



**HAL**  
open science

# Engineering fast almost optimal algorithms for bipartite graph matching

Ioannis Panagiotas, Bora Uçar

► **To cite this version:**

Ioannis Panagiotas, Bora Uçar. Engineering fast almost optimal algorithms for bipartite graph matching. ESA 2020 - European Symposium on Algorithms, Sep 2020, Pisa, Italy. hal-02463717v3

**HAL Id: hal-02463717**

**<https://inria.hal.science/hal-02463717v3>**

Submitted on 1 Jul 2020

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# 1 Engineering fast almost optimal algorithms for 2 bipartite graph matching

3 Ioannis Panagiotas 

4 ENS Lyon, France

5 ioannis.panagiotas@ens-lyon.fr

6 Bora Uçar 

7 CNRS and LIP, ENS Lyon, France

8 bora.ucar@ens-lyon.fr

---

## 9 Abstract

10 We consider the maximum cardinality matching problem in bipartite graphs. There are a number  
11 of exact, deterministic algorithms for this purpose, whose complexities are high in practice. There  
12 are randomized approaches for special classes of bipartite graphs. Random 2-out bipartite graphs,  
13 where each vertex chooses two neighbors at random from the other side, form one class for which  
14 there is an  $O(m + n \log n)$ -time Monte Carlo algorithm. Regular bipartite graphs, where all vertices  
15 have the same degree, form another class for which there is an expected  $O(m + n \log n)$ -time Las  
16 Vegas algorithm. We investigate these two algorithms and turn them into practical heuristics with  
17 randomization. Experimental results show that the heuristics are fast and obtain near optimal  
18 matchings. They are also more robust than the state of the art heuristics used in the cardinality  
19 matching algorithms, and are generally more useful as initialization routines.

20 **2012 ACM Subject Classification** Theory of computation → Design and analysis of algorithms

21 **Keywords and phrases** bipartite graphs, matching, randomized algorithm

22 **Supplement Material** <https://gitlab.inria.fr/bora-ucar/fast-matching>

## 23 **1** Introduction

24 A matching in a graph is a set of edges, such that no two of them share a common vertex.  
25 We consider the *maximum cardinality problem* in bipartite graphs which asks for a matching  
26 with maximum cardinality. There are a number of exact algorithms for this problem. The  
27 best known algorithms [21] run in  $O(m\sqrt{n})$  time for a graph with  $n$  vertices and  $m$  edges.  
28 Such complexity can be prohibiting for large instances. For this reason, there is significant  
29 interest in algorithms which can find large matchings in linear or near linear time [37]. The  
30 practical use of approximate matchings in applications [33] and as an initialization to exact  
31 algorithms [30] are well known.

32 We investigate two randomized algorithms by Karp et al. [22] and Goel et al. [18], both  
33 of which run in  $O(m + n \log n)$  time. The former algorithm finds, almost surely, maximum  
34 cardinality matchings on random graphs formed by allowing each vertex to select two  
35 vertices from the other side uniformly at random. The latter algorithm finds maximum  
36 cardinality matchings in regular bipartite graphs, where all vertices have equal degree. In  
37 both of these classes of graphs, the bipartite graphs have equal number of vertices in each  
38 part, and the maximum cardinality matchings cover all vertices (such matchings are called  
39 perfect). We investigate these two theoretical algorithms for very special cases of bipartite  
40 graphs and convert them to efficient heuristics for general bipartite graphs. We discuss  
41 our implementations and investigate the performance of the resulting heuristics in terms of  
42 run time and the matching cardinality. Both heuristics run in near linear time and obtain  
43 matchings whose cardinality is more than 0.99 of the maximum, even in cases where the  
44 current state of the art approaches have difficulties.

45 The rest of the paper is organized as follows. In Section 2, we give the necessary  
 46 background. In Sections 3.1 and 3.2 we review the existing randomized algorithms and then  
 47 discuss how we adapt them. Section 4 contains the experimental results, and Section 5  
 48 concludes the paper. Appendices A–D provide some additional results and discussion.

## 49 **2 Background and notation**

50 Let  $G = (R \cup C, E)$  be a bipartite graph, where  $R$  and  $C$  are two disjoint set of vertices, and  
 51  $E$  is the set of edges. The bipartite graph  $G$  can be represented with a matrix  $\mathbf{A}_G$ . The  
 52 vertex  $r_i \in R$  corresponds to the  $i$ th row, and the vertex  $c_j \in C$  corresponds to the  $j$ th  
 53 column, so that  $\mathbf{A}_G(i, j) = 1$  if and only if  $(r_i, c_j) \in E$ . We will refer to vertices of  $R$  as *rows*  
 54 and to those of  $C$  as *columns* from this point on, and use  $\mathbf{A}$  to refer to  $\mathbf{A}_G$ .

55 Let  $\mathcal{M}$  be a matching. For  $(u, v) \in \mathcal{M}$ , the vertices  $u$  and  $v$  are matched, and they are  
 56 each other's mate. A vertex is called free if it is not matched by  $\mathcal{M}$ . If there are no free  
 57 vertices in  $R$  or in  $C$ , then  $\mathcal{M}$  is called perfect. An augmenting path with respect to  $\mathcal{M}$  is a  
 58 path which starts with a free vertex and ends at another free vertex, where every second  
 59 edge is in  $\mathcal{M}$ . A matching is maximum if and only if there are no augmenting paths [7].

60 A square matrix is called doubly stochastic if the sum of entries in each row and column is  
 61 equal to one. An  $n \times n$  matrix  $\mathbf{A}$  has *support* if there is a perfect matching in the associated  
 62 bipartite graph  $G$ .  $\mathbf{A}$  is said to have *total support* if each edge in  $G$  is used in a perfect  
 63 matching. A square matrix is fully indecomposable, if it has total support and cannot be  
 64 permuted into a block diagonal matrix. Any nonnegative matrix  $\mathbf{A}$  with total support can be  
 65 scaled with two positive diagonal matrices  $\mathbf{D}_R$  and  $\mathbf{D}_C$  such that  $\mathbf{A}_S = \mathbf{D}_R \mathbf{A} \mathbf{D}_C$  is doubly  
 66 stochastic, and if  $\mathbf{A}$  is fully indecomposable, then the matrices  $\mathbf{D}_R$  and  $\mathbf{D}_C$  are unique. The  
 67 Sinkhorn–Knopp algorithm [38] is a well-known method for finding such  $\mathbf{D}_R$  and  $\mathbf{D}_C$  for a  
 68 given matrix. This is an iterative algorithm, where at each iteration each row is normalized to  
 69 have unit length, and then each column is normalized to have unit length. If a given matrix  
 70  $\mathbf{A}$  has total support, then Sinkhorn–Knopp algorithm finds the unique scaling matrices. If  $\mathbf{A}$   
 71 has support but not total support, then entries that cannot be put into a perfect matching  
 72 tend to zero. The method converges with an asymptotical convergence rate depending on  
 73 the second singular value of the final doubly stochastic matrix. There are other iterative,  
 74 faster converging methods [1, 10, 28], whose iterations are more sophisticated than that of  
 75 Sinkhorn–Knopp's.

76 A  $k$ -out subgraph  $G_k$  of a host graph  $G$  is defined by allowing each vertex in  $G$  to  
 77 randomly select uniformly  $k$  of its neighbors, and the union of all selections forms the edge  
 78 set of  $G_k$ . Walkup [40] shows that in the pure random  $k$ -out setting, where the host graph is  
 79 the complete bipartite graph, the resulting  $G_k$  has a perfect matching with high probability  
 80 for  $k \geq 2$ . We do not know any general result about properties of  $G_2$  sampled from any  
 81 arbitrary host graph. Frieze and Johansson [17] investigate some other properties of  $G_k$ s on  
 82 host graphs where the minimum degree of a vertex is at least  $n/2$ . Dufossé et al. [16] propose  
 83 using the doubly stochastic matrix  $\mathbf{A}_S$  (scaled version of the matrix representation) for  
 84 sampling and show an approximation result for  $G_1$ , when  $\mathbf{A}$  has total support. We give some  
 85 experiments in which  $G_2$ s generated using the same probabilities have perfect matchings in  
 86 majority of the cases.

87 Two popular classes of randomized algorithms are *Las Vegas* and *Monte Carlo* algorithms.  
 88 *Las Vegas* algorithms always return a correct answer, but their run time can depend on  
 89 random choices, whereas Monte Carlo algorithms can fail with small probability, but their  
 90 complexity is independent of the random choices made (see for example [34, p. 70]).

91 There are a number of heuristics for the cardinality matching problem [30, 37] (see  
 92 Appendix A for a relevant discussion). Among those, that by Karp and Sipser [23] is very  
 93 well known and widely used. This heuristic eliminates vertices of degree at most two in the  
 94 following way. It matches any degree-1 vertices with their neighbors (and discards both), or  
 95 merges the neighbors of a degree-2 vertex (which is then discarded) to a single node, and  
 96 removes any parallel edges that occur. If neither operation can be done, it matches a pair of  
 97 vertices randomly.

### 98 **3 Two heuristics**

99 We describe the original Monte Carlo algorithm [22] for finding perfect matchings in 2-out  
 100 bipartite graphs in Section 3.1 and the original Las Vegas algorithm [18] for finding perfect  
 101 matchings in  $d$ -regular bipartite graphs in Section 3.2. These two algorithms are based on  
 102 uniform sampling. We generalize these two algorithms to general bipartite graphs within a  
 103 common framework. The framework we propose scales the adjacency matrix of the input  
 104 bipartite graph and uses the nonzero values of the scaled matrix for sampling. We also  
 105 identify and fix an oversight in the description of the Monte Carlo algorithm, and describe  
 106 efficient implementations of the two heuristics.

## 107 **3.1 2OUTMC: Monte Carlo on 2-out graphs**

### 108 **3.1.1 Description of the algorithm**

109 The Monte Carlo algorithm by Karp et al. [22] finds a perfect matching, with high probability,  
 110 in a random 2-out bipartite graph, sampled from the complete bipartite graph. A random  
 111 2-out bipartite graph  $B_{2o}$  is constructed by selecting uniformly at random two row vertices  
 112 for each column, and two column vertices for each row. These selections form the edges  
 113 of  $B_{2o}$ . Given the edges of  $B_{2o}$ , Karp et al. define two multigraphs. The *Column-Graph*  
 114 (CG) is the multigraph whose vertices are the rows, and whose edges are the choices of the  
 115 columns. That is, there is an edge in CG for a column vertex in  $B_{2o}$ . Parallel edges occur  
 116 if two columns select the same rows. The *Row-Graph* (RG) is defined similarly. The main  
 117 idea to show that  $B_{2o}$  has a perfect matching is the following. In a component of CG that  
 118 contains a cycle, it is possible to match all rows (vertices in CG) with one of the columns  
 119 that have selected them (edges in CG). On the other hand in a tree component of CG, in  
 120 any matching (pairing of edges with vertices) there will always be a free row vertex. As a  
 121 consequence, when one or more trees appear in CG, the choices of the columns alone do  
 122 not suffice to find a perfect matching, and those of the rows must be used. The algorithm  
 123 thus keeps track of the tree components of CG and tries to identify one row vertex per tree  
 124 component whose selections should be taken into account. The columns selected by such a  
 125 row could be used for a set of rows belonging in tree components. Thus one should go back  
 126 and forth identifying trees in CG and analyzing components in RG. Karp et al.'s algorithm,  
 127 which is described in Algorithm 1, formalizes this approach.

128 The algorithm operates on  $H_1$ , a copy of CG, and  $H_2$ , a copy of RG initially devoid of  
 129 edges. It furthermore uses two arrays `checked` for columns and `marked` for rows. These two  
 130 arrays together signal whether a vertex will be matched with one of its two selections or not.  
 131 More specifically, if a row vertex  $r$  is marked (i.e., `marked[r]=true`), then the algorithm will  
 132 match  $r$  with one of its two selections. On the other hand, if a column  $c$  is checked (i.e.,  
 133 `checked[c]=true`), then the algorithm will match  $c$  with one of the marked row vertices that  
 134 have selected it.

