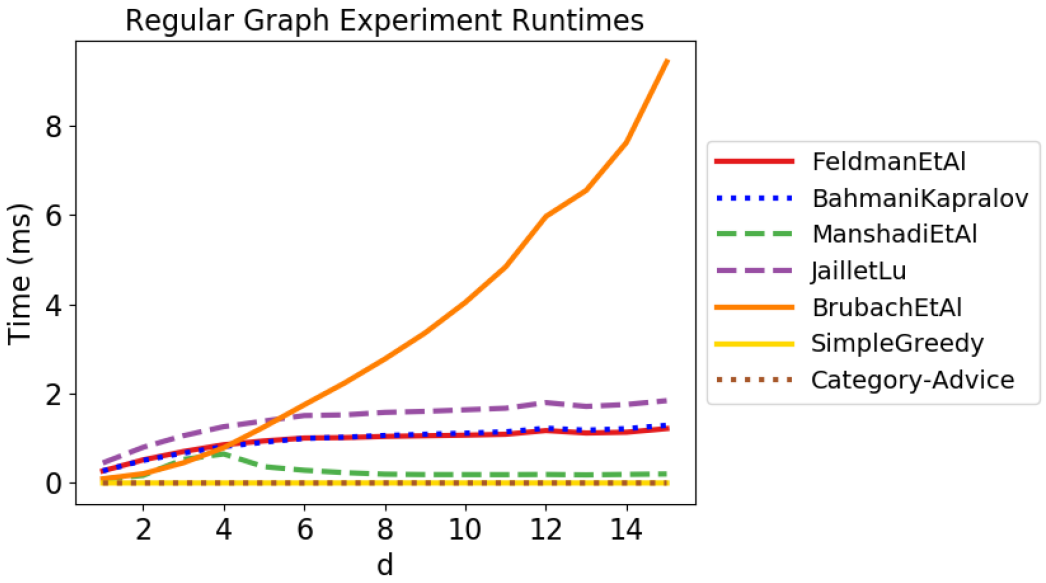

Fig. 36. Running times left-side regular family of graphs.

solving the LP. The number of constraints in the LP of BrubachEtAl is asymptotically larger than the number of edges (assuming $|V| = o(|E|)$). More specifically, let $e(r)$ denote the number of edges incident on a node $r \in R$ in the given type graph. The number of constraints in the LP of BrubachEtAl is at least $\sum_{r \in R} e(r)^2 \geq |E|^2/n$ (by Cauchy-Schwarz). Using the simplex method from the GNU Linear Programming Toolkit even with moderately dense graphs (e.g., $d = 100$) already results in excessive runtimes and in memory consumption of over 6 GB. One could potentially try to optimize this step, possibly applying interior-point methods to very large instances and designing new heuristics to speed up this computation. We suspect that such efforts would not be worth it; it does not seem feasible to run BrubachEtAl on very large instances (e.g., number of edges on the order of tens of millions), and furthermore, simpler methods either match or surpass its performance in terms of the competitive ratio on many instances that we consider in this study.

5 DISCUSSION

As seen in Figure 9, in the **Erdős-Rényi** family of graphs, all algorithms exhibit somewhat similar performance. All algorithms perform much better than their theoretical worst-case guarantees. This is expected because of the randomness in the input. Figures 10–14 showcase the experimental competitive ratios of each algorithm along with its greedy version, SimpleGreedy, and Category-Advice (2-Pass). All algorithms seem to follow similar trends. More specifically, the non-greedy versions show a drop in performance as c increases. This is to be expected since non-greedy algorithms ignore a certain fraction of the input while the offline optimum increases when the graph is getting denser. Greedy algorithms always perform close to optimum when the graph is very sparse or very dense and this behavior is evident as the greedy versions achieve a global minimum around $c = 4.9$. For a theoretical explanation of this behavior, see [22] and [5]. Figures 15–17 show an experimental ranking of the algorithms before, around, and after the global minimum, respectively. What stands out the most is that even the simplest greedy algorithms, SimpleGreedy, Ranking, MinDegree, and KarpSipser, always outperform the more sophisticated non-greedy algorithms that make use of the type graph, while the greedy versions of the latter perform only slightly

