Engineering Fast Almost Optimal Algorithms for Bipartite Graph Matching

Ioannis Panagiotas
ENS Lyon, France
ioannis.panagiotas@ens-lyon.fr

Bora Uçar
CNRS and LIP, ENS Lyon, France
bora.ucar@ens-lyon.fr

Abstract

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

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

Keywords and phrases bipartite graphs, matching, randomized algorithm

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.76

Supplementary Material `https://gitlab.inria.fr/bora-ucar/fast-matching`

1 Introduction

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

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

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

2 Background and notation

Let $G = (R \cup C, E)$ be a bipartite graph, where R and C are two disjoint set of vertices, and E is the set of edges. The bipartite graph G can be represented with a matrix $\mathbf{A}_G$. The vertex $r_i \in R$ corresponds to the ith row, and the vertex $c_j \in C$ corresponds to the jth 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* and to those of C as *columns* from this point on, and use $\mathbf{A}$ to refer to $\mathbf{A}_G$.

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

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

A k-out subgraph G_k of a host graph G is defined by allowing each vertex in G to randomly select uniformly k of its neighbors, and the union of all selections forms the edge set of G_k. Walkup [40] shows that in the pure random k-out setting, where the host graph is the complete bipartite graph, the resulting G_k has a perfect matching with high probability for $k \geq 2$. We do not know any general result about properties of G_2 sampled from any arbitrary host graph. Frieze and Johansson [17] investigate some other properties of G_ks on host graphs where the minimum degree of a vertex is at least $n/2$. Dufossé et al. [16] propose using the doubly stochastic matrix $\mathbf{A_S}$ (scaled version of the matrix representation) for sampling and show an approximation result for G_1, when $\mathbf{A}$ has total support. We give some experiments in which G_2s generated using the same probabilities have perfect matchings in majority of the cases.

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

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

3 Two heuristics

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

3.1 2outMC: Monte Carlo on 2-out graphs

3.1.1 Description of the algorithm

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

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

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

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

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

Task B can be efficiently done in amortized near linear time (over the course of the algorithm) by using a union-find data-structure and keeping the identity of the single unchecked vertex in a component of H_2 at the root of the component. For Task A, Karp

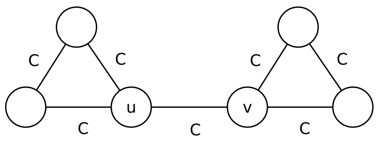


Figure 1 Algorithm 1 does not recognize new trees, if another edge is deleted after (u, v).

et al. propose the following. In the beginning, the edges of CG are labeled as $\mathcal{F}$, if their deletion creates a tree; $\mathcal{T}$, if they belong to a tree component; and $\mathcal{C}$ otherwise. Let c-degree of a vertex v be the number of $\mathcal{C}$ edges incident on v. During deleting the edge (u, v) from H_1, one of the following is performed depending on the label of (u, v).

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

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

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

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

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

Case 1-continuation is therefore as follows:

- Once there are no vertices with c-degree equal to 1, take the last vertex v whose c-degree was reduced. If c-degree$[v] = 0$, then relabel all edges in vs component from $\mathcal{F}$ to $\mathcal{T}$.

This addition has overall $O(n)$ cost, because each edge can change label at most twice.

3.1.2 Conversion to an efficient general heuristic

Algorithm 1 works well when the random 2-out graph is sampled from $K_{n,n}$. However, in the case of an arbitrary host graph, the underlying theory is not shown to hold, and the algorithm can make erroneous decisions. Here we discuss how to turn Algorithm 1 into a general heuristic. Apart from the aim of obtaining a practical heuristic for bipartite matching, there is another reason to investigate the matching problem in 2-out bipartite graphs. We show in Appendix D that an $O(f(n,m))$ time algorithm to find a maximum cardinality matching in a 2-out bipartite graph can be used to find a maximum cardinality matching in any bipartite graph with m edges in $O(f(m,m))$ time, where f is a function on the number of vertices n and edges m. Such a reduction is important because it shows that an algorithm for finding maximum cardinality matchings in 2-out graphs with similar complexity to 2OUTMC can be used to obtain an $O(m \log m)$ algorithm for matchings in general bipartite graphs.

