Algorithms for the bottleneck matching problem

Abstract

We investigate the maximum bottleneck matching problem in bipartite graphs. Given a bipartite graph with nonnegative edge weights, the problem is to determine a maximum cardinality matching in which the minimum weight of an edge is the maximum. To the best of our knowledge, there are two widely used solvers for this problem based on two different approaches. There exists a third known approach in the literature, which seems inferior to those two which is presumably why there is no implementation of it. We take this third approach, make theoretical observations to improve its behavior, and implement the improved method. Experiments with the existing two solvers show that their run times can be too high to be useful in many interesting cases. Furthermore, their performance is not predictable, and slight perturbations of the input graph lead to considerable changes in the run time. On the other hand, the proposed solver's performance is much more stable; it is almost always faster than or comparable to the two existing solvers, and its run time always remains low.

1 Introduction

A matching in a graph is a set of edges without any common vertices. A maximum cardinality matching in a graph has the largest number of edges among all matchings. We investigate algorithms for the variant of the problem on bipartite graphs in which there are edge weights, and the aim is to find a maximum cardinality matching whose minimum edge weight is the maximum. This is called the bottleneck matching problem or linear bottleneck assignment problem when all vertices can be matched [3, Section 6.2]. Formally, the bottleneck matching problem is to find a maximum cardinality matching $\mathcal{M}$ which maximizes

$$\min_{(r_i,c_j)\in\mathcal{M}} w_{i,j}.$$

This problem can be solved in polynomial time.

The bottleneck matching problem arises in different contexts [3, Section 6.2.8]. We are particularly motivated by the Birkhoff–von Neumann (BvN) decomposition of doubly stochastic matrices [2], which arises in practical applications [1, 4, 16]. In this case, the bipartite graphs associated with matrices have equal number of vertices on both sides and contain perfect matchings.

	A		**AP**		**S**	
matrix	MC64J2	MC64J3	MC64J2	MC64J3	MC64J2	MC64J3
Bump_2911	0.56	5.14	1.15	5.44	9343.49	6685.96
cage15	0.68	2.81	1.96	4.58	861.81	42656.20
ss	2.42	0.90	2426.27	1.68	892.55	17146.19
vas_stokes_4M	0.68	41.09	18224.39	834.74	1883.45	12417.90

Table 1: Run time, in seconds, of MC64J2 and MC64J3 on 12 problems. **A** is the original matrix, **AP** is the same matrix with a random column permutation; $\mathbf{S} = \mathbf{D}P(\mathbf{A})\mathbf{E}$, where $P(\mathbf{A})$ is the 0-1 matrix with 1s at the nonzero positions of **A**, **D** and **E** are positive diagonal matrices scaling $P(\mathbf{A})$ to be doubly stochastic.

A known heuristic for the BvN decomposition [10] repeatedly calls a bottleneck matching algorithm on the bipartite graph of a dynamically changing matrix.

The software MC64 [7, 8] implements two algorithms, denoted MC64J2 and MC64J3, for the bottleneck matching problem. To the best of our knowledge these are the only available implementations that can handle bipartite graphs corresponding to large sparse matrices. Their worst-case run time are $O(n(m+n)\log_2 n)$ and $O(nm\log_2 n)$, on bipartite graphs with n vertices on each side and m edges [8].

MC64 is a well developed software but the underlying algorithms for the bottleneck matching problem do not have stable run time behavior, in two senses. First, when run on the same bipartite graph twice with different edge weights the difference in run times can be in the order of hours. Second, on two equivalent problem instances, where one is obtained from the other by just reordering the vertices, the run time can change dramatically. We report run time of MC64 on four matrices, from the SuiteSparse Matrix Collection [5], in Table 1 to explain this—more experiments of similar nature are available in Section 4. Here, the set of rows and the set of columns of a matrix correspond to the two parts of the bipartite graph with an edge between two vertices if the corresponding entry in the matrix is nonzero, and the nonzero values are the edge weights. The table contains results for **A**, for **AP** where **P** is a random permutation matrix, and for a doubly stochastic matrix **S** which is obtained by scaling the pattern of **A** with Sinkhorn-Knopp algorithm [19].

In Table 1, the bipartite graphs associated with **A** and **AP** are the same apart from renumbering of the vertices in one part. We see that on **A** MC64J2 runs very fast, MC64J3 is not as fast but its run time can still be considered acceptable. However on **AP**, both methods suffer; the run time of MC64J2 is not acceptable for ss and vas_stokes_4M, and that of MC64J3 for vas_stokes_4M is high as well. The bottleneck matching problems on **A** and **AP** are essentially the same, as permuting the columns does not change the values, nor the bottleneck matching and its value. Therefore, it is likely that the matrices are numbered in a special way. The bipartite graph of $\mathbf{S} = \mathbf{D}P(\mathbf{A})\mathbf{E}$ is the same as that of **A** with different edge weights. Now, the run time of both methods are too much for all instances—for Bump_2911, MC64J2 takes nearly three hours; for cage15, MC64J3 runs for more than 11 hours. As the weights in the bipartite graph of **S** are different from those in **A**, the two problems are not equivalent, despite sharing the same bipartite graph structure. The wildly varying run times of both routines from MC64 on **S** in comparison to those on **A** again highlight the instability in their performance.