better than simple greedy ones. Interestingly, Category-Advice is always the best performing algorithm, while MinDegree is the best performing purely online algorithm on very dense graphs ($c = 14.9$). For $c = 14.9$, the experimental ranking of the non-greedy algorithms is consistent with Table 1. For $c = 1.9$ and $c = 4.9$, BahmaniKapralov is the best and worst non-greedy algorithm, respectively, while in both cases JailletLu is doing slightly better than BrubachEtAl. It's worth noting that the proven competitive ratios of JailletLu and BrubachEtAl only differ by 0.0006 and the number of simulations required to achieve a low enough error margin is beyond our computational resources. We also note that 3-Pass is always guaranteed to perform at least as well as Category-Advice, but sometimes our plots list Category-Advice above 3-Pass. In these instances, the two algorithms gave identical performance, and the plot generating procedure broke ties in favor of Category-Advice. In all our experiments, 3-Pass was either identical to Category-Advice or gave minuscule improvements. This is explained by an already impressive performance of Category-Advice. When the instance graph has a near-perfect matching, one can expect the first and second pass to cover almost all offline nodes. The third pass differs from the first two only in its behavior on the nodes that were not matched in either of the first two passes. If the first two passes already cover all or almost all nodes, then the third pass doesn't actually do anything, and this is what we observed.

The **Random Regular** family of graphs paints a similar picture. The results are almost identical for *Left Regular* and *Right Regular*, so we only present the former. The performance of all non-greedy algorithms is depicted in Figure 18. The main difference compared to the Erdős-Rényi family is that the algorithms converge a bit faster as the graph gets denser. The results of the non-greedy Feldman algorithm compared to its greedy version and other greedy algorithms are shown in Figure 19. Other non-greedy algorithms exhibit similar behavior. We see that the greedy algorithms achieve a global minimum around $d = 5$ and that there seems to be a slightly bigger gap between the non-greedy algorithms and their greedy versions than in the Erdős-Rényi experiment. As seen in Figures 20–22, for $d = 5$ and $d = 30$, the ranking of the non-greedy algorithms agrees with their respective theoretical guarantees, while for $d = 2$ BahmaniKapralov performs the best, and BrubachEtAl is second worst after FeldmanEtAl. Simple greedy algorithms are consistently better than all non-greedy algorithms, while greedy algorithms that use information from the type graph perform marginally better than simple greedy ones on sparser graphs. Interestingly, MinDegree quickly becomes the best performing online algorithm, even for $d = 5$.

For the **Molloy-Reed** family of graphs, Figures 23 and 24 show the performance of non-greedy algorithms as functions of τ and κ, respectively. The behavior of all algorithms is again very similar. We can see that as κ increases, the graph is getting denser, which results in an expected drop in performance as more nodes that could be included in an optimal matching are being rejected. On the contrary, as τ increases, the graph is getting less dense and the performance increases. In fact, as shown in the indicative plot of BahmaniKapralov (Figure 25), as τ increases and the graph becomes sparse, non-greedy algorithms achieve performance close to that of greedy ones. The experimental performance-based ranking of algorithms in various settings is shown in Figures 27–30. Simple greedy algorithms consistently outperform non-greedy algorithms but the greedy versions of algorithms that use the type graph achieve slightly better results (although MinDegree manages to outperform BrubachEtAl(g) and JailletLu(g) in the setting of $\tau = 0.5, \kappa = 41$). The ranking of non-greedy algorithms varies a lot among different parameter settings. As opposed to other graphs, FeldmanEtAl does not always come last, and it even outperforms BrubachEtAl in the setting of $(\tau = 2, \kappa = 96.0)$.