135 Initially, all row vertices are unmarked and all column vertices are unchecked. The  
 136 algorithm at each step picks a tree from  $H_1$  and marks one of its vertices  $x$ . This signifies  
 137 that  $x$  can only be matched with one of its choices. Then, the edge of  $x$  is inserted in  $H_2$ .  
 138 The algorithm then finds the component  $Q_x$  in  $H_2$  containing the edge  $x$ , and selects an  
 139 unchecked column  $y$  from  $Q_x$ . Column  $y$  is checked, which means that it can only be matched  
 140 with a marked vertex. As  $y$ 's choices are rendered useless now, the corresponding edge is  
 141 removed from  $H_1$  upon which new trees can arise. For each tree vertex  $x$  identified in  $H_1$ , one  
 142 should be able to find a vertex in the associated component  $Q_x$ , so that  $x$  can be matched in  
 143 that component. Otherwise,  $Q_x$  has more edges than vertices, and any matching of vertices  
 144 with edges in  $Q_x$  will hence leave some edges unpaired. In other words, Algorithm 1 has  
 145 decided that all columns that correspond to edges in  $Q_x$  should be matched with one of their  
 146 two selections. However, the union of the rows denoted by these selections has cardinality  
 147 strictly smaller than the number of such columns, and that is why a column is always left  
 148 unmatched by the algorithm if this scenario occurs. The algorithm returns failure upon  
 149 detecting this case (Line 10). The algorithm terminates successfully if all trees have a marked  
 150 vertex. If this happens, each component in  $H_1$  will have as many edges as unmarked vertices.  
 151 Likewise, each component in  $H_2$  will have as many edges as checked vertices. It is therefore  
 152 possible to orient the edges in either  $H_1$  or  $H_2$  such that each vertex (excluding marked rows  
 153 or unchecked columns) is matched with a unique adjacent edge. This gives a perfect matching  
 154 in  $B_{2o}$ , which can be found by the Karp–Sipser heuristic in linear time. Algorithm 1 finds a  
 155 perfect matching with probability  $1 - O(n^{-\alpha})$ , where  $\alpha$  is a positive constant.

■ **Algorithm 1** 2OUTMC: Monte Carlo on 2-out graphs

---

```

1:  $H_1 \leftarrow CG$ ,  $H_2 \leftarrow$  empty graph with columns as vertices;
2: All vertices in  $H_1$  are unmarked, all vertices in  $H_2$  are unchecked;
3:  $CORE \leftarrow$  edges in cycles of  $CG$ 
4: while there exists a tree  $T$  in  $H_1$  with no marked vertex do
5:   Let  $x$  be a random vertex of  $T$    ►  $x$  is a column vertex
6:    $marked[x] \leftarrow true$    ►  $x$  must be matched with one of its choices
7:   Add the edge of  $x$  in  $H_2$ 
8:   Let  $Q_x$  be the component in  $H_2$  containing the edge of  $x$ 
9:   if  $Q_x$  has no unchecked vertices then
10:    Return Fail   ►  $Q_x$  has more edges than vertices (no 1-1 pairing possible)
11:   else
12:    Select an unchecked vertex  $y$  of  $Q_x$ . In case of ties, prefer one from  $CORE$ 
13:     $checked[y] \leftarrow true$    ►  $y$  will be matched with a row that selected it
14:    delete  $y$  in  $H_1$    ► The algorithm forgets  $y$ 's choices
15: Create  $B'_{2o}$  from  $B_{2o}$  by keeping only edges between marked rows and checked columns (edges
    in  $H_2$ ) or unmarked rows and unchecked columns (edges in  $H_1$ )
16: Apply Karp–Sipser on  $B'_{2o}$  to find a perfect matchin

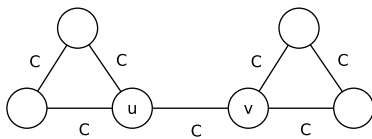
```

---

156 The authors then describe how to efficiently implement the algorithm such that it runs  
 157 in  $O(n \log n)$  worst case time. They identify two main tasks:

- 158 ■ **Task A:** Keep track of the tree components during edge deletions in  $H_1$ .
- 159 ■ **Task B:** Keep track of the connected components during edge insertions in  $H_2$ , and the  
 160 single unchecked vertex in each component.

161 Task B can be efficiently done in amortized near linear time (over the course of the  
 162 algorithm) by using a union–find data-structure and keeping the identity of the single  
 163 unchecked vertex in a component of  $H_2$  at the root of the component. For Task A, Karp  
 164 et al. propose the following. In the beginning, the edges of  $CG$  are labeled as  $\mathcal{F}$ , if their



■ **Figure 1** Algorithm 1 does not recognize new trees, if another edge is deleted after  $(u, v)$ .

165 deletion creates a tree;  $\mathcal{T}$ , if they belong to a tree component; and  $\mathcal{C}$  otherwise. Let **c-degree**  
 166 of a vertex  $v$  be the number of  $\mathcal{C}$  edges incident on  $v$ . During deleting the edge  $(u, v)$  from  
 167  $H_1$ , one of the following is performed depending on the label of  $(u, v)$ .

- 168 ■ **Case 1:**  $(u, v)$  is  $\mathcal{C}$ : The **c-degrees** of  $u$  and  $v$  are decreased by one. Then, while there is  
 169 a vertex with a single  $\mathcal{C}$  edge; its  $\mathcal{C}$  edge is relabeled as  $\mathcal{F}$ .
- 170 ■ **Case 2:**  $(u, v)$  is  $\mathcal{F}$ : Using a dove-tailed depth-first search, where depth-first searches from  
 171  $u$  and  $v$  are interleaved, the tree component created can be found in time proportional to  
 172 its size. One then changes the labels of all edges in this tree from  $\mathcal{F}$  to  $\mathcal{T}$ .
- 173 ■ **Case 3:**  $(u, v)$  is  $\mathcal{T}$ : Deleting  $(u, v)$  creates two trees. As in the previous case, a dove-  
 174 tailed DFS is used to find these two trees in time proportional to the size of the smaller  
 175 one. The new trees are to be examined by the algorithm.

176 We identify an oversight in this procedure, where the algorithm fails to keep track of some  
 177 trees in  $H_1$ . We demonstrate this by an example. In Figure 1, if the edge between vertices  $u$   
 178 and  $v$  gets deleted, then the connected component is split into two triangles. The **c-degree** of  
 179 both  $u$  and  $v$  decreases to two, and as both are greater to one, the deletion procedure stops  
 180 without any action. However, both triangles are unicyclic. If an edge is deleted from either  
 181 triangle, then Case-1 does not recognize that the remaining edges should be relabeled as  $\mathcal{T}$   
 182 not  $\mathcal{F}$ .

183 If Algorithm 1 is not able to keep track of all the trees in  $H_1$ , then it can exit the loop  
 184 of Line 4 prematurely. As a consequence Karp–Sipser in Line 16 will return a suboptimal  
 185 matching. We propose a fix for this oversight in Lemma 1.

186 ► **Lemma 1.** *Let  $u$  be an endpoint of a deleted edge  $(u, v)$  with label  $\mathcal{C}$ . Apply the procedure*  
 187 *of Case-1 until we arrive at a vertex  $p$  with  $\text{c-degree}[p] \neq 1$ . If  $\text{c-degree}[p] = 0$ , then  $u$ 's*  
 188 *component has become a tree.*

189 **Proof.** We claim that if  $\text{c-degree}[p] = 0$ , then  $p$  and  $v$  are the same vertex. Each vertex on  
 190 the path from  $u$  to  $p$  had its **c-degree** affected twice (from 2 to 0), except  $p$ . Hence for  $p$  to  
 191 become 0, its **c-degree** must have been equal to 1. If  $p \neq v$ , then  $p$  should had its  $\mathcal{C}$  edge  
 192 relabeled during another deletion process. Therefore, prior to the deletion of  $(u, v)$ , there was  
 193 a cycle on  $H_1$  with all vertices having **c-degree** equal to 2, and both their  $\mathcal{C}$  edges participated  
 194 in the cycle. Any outgoing edges from vertices of the cycle therefore were labeled  $\mathcal{F}$  and by  
 195 definition, their deletion led to a tree being formed. The component was hence unicyclic  
 196 before. ◀

197 **Case 1-continuation** is therefore as follows:

- 198 ■ Once there are no vertices with **c-degree** equal to 1, take the last vertex  $v$  whose **c-degree**  
 199 was reduced. If  $\text{c-degree}[v] = 0$ , then relabel all edges in  $vs$  component from  $\mathcal{F}$  to  $\mathcal{T}$ .
- 200 This addition has overall  $O(n)$  cost, because each edge can change label at most twice.

### 201 3.1.2 Conversion to an efficient general heuristic

202 Algorithm 1 works well when the random 2-out graph is sampled from  $K_{n,n}$ . However, in the  
 203 case of an arbitrary host graph, the underlying theory is not shown to hold, and the algorithm

204 can make erroneous decisions. Here we discuss how to turn Algorithm 1 into a general  
 205 heuristic. Apart from the aim of obtaining a practical heuristic for bipartite matching, there  
 206 is another reason to investigate the matching problem in 2-out bipartite graphs. We show in  
 207 Appendix D that an  $O(f(n, m))$  time algorithm to find a maximum cardinality matching in  
 208 a 2-out bipartite graph can be used to find a maximum cardinality matching in any bipartite  
 209 graph with  $m$  edges in  $O(f(m, m))$  time, where  $f$  is a function on the number of vertices  $n$   
 210 and edges  $m$ . Such a reduction is important because it shows that an algorithm for finding  
 211 maximum cardinality matchings in 2-out graphs with similar complexity to 2OUTMC can be  
 212 used to obtain an  $O(m \log m)$  algorithm for matchings in general bipartite graphs.

213 If the algorithm reaches Line 10 during execution, it quits immediately before examining  
 214 all trees in  $H_1$ . We instead propose to continue with the execution of the algorithm to make  
 215 the returned matching as large as possible. To achieve this efficiently, we keep for each tree  $T$   
 216 a list  $L_T$  of unmarked vertices. At Line 5 we randomly sample  $x$  from  $L_T$  and discard it from  
 217  $L_T$ . Contrary to Algorithm 1, we neither mark  $x$  nor insert it in  $H_2$  yet. Instead, we examine  
 218 first whether the component in  $H_2$  of either of the two choices of  $x$  has an unchecked column  
 219  $y$ . If  $y$  exists, we mark  $x$ , insert it to  $H_2$  and continue by deleting  $y$  from  $H_1$ . Otherwise,  
 220 we perform the same set of actions with another randomly sampled vertex from  $L_T$ . If  $L_T$   
 221 becomes empty, and no vertex was marked, we abandon  $T$  and proceed to another tree. Each  
 222 such tree in the final state of  $H_1$  decreases the cardinality of the returned matching by one,  
 223 as a row is left free. If  $T$  is split into two trees, the lists of unmarked vertices for the new  
 224 trees contain only those vertices still inside  $L_T$  at the moment of splitting. This is necessary  
 225 to avoid sampling vertices more than once.