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

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

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

3.2 TruncRW: Truncated random walk with nonuniform sampling

3.2.1 Description of the algorithm for regular bipartite graphs

Goel et al. [18] propose a randomized algorithm (of the Las Vegas type) that finds a perfect matching in a d-regular bipartite graph with n vertices in each side in $O(n \log n)$ time in expectation. This algorithm starts a random walk from a randomly chosen free column-vertex.

At a column vertex c, the algorithm selects uniformly at random one of the row-vertices that are not matched to c, and goes to the chosen row vertex r. If r is free, then an augmenting path is obtained by removing possible loops from the walk. If r is matched, then the random walk goes to the mate of r. Goel et al. show that the total length of the random walks is $O(n \log n)$ in expectation, and thus the algorithm obtains a perfect matching in the stated time [18, Theorem 4]. They also show that one can obtain a Monte Carlo-type algorithm by truncating the random walks. The expected length of an augmenting path with respect to a given matching of cardinality j is $2(4 + 2n/(n - j))$, and the random walks could be truncated at this length to obtain near optimal matchings in $O(n \log n)$ time.

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

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

3.2.2 Conversion to an efficient general heuristic

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

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

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

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

Figure 2 The matrix $\mathbf{A}$ associated with a 4×4 Hessenberg matrix, the scaling matrices $\mathbf{D_R}$ and $\mathbf{D_C}$, and the resulting doubly stochastic matrix $\mathbf{A_S} = \mathbf{D_R A D_C}$. In general, $\mathbf{A_S}(n, 1) = 1/2^{n-1}$.

- $\mathsf{adj}_c[1, \ldots, d_c]$: an array keeping the neighbors of c.
- $\mathsf{wghts}_c[1, \ldots, d_c]$: the weight of the edges incident on c. This array is parallel to the first one so that the weight of the edge $(c, \mathsf{adj}_c[i])$ is $\mathsf{wghts}_c[i]$.
- $\mathsf{medge}[c]$: the position of the mate of c in the array adj_c, or -1 if c is not matched.

At the beginning, we compute the prefix sum of $\mathsf{wghts}_c[1, \ldots, d_c]$. After this operation, the total weight of the edges incident on c is $\mathsf{wghts}_c[d_c]$, and the weight of the edge $(c, \mathsf{mate}[c])$ is $\mathsf{wghts}_c[\mathsf{medge}[c]] - \mathsf{wghts}_c[\mathsf{medge}[c] - 1]$, assuming that $\mathsf{wghts}_c[0]$ signifies zero.

Given the prefix sums in $\mathsf{wghts}_c[1, \ldots, d_c]$, the position of the mate of c at $\mathsf{medge}[c]$, we can choose a random neighbor (which is not equal to $\mathsf{mate}[c]$) as shown in Algorithm 2. We use a binary search function, binSearch, which takes an array, the array's start and end positions, a target value, and returns the smallest index of an array element which is larger than the given value with binary search (we skip the details of this search function). At Line 5, since c does not have a mate, we search in the whole list. At Line 8, since the prefix sum just before $\mathsf{medge}[c]$ is larger than the target value, we search in the first part of wghts_c until the current mate located at $\mathsf{medge}[c]$. At Line 10, we search on the right of $\mathsf{medge}[c]$, by a modified target value. This last part is the gist of the algorithm's efficiency as it avoids updating the prefix sums when the mate changes.