Results for the **Preferential Attachment Bigraph** family (Figures 31–35) agree with the patterns observed thus far. The greedy version of only one of the non-greedy algorithms is presented in a plot (Figure 32), and the rest are similar. Simple greedy algorithms do better than all non-greedy

algorithms but, with the exception of MinDegree, are outperformed by the greedy versions of algorithms using the type graph. BahmaniKapralov(g) is the best performing online algorithm for the sparse case but afterwards the heuristic algorithm MinDegree climbs to the top. For $c = 8.1$ and $c = 14.9$, ManshadiEtAl is the best non-greedy algorithm, whereas for $c = 2.1$ it comes second to BahmaniKapralov.

Similar trends appear in the results for stand-alone graphs (Table 3), with algorithms performing much better than their worst-case guarantees. One exception is graph **UT**, where SimpleGreedy is the worst performing algorithm. This is to be expected as that is the worst-case graph for that algorithm. Besides SimpleGreedy, Category-Advice and 3-Pass algorithms also achieve low performance on graph UT. On the other hand, MinDegree is the best algorithm for UT, followed by Ranking, even though in the adversarial setting this is its worst-case graph for the latter. Moreover, it seems that 0.78 is an upper bound of all other non-greedy algorithms on UT, indicating that this graph might be a useful theoretical benchmark. Graph **MH** is a hard instance that produces the best known upper bound for the known i.i.d. *input model* with integral types. All greedy online algorithms achieve the same, best online performance, while Category-Advice and 3-Pass result in a substantial improvement. The exceptionally low performance of FeldmanEtAl on graph **FH** verifies it as its worst-case graph. What is also interesting about FH is that the experimental performance-based ranking of the algorithms is not consistent with the ranking of Table 1. Specifically, the best non-greedy algorithm is ManshadiEtAl with 0.84, followed by BahmaniKapralov and the remaining algorithms in their usual ordering. **Rope** seems to be the easiest class that all algorithms can handle quite well. The worst performance on graph Rope is 0.91 achieved by ManshadiEtAl. The second easiest graph is **Zipf** where the worst performing algorithm is FeldmanEtAl with a competitive ratio of 0.86. On graphs **FewG** and **ManyG**, the ranking of the non-greedy algorithms based on their performance follows their ranking based on their worst-case analysis. Overall, **ManyG** appears to be an easy graph, which might be due to a quite uniform distribution of edges over the offline nodes. **Hexa** appears to be an instance of similar hardness to **MH**. The best performing algorithms get 0.89, only slightly better than the best performance on **MH** (0.87), while the worst algorithm (FeldmanEtAl with 0.75) is slightly worse than the worst performance on **MH** (0.76). It is also worth noting that the multiple-pass offline algorithms do not result in a performance increase as big as on graph **MH**.

Overall, the ranking of the non-greedy algorithms as presented in Table 1 remains fairly consistent in the stand-alone graphs, except for **FH** and **Zipf** where ManshadiEtAl and BahmaniKapralov take the lead. Additionally, simple greedy algorithms always outperform the non-greedy algorithms that make use of the type graph, with the former always being slightly better, and with the only exception being graph **UT**, where SimpleGreedy experiences a drop in performance. For most graphs, MinDegree is in the set of algorithms that achieve the best performance. In just two graphs, namely, **UT** and **Zipf**, there exist online algorithms that beat the offline multiple-pass algorithms.

As shown in Tables 4 and 5, the performance on real instances is also much better than the worst-case guarantees, which justifies looking at random graphs for an indication of real-world performance. The results using both the random-bipartition method and the duplicating method are very similar in terms of the experimental performance-based ranking of the algorithms, with the graphs produced using the random-bipartition method being seemingly easier. The numbers achieved by Ranking and SimpleGreedy are pretty much identical when random-bipartition is used, while Ranking is significantly better when the duplicating method is used. MinDegree is the best out of the simple greedy algorithms, but with the exception of dataset *mbeaflw*, there's always a greedy version of an algorithm that uses the type graph that performs just as well. In datasets *Caltech36* and *Reed98*, the ranking of Table 1 is maintained for non-greedy algorithms. In *CE-GN*

and *CE-PG*, ManshadiEtAl and BahmaniKapralov outperform the rest. In *beause* and *mbeaflw* ManshadiEtAl is the best algorithm, while in *beause* using the random-bipartition method, BahmaniKapralov comes in a close second, outperforming BrubachEtAl and JailletLu.