226 The overall algorithm 2OUTMC is as follows. It takes the matrix representation of the  
 227 given bipartite graph and scales it with a few steps of the Sinkhorn–Knopp algorithm to  
 228 obtain  $\mathbf{A}_S$ . It then chooses two random neighbors for each column and row using their  
 229 respective probability distributions in the corresponding row and column of  $\mathbf{A}_S$ , which are  
 230 given as input to Algorithm 1. Then, the auxiliary graph  $B_{2o}$  is constructed and Karp–Sipser  
 231 is run on this graph to retrieve a maximum cardinality matching in  $B_{2o}$ . If one allows vertices  
 232 to choose neighbors uniformly, then there are no guarantees on the maximum cardinality of  
 233 a matching in  $B_{2o}$ . As an example, consider the graph where the  $i$ th row and  $i$ th column  
 234 are connected for  $i = 1, \dots, n$ , and additionally the first  $\ell$  rows and columns are connected  
 235 with every vertex on the opposite side. Then, in expectation  $O(\frac{\ell-1}{\ell+1} \cdot n)$  rows (resp. columns)  
 236 make both choices from the first  $\ell$  columns (resp. rows), such that in the generated  $B_{2o}$  the  
 237 maximum cardinality matching is of size  $O(\frac{n}{\ell} + \ell)$ . Using  $\mathbf{A}_S$ 's values to perform the random  
 238 choices spreads the choices so that the maximum cardinality of the matching in the subgraph  
 239 increases (see Theorem 2 and Lemmas 6–8 in [16] that examines the 1-out subgraph model).

240 In Appendix B we describe two heuristics for 2OUTMC which can lead to an increase in  
 241 the cardinality of the returned matching. The main idea of both heuristics is to reduce the  
 242 chance that an edge deletion in  $H_1$  creates a new tree.

## 243 3.2 TRUNCERW: Truncated random walk with nonuniform sampling

### 244 3.2.1 Description of the algorithm for regular bipartite graphs

245 Goel et al. [18] propose a randomized algorithm (of the Las Vegas type) that finds a perfect  
 246 matching in a  $d$ -regular bipartite graph with  $n$  vertices in each side in  $O(n \log n)$  time in  
 247 expectation. This algorithm starts a random walk from a randomly chosen free column-vertex.  
 248 At a column vertex  $c$ , the algorithm selects uniformly at random one of the row-vertices that  
 249 are not matched to  $c$ , and goes to the chosen row vertex  $r$ . If  $r$  is free, then an augmenting  
 250 path is obtained by removing possible loops from the walk. If  $r$  is matched, then the random

$$\begin{pmatrix} \sqrt{2} & & & \\ & \frac{1}{\sqrt{2}} & & \\ & & \frac{1}{\sqrt{8}} & \\ & & & \frac{1}{\sqrt{8}} \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{8}} & & & \\ & \frac{1}{\sqrt{8}} & & \\ & & \frac{1}{\sqrt{2}} & \\ & & & \sqrt{2} \end{pmatrix} = \begin{pmatrix} 1/2 & 1/2 & 0 & 0 \\ 1/4 & 1/4 & 1/2 & 0 \\ 1/8 & 1/8 & 1/4 & 1/2 \\ 1/8 & 1/8 & 1/4 & 1/2 \end{pmatrix}$$

■ **Figure 2** The matrix  $\mathbf{A}$  associated with a  $4 \times 4$  Hessenberg matrix, the scaling matrices  $\mathbf{D}_R$  and  $\mathbf{D}_C$ , and the resulting doubly stochastic matrix  $\mathbf{A}_S = \mathbf{D}_R \mathbf{A} \mathbf{D}_C$ . In general,  $\mathbf{A}_S(n, 1) = 1/2^{n-1}$ .

251 walk goes to the mate of  $r$ . Goel et al. show that the total length of the random walks is  
 252  $O(n \log n)$  in expectation, and thus the algorithm obtains a perfect matching in the stated  
 253 time [18, Theorem 4]. They also show that one can obtain a Monte Carlo-type algorithm  
 254 by truncating the random walks. The expected length of an augmenting path with respect  
 255 to a given matching of cardinality  $j$  is  $2(4 + 2n/(n - j))$ , and the random walks could be  
 256 truncated at this length to obtain near optimal matchings in  $O(n \log n)$  time.

257 A random walk is easy to implement for  $d$ -regular bipartite graphs. At a column vertex  $c$ ,  
 258 one can create a random number between 1 and  $d$  in  $O(1)$  time and choose the neighbor at  
 259 that position, and repeat the experiment if the mate of  $c$  is chosen. This will take  $O(1)$  time  
 260 in expectation for each step of the walk, and the run time bound of  $O(n \log n)$  is maintained.

261 Goel et al. show that the random-walk based algorithm will work for finding perfect  
 262 matchings in the bipartite graph representation of a doubly stochastic matrix. They also  
 263 suggest using an existing data structure [20] when the row and column sums are constant  
 264 with nonnegative integer entries bounded by a polynomial in  $n$ , to attain an  $O(n \log n)$  run  
 265 time bound. A more recent paper [32] removes the restriction on the entries, and obtains  
 266 an expected constant time per update and sampling. Further investigations and a careful  
 267 implementation are necessary to apply the mentioned sampling approaches in our context.  
 268 Instead, for general doubly stochastic matrices without any bound on the entries, Goel et  
 269 al. propose an augmented binary search tree with which each selection step of the random  
 270 walk can be implemented in  $O(\log n)$  time, and obtain a run time of  $O(m + n \log^2 n)$  in  
 271 expectation, with a total of  $O(m)$  preprocessing time.

### 272 3.2.2 Conversion to an efficient general heuristic

273 Let  $c$  be a free column vertex with respect to a given matching of cardinality  $j$ . Assuming  
 274 there is a perfect matching, one can find an augmenting path to match  $c$ , and a random walk  
 275 can find it. The  $O(\frac{n}{n-j})$  bound on the expected length of such a path will not hold if the  
 276 bipartite graph is not regular. One may perform more than  $m$  steps, which is the worst case  
 277 time complexity of deterministically finding an augmenting path starting from a free vertex.  
 278 We propose two methods to make the random walks more useful and to sample efficiently in  
 279 a random walk. We also discuss an efficient implementation of the whole approach.

280 The first proposed method is to scale the matrix representation  $\mathbf{A}$  of a given bipartite  
 281 graph to obtain a doubly stochastic matrix  $\mathbf{A}_S$  for random selections. The expected length  
 282 of a random walk to find an augmenting path holds when  $\mathbf{A}_S$  has bounded nonzero entries.  
 283 In general, one does not have any bound on the entries of  $\mathbf{A}_S$ . Consider the matrix  $\mathbf{A}$   
 284 associated with an upper Hessenberg matrix of size  $n$ .  $\mathbf{A}$  has a full lower triangular part,  
 285 and additional  $n - 1$  entries  $\mathbf{A}(i - 1, i) = 1$  for  $i = 2, \dots, n$ , and fully indecomposable. The  
 286  $4 \times 4$  example along with its unique scaling matrices are shown in Fig. 2. In the resulting  
 287 scaled matrix  $\mathbf{A}_S(n, 1) = 1/2^{n-1}$  whose inverse is not bounded polynomially in  $n$ .

288 As highlighted at the end of Section 3.2, one needs an  $O(\log n)$  time algorithm to select a  
 289 row vertex randomly from a given column vertex. The second proposed method is a simple  
 290 yet efficient algorithm for this purpose, rather than a sophisticated augmented tree. The  
 291 main components of the proposed sampling method are as follows. For each column vertex  $c$ ,  
 292 with  $d_c$  neighbors, we have:



## 8 Almost optimal algorithms for bipartite matching

293 ■  $\text{adj}_c[1, \dots, d_c]$ : an array keeping the neighbors of  $c$ .  
 294 ■  $\text{wghts}_c[1, \dots, d_c]$ : the weight of the edges incident on  $c$ . This array is parallel to the first  
 295 one so that the weight of the edge  $(c, \text{adj}_c[i])$  is  $\text{wghts}_c[i]$ .  
 296 ■  $\text{medge}[c]$ : the position of the mate of  $c$  in the array  $\text{adj}_c$ , or  $-1$  if  $c$  is not matched.  
 297 At the beginning, we compute the prefix sum of  $\text{wghts}_c[1, \dots, d_c]$ . After this operation, the  
 298 total weight of the edges incident on  $c$  is  $\text{wghts}_c[d_c]$ , and the weight of the edge  $(c, \text{mate}[c])$   
 299 is  $\text{wghts}_c[\text{medge}[c]] - \text{wghts}_c[\text{medge}[c] - 1]$ , assuming that  $\text{wghts}_c[0]$  signifies zero.  
 300 Given the prefix sums in  $\text{wghts}_c[1, \dots, d_c]$ , the position of the mate of  $c$  at  $\text{medge}[c]$ , we  
 301 can choose a random neighbor (which is not equal to  $\text{mate}[c]$ ) as shown in Algorithm 2. We  
 302 use a binary search function, `binSearch`, which takes an array, the array's start and end  
 303 positions, a target value, and returns the smallest index of an array element which is larger  
 304 than the given value with binary search (we skip the details of this search function). At  
 305 Line 5, since  $c$  does not have a mate, we search in the whole list. At Line 8, since the prefix  
 306 sum just before  $\text{medge}[c]$  is larger than the target value, we search in the first part of  $\text{wghts}_c$   
 307 until the current mate located at  $\text{medge}[c]$ . At Line 10, we search on the right of  $\text{medge}[c]$ ,  
 308 by a modified target value. This last part is the gist of the algorithm's efficiency as it avoids  
 309 updating the prefix sums when the mate changes.

■ **Algorithm 2** Sampling a random neighbor of the column vertex  $c$  with  $d_c$  neighbors.

---

**Require:**  $\text{adj}_c[1, \dots, d_c]$ ,  $\text{wghts}_c[1, \dots, d_c]$ , and  $\text{medge}[c]$ .

- 1:  $\text{mwght} \leftarrow \text{wghts}_c[\text{medge}[c]] - \text{wghts}_c[\text{medge}[c] - 1]$  if  $\text{medge}[c] \neq -1$ , otherwise 0
- 2:  $\text{totalW} \leftarrow \text{wghts}_c[d_c] - \text{mwght}$    ► The total weight of the edges that can be sampled
- 3: create a random value  $rv$  between 0 and  $\text{totalW}$
- 4: **if**  $\text{medge}_c = -1$  **then**
- 5:   **return** `binarySearch`( $\text{wghts}_c[1, \dots, d_c], rv$ )
- 6: **else**
- 7:   **if**  $\text{wghts}_c[\text{medge}_c] - \text{mwght} \geq rv$  **then**
- 8:     **return** `binSearch`( $\text{wghts}_c[1, \dots, \text{medge}[c] - 1], rv$ )
- 9:   **else**
- 10:    **return** `binSearch`( $\text{wghts}_c[\text{medge}[c] + 1, \dots, d_c], rv + \text{mwght}$ ) +  $\text{medge}_c$

---

310 The sampling algorithm returns the index of the neighbor in  $\text{adj}_c$  different from the  
 311 current mate in time  $O(\log d_c)$ , independent of the values of the edges. It thus respects the  
 312 required run time bound. If we were to apply the rejection sampling (as discussed before for  
 313 the regular bipartite graphs), the run time would depend on the value of the matching edge  
 314 that we want to avoid. This could of course lead to an expected run time of more than  $O(n)$ .

315 There are two key components of Algorithm 2. The first one is the prefix sum, which  
 316 is computed once before the random walks start and does not change. The second one is  
 317  $\text{medge}[c]$ , the position of  $\text{mate}[c]$  in  $\text{adj}_c$ . The value  $\text{medge}[c]$  changes and needs to be updated  
 318 when we perform an augmentation. We handle this update as follows. We keep the random  
 319 walk in a stack by storing only the column vertices, as the row vertices direct the walk to  
 320 their mate, or terminate the walk if not matched. We discard the cycles from the random  
 321 walk as soon as they arise—this way we only store a path on the stack, and its length can  
 322 be at most  $n$ . Storing a path also enables keeping the  $\text{medge}[\cdot]$  up-to-date. Every time we  
 323 sample an outgoing edge from a column vertex  $c$ , we assign the location of the sampled  
 324 row vertex in  $\text{adj}_c$  to a variable  $\text{nmedge}[c]$ . When we find a free row, the stack contains the  
 325 column vertices of the corresponding augmenting path, whose new mates' locations are in  
 326  $\text{nmedge}[\cdot]$  and thus can be used to update  $\text{medge}[\cdot]$ .