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

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

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

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

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

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

4 Experiments

We implemented 2outMC and TruncRW in C/C++, and the codes are accessible from `https://gitlab.inria.fr/bora-ucar/fast-matching`. The codes, all are sequential, were compiled with "-O3" and run on a machine with 2 x Intel Xeon CPU Gold 6136 CPUs and 187 GB RAM. We evaluate 2outMC and TruncRW both on real-life and synthetic bipartite graphs with equal number of vertices in each side. We compared the two algorithms against KaSi, the widely used version of Karp–Sipser which applies degree-1 reduction (own implementation), and KaSi2, the original version of Karp–Sipser with both reduction rules. We use a publicly available implementation of KaSi2 (`https://gitlab.inria.fr/bora-ucar/karp--sipser-reduction`) which is the fastest of recent implementations [26, 29]. We note that there are other heuristics (a short summary and further references are in Appendix A) which deliver very good results in practice. For most of these heuristics, especially for those based on vertex degree, there are known worst case upper bounds close to 1/2. We therefore restrict the focus on KaSi and KaSi2, which are efficient and very effective in practice [14, 25, 30]. We also investigated if random 2-out bipartite graphs of a general host graph have perfect matchings if rows and columns select neighbors with the probabilities in the scaled matrix representation. The quality of a matching refers to the ratio of the cardinality of the matching to the maximum cardinality of a matching in a given graph. The practical version of Sinkhorn-Knopp is referred to as SK-t, where t is the number of allowed iterations. All run times are reported in seconds.

Table 1 We divide the real-life graphs into five groups. The ith 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_1 and M_2 has a perfect matching and the largest difference from the maximum cardinality of a matching.

$\frac{m}{n}$	[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_1	0	223	0	8	1	20	0	2	1	0
Model M_2	27	0	3	3	1	10	0	1	0	1

4.1 Investigation of perfect matchings in 2-out graphs

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

We consider two different models to create G_2. In the model M_1, row choices are independent of the column choices. Under this model, a row and a column can select each other resulting in parallel edges – only one of them is kept. The model M_2 tries to avoid parallel edges. In this model, all columns perform their selections. Then, each row r attempts to randomly choose two columns, only from those that did not select r. These selections again are based on the scaled matrix. In this model, parallel edges can arise (and be discarded) only when a vertex v is connected in the 2-out graph with all of its neighbors in G, because it is impossible for v to select otherwise. We experimented three times with each real-life graph. M_i's result is the maximum of those three experiments. In each test, we first created the choices of all columns. Then we allowed the two models to generate the choices of the rows accordingly.

The results are shown in Table 1 for the 39 real-life graphs and are with SK-5. As seen in this table, the random G_2 graphs generated with the model M_1 have near perfect matchings, but they do not contain perfect matchings in most cases. In contrast, the random G_2 graphs generated by M_2 in many cases contain a perfect matching. In only a few graphs this does not hold true, and in these cases the deficiency is no more than 10.

4.2 On synthetic graphs

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

As seen in Table 2, KaSi and KaSi2 have more and more difficulty with increasing h. The matching quality drops over 30% between $h = 2$ and $h = 512$ for KaSi and almost 40% for KaSi2. On the contrary, 2outMC and TruncRW both obtain a near perfect matching, with SK-5. Even though the matrices associated with the graphs of $\mathcal{I}$ lack total support, SK-5 sufficed to obtain near optimal matchings. We notice the effect of scaling: if vertices select without scaling (Uniform), the matching quality reduces. This is particularly true for 2outMC, which exhibits the worst overall performance with uniform selection. Family

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.

			2outMC		TruncRW	
h	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 3 Average quality of the matchings found by the algorithms on graphs from the synthetic family $\mathcal{J}$ for $n \in \{10000, 20000, 30000\}$.

	KaSi	KaSi2	2outMC			TruncRW		
n	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