The experimental performance-based ranking of the non-greedy algorithms is generally consistent with their provable competitive ratios. BrubachEtAl and JailletLu perform very similarly and are usually on top, although on some graphs, ManshadiEtAl and BahmaniKapralov can outperform the rest. FeldmanEtAl almost always comes last (with a few exceptions, e.g., Molloy-Reed), but FeldmanEtAl(g) is often one of the best online algorithms. MinDegree, KarpSipser, Ranking, and SimpleGreedy get excellent results and almost always outperform all non-greedy algorithms. MinDegree seems to be the most powerful out of the simple greedy algorithms studied here, and with its heuristic having such a minimal overhead, it is quite impressive. After turning non-greedy algorithms into greedy algorithms, the ranking of the resulting algorithms is not very predictable. FeldmanEtAl(g) does quite well and its place in the ranking improves. BrubachEtAl(g) and JailletLu(g) can drop in rankings and become some of the worst greedy algorithms, while it is not unusual for BahmaniKapralov(g) to become the best. It is not straightforward to predict how an algorithm will behave after it turns greedy but the transformation appears to be very beneficial as the greedy versions significantly outperform non-greedy algorithms. Category-Advice seems to be a great algorithm to use in streaming models and when dealing with massive datasets. Even though it can improve substantially over the SimpleGreedy solution, an additional pass (as done in 3-Pass) does not seem to provide much benefit, as discussed before.

6 CONCLUSION

In this article, we experimentally studied various online bipartite matching algorithms under the known i.i.d. input model with integral types. Type graphs that were used in our evaluations came from different sources, including random models of social networks, real-life networks, and standalone graphs that appeared previously in matching-related literature. Broadly speaking, algorithms under consideration can be split into two groups: simple algorithms that do not make use of the additional information (i.e., the type graph), and more complicated algorithms that often have a computationally intensive preprocessing step that tries to utilize the type graph for future predictions. The more complicated algorithms were developed and analyzed by researchers in the worst-case known i.i.d. setting so as to demonstrate more realistic performance bounds in contrast to the purely adversarial setting. These algorithms are often presented as being non-greedy to simplify the analysis. In contrast, most simple algorithms are naturally greedy. It is relatively easy to convert complicated algorithms into greedy ones without hurting the worst-case performance guarantee and without any significant computational overhead. Thus, intuitively an algorithm for online bipartite matching can be viewed as consisting of two parts; namely, the complicated preprocessing part, and the greedy part. One of the main questions we try to answer in this work is how much each part is contributing to the competitive ratio on "practical instances," where practical instances are modeled by the type graphs discussed above. It turns out that most of the work is done by the greedy part. In particular, the simple greedy algorithm tends to outperform all non-greedy versions, sometimes quite significantly. It also tends to perform comparably to the greedy versions of more complicated algorithms. In certain scenarios, the more complicated algorithms turned into greedy ones outperform the simple greedy algorithm, although it is questionable whether the performance boost is worth the extra computational effort in practice. In certain cases, this overhead can become computationally intractable in practice (e.g., running BrubachEtAl on type graphs with millions of nodes).