327 The described procedure will work gracefully in expected  $O(m + n \log n)$  time for regular

328 bipartite graphs and for doubly stochastic matrices where the nonzero values do not differ by  
 329 large. On the other hand, when there are large differences in edge weights, a random walk can  
 330 get stuck in a cycle. That is why truncating the long walks is necessary to make the algorithm  
 331 work for any given doubly stochastic matrix. Furthermore, such a truncation is necessary  
 332 with the proposed matrix scaling approach for defining random choices. For the overall  
 333 approach to be practical, we should not apply the scaling algorithms until convergence. As  
 334 in the previous approaches [15, 16], we allot a linear time of  $O(m + n)$  for scaling. Applying  
 335 Sinkhorn–Knopp algorithm for a few iterations will thus be allowable. The known convergence  
 336 bounds for the Sinkhorn–Knopp algorithm [27, Thm. 4.5] apply asymptotically, therefore  
 337 we do not have any bounds on the error after a few iterations; it can be large. That is why  
 338 truncation makes the random walk based augmenting path search practical.

339 The overall algorithm TRUNCROW is thus as follows. It takes the matrix representation of  
 340 the given bipartite graph and scales it with a few steps of the Sinkhorn–Knopp algorithm.  
 341 Then for  $j = 0$  to  $n - 1$ , it uniformly at random picks a free column vertex, and starts a  
 342 random walk starting from that column, for at most  $2(4 + 2n/(n - j))$  steps, after which  
 343 the walk is truncated. Some follow discussion and experiments with different parameters for  
 344 TRUNCROW may be found in Appendix C.

## 345 4 Experiments

346 We implemented 2OUTMC and TRUNCROW in C/C++, and the codes are accessible from  
 347 <https://gitlab.inria.fr/bora-ucar/fast-matching>. The codes, all are sequential, were  
 348 compiled with "-O3" and run on a machine with 2 x Intel Xeon CPU Gold 6136 CPUs and 187  
 349 GB RAM. We evaluate 2OUTMC and TRUNCROW both on real-life and synthetic bipartite  
 350 graphs with equal number of vertices in each side. We compared the two algorithms  
 351 against KASI, the widely used version of Karp–Sipser which applies degree-1 reduction  
 352 (own implementation), and KASI2, the original version of Karp–Sipser with both reduction  
 353 rules. We use a publicly available implementation of KASI2 ([https://gitlab.inria.fr/  
 354 bora-ucar/karp--sipser-reduction](https://gitlab.inria.fr/bora-ucar/karp--sipser-reduction)) which is the fastest of recent implementations [26,  
 355 29]. We note that there are other heuristics (a short summary and further references are  
 356 in Appendix A) which deliver very good results in practice. For most of these heuristics,  
 357 especially for those based on vertex degree, there are known worst case upper bounds close  
 358 to  $1/2$ . We therefore restrict the focus on KASI and KASI2, which are efficient and very  
 359 effective in practice [14, 25, 30]. We also investigated if random 2-out bipartite graphs of a  
 360 general host graph have perfect matchings if rows and columns select neighbors with the  
 361 probabilities in the scaled matrix representation. The quality of a matching refers to the  
 362 ratio of the cardinality of the matching to the maximum cardinality of a matching in a given  
 363 graph. The practical version of Sinkhorn-Knopp is referred to as SK- $t$ , where  $t$  is the number  
 364 of allowed iterations. All run times are reported in seconds.

### 365 4.1 Investigation of perfect matchings in 2-out graphs

366 Here, we investigate the claim that  $G_2$  will likely have a perfect matching for  $G$ , if created  
 367 with the probabilities in the scaled matrix. We used a set of 39 large sparse square matrices  
 368 from the SuiteSparse Collection [12], whose bipartite graphs have perfect matchings. These  
 369 matrices are automatically selected from all square matrices available at the collection with  
 370  $10^6 \leq n \leq 28 \times 10^6$ , and with at least two nonzeros per row or column.

371 We consider two different models to create  $G_2$ . In the model  $M_1$ , row choices are  
 372 independent of the column choices. Under this model, a row and a column can select each  
 373 other resulting in parallel edges—only one of them is kept. The model  $M_2$  tries to avoid

$\frac{m}{n}$	[0,10)		[10,20)		[20,30)		[30,40)		[40,50)	
#Instances	27		5		5		1		1	
	#PM	deficiency	#PM	deficiency	#PM	deficiency	#PM	deficiency	#PM	deficiency
Model M <sub>1</sub>	0	223	0	8	1	20	0	2	1	0
Model M <sub>2</sub>	27	0	3	3	1	10	0	1	0	1

■ **Table 1** We divide the real-life graphs into five groups. The  $i$ th group consists of graphs whose  $\frac{m}{n}$  ratio is between  $10(i - 1)$  and  $10i$ . For each group, we give the number of instances in which a 2-out graph built using the models M<sub>1</sub> and M<sub>2</sub> has a perfect matching and the largest difference from the maximum cardinality of a matching.

$h$	KASi		2OUTMC		TRUNCRW	
	KASi	KASi2	Uniform	SK-5	Uniform	SK-5
2	0.93	1.00	0.78	0.99	0.88	0.99
8	0.80	0.85	0.59	0.99	0.91	0.99
32	0.69	0.72	0.52	0.99	0.83	0.99
128	0.64	0.65	0.51	0.99	0.78	0.99
512	0.61	0.63	0.52	0.99	0.76	0.99

■ **Table 2** Average quality of the matchings found by the algorithms on graphs from the synthetic family  $\mathcal{I}$  for  $n = 30000$  and various values of  $h$ .

374 parallel edges. In this model, all columns perform their selections. Then, each row  $r$  attempts  
 375 to randomly choose two columns, only from those that did not select  $r$ . These selections again  
 376 are based on the scaled matrix. In this model, parallel edges can arise (and be discarded)  
 377 only when a vertex  $v$  is connected in the 2-out graph with all of its neighbors in  $G$ , because  
 378 it is impossible for  $v$  to select otherwise. We experimented three times with each real-life  
 379 graph. M <sub>$i$</sub> 's result is the maximum of those three experiments. In each test, we first created  
 380 the choices of all columns. Then we allowed the two models to generate the choices of the  
 381 rows accordingly.

382 The results are shown in Table 1 for the 39 real-life graphs and are with SK-5. As seen in  
 383 this table, the random  $G_2$  graphs generated with the model M<sub>1</sub> have near perfect matchings,  
 384 but they do not contain perfect matchings in most cases. In contrast, the random  $G_2$  graphs  
 385 generated by M<sub>2</sub> in many cases contain a perfect matching. In only a few graphs this does  
 386 not hold true, and in these cases the deficiency is no more than 10.

### 387 4.2 On synthetic graphs

388 In Table 2, we give results with a synthetic family  $\mathcal{I}$  of graphs from literature [16], whose  
 389 matrix representations do not have total support. To create a member of  $\mathcal{I}$ , we separate  
 390 the vertex set  $R$  into  $R_1 = \{r_1, \dots, r_{n/2}\}$  and  $R_2 = \{r_{n/2+1}, \dots, r_n\}$  and likewise for  $C$ .  
 391 All vertices of  $R_1$  are connected to all vertices of  $C_1$ . Edges  $(r_i, c_{n/2+i})$  and  $(r_{n/2+i}, c_i)$  for  
 392  $i = 1, \dots, n/2$  are added to introduce a perfect matching. A parameter  $h$  is used to connect  
 393  $h$  vertices from  $R_1$ , and  $h$  vertices from  $C_1$  to every vertex on the opposite side.

394 As seen in Table 2, KASi and KASi2 have more and more difficulty with increasing  $h$ .  
 395 The matching quality drops over 30% between  $h = 2$  and  $h = 512$  for KASi and almost 40%  
 396 for KASi2. On the contrary, 2OUTMC and TRUNCRW both obtain a near perfect matching,  
 397 with SK-5. Even though the matrices associated with the graphs of  $\mathcal{I}$  lack total support,  
 398 SK-5 sufficed to obtain near optimal matchings. We notice the effect of scaling: if vertices  
 399 select without scaling (Uniform), the matching quality reduces. This is particularly true  
 400 for 2OUTMC, which exhibits the worst overall performance with uniform selection. Family  
 401  $\mathcal{I}$  shows the importance of scaling, and more importantly highlights the robustness of the  
 402 proposed methods. An adversary can create graphs which make degree-based randomized

$n$	KASi	KASi2	2OUTMC			TRUNCRW		
	quality	quality	uniform	SK-5	SK-20	uniform	SK-5	SK-20
10000	0.76	0.84	0.81	0.92	0.95	0.97	0.97	0.97
20000	0.73	0.83	0.81	0.92	0.95	0.97	0.97	0.97
30000	0.73	0.83	0.81	0.92	0.95	0.97	0.97	0.97

■ **Table 3** Average quality of the matchings found by the algorithms on graphs from the synthetic family  $\mathcal{J}$  for  $n \in \{10000, 20000, 30000\}$ .

403 approaches lose quality—some of those heuristics are briefly mentioned in Appendix A, and  
 404 the full details including negative results on KASi2 can be found elsewhere [9]. On the other  
 405 hand, the use of scaling helps to avoid such cases for 2OUTMC and TRUNCRW.

406 We now discuss another synthetic family of graphs  $\mathcal{J}$  in which the proposed approaches  
 407 obtain matchings of much higher quality than KASi and KASi2. A bipartite graph with  $n$   
 408 vertices per side belonging to  $\mathcal{J}$  contains the following edges:  $(r_i, c_j)$  for all  $i \leq j$ ;  $(r_2, c_1)$ ,  
 409  $(r_n, c_{n-1})$ ;  $(r_3, c_1)$ ,  $(r_3, c_2)$ ,  $(r_n, c_{n-2})$ ; and  $(r_{n-1}, c_{n-2})$ . The graphs in  $\mathcal{J}$  are hard for  
 410 Karp–Sipser-based heuristics because only few of the edges participate in a perfect matching,  
 411 the deterministic rules do not apply, and hence they resort to multiple suboptimal random  
 412 decisions. Likewise, due to the large number of entries without support in the matrix  
 413 representation, Sinkhorn–Knopp will take many iterations to properly scale the matrix.

414 In Table 3, we give results of the algorithms for a few graphs from this family. In the  
 415 table, we also show the effects of scaling on 2OUTMC and TRUNCRW by showing results  
 416 without scaling (under column “uniform”, in which a column vertex chooses a neighbor  
 417 uniformly at random), with SK-5, and with SK-20. As can be seen, despite the lack of  
 418 total support, both 2OUTMC and TRUNCRW obtain matchings whose cardinality is more  
 419 than 0.92 of the maximum, when SK-5 or SK-20 is used. TRUNCRW in particular is nearly  
 420 optimal. These results are always better than that of KASi and KASi2, with the difference  
 421 in matching quality being about 20–25% for the former, and 10–15% for the latter. With  
 422 increased iterations of Sinkhorn–Knopp, 2OUTMC increases the cardinality of its matchings  
 423 by 3%. If we do not use scaling (“uniform”), while there’s no noticeable effect on TRUNCRW’s  
 424 matchings, 2OUTMC matchings decrease by roughly 10%. Even so, its results remain better  
 425 than KASi’s and on par with those of KASi2.