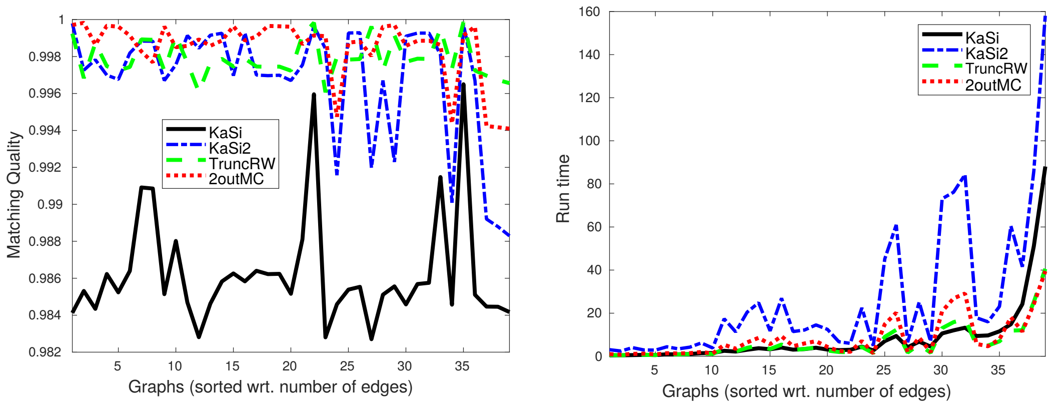
$\mathcal{I}$ shows the importance of scaling, and more importantly highlights the robustness of the proposed methods. An adversary can create graphs which make degree-based randomized approaches lose quality – some of those heuristics are briefly mentioned in Appendix A, and the full details including negative results on KaSi2 can be found elsewhere [9]. On the other hand, the use of scaling helps to avoid such cases for 2outMC and TruncRW.

We now discuss another synthetic family of graphs $\mathcal{J}$ in which the proposed approaches obtain matchings of much higher quality than KaSi and KaSi2. A bipartite graph with n vertices per side belonging to $\mathcal{J}$ contains the following edges: (r_i, c_j) for all $i \leq j$; (r_2, c_1), (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 Karp–Sipser-based heuristics because only few of the edges participate in a perfect matching, the deterministic rules do not apply, and hence they resort to multiple suboptimal random decisions. Likewise, due to the large number of entries without support in the matrix representation, Sinkhorn–Knopp will take many iterations to properly scale the matrix.

In Table 3, we give results of the algorithms for a few graphs from this family. In the table, we also show the effects of scaling on 2outMC and TruncRW by showing results without scaling (under column "uniform", in which a column vertex chooses a neighbor uniformly at random), with SK-5, and with SK-20. As can be seen, despite the lack of total support, both 2outMC and TruncRW obtain matchings whose cardinality is more than 0.92 of the maximum, when SK-5 or SK-20 is used. TruncRW in particular is nearly optimal. These results are always better than that of KaSi and KaSi2, with the difference in matching quality being about 20–25% for the former, and 10–15% for the latter. With increased iterations of Sinkhorn–Knopp, 2outMC increases the cardinality of its matchings by 3%. If we do not use scaling ("uniform"), while there's no noticeable effect on TruncRW's matchings, 2outMC matchings decrease by roughly 10%. Even so, its results remain better than KaSi's and on par with those of KaSi2.

4.3 On real-life graphs

We compared TruncRW and 2outMC with KaSi and KaSi2 on all 39 real-life graphs from Section 4.1. Figure 3a and Figure 3b present the high level picture. For the experiments, we did not permute the matrices randomly, which generally increases the experimentation time.



(a) Quality results. **(b)** Run time results.

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

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

Figure 3a also shows the matching quality of KaSi2 and KaSi. KaSi obtains matchings of quality 0.9862 on average, with always smaller cardinality than TruncRW and 2outMC. KaSi2 fares better and its average quality is 0.9968. Even so, in the majority of cases, it obtains matchings that are inferior quality-wise to both TruncRW and 2outMC.

While all algorithms obtain matchings of high quality, the absolute different is remarkable in some cases. For example, the largest difference observed between the matching cardinalities obtained by 2outMC and KaSi was 346577, in favor of 2outMC.

Figure 3b shows the run time of all examined heuristics, where the graphs are again indexed in nondecreasing number of edges. KaSi is in general the fastest of these four algorithms when there are not too many edges. TruncRW and 2outMC are close run-time wise to KaSi and in some instances faster than it. This is especially true in instances with many edges because KaSi depends more on m. KaSi2 has the slowest performance overall.

For a detailed study, we show results on the five largest graphs from the mentioned dataset and `Circuit5M`, which was identified as a challenging instance in earlier work [25]. Degree-1 vertices from `Circuit5M` are removed by applying Rule-1 of KaSi2 as a preprocessing step – this is without loss of generality of the heuristics. For each graph we relabeled its row-vertices randomly and executed five tests with each algorithm.