Our aim in this paper is to develop an algorithm for the bottleneck matching problem which is better than the state of the art MC64 on all types of problem instances mentioned. For this purpose, we study an overlooked alternative from the literature. We make observations that pave the way for an efficient algorithm, implement it and compare it with both variants of MC64. We conduct a large set of experiments to show that our approach is usually much faster than MC64 and in addition it exhibits stable and robust performance.

We give a brief background in Section 2, in which we also summarize the three known algorithms. Section 3 contains the proposed algorithm. Section 4 presents the experimental results, and Section 5 concludes the paper.

2 Background and related work

2.1 Background. A matrix **A** can be represented with a bipartite graph $G = (R \cup C, E)$ where each row of **A** corresponds to a unique vertex in R, each column of **A** corresponds to a unique vertex in C, and there is an edge (r_i, c_j) whenever $a_{ij} \neq 0$. When the edges are weighted, the weight of the edge (r_i, c_j) is $|a_{ij}|$. For a vertex v, we use $\text{adj}(v)$ to denote the set of its neighbors.

A *matching* is a set of edges with no common vertices. A matching is of *maximum cardinality* if it has the largest number of edges. Given a matching $\mathcal{M}$, a vertex is *matched* if an edge from $\mathcal{M}$ is incident on it and *free* otherwise. A matching is called perfect if it matches all vertices. The *deficiency* of a matching $\mathcal{M}$ is the difference between the maximum cardinality of a matching and $|\mathcal{M}|$.

Let $\mathcal{M}$ be a matching in given graph $G = (R \cup C, E)$ A path in G is $\mathcal{M}$-*alternating* if its edges are alternately in $\mathcal{M}$. An $\mathcal{M}$-alternating path $\mathcal{P}$ is $\mathcal{M}$-*augmenting* if the start and end vertices of $\mathcal{P}$ are both free.

A *vertex cover* is a set of vertices that includes at least one vertex from each edge. A vertex cover with the smallest number of vertices is a *minimum vertex cover*. König's theorem states that in a bipartite graph the maximum cardinality of a matching is equal to the minimum cardinality of a vertex cover [3, Theorem 2.7].

Given a bipartite graph, any of its maximum cardinality matchings can be used to obtain a canonical decomposition called Dulmage-Mendelsohn (DM) decomposition [11]. Based on the DM decomposition, Pothen and Fan [17] describe algorithms to permute sparse matrices in a block upper triangular form (BTF):

$$\mathbf{A} = \begin{array}{c} \\ H_R \\ S_R \\ V_R \end{array} \begin{array}{c} \begin{array}{ccc} H_C & S_C & V_C \end{array} \\ \begin{pmatrix} \mathbf{A}_H & * & * \\ O & \mathbf{A}_S & * \\ O & O & \mathbf{A}_V \end{pmatrix} \end{array}. \tag{2.1}$$

In a BTF, the submatrix $\mathbf{A}_H$ has more columns than rows, and all rows in H_R are matched to a column in H_C in any maximum cardinality matching; the submatrix $\mathbf{A}_S$ is square with at least one perfect matching; the submatrix $\mathbf{A}_V$ has more rows than columns, and all columns in V_C are matched to a row in V_R. The rows/columns in each block are defined by the relations

$$\begin{aligned} H_R &= \{\text{row vertices reachable from free column vertices via alternating paths}\}, \\ H_C &= \{\text{free column vertices or column vertices reachable from free column vertices via alternating paths}\}, \\ V_R &= \{\text{free row vertices or row vertices reachable from free row vertices via alternating paths}\}, \\ V_C &= \{\text{column vertices reachable from free row vertices via alternating paths}\}, \\ S_R &= R \setminus (H_R \cup V_R), \text{and} \\ S_C &= C \setminus (H_C \cup V_C). \end{aligned}$$

A standard BFS/DFS-based graph traversal algorithm will find these sets in linear time.

We make some observation on the BTF form of a matrix. First, the DM decomposition reveals a minimum cover [3, Algorithm 3.1 and its discussion]. As all nonzeros in the BTF (2.1) are confined in rows $H_R \cup S_R$ and the columns in V_C, the vertex set $\mathcal{C} =$

$H_R \cup S_R \cup V_C$ is a cover. Since the cardinality of the set $\mathcal{C}$ is equal to the maximum cardinality of a matching, $\mathcal{C}$ is a minimum cover. Second, if one adds new nonzeros to the diagonal blocks, upper diagonal blocks, or to the blocks (S_R, H_C) and (V_R, S_C), the maximum cardinality of a matching does not change. This is so, as the new nonzeros cannot create augmenting paths from an unmatched column in H_C to an unmatched row in V_R.

Hall's theorem [13] states that for a bipartite graph G to have a column-perfect matching, the relation $|S| \leq |\bigcup_{c \in S} \text{adj}(c)|$ must hold for any subset S of columns. If a similar relation holds for all subsets of rows, then G will have perfect matchings.