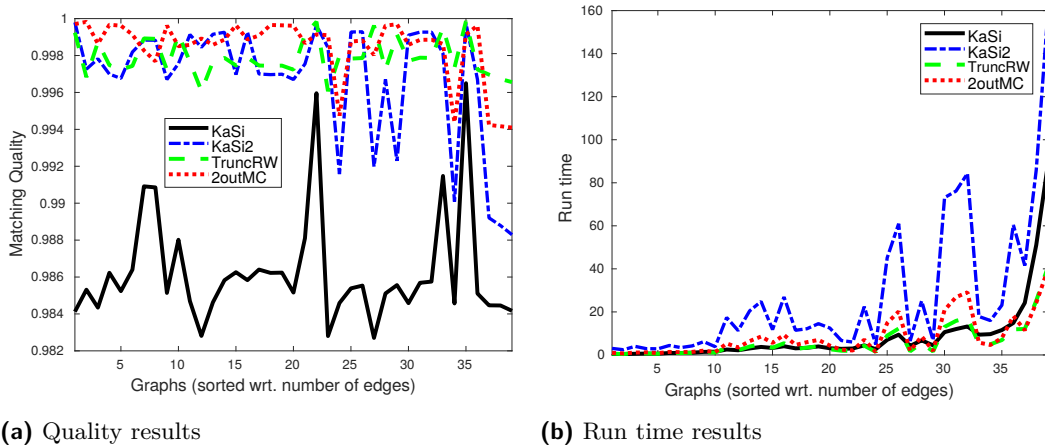
### 426 4.3 On real-life graphs

427 We compared TRUNCRW and 2OUTMC with KASi and KASi2 on all 39 real-life graphs from  
 428 Section 4.1. Figure 3a and Figure 3b present the high level picture. For the experiments, we  
 429 did not permute the matrices randomly, which generally increases the experimentation time.

430 The results for matching quality can be seen in Figure 3a, where we plot the ratio of the  
 431 cardinality of the matchings found by different algorithms to the maximum cardinality of  
 432 the matching. The graphs are indexed in nondecreasing number of edges. 2OUTMC and  
 433 TRUNCRW use SK-3 for scaling. As can be observed, both 2OUTMC and TRUNCRW obtain  
 434 near perfect matchings. The average matching quality obtained by 2OUTMC is 0.9979 and  
 435 that obtained by TRUNCRW is 0.9984. Both algorithms never drop below 0.9900 in any of  
 436 the 39 cases.

437 Figure 3a also shows the matching quality of KASi2 and KASi. KASi obtains matchings  
 438 of quality 0.9862 on average, with always smaller cardinality than TRUNCRW and 2OUTMC.  
 439 KASi2 fares better and its average quality is 0.9968. Even so, in the majority of cases, it  
 440 obtains matchings that are inferior quality-wise to both TRUNCRW and 2OUTMC.

441 While all algorithms obtain matchings of high quality, the absolute different is remarkable  
 442 in some cases. For example, the largest difference observed between the matching cardinalities



(a) Quality results

(b) Run time results

■ **Figure 3** Quality (left) and run time (right) results for all 39 graphs from Section 4.1.

443 obtained by 2OUTMC and KASI was 346577, in favor of 2OUTMC.

444 Figure 3b shows the run time of all examined heuristics, where the graphs are again  
 445 indexed in nondecreasing number of edges. KASI is in general the fastest of these four  
 446 algorithms when there are not too many edges. TRUNCROW and 2OUTMC are close run-time  
 447 wise to KASI and in some instances faster than it. This is especially true in instances with  
 448 many edges because KASI depends more on  $m$ . KASI2 has the slowest performance overall.

449 For a detailed study, we show results on the five largest graphs from the mentioned dataset  
 450 and `Circuit5M`, which was identified as a challenging instance in earlier work [25]. Degree-  
 451 1 vertices from `Circuit5M` are removed by applying Rule-1 of KASI2 as a preprocessing  
 452 step—this is without loss of generality of the heuristics. For each graph we relabeled its  
 453 row-vertices randomly and executed five tests with each algorithm.

454 Table 4 shows the matching quality and the run time of the four heuristics. 2OUTMC  
 455 and TRUNCROW used SK-3 for this set of experiments for speed. For each graph, we give the  
 456 minimum, maximum, and averages over five runs. As already discussed, all heuristics obtain  
 457 high quality matchings. On a closer look, we see that TRUNCROW, on average, matched  
 458 158410 more edges than KASI, and 50847 more edges than KASI2. Similarly 2OUTMC  
 459 matched 139220 more edges than KASI on average, and 31652 more edges than KASI2.  
 460 Interestingly, on graph `Channel-500` TRUNCROW was able to find the maximum matching.

461 Concerning run time, as KASI is a linear time heuristic it is expected to be the fastest.  
 462 Surprisingly, TRUNCROW even with the scaling time added is faster than KASI in three  
 463 instances. This is due to the fact that each iteration of the scaling algorithm takes linear time  
 464 with small constants. As an algorithm on its own (without scaling time), TRUNCROW becomes  
 465 the fastest one, thanks to its run time not depending on  $m$  after the initialization. 2OUTMC,  
 466 though slower, also exhibits good behavior, except in `nlpkkt240`. KASI2 has the worst run  
 467 time overall. Its initialization takes more time, and its implementation is more involved.  
 468 SK-3 is fast except for `nlpkkt240` where it requires about 30 seconds. The reason that SK-3  
 469 requires 30 seconds for this particular graph is due to the random permutation of its rows,  
 470 which is not cache-friendly (if SK-3 is run on `nlpkkt240` using the initial ordering of rows, it  
 471 finishes in less than 10 seconds). In the other cases and despite the large size of the graphs,  
 472 scaling finishes in less than seven seconds. Table 4 additionally shows that TRUNCROW and  
 473 2OUTMC’s run time performance does not seem to be affected by their random decisions.  
 474 The largest difference between the result of the minimum, and the maximum run is no more  
 475 than two seconds for both of these algorithms.

name	$n$	statistics	KAS1		KAS2		SK-3	2OUTMC		TRUNCRW	
			quality	time	quality	time	time	quality	time	quality	time
cage15	5.15	min.	0.99	12.67	0.99	26.89	4.59	0.99	8.82	0.99	8.27
		avg.	0.99	12.81	0.99	27.08	4.68	0.99	8.88	0.99	9.32
		max.	0.99	13.17	0.99	27.27	4.83	0.99	8.96	0.99	10.23
Channel-500	4.80	min.	0.99	10.12	0.99	20.63	2.74	0.99	7.63	1.00	3.86
		avg.	0.99	10.16	0.99	20.94	2.75	0.99	7.66	1.00	4.48
		max.	0.99	10.18	0.99	21.87	2.75	0.99	7.70	1.00	5.11
Circuit5M	5.55	min.	0.99	6.57	0.99	24.74	2.45	0.99	4.40	0.99	2.07
		avg.	0.99	6.76	0.99	24.93	2.84	0.99	4.56	0.99	2.19
		max.	0.99	7.03	0.99	25.33	4.16	0.99	4.81	0.99	2.35
Delaunay_24	16.00	min.	0.99	11.58	0.99	65.97	4.32	0.99	23.34	0.99	11.21
		avg.	0.99	11.61	0.99	68.30	4.44	0.99	23.58	0.99	11.31
		max.	0.99	11.66	0.99	72.47	4.48	0.99	24.38	0.99	11.37
Hugebub-20	21.19	min.	0.99	14.97	0.99	91.42	6.26	0.99	30.96	0.99	14.25
		avg.	0.99	15.04	0.99	97.77	6.29	0.99	31.28	0.99	14.38
		max.	0.99	15.15	0.99	106.78	6.31	0.99	31.59	0.99	14.57
nlpkkt240	27.99	min.	0.98	98.58	0.99	182.08	29.77	0.99	52.34	0.99	34.34
		avg.	0.98	98.66	0.99	183.10	29.92	0.99	52.53	0.99	34.50
		max.	0.98	98.76	0.99	186.08	30.27	0.99	52.76	0.99	34.70

■ **Table 4** Full run time comparisons with heuristics for the graphs of Section 4.3. The run time of SK-3 should be added to TRUNCRW and 2OUTMC. For each instance we give the minimum, the average, and the maximum of five runs for all columns regarding the quality and the run time. The number of vertices  $n$  per side is in the order of millions. Hugebub-20 stands for Hugebubbles-0020.

476 Combined with the results in the previous section, we conclude thus that (i) 2OUTMC  
477 and TRUNCRW always obtain near perfect matchings, while KAS1 and KAS2 are not as  
478 robust; (ii) 2OUTMC and TRUNCRW are nearly as fast as the linear time algorithm KAS1,  
479 and are much faster than KAS2.

480 Next, we consider the impact of our heuristics as initialization to an exact algorithm for  
481 finding a maximum cardinality matching. We first run the heuristics to obtain an initial  
482 matching, then call an exact algorithm to augment the initial matchings for maximum  
483 cardinality. We consider three different exact algorithms MC21, PR, and PF+ for the  
484 augmentation steps. MC21 [13] from `mmaker` [14, 25] visits free vertices one by one and  
485 tries to match the visited vertex with a depth-first search, and hence is closely related  
486 to TRUNCRW. In this setting, differences among the qualities of initial matchings should  
487 be observable while computing an exact matching. PR [25] is based on the Push-Relabel  
488 method [19], and PF+ which is a depth-first search based method [14, 36]. The last two  
489 algorithms are more elaborate than MC21, and the cardinality difference between two different  
490 initial matchings does not necessarily correlate with the run time.

491 The statistics of five runs with MC21 are given in Table 5. In this table, the time  
492 spent in augmentations is given in column “augment.”. The overall time to compute a  
493 maximum cardinality matching is given in column “overall”, which includes the time spent in  
494 heuristics—in case of 2OUTMC and TRUNCRW it includes the scaling time as well. The  
495 runs on `nlpkkt240` did not finish within an hour and are not presented. As seen in the  
496 table, the overall time to obtain a maximum cardinality matching is always the smallest  
497 with TRUNCRW initialization. 2OUTMC is usually competitive with the faster of KAS2  
498 and KAS1, without a clear winner. It is also interesting to note that in all graphs the worst  
499 behavior of TRUNCRW is better than the best behavior of KAS2 and KAS1 and in some  
500 cases (see `cage15` or `Channel-500`) significantly so. The same is almost true for 2OUTMC as  
501 well except for graphs `Delaunay_24` and `Hugebubbles-0020` where 2OUTMC’s worst result  
502 is only a few seconds slower than KAS1’s best result, or `cage15` versus KAS2.

name	statistics	KASi		KASi2		2OUTMC		TRUNCRW	
		augment	overall.	augment	overall.	augment	overall.	augment	overall
cage15	min.	133.85	146.52	7.42	34.47	27.29	40.75	0.22	14.07
	avg.	140.13	152.94	8.81	35.90	31.44	45.00	1.85	15.84
	max.	144.42	157.28	10.70	37.59	37.84	51.47	2.46	16.84
Channel-500	min.	64.29	74.46	9.15	29.81	12.18	22.62	0.04	6.65
	avg.	71.61	81.76	10.93	31.86	15.28	25.68	0.14	7.36
	max.	78.81	88.98	11.71	33.58	18.84	29.25	0.25	8.11
Circuit5M	min.	14.33	20.94	10.51	35.32	4.38	12.21	0.50	5.02
	avg.	15.26	22.01	13.11	38.04	5.70	13.09	0.77	5.80
	max.	16.00	22.72	14.42	39.43	6.81	13.68	1.31	7.79
Delaunay_24	min.	49.95	61.54	26.93	94.02	35.10	63.71	26.77	42.49
	avg.	54.79	66.40	29.99	98.29	36.68	64.70	31.06	46.81
	max.	61.23	72.81	32.70	104.13	40.30	68.11	34.09	49.77
Hugebub-20	min.	68.17	83.14	55.79	148.64	44.83	82.31	42.02	62.56
	avg.	73.15	88.20	58.95	156.72	50.65	88.21	44.54	65.21
	max.	75.99	91.10	61.18	166.98	54.35	91.60	47.11	67.68

■ **Table 5** Detailed run times when MC21 is used for augmentations on the graphs described in Section 4.3. The quality of heuristics are in Table 4. We have omitted graph `nlpkkt240` for which MC21 did not finish within a reasonable amount of time. For each instance we give the minimum, the average, and the maximum run time of five runs. `Hugebub-20` stands for `Hugebubbles-0020`.