Table 4 shows the matching quality and the run time of the four heuristics. 2outMC and TruncRW used SK-3 for this set of experiments for speed. For each graph, we give the minimum, maximum, and averages over five runs. As already discussed, all heuristics obtain high quality matchings. On a closer look, we see that TruncRW, on average, matched 158410 more edges than KaSi, and 50847 more edges than KaSi2. Similarly 2outMC matched 139220 more edges than KaSi on average, and 31652 more edges than KaSi2. Interestingly, on graph `Channel-500` TruncRW was able to find the maximum matching.

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`.

			KaSi		KaSi2		SK-3	2outMC		TruncRW	
name	n	statistics	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

Concerning run time, as KaSi is a linear time heuristic it is expected to be the fastest. Surprisingly, TruncRW even with the scaling time added is faster than KaSi in three instances. This is due to the fact that each iteration of the scaling algorithm takes linear time with small constants. As an algorithm on its own (without scaling time), TruncRW becomes the fastest one, thanks to its run time not depending on m after the initialization. 2outMC, though slower, also exhibits good behavior, except in `nlpkkt240`. KaSi2 has the worst run time overall. Its initialization takes more time, and its implementation is more involved. SK-3 is fast except for `nlpkkt240` where it requires about 30 seconds. The reason that SK-3 requires 30 seconds for this particular graph is due to the random permutation of its rows, which is not cache-friendly (if SK-3 is run on `nlpkkt240` using the initial ordering of rows, it finishes in less than 10 seconds). In the other cases and despite the large size of the graphs, scaling finishes in less than seven seconds. Table 4 additionally shows that TruncRW and 2outMC's run time performance does not seem to be affected by their random decisions. The largest difference between the result of the minimum, and the maximum run is no more than two seconds for both of these algorithms.

Combined with the results in the previous section, we conclude thus that (i) 2outMC and TruncRW always obtain near perfect matchings, while KaSi and KaSi2 are not as robust; (ii) 2outMC and TruncRW are nearly as fast as the linear time algorithm KaSi, and are much faster than KaSi2.

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

The statistics of five runs with MC21 are given in Table 5. In this table, the time spent in augmentations is given in column "augment.". The overall time to compute a maximum cardinality matching is given in column "overall', which includes the time spent in heuristics – in case of 2outMC and TruncRW it includes the scaling time as well. The runs on `nlpkkt240` did not finish within an hour and are not presented. As seen in the table, the overall time to obtain a maximum cardinality matching is always the smallest with TruncRW initialization. 2outMC is usually competitive with the faster of KaSi2 and KaSi, without a clear winner. It is also interesting to note that in all graphs the worst behavior of TruncRW is better than the best behavior of KaSi2 and KaSi and in some cases (see `cage15` or `Channel-500`) significantly so. The same is almost true for 2outMC as well except for graphs `Delaunay_24` and `Hugebbubles-0020` where 2outMC's worst result is only a few seconds slower than KaSi's best result, or `cage15` versus KaSi2.

In Table 6, we observe the behavior of the heuristics when used for initializing the PF+ algorithm. The table shows the minimum, average, and maximum time over the five runs. As can be observed, TruncRW exhibits the best overall behavior. TruncRW has the fastest performance in four out of six instances, and in the remaining two instances it is very close to KaSi. The largest difference between the two can be observed in `nlpkkt240` where KaSi is overall almost 50 seconds slower. The total run time with KaSi2 is never better than that with TruncRW. It roughly takes the same amount of time for PF+ to augment 2outMC's initial matching, as it takes for it to augment the matching of TruncRW. Therefore, when 2outMC has a run time similar to TruncRW their overall run times are similar. In the largest of instances 2outMC's and TruncRW's performance diverge, but 2outMC's overall behavior is superior to KaSi2 and competitive with that of KaSi.

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

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

5 Conclusions

We have examined two randomized algorithms for the maximum cardinality matching problem in bipartite graphs. These algorithms originally were designed for two very special classes of bipartite graphs. We have discussed how to convert them into efficient and effective heuristics.

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`.

		KaSi		KaSi2		2outMC		TruncRW	
name	statistics	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 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`.