There are many problems suggested by our work and many future directions are worth exploring. We list some of them here:

Open Problem 6.1. *We conjecture that a practical study of online bipartite matching under the known i.i.d. with fractional types would result in very similar results and conclusions to what we observed with integral types. Does there exist a "practical instance" with fractional types that highlights the necessity to use more complicated algorithms?*

Open Problem 6.2. *It is important to perform a similar evaluation on real-life data for online advertising, which is one of the main applications of online bipartite matching. Such data is proprietary and is not available to the public. Creating a public repository of such benchmarks would be a great contribution to the field on its own.*

Open Problem 6.3. *In order to bridge the gap between theory and practice one needs to consider models other than worst-case. Known i.i.d. was the first step in this direction for online bipartite matching, since worst-case over type graphs allows for much better competitive ratios than worst-case over adversarial inputs. However, the area does not have to stop at known i.i.d. It is important to design and analyze new stochastic input models that better match practical inputs for certain application domains, e.g., online advertising.*

Open Problem 6.4. *In an attempt at being fair and testing all algorithms on the same type graphs, we were limited to consider graphs with at most 1,000 nodes due to prohibitive computational requirements of* BrubachEtAl. *One could perform a study on extremely large instances with millions or billions of nodes by excluding algorithms with too much preprocessing. We suspect that the results of such a study would be similar to ours, and they would highlight the importance of using very simple greedy algorithms in large-scale applications. Would any of the complicated algorithms be able to handle such instances? Would the extra computation be worth it?*

Open Problem 6.5. *There are many extensions of online bipartite matching, including edge-weighted matching, vertex-weighted matching, matching with capacity constraints (i.e., the so-called b-matching), and so on. Performing an experimental study similar to the current one for those extensions is an interesting open problem. Such a study might reveal certain ranges of parameters which make extensions of the matching problem behave very differently from the vanilla matching problem with respect to the algorithms considered in this article.*

ACKNOWLEDGMENTS

We thank Michael Kapralov for discussing the BahmaniKapralov algorithm. We are thankful to the anonymous reviewers who provided a lot of constructive comments that led to an improved version of the article. In particular, the MinDegree algorithm in the known i.i.d. input model that we consider in this article was inspired by the offline MinDegree algorithm, which was suggested by one of the reviewers. This ended up being an excellent algorithm and we are thankful to the reviewer for the suggestion.

REFERENCES

[1] [n.d.]. GNU Linear Programming Kit, Version 4.32. Retrieved May 11, 2018 from http://www.gnu.org/software/glpk/glpk.html.

[2] Denis Pankratov, Allan Borodin, and Christodoulos Karavasilis. [n.d.]. Online Bipartite Matching Library. Retrieved August 8, 2018 from https://users.encs.concordia.ca/~denisp/online-bm-lib.tar.gz.

[3] Bahman Bahmani and Michael Kapralov. 2010. Improved bounds for online stochastic matching. In *Proc. of ESA*. 170–181.

[4] Yonatan Bilu and Nathan Linial. 2012. Are stable instances easy? *Combinatorics, Probability & Computing* 21, 5 (2012), 643–660.