503 In Table 6, we observe the behavior of the heuristics when used for initializing the PF+  
504 algorithm. The table shows the minimum, average, and maximum time over the five runs. As  
505 can be observed, TRUNCRW exhibits the best overall behavior. TRUNCRW has the fastest  
506 performance in four out of six instances, and in the remaining two instances it is very close  
507 to KASi. The largest difference between the two can be observed in `nlpkkt240` where KASi  
508 is overall almost 50 seconds slower. The total run time with KASi2 is never better than that  
509 with TRUNCRW. It roughly takes the same amount of time for PF+ to augment 2OUTMC's  
510 initial matching, as it takes for it to augment the matching of TRUNCRW. Therefore, when  
511 2OUTMC has a run time similar to TRUNCRW their overall run times are similar. In the  
512 largest of instances 2OUTMC's and TRUNCRW's performance diverge, but 2OUTMC's overall  
513 behavior is superior to KASi2 and competitive with that of KASi.

514 In Table 7, we observe the behavior of the heuristics when used for initializing the  
515 PR algorithm. The behavior of KASi in `Circuit5M` demonstrates the robustness of our  
516 approaches. The average behavior of PR initialized with KASi is 339 seconds with the  
517 maximum run time exceeding 500 seconds. In stark contrast, PR with TRUNCRW's input  
518 never needs more than 25 seconds, whereas with 2OUTMC it never surpasses 150 seconds. In  
519 the remaining instances, the proposed algorithms are competitive with KASi or even faster.

520 In summary, the effects of the proposed methods as an initialization routine are more  
521 observable with MC21 on all instances. With PF+, we see that the augmentations take  
522 less time on average with 2OUTMC and TRUNCRW, but the overall time with KASi can  
523 be sometimes better than that of TRUNCRW slightly thanks to KASi being faster. When  
524 PR is used, the augmentations take less time with KASi in three instances compared to  
525 TRUNCRW; and in four instances compared to 2OUTMC. When 2OUTMC and TRUNCRW  
526 serve better than KASi as an initialization to PR, the difference is more significant. The  
527 above results with three different algorithms demonstrate the merits of the two proposed  
528 algorithms for use as initialization routines in exact matching algorithms.

name	statistics	KAS1		KAS2		2OUTMC		TRUNCRW	
		augment.	overall	augment.	overall	augment.	overall	augment.	overall
cage15	min.	2.19	14.89	2.11	29.18	1.90	15.46	0.73	14.22
	avg.	2.51	15.33	2.59	29.67	1.97	15.53	1.16	15.15
	max.	2.98	16.15	3.16	30.43	2.01	15.69	1.55	15.63
Channel-500	min.	1.70	11.84	1.82	22.50	1.19	11.60	0.04	6.66
	avg.	1.91	12.06	2.07	23.01	1.30	11.71	0.04	7.27
	max.	2.60	12.77	2.89	23.69	1.40	11.84	0.05	7.90
Circuit5M	min.	0.63	7.20	0.45	25.28	0.45	7.34	0.48	5.01
	avg.	0.77	7.53	0.62	25.55	0.53	7.93	0.58	5.61
	max.	0.97	7.97	0.90	25.92	0.67	9.55	0.64	7.04
Delaunay_24	min.	18.47	30.06	13.88	80.75	14.24	42.05	14.20	29.92
	avg.	20.83	32.44	14.89	83.19	15.47	43.49	17.67	33.41
	max.	22.33	33.91	16.17	86.35	17.12	44.98	20.40	36.09
Hugebub-20	min.	23.09	38.09	14.99	106.41	23.27	60.75	21.97	42.54
	avg.	28.13	43.17	19.63	117.40	26.97	64.53	24.49	45.16
	max.	34.11	49.26	23.00	127.49	30.38	68.17	29.65	50.53
nlpkkt240	min.	27.01	125.69	28.19	210.27	14.91	97.26	13.76	77.87
	avg.	27.09	125.76	29.63	212.73	17.56	100.01	13.96	78.38
	max.	27.24	125.83	30.27	216.15	20.99	103.47	14.09	79.06

■ **Table 6** Detailed run times when PF+ is used for augmentations on the graphs described in Section 4.3. The quality of heuristics are in Table 4. For each instance we give the minimum, the average, and the maximum run time of five runs. **Hugebub-20** stands for **Hugebubbles-0020**.

name	statistics	KAS1		KAS2		2OUTMC		TRUNCRW	
		augment.	overall	augment.	overall	augment.	overall	augment.	overall
cage15	min.	2.15	14.85	3.63	30.52	1.19	14.67	1.10	14.03
	avg.	2.41	15.22	3.80	30.88	1.39	14.95	1.28	15.28
	max.	2.68	15.85	4.01	31.08	1.69	15.32	1.69	16.59
Channel-500	min.	1.57	11.75	2.83	23.47	1.63	12.03	0.04	6.68
	avg.	1.66	11.81	2.92	23.86	1.75	12.16	0.06	7.28
	max.	1.70	11.85	3.01	24.88	2.02	12.44	0.08	7.92
Circuit5M	min.	116.67	123.24	107.51	132.34	2.02	8.89	0.74	5.26
	avg.	332.29	339.05	235.54	260.47	37.11	44.51	5.37	10.40
	max.	559.09	566.09	378.31	403.12	139.61	148.58	18.30	24.78
Delaunay_24	min.	40.52	52.15	32.09	98.89	41.66	69.52	48.63	64.32
	avg.	45.48	57.09	36.90	105.20	46.94	74.96	52.48	68.23
	max.	52.47	64.06	43.74	110.18	53.19	81.04	58.07	73.91
Hugebub-20	min.	41.01	56.16	55.22	146.78	44.71	81.96	49.46	70.34
	avg.	47.53	62.58	58.56	156.33	51.59	89.15	53.16	73.84
	max.	52.59	67.56	61.17	166.57	58.54	96.15	54.82	75.36
nlpkkt240	min.	13.98	112.59	22.87	205.18	15.49	97.63	19.74	84.26
	avg.	14.13	112.80	24.17	207.27	17.34	99.79	28.70	93.13
	max.	14.51	113.27	25.77	211.10	19.01	101.46	47.31	112.28

■ **Table 7** Detailed run times when PR is used for augmentations on the graphs described in Section 4.3. The quality of heuristics are in Table 4. For each instance we give the minimum, the average, and the maximum run time of five runs. **Hugebub-20** stands for **Hugebubbles-0020**.



529 **5 Conclusions**

530 We have examined two randomized algorithms for the maximum cardinality matching problem  
 531 in bipartite graphs. These algorithms originally were designed for two very special classes of  
 532 bipartite graphs. We have discussed how to convert them into efficient and effective heuristics.  
 533 Our experimental results show that these approaches obtain near perfect matchings in real-life  
 534 and synthetic instances and have a near linear time run time. The two approaches are also  
 535 shown to be more robust than the state of the art heuristics used in the cardinality matching  
 536 algorithms, and are generally more useful as initialization routines.

537 Our adaptation of 2OUTMC is based on the premise that 2-out graphs sampled from  
 538 a host graph have perfect matchings, assuming that the matrix representation of the host  
 539 graph have total support. We showed evidence that this may be true and even if not, the  
 540 sampled graphs have close to perfect matchings. A proof or the disproof of such 2-out graphs  
 541 having perfect matchings is certainly welcome. Furthermore, this was the first attempt to  
 542 implement 2OUTMC, and there is room for improved performance.

543 **A Other heuristics for bipartite matching and recent work**

544 In the main text, we compared the proposed heuristics with KASI and KASI2. There are a  
 545 few other effective heuristics, which we briefly review here (see a recent survey [37]).

546 Hopcroft and Karp's original algorithm [21] proceeds in phases. At each phase, it finds  
 547 shortest augmenting paths, and augments the current matching along a maximal set of  
 548 disjoint such paths, where each phase runs in  $O(n + m)$  time. Stopping when the shortest  
 549 augmenting paths is of length  $2k + 1$  at a phase no larger than  $k$  results in an  $1 - 1/(k + 1)$   
 550 approximate matching in  $O(k(m + n))$  time in the worst case. Greedy [39] chooses a random  
 551 edge and matches the two endpoints and discards both vertices and the edges incident on  
 552 them. Modified Greedy [39] chooses a free vertex and then randomly matches it to one of  
 553 the available neighbors. MinGreedy [39] (see also Magun [31] and Langguth et al. [30] for  
 554 related algorithms) improves upon Modified Greedy by selecting a random vertex with the  
 555 minimum degree at the first step. The Greedy-like algorithms obtain maximal matchings  
 556 and therefore are  $1/2$  approximate. Slight improvements in the form of  $1/2 + \varepsilon$  are shown  
 557 for these algorithms [2, 35], but there are theoretical bounds in the same vicinity [9]. Duff et  
 558 al. [14] and Langguth et al. [25, 30] compare these algorithms for initialization in maximum  
 559 cardinality matching algorithms and suggest using KASI as initialization for general problems  
 560 especially with the push-relabel based algorithms.

561 Another class of heuristics use randomization for breaking the  $1/2$  barrier. RANKING [24]  
 562 algorithm achieves an approximation ratio of  $1 - 1/e$ , where  $e$  is the base of the natural  
 563 logarithm. The same approximation ration is also achieved by a very simple parallel algorithm  
 564 [16] whose most involved step is the application of a matrix scaling algorithm. This last  
 565 paper also proposes an algorithm based on sampling 1-out subgraphs of a general bipartite  
 566 graph (as we did in this paper) to obtain matchings of size about 0.86 times the maximum  
 567 cardinality.

568 Matching has stirred some recent interest in the theoretical computer science community,  
 569 with works focusing on parallel and distributed settings [4, 5, 11, 3] or on the fully dynamic  
 570 version [6, 8] among others. Among the recent work, a method by Assadi et al. [4] shares  
 571 similarities with the 2OUTMC algorithm. Their approach similarly sparsifies a given graph  
 572  $G$  to produce a subgraph with some approximation guarantees for the maximum cardinality  
 573 matching. A detailed experimentation with this sparsification approach will reveal useful.

## 574 **B** Further comments on 2OUTMC

575 As demonstrated in the experiments in Section 4, 2OUTMC obtains matchings of very high  
 576 cardinality. We can improve its matching quality by the following two heuristics. These two  
 577 heuristics are not used in the given experiments. We plan to improve their run time.

### 578 **B.1** Heuristic 1: Delayed tree vertex selection during Line 5

579 The ideal case at Line 5 of Algorithm 1 is to select an  $x$  such that  $x$ 's insertion as an edge  
 580 to  $H_2$  does not lead to a new tree in  $H_1$  after the deletion of the edge corresponding to the  
 581 unchecked vertex of the connected component  $Q_x$ . This is only possible if  $Q_x$  contains an  
 582 unchecked column labeled as  $\mathcal{C}$  in  $H_1$ . Otherwise, a new tree will be created in  $H_1$ , and the  
 583 algorithm will have to process it in a future step. For the first heuristic, we greedily select  
 584 an  $x$  such that, if possible, the creation of a tree in  $H_1$  is avoided.