		KaSi		KaSi2		2outMC		TruncRW	
name	statistics	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 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`.

		KaSi		KaSi2		2outMC		TruncRW	
name	statistics	augment.	overall	augment.	overall	augment	overall	augment	overall
	min.	2.15	14.85	3.63	30.52	1.19	14.67	1.10	14.03
`cage15`	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
	min.	1.57	11.75	2.83	23.47	1.63	12.03	0.04	6.68
`Channel-500`	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
	min.	116.67	123.24	107.51	132.34	2.02	8.89	0.74	5.26
`Circuit5M`	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
	min.	40.52	52.15	32.09	98.89	41.66	69.52	48.63	64.32
`Delaunay_24`	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
	min.	41.01	56.16	55.22	146.78	44.71	81.96	49.46	70.34
`Hugebub-20`	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
	min.	13.98	112.59	22.87	205.18	15.49	97.63	19.74	84.26
`nlpkkt240`	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

Our experimental results show that these approaches obtain near perfect matchings in real-life and synthetic instances and have a near linear time run time. The two approaches are also shown to be more robust than the state of the art heuristics used in the cardinality matching algorithms, and are generally more useful as initialization routines.

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

References

1 Z. Allen-Zhu, Y. Li, R. Mendes de Oliveira, and A. Wigderson. Much faster algorithms for matrix scaling. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 890–901, Berkeley, CA, USA, October 2017.

2 J. Aronson, M. Dyer, A. Frieze, and S. Suen. Randomized greedy matching II. *Random Structures & Algorithms*, 6(1):55–73, 1995.

3 S. Assadi, M. Bateni, A. Bernstein, V. Mirrokni, and C. Stein. Coresets meet edcs: algorithms for matching and vertex cover on massive graphs. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1616–1635. SIAM, 2019.

4 S. Assadi and A. Bernstein. Towards a unified theory of sparsification for matching problems. *arXiv preprint*, 2018. `arXiv:1811.02009`.

5 S. Behnezhad, S. Brandt, M. Derakhshan, M. Fischer, M. Hajiaghayi, R.M. Karp, and J. Uitto. Massively parallel computation of matching and mis in sparse graphs. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 481–490, 2019.

6 S. Behnezhad, J. Łącki, and V. Mirrokni. Fully dynamic matching: Beating 2-approximation in δ^ϵ update time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2492–2508. SIAM, 2020.

7 C. Berge. Two theorems in graph theory. *Proceedings of the National Academy of Sciences of the USA*, 43:842–844, 1957.

8 A. Bernstein and C. Stein. Fully dynamic matching in bipartite graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 167–179. Springer, 2015.

9 B. Besser and M. Poloczek. Greedy matching: Guarantees and limitations. *Algorithmica*, 77(1):201–234, 2017.

10 M. B. Cohen, A. Madry, D. Tsipras, and A. Vladu. Matrix scaling and balancing via box constrained newton's method and interior point methods. In *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS*, pages 902–913, Berkeley, CA, USA, October 2017.

11 A. Czumaj, J. Łącki, A. Madry, S. Mitrovic, K. Onak, and P. Sankowski. Round compression for parallel matching algorithms. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 471–484. Association for Computing Machinery, 2018.

12 T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

13 I. S. Duff. On algorithms for obtaining a maximum transversal. *ACM Transactions on Mathematical Software*, 7(3):315–330, 1981.

14 I. S. Duff, K. Kaya, and B. Uçar. Design, implementation, and analysis of maximum transversal algorithms. *ACM Transactions on Mathematical Software*, 38:13:1–13:31, 2011.

15 F. Dufossé, K. Kaya, I. Panagiotas, and B. Uçar. Approximation algorithms for maximum matchings in undirected graphs. In *2018 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*, pages 56–65, 2018.

16 F. Dufossé, K. Kaya, and B. Uçar. Two approximation algorithms for bipartite matching on multicore architectures. *Journal of Parallel and Distributed Computing*, 85:62–78, 2015.

17 A. Frieze and T. Johansson. On random k-out subgraphs of large graphs. *Random Structures & Algorithms*, 50(2):143–157, 2017.