[5] A. Borodin, C. Karavasilis, and D. Pankratov. 2018. Greedy bipartite matching in random type Poisson arrival model. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2018*. 5:1–5:15. DOI: https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2018.5
[6] Allan Borodin, Denis Pankratov, and Amirali Salehi-Abari. 2018. On conceptually simple algorithms for variants of online bipartite matching. In *Approximation and Online Algorithms*, Roberto Solis-Oba and Rudolf Fleischer (Eds.). Springer International Publishing, 253–268.
[7] Joan Boyar, Lene M. Favrholdt, Christian Kudahl, Kim S. Larsen, and Jesper W. Mikkelsen. [n.d.]. Online algorithms with advice: A survey. *ACM Comput. Surv.* 50, 2 ([n. d.]), 19:1–19:34.
[8] Brian Brubach, Karthik Abinav Sankararaman, Aravind Srinivasan, and Pan Xu. 2016. New algorithms, better bounds, and a novel model for online stochastic matching. In *Proc. of ESA*. 24:1–24:16.
[9] Boris V. Cherkassky, Andrew V. Goldberg, Joao C. Setubal, and Jorge Stolfi. 1998. Augment or push: A computational study of bipartite matching and unit-capacity flow algorithms. *Journal of Experimental Algorithmics* 3, 8 (1998).
[10] Nikhil R. Devanur, Kamal Jain, and Robert D. Kleinberg. 2013. Randomized primal-dual analysis of RANKING for online bipartite matching. In *Proc. of SODA*. 101–107.
[11] Christoph Dürr, Christian Konrad, and Marc Renault. 2016. On the power of advice and randomization for online bipartite matching. In *Proc. of ESA*. 37:1–37:16.
[12] Jon Feldman, Aranyak Mehta, Vahab S. Mirrokni, and S. Muthukrishnan. 2009. Online stochastic matching: Beating 1-1/e. In *FOCS 2009*. 117–126.
[13] Rajiv Gandhi, Samir Khuller, Srinivasan Parthasarathy, and Aravind Srinivasan. 2006. Dependent rounding and its applications to approximation algorithms. *Journal of the ACM* 53, 3 (May 2006), 324–360. DOI: https://doi.org/10.1145/1147954.1147956
[14] Gagan Goel and Aranyak Mehta. 2008. Online budgeted matching in random input models with applications to adwords. In *Proc. of SODA*. 982–991.
[15] Patrick Jaillet and Xin Lu. 2014. Online stochastic matching: New algorithms with better bounds. *Mathematics of Operations Research* 39, 3 (2014), 624–646.
[16] Richard M. Karp and Michael Sipser. 1981. Maximum matchings in sparse random graphs. In *22nd Annual Symposium on Foundations of Computer Science, 1981*. 364–375.
[17] R. M. Karp, U. V. Vazirani, and V. V. Vazirani. 1990. An optimal algorithm for on-line bipartite matching. In *Proc. of STOC*. 352–358.
[18] Johannes Langguth, Fredrik Manne, and Peter Sanders. 2010. Heuristic initialization for bipartite matching problems. *ACM Journal of Experimental Algorithmics* 15 (2010).
[19] Jacob Magun. 1998. Greedy matching algorithms, an experimental study. *ACM Journal of Experimental Algorithms* 3 (1998).
[20] Mohammad Mahdian and Qiqi Yan. 2011. Online bipartite matching with random arrivals: An approach based on strongly factor-revealing LPs. In *Proc. of STOC*. 597–606.
[21] Vahideh H. Manshadi, Shayan Oveis Gharan, and Amin Saberi. 2011. Online stochastic matching: Online actions based on offline statistics. In *Proc. of SODA*. 1285–1294.
[22] A. Mastin and P. Jaillet. 2013. Greedy online bipartite matching on random graphs. *ArXiv e-prints* (July 2013). arxiv:cs.DS/1307.2536
[23] A. Mehta. 2012. Online matching and ad allocation. *Theoretical Computer Science* 8, 4 (2012), 265–368,
[24] Molloy Michael and Reed Bruce. 1995. A critical point for random graphs with a given degree sequence. *Random Structures & Algorithms* 6, 2–3 (1995), 161–180. DOI: https://doi.org/10.1002/rsa.3240060204 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/rsa.3240060204.
[25] Mark E. J. Newman, Duncan J. Watts, and Steven H. Strogatz. 2002. Random graph models of social networks. *Proceedings of the National Academy of Sciences USA* 99, Suppl 1 (2002), 2566–2572.
[26] Nicolas Pena and Allan Borodin. 2019. On extensions of the deterministic online model for bipartite matching and max-sat. *Theoretical Computer Science* 770 (2019), 1–24.
[27] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence*. http://networkrepository.com.
[28] Alvin E. Roth, Tayfun Sönmez, and M. Utku Ünver. 2007. Efficient kidney exchange: Coincidence of wants in markets with compatibility-based preferences. *American Economic Review* 97, 3 (June 2007), 828–851.
[29] Daniel A. Spielman and Shang-Hua Teng. 2009. Smoothed analysis: An attempt to explain the behavior of algorithms in practice. *Communications of the ACM* 52, 10 (2009), 76–84.

Received November 2018; revised September 2019; accepted October 2019