585 We replace  $L_T$  is with two lists  $L_T^1$  and  $L_T^2$ . The lists  $L_T^1$  contains those unmarked  
 586 vertices of  $T$  whose insertion in  $H_2$  leads to a new tree;  $L_T^2$  contains all other  $L_T$  vertices  
 587 that have not been tried yet. At first, we sample  $x$  from  $L_T^2$  and see whether the components  
 588 of  $x$ 's choices in  $H_2$  have an unchecked vertex of type  $\mathcal{C}$  in  $H_1$ . If they have,  $x$  is marked and  
 589 inserted to  $H_2$ . Otherwise,  $x$  is inserted in  $L_T^1$ , and we consider another random vertex of  
 590  $L_T^2$ . If  $L_T^2$  becomes empty, we start sampling from  $L_T^1$ .

591 With the union-find data structure, this heuristic requires constant amortized time per  
 592 sample and each vertex can be sampled at most twice. Therefore the overhead associated  
 593 with this heuristic is almost linear in  $n$ .

### 594 **B.2** Heuristic 2: Online creation of the RG multigraph

595 In this heuristic, the decisions of the rows are not given as input, but are instead defined  
 596 during the course of the algorithm. Similar to the previous idea, this heuristic aims to reduce  
 597 the possibility that a tree in  $H_1$  gets created following an edge insertion into  $H_2$ .

598 More specifically, consider a vertex  $x$  randomly chosen at Line 5. In this heuristic,  $x$   
 599 has not picked its two choices yet, and we let  $x$  choose them at this point, in the way that  
 600 benefits the algorithm the most. This is done as follows. Initially, we iterate over all of  $x$ 's  
 601 neighbors in the host graph  $G$ . Let  $c$  be one of  $x$ 's neighbors and  $c^*$  be the sole unchecked  
 602 vertex in  $c$ 's connected component in  $H_2$ , or  $c^* = -1$  if no unchecked vertices exist. We  
 603 assign values to  $x$ 's neighbors to classify them. If  $c^*$  is equal to  $-1$ ,  $c$ 's value is 0. If  $c^*$  has  
 604 label  $\mathcal{F}$  or  $\mathcal{T}$  in  $H_1$ ,  $c$ 's value is 1. Otherwise,  $c$ 's value is 2. Based on these assigned values,  
 605 we partition the neighbors of  $x$  in  $G$  into three disjoint sets  $C_0$ ,  $C_1$  and  $C_2$  such that  $C_i$   
 606 contains all neighbors of  $x$  with value equal to  $i$ . Selecting columns from  $C_2$  is preferred, as  
 607 they can avoid creating a tree in  $H_1$ . Vertex  $x$  will attempt to sample first from  $C_2$ , and if  
 608 needed from  $C_1$  or  $C_0$ , with a preference for  $C_1$  over  $C_0$ . The sets  $C_0$ ,  $C_1$  and  $C_2$  are kept  
 609 implicitly, and each vertex  $x$  requires amortized  $O(d_x)$  to make its choices, where  $d_x$  is its  
 610 degree. Hence, the overhead associated with this heuristic is almost linear in  $m$ .

### 611 **B.3** Comparison with 2OUTMC

612 Here, we briefly discuss the effects that the above two heuristics have on the performance of the  
 613 2OUTMC algorithm. Since 2OUTMC obtains high quality results, the two heuristics can only  
 614 yield a relatively small improvement. When they are enabled and used with SK-5 2OUTMC  
 615 finds matchings with average quality of 0.9997 for the real-world graphs from Section 4.3 for

616 which 2OUTMC obtained matchings of quality 0.9983. This difference corresponds to about  
 617 13113 additionally matched edges, and hence signals that 13113 augmentations are avoided.

618 It is also interesting to consider the effects that these heuristics can have on cases where  
 619 2OUTMC did not deliver near-optimal matchings. As an example, we consider the synthetic  
 620 family  $\mathcal{J}$  from Section 4.2. When scaling was not enabled, 2OUTMC found matchings of  
 621 average cardinality 0.80 – 0.81% of the maximum. If however one uses the two heuristics  
 622 proposed in this section, then there is a significant improvement in performance, and 2OUTMC  
 623 finds matchings of cardinality 0.89 of the maximum.

## 624 **C Further comments on TRUNCRW**

625 We incorporated a known heuristic called look-ahead [13, 14] for speeding up the augmenting  
 626 path search in practice. All our experiments with TRUNCRW in Section 4 were with the  
 627 look-ahead approach. In this heuristic, before sampling an arbitrary row-vertex from a  
 628 column-vertex  $c$ , we check if there is a free row vertex in the adjacency list of  $c$ . If so, such a  
 629 row is returned, and the random walk terminates. The implementation of this heuristic has  
 630 a total overhead of  $O(m)$  for the whole course of the algorithm [13, 14]. We note that the  
 631 look-ahead technique trades the quality of TRUNCRW with run time. In our experiments, the  
 632 look-ahead heuristic reduced the run time significantly; it interferes with the randomization  
 633 though.

634 We can easily apply TRUNCRW to bipartite graphs with different number of vertices  
 635 in each side. This is based on the fact that we can scale a rectangular  $n_1 \times n_2$  matrix (say  
 636  $n_1 \geq n_2$ ) so that all columns have sum of 1, and all rows have equal sum of  $n_2/n_1$ , if there  
 637 is matching covering all columns, and all entries can be put in such a matching. Then, all  
 638 components of TRUNCRW work without any change.

639 If there is no total support, then Sinkhorn–Knopp works in such a way that the entries  
 640 that cannot put into a perfect matching tend to zero. This is helpful in TRUNCRW’s context,  
 641 as the corresponding edges will not likely be selected in a random walk. If there is no perfect  
 642 matching, then little is known about scaling. It is our experience that the Sinkhorn–Knopp  
 643 iterations tend to zero out entries that cannot be put into a maximum cardinality matching.  
 644 Therefore, in this case again, scaling, random selection, and truncation should help. We  
 645 present some experiments to support this observation and leave the question of showing this  
 646 theoretically as an open problem.

647 We experimented with bipartite graphs without total support which correspond to square  
 648 ( $10000 \times 10000$ ) and rectangular matrices ( $12000 \times 10000$ ) with a uniform nonzero distribution.  
 649 These matrices are generated with `sprand` command of Matlab and have about  $d \times 10000$   
 650 nonzeros for  $d = 2, 3, 4, 5$ . The matrix representation of the bipartite graphs were scaled with  
 651 10 iterations of SK. For each  $d$ , we created five random matrices and ran TRUNCRW on the  
 652 corresponding five instances. We report the worst quality of the five instances in Table 8. As  
 653 seen in this table, TRUNCRW works just fine for this case. We did not report in the table  
 654 but with increased SK iterations, the results improve, which is in accordance with earlier  
 655 work [16].

### 656 **C.1 Engineering TRUNCRW**

657 The experiments here are on real-life instances from Subsection 4.3 and with SK-5.

658 Recall that TRUNCRW tries to find an augmenting path starting from a column vertex a  
 659 certain number of times before giving up and moving to the next column vertex. When we  
 660 allowed TRUNCRW just a single attempt, it was unable to find a perfect matching in any

$d$	10000 × 10000		12000 × 10000	
	sprank	TRUNCRW	sprank	TRUNCRW
2	7787	0.9888	8724	0.9919
3	9266	0.9697	9667	0.9958
4	9761	0.9828	9899	0.9995
5	9918	0.9922	9973	1.0000

■ **Table 8** The quality of TRUNCRW on bipartite graphs without perfect matchings.

661 of the cases, and its average matching quality was 0.9984. When we allowed five attempts,  
 662 TRUNCRW found a perfect matching for 13 graphs, and its average matching quality was  
 663 0.9999. With 10 attempts, it managed to find a perfect matching in 5 additional graphs.  
 664 This verifies that allowing more attempts indeed improves the performance of the algorithm.  
 665 The drawback, however, was the increased run time, which we did not think worth. That is  
 666 why our implementation of TRUNCRW starts a random walk from a vertex only once.

667 We also test the effects of the look-ahead mechanism. Let us define the walk efficiency of  
 668 TRUNCRW as the ratio of the cardinality of the matching found to the total length of the  
 669 random walks. The higher this ratio, the more useful the random walks are. We evaluate  
 670 the walk efficiency on a set of seven instances (real-life instances having at most 10000000  
 671 edges). We test both with and without scaling and report the results of the 14 tests. In 13  
 672 cases, the look-ahead mechanism improved the walk efficiency. The geometric mean (of 14  
 673 cases) of the ratios of walk efficiencies with look-ahead to that of without was 1.37. In the  
 674 case where the look-ahead did not help (ratio was 0.71 in an instance named Hamr1e3), the  
 675 maximum deviation of a row or column sum from one after SK-5 was 0.28, which is high.  
 676 We conclude that the look-ahead mechanism is very helpful.

677 Finally we test the effects that the length of the augmenting walk has on TRUNCRW. We  
 678 doubled the allowed length of a random walk to  $4(4 + 2n/(n - j))$ . On average, the matching  
 679 quality rose from 0.9984 to 0.9998. This modification was not able to find a perfect matching  
 680 in any of the 39 instances. This led to an increase in the run time, which we deemed too  
 681 large. We therefore keep  $2(4 + 2n/(n - j))$  as the truncation length.

## 682 **D Reducing bipartite graph matching to matching on 2-out graphs**

683 Here, we prove our claim in Section 3.1 that bipartite matching can be reduced to matching  
 684 on a 2-out bipartite graph. Let  $G = (V_G, E_G)$ , with be a graph with minimum degree at least  
 685 two. If  $G$ 's minimum degree is one, we can apply the first deterministic rule of Karp–Sipser  
 686 to match degree-1 vertices with their neighbors and consider as  $G$  the resulting graph.

687 We produce a new graph  $G'$  from  $G$  in the following way. For any edge  $e = (a, b) \in E$   
 688 we add edges  $e' = (a, a_e)$ ,  $e'' = (a_e, b_e)$ , and  $e''' = (b_e, b)$  to  $G'$ . We hence introduce two  
 689 new vertices  $a_e, b_e$  s.t  $d_{G'}(a_e) = d_{G'}(b_e) = 2$  for each edge  $e \in E_G$ . The degree of nodes in  $V_G$   
 690 remains unchanged in  $G'$ .

691 ► **Lemma 2.** *Let  $H$  be a random 2-out subgraph  $G'$ . Then  $H = G'$ .*

692 **Proof.** The added vertices  $a_e, b_e$  have degree two and will select both neighbors, hence no  
 693 edge will remain unpicked. ◀

694 In what follows, we refer to the second reduction rule of Karp–Sipser which merges the  
 695 neighbors of a degree-2 vertex, which is then discarded, as a degree-2 reduction.

696 ► **Lemma 3.** *It is possible to obtain  $G$  by doing only degree-2 reductions on  $G'$ .*

697 **Proof.** Let  $a_e$  be a vertex of  $G'$ , introduced due to the edge  $e = (a, b)$ . Since  $d_{G'}(a_e) = 2$  we  
 698 can apply a degree-2 reduction which will merge  $a$  with  $b_e$  to create a single node  $ab_e$ . As a  
 699 consequence of this merge, the edge  $(ab_e, b)$  will be created and edges  $(a, a_e), (a_e, b_e), (b_e, b)$   
 700 will be erased. We simply relabel  $ab_e$  to  $a$  again. The proof then follows similarly by applying  
 701 degree-2 reduction for all  $a_e$  corresponding to  $e \in E$  until we obtain  $G$ . ◀

702 Now we show that maximum matchings in  $G'$  are related to those on  $G$  and vice versa.

703 ▶ **Lemma 4.** *Any maximum cardinality matching  $M'$  on  $G'$  corresponds to a maximum  
 704 cardinality matching  $M$  on  $G$ .*

705 **Proof.** Let  $M'$  be a maximum cardinality matching on  $G'$ . A matching  $M$  for  $G$  can be  
 706 generated in the following way: If both  $(a, a_e)$  and  $(b_e, b)$  appear in  $M'$ ,  $e$  is added to  $M$ .  
 707 Hence it suffices to show that any maximum cardinality matching  $M'$  in  $G'$  necessarily  
 708 contains  $|M|$  pair of matched edges  $(a, a_e)$  and  $(b, b_e)$ .