18 A. Goel, M. Kapralov, and S. Khanna. Perfect matchings in $O(n \log n)$ time in regular bipartite graphs. *SIAM Journal on Computing*, 42(3):1392–1404, 2013.

19 A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

20 T. Hagerup, K. Mehlhorn, and J. I. Munro. Maintaining discrete probability distributions optimally. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *20th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 253–264, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

21 J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

22 R. M. Karp, A. H. G. Rinnooy Kan, and R. V. Vohra. Average case analysis of a heuristic for the assignment problem. *Mathematics of Operations Research*, 19(3):513–522, 1994.

23 R. M. Karp and M. Sipser. Maximum matching in sparse random graphs. In *22nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–375, Los Alamitos, CA, USA, 1981. IEEE Computer Society.

24 R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 352–358, New York, NY, USA, 1990. ACM.

25 K. Kaya, J. Langguth, F. Manne, and B. Uçar. Push-relabel based algorithms for the maximum transversal problem. *Computers & Operations Research*, 40(5):1266–1275, 2013.

26 K. Kaya, J. Langguth, I. Panagiotas, and B. Uçar. Karp–Sipser based kernels for bipartite graph matching. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 134–145, Salt Lake City, Utah, US, January 2020.

27 P. A. Knight. The Sinkhorn–Knopp algorithm: Convergence and applications. *SIAM Journal on Matrix Analysis and Applications*, 30(1):261–275, 2008.

28 P. A. Knight and D. Ruiz. A fast algorithm for matrix balancing. *IMA Journal of Numerical Analysis*, 33(3):1029–1047, 2013.

29 V. Korenwein, A. Nichterlein, R. Niedermeier, and P. Zschoche. Data reduction for maximum matching on real-world graphs: Theory and experiments. In *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112, pages 53:1–53:13, Dagstuhl, Germany, 2018.

30 J. Langguth, F. Manne, and P. Sanders. Heuristic initialization for bipartite matching problems. *Journal of Experimental Algorithmics (JEA)*, 15:1–22, 2010.

31 J. Magun. Greedy matching algorithms, an experimental study. *Journal of Experimental Algorithmics*, 3:6, 1998.

32 Y. Matias, J. S. Vitter, and W.-C. Ni. Dynamic generation of discrete random variates. *Theory of Computing Systems*, 36(4):329–358, 2003.

33 N. McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 7:188–201, 1999.

34 M. Mitzenmacher and E. Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 1st edition, 2005.

35 M. Poloczek and M. Szegedy. Randomized greedy algorithms for the maximum matching problem with new analysis. In *Foundations of Computer Science (FOCS), 2012 IEEE 53rd Annual Symposium on*, pages 708–717, 2012.

36 A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, 1990.

37 A. Pothen, S. M. Ferdous, and F. Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28:541–633, 2019.

38 R. Sinkhorn and P. Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific Journal of Mathematics*, 21(2):343–348, 1967.

39 G. Tinhofer. A probabilistic analysis of some greedy cardinality matching algorithms. *Annals of Operations Research*, 1(3):239–254, 1984.

40 D. Walkup. Matchings in random regular bipartite digraphs. *Discrete Mathematics*, 31(1):59–64, 1980.

A Other heuristics for bipartite matching and recent work

In the main text, we compared the proposed heuristics with KaSi and KaSi2. There are a few other effective heuristics, which we briefly review here (see a recent survey [37]).

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

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

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

B Further comments on 2outMC

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

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

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

We replace L_T is with two lists L_T^1 and L_T^2. The lists L_T^1 contains those unmarked vertices of T whose insertion in H_2 leads to a new tree; L_T^2 contains all other L_T vertices that have not been tried yet. At first, we sample x from L_T^2 and see whether the components 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 inserted to H_2. Otherwise, x is inserted in L_T^1, and we consider another random vertex of L_T^2. If L_T^2 becomes empty, we start sampling from L_T^1.