709 First, we have that  $|M'| = |E_G| + |M|$ . To see this, note that per Lemma 2 we perform  
 710  $|E_G|$  degree-2 reductions, and result in  $G$ . Each of this reductions corresponds with a matched  
 711 edge in  $M'$ . Then, we only need to find the maximum cardinality on  $G$  which is  $|M|$ .

Let  $S_a$  contain all indices  $e$  such that  $(a, a_e)$  is in  $M'$  and  $(b_e, b)$  is not in  $M'$ . Set  $S_b$  is  
 defined similarly. Set  $S_\emptyset$  contains all indices  $e$  such that  $(a_e, b_e)$  appears in  $M'$ . Finally,  $S_{ab}$   
 contains all indices  $e$  such that  $(a, a_e)$  and  $(b, b_e)$  are matched together in  $M'$ . Then, since  
 $M'$  is a maximum cardinality matching we have

$$|S_a| + |S_b| + |S_\emptyset| + 2 \cdot |S_{ab}| = |E_G| + |M| .$$

712 This is true because of the fact that for each edge  $e$  exactly one matched edge appears in  $M'$   
 713 in case  $e \in S_a \cup S_b \cup S_\emptyset$  and two edges are added if  $e \in S_{ab}$ .

714 However,  $|S_a| + |S_b| + |S_\emptyset| + |S_{ab}| = |E_G|$ , since each edge  $e$  must appear in one of those  
 715 sets and there exist exactly  $|E_G|$  of them.

716 Hence,  $|S_{ab}| = |M|$  necessarily. As they define a matching in  $G$  and their cardinality is  
 717  $|M|$ , the matching is maximum. ◀

718

719 Using the above lemma, we can prove Theorem 5 below.

720 ▶ **Theorem 5.** *Assume there is an algorithm ALG working in  $O(f(n, m))$  time for finding a  
 721 maximum cardinality matching in a 2-out graph. Then we can find a maximum cardinality  
 722 matching in  $O(f(m, m))$  time for any given graph.*

723 **Proof.** Let  $G$  be any bipartite graph without degree-1 vertices and  $m = |E_G|$ . In  $O(m)$  time  
 724 we generate  $G'$ . By Lemma 2, the 2-out subgraph of  $G'$  corresponds to  $G'$  itself. In addition  
 725  $|E_{G'}|, |V_{G'}| \in O(m)$ . Using ALG, we can find a maximum cardinality  $M'$  for  $G'$  in  $O(f(m, m))$   
 726 time. By Lemma 4 then, we can convert  $M'$  to a maximum cardinality matching  $M$  for  $G$  in  
 727  $O(m)$  time. ◀

728 As a byproduct of Lemma 4, we observe that the transformation of  $G$  to  $G'$  also eliminates  
 729 the need to perform SK as a preprocessing step. We briefly experimented with this method  
 730 on the real-world graphs of Section 4.3. For each graph  $G$  of the test-set, we generated  
 731 its extension  $G'$  and executed the 2OUTMC algorithm on 2-out graphs sampled from  $G'$ ,  
 732 with uniform selections. The behavior of 2OUTMC was similar with that of the previous  
 733 experiments. It was not able to obtain a perfect matching in  $G'$  (and consequently  $G$ ), but  
 734 it always returned near-optimal matchings of quality over 0.99. These matchings, when  
 735 converted into matchings of  $G$  (following the idea in Lemma 4) yielded also near-optimal  
 736 matchings with quality over 0.99.

737 ——— **References** ———

- 738 1 Z. Allen-Zhu, Y. Li, R. Mendes de Oliveira, and A. Wigderson. Much faster algorithms for  
739 matrix scaling. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS*,  
740 pages 890–901, Berkeley, CA, USA, October 2017.
- 741 2 J. Aronson, M. Dyer, A. Frieze, and S. Suen. Randomized greedy matching II. *Random*  
742 *Structures & Algorithms*, 6(1):55–73, 1995.
- 743 3 S. Assadi, M. Bateni, A. Bernstein, V. Mirrokni, and C. Stein. Coresets meet edcs: algorithms  
744 for matching and vertex cover on massive graphs. In *Proceedings of the Thirtieth Annual*  
745 *ACM-SIAM Symposium on Discrete Algorithms*, pages 1616–1635. SIAM, 2019.
- 746 4 S. Assadi and A. Bernstein. Towards a unified theory of sparsification for matching problems.  
747 *arXiv preprint arXiv:1811.02009*, 2018.
- 748 5 S. Behnezhad, S. Brandt, M. Derakhshan, M. Fischer, M. Hajiaghayi, R.M. Karp, and J. Uitto.  
749 Massively parallel computation of matching and mis in sparse graphs. In *Proceedings of the*  
750 *2019 ACM Symposium on Principles of Distributed Computing*, pages 481–490, 2019.
- 751 6 S. Behnezhad, J. Łącki, and V. Mirrokni. Fully dynamic matching: Beating 2-approximation in  
752  $\delta^\epsilon$  update time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete*  
753 *Algorithms*, pages 2492–2508. SIAM, 2020.
- 754 7 C. Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of*  
755 *the USA*, 43:842–844, 1957.
- 756 8 A. Bernstein and C. Stein. Fully dynamic matching in bipartite graphs. In *International*  
757 *Colloquium on Automata, Languages, and Programming*, pages 167–179. Springer, 2015.
- 758 9 B. Besser and M. Poloczek. Greedy matching: Guarantees and limitations. *Algorithmica*,  
759 77(1):201–234, 2017.
- 760 10 M. B. Cohen, A. Madry, D. Tsipras, and A. Vladu. Matrix scaling and balancing via box  
761 constrained newton’s method and interior point methods. In *58th IEEE Annual Symposium*  
762 *on Foundations of Computer Science, FOCS*, pages 902–913, Berkeley, CA, USA, October  
763 2017.
- 764 11 A. Czumaj, J. Łącki, A. Madry, S. Mitrovic, K. Onak, and P. Sankowski. Round compression  
765 for parallel matching algorithms. In *Proceedings of the 50th Annual ACM SIGACT Symposium*  
766 *on Theory of Computing*, pages 471–484. Association for Computing Machinery, 2018.
- 767 12 T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions*  
768 *on Mathematical Software*, 38(1):1:1–1:25, 2011.
- 769 13 I. S. Duff. On algorithms for obtaining a maximum transversal. *ACM Transactions on*  
770 *Mathematical Software*, 7(3):315–330, 1981.
- 771 14 I. S. Duff, K. Kaya, and B. Uçar. Design, implementation, and analysis of maximum transversal  
772 algorithms. *ACM Transactions on Mathematical Software*, 38:13:1–13:31, 2011.
- 773 15 F. Dufossé, K. Kaya, I. Panagiotas, and B. Uçar. Approximation algorithms for maximum  
774 matchings in undirected graphs. In *2018 Proceedings of the Seventh SIAM Workshop on*  
775 *Combinatorial Scientific Computing*, pages 56–65, 2018.
- 776 16 F. Dufossé, K. Kaya, and B. Uçar. Two approximation algorithms for bipartite matching on  
777 multicore architectures. *Journal of Parallel and Distributed Computing*, 85:62–78, 2015.
- 778 17 A. Frieze and T. Johansson. On random  $k$ -out subgraphs of large graphs. *Random Structures*  
779 *& Algorithms*, 50(2):143–157, 2017.
- 780 18 A. Goel, M. Kapralov, and S. Khanna. Perfect matchings in  $O(n \log n)$  time in regular bipartite  
781 graphs. *SIAM Journal on Computing*, 42(3):1392–1404, 2013.
- 782 19 A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*,  
783 35(4):921–940, 1988.
- 784 20 T. Hagerup, K. Mehlhorn, and J. I. Munro. Maintaining discrete probability distributions  
785 optimally. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *20th International Colloquium*  
786 *on Automata, Languages, and Programming (ICALP)*, pages 253–264, Berlin, Heidelberg, 1993.  
787 Springer Berlin Heidelberg.

- 788 21 J. E. Hopcroft and R. M. Karp. An  $n^{5/2}$  algorithm for maximum matchings in bipartite  
789 graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.
- 790 22 R. M. Karp, A. H. G. Rinnooy Kan, and R. V. Vohra. Average case analysis of a heuristic for  
791 the assignment problem. *Mathematics of Operations Research*, 19(3):513–522, 1994.
- 792 23 R. M. Karp and M. Sipser. Maximum matching in sparse random graphs. In *22nd Annual*  
793 *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–375, Los Alamitos,  
794 CA, USA, 1981. IEEE Computer Society.
- 795 24 R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite  
796 matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*,  
797 STOC '90, pages 352–358, New York, NY, USA, 1990. ACM.
- 798 25 K. Kaya, J. Langguth, F. Manne, and B. Uçar. Push-relabel based algorithms for the maximum  
799 transversal problem. *Computers & Operations Research*, 40(5):1266–1275, 2013.
- 800 26 K. Kaya, J. Langguth, I. Panagiotas, and B. Uçar. Karp–Sipser based kernels for bipartite  
801 graph matching. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*,  
802 pages 134–145, Salt Lake City, Utah, US, January 2020.
- 803 27 P. A. Knight. The Sinkhorn–Knopp algorithm: Convergence and applications. *SIAM Journal*  
804 *on Matrix Analysis and Applications*, 30(1):261–275, 2008.
- 805 28 P. A. Knight and D. Ruiz. A fast algorithm for matrix balancing. *IMA Journal of Numerical*  
806 *Analysis*, 33(3):1029–1047, 2013.
- 807 29 V. Korenwein, A. Nichterlein, R. Niedermeier, and P. Zschoche. Data reduction for maximum  
808 matching on real-world graphs: Theory and experiments. In *26th Annual European Symposium*  
809 *on Algorithms (ESA 2018)*, volume 112, pages 53:1–53:13, Dagstuhl, Germany, 2018.
- 810 30 J. Langguth, F. Manne, and P. Sanders. Heuristic initialization for bipartite matching problems.  
811 *Journal of Experimental Algorithmics (JEA)*, 15:1–22, 2010.
- 812 31 J. Magun. Greedy matching algorithms, an experimental study. *Journal of Experimental*  
813 *Algorithmics*, 3:6, 1998.
- 814 32 Y. Matias, J. S. Vitter, and W.-C. Ni. Dynamic generation of discrete random variates. *Theory*  
815 *of Computing Systems*, 36(4):329–358, 2003.
- 816 33 N. McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM*  
817 *Transactions on Networking*, 7:188–201, 1999.
- 818 34 M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and*  
819 *probabilistic analysis*. Cambridge University Press, 1st edition, 2005.
- 820 35 M. Poloczek and M. Szegedy. Randomized greedy algorithms for the maximum matching  
821 problem with new analysis. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd*  
822 *Annual Symposium on*, pages 708–717, 2012.
- 823 36 A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM*  
824 *Transactions on Mathematical Software*, 16(4):303–324, 1990.
- 825 37 A. Pothen, S. M. Ferdous, and F. Manne. Approximation algorithms in combinatorial scientific  
826 computing. *Acta Numerica*, 28:541–633, 2019.
- 827 38 R. Sinkhorn and P. Knopp. Concerning nonnegative matrices and doubly stochastic matrices.  
828 *Pacific Journal of Mathematics*, 21(2):343–348, 1967.
- 829 39 G. Tinhofer. A probabilistic analysis of some greedy cardinality matching algorithms. *Annals*  
830 *of Operations Research*, 1(3):239–254, 1984.
- 831 40 D. Walkup. Matchings in random regular bipartite digraphs. *Discrete Mathematics*, 31(1):59–64,  
832 1980.