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

B.2 Heuristic 2: Online creation of the RG multigraph

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

More specifically, consider a vertex x randomly chosen at Line 5. In this heuristic, x has not picked its two choices yet, and we let x choose them at this point, in the way that benefits the algorithm the most. This is done as follows. Initially, we iterate over all of x's neighbors in the host graph G. Let c be one of x's neighbors and c^* be the sole unchecked vertex in c's connected component in H_2, or $c^* = -1$ if no unchecked vertices exist. We assign values to x's neighbors to classify them. If c^* is equal to -1, c's value is 0. If c^* has 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, we partition the neighbors of x in G into three disjoint sets C_0, C_1 and C_2 such that C_i contains all neighbors of x with value equal to i. Selecting columns from C_2 is preferred, as they can avoid creating a tree in H_1. Vertex x will attempt to sample first from C_2, and if 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 implicitly, and each vertex x requires amortized $O(d_x)$ to make its choices, where d_x is its degree. Hence, the overhead associated with this heuristic is almost linear in m.

B.3 Comparison with 2outMC

Here, we briefly discuss the effects that the above two heuristics have on the performance of the 2outMC algorithm. Since 2outMC obtains high quality results, the two heuristics can only yield a relatively small improvement. When they are enabled and used with SK-5 2outMC finds matchings with average quality of 0.9997 for the real-world graphs from Section 4.3 for which 2outMC obtained matchings of quality 0.9983. This difference corresponds to about 13113 additionally matched edges, and hence signals that 13113 augmentations are avoided.

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

C Further comments on TruncRW

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

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

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

We experimented with bipartite graphs without total support which correspond to square (10000×10000) and rectangular matrices (12000×10000) with a uniform nonzero distribution. These matrices are generated with `sprand` command of Maltab and have about $d \times 10000$

Table 8 The quality of TruncRW on bipartite graphs without perfect matchings.

	10000 × 10000		12000 × 10000	
d	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

nonzeros for $d = 2, 3, 4, 5$. The matrix representation of the bipartite graphs were scaled with 10 iterations of SK. For each d, we created five random matrices and ran TruncRW on the corresponding five instances. We report the worst quality of the five instances in Table 8. As seen in this table, TruncRW works just fine for this case. We did not report in the table but with increased SK iterations, the results improve, which is in accordance with earlier work [16].

C.1 Engineering TruncRW

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

Recall that TruncRW tries to find an augmenting path starting from a column vertex a certain number of times before giving up and moving to the next column vertex. When we allowed TruncRW just a single attempt, it was unable to find a perfect matching in any of the cases, and its average matching quality was 0.9984. When we allowed five attempts, TruncRW found a perfect matching for 13 graphs, and its average matching quality was 0.9999. With 10 attempts, it managed to find a perfect matching in 5 additional graphs. This verifies that allowing more attempts indeed improves the performance of the algorithm. The drawback, however, was the increased run time, which we did not think worth. That is why our implementation of TruncRW starts a random walk from a vertex only once.

We also test the effects of the look-ahead mechanism. Let us define the walk efficiency of TruncRW as the ratio of the cardinality of the matching found to the total length of the random walks. The higher this ratio, the more useful the random walks are. We evaluate the walk efficiency on a set of seven instances (real-life instances having at most 10000000 edges). We test both with and without scaling and report the results of the 14 tests. In 13 cases, the look-ahead mechanism improved the walk efficiency. The geometric mean (of 14 cases) of the ratios of walk efficiencies with look-ahead to that of without was 1.37. In the case where the look-ahead did not help (ratio was 0.71 in an instance named `Hamrle3`), the maximum deviation of a row or column sum from one after SK-5 was 0.28, which is high. We conclude that the look-ahead mechanism is very helpful.

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

D Reducing bipartite graph matching to matching on 2-out graphs

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

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

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

Proof. The added vertices a_e, b_e have degree two and will select both neighbors, hence no edge will remain unpicked. ◄

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

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

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

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

► **Lemma 4.** *Any maximum cardinality matching M' on G' corresponds to a maximum cardinality matching M on G.*

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

First, we have that $|M'| = |E_G| + |M|$. To see this, note that per Lemma 2 we perform $|E_G|$ degree-2 reductions, and result in G. Each of this reductions corresponds with a matched 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

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

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

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

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

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

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

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

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