An $n \times n$ matrix $\mathbf{A} \neq 0$ is called *doubly stochastic* if every entry is nonnegative and the sum of entries in each row and each column is equal to 1. Any nonnegative square matrix, whose bipartite graph has perfect matchings, can be scaled with two diagonal matrices to be doubly stochastic [19]. A *permutation matrix* is a square matrix where each row and column contain exactly one nonzero value equal to 1. A perfect matching in the bipartite graph representation of $\mathbf{A}$ corresponds to a permutation matrix. The bipartite graphs of doubly stochastic matrices have perfect matchings.

From now on, we assume that there are perfect matchings in the given bipartite graph. We comment on the cases where we have rectangular matrices or matrices without perfect matchings later in Section 3.3.2.

2.2 Related work. We review three algorithms from the literature [3, 7, 8]. The first two are implemented in MC64, and to the best of our knowledge, are currently the best practical algorithms. The book by Burkard et al. [3, Section 6.2.4] describes two other algorithms [12, 18], which are more theoretically motivated.

2.2.1 Threshold-based algorithms. Let G be a weighted bipartite graph. Given a value v, let $G[v]$ contain only those edges of G whose values are no smaller than v. Threshold-based algorithms find the largest v for which $G[v]$ has a perfect matching. This is done by considering different values of v, and testing whether $G[v]$ has a perfect matching or not and tuning the next v to a higher or lower value accordingly. MC64's implementation, MC64J3, discusses initialization and algorithmic choices to reduce the number of tests [7].

2.2.2 A Shortest-augmenting path based algorithm. Algorithms based on shortest augmenting paths start with a matching which has the maximum bottleneck value for the currently matched vertices C' in one part, say C. In order to augment the matching, a shortest augmenting path from a free vertex c of C is found with a variant of Dijkstra's shortest path algorithm. Augmenting along the shortest paths maintains the invariant that the current matching has the bottleneck value for any matching that matches the vertices $C' \cup \{c\}$. The process continues until a perfect matching is obtained. The implementation in MC64 [8], MC64J2, includes effective initialization procedures to obtain a large initial matching. It also includes an efficient adaptation of Dijkstra's algorithm for the task at hand.

2.2.3 An algorithm based on duality. This algorithm is also threshold-based and uses the duality of matchings and coverings to find the next threshold. We explain this algorithm in more details as ours improves it. Let v be a value where $G[v]$ does not contain a perfect matching. Let $\mathcal{M}$ be a maximum cardinality matching in $G[v]$. From $\mathcal{M}$ one defines a minimum vertex cover $\mathcal{C} = H_R \cup S_R \cup V_C$ of $G[v]$ with vertex sets $I = H_R \cup S_R$ and $J = V_C$ [3, Section 6.2.3]. Since $\mathcal{M}$ is not perfect, there must be an edge in G from a vertex in $\bar{I} = R \setminus I$ to another vertex in $\bar{J} = C \setminus J$. The value $\max_{i \in \bar{I}, j \in \bar{J}} a_{ij}$ thus cannot be smaller than the bottleneck matching value of G and can be used as the next threshold. This approach is shown in Algorithm 1 [3, Section 6.2.3].

Algorithm 1: Duality-based algorithm

```
Input  : G, an edge weighted bipartite graph having
         perfect matchings
Output: M, a bottleneck perfect matching
Let v be an upper bound on the bottleneck matching
  value
M ← ∅
while |M| < n do
    M ← a maximum cardinality matching in G[v]
    if |M| < n then
1       Let I ⊆ R and J ⊆ C be the vertex sets of the
          associated cover of G[v]
2       v ← max_{i∈R\I, j∈C\J} w_{i,j}  /* in G, not G[v] */
```

The maximum cardinality matching in $G[v]$ can be found in $O(\sqrt{n}\,\text{nnz}(\mathbf{A}[v]))$ time in the worst case [14]. Once such a matching is found, the associated minimum cover and the maximum uncovered value at Line 2 can be obtained in linear time. Therefore, the worst case time complexity of Algorithm 1 is $O(\sqrt{n}\,\text{nnz}(\mathbf{A}))$ times the number of iterations. Thus, the worst case run time for Algorithm 1 can be too high. This is so as a new edge, due to the reduced v, does not mean one more edge in the maximum matching, and hence the while loop can run for more than n iterations.

3 The proposed algorithm

Our algorithm is based on Algorithm 1, and as such it combines the duality and threshold techniques.

3.1 Theoretical findings. Let $G = (R \cup C, E)$ be an edge weighted bipartite graph, with $w_{i,j}$ being the weight of the edge (r_i, c_j). Let v be a nonnegative value and define $G[v] = (R \cup C, E')$ where $E' = \{(r_i, c_j) : w_{i,j} \geq v\}$. That is, $G[v]$ contains all vertices of G but only those edges of G with weight at least v. The bottleneck matching value $b^\star$ is the largest v for which $G[v]$ has a perfect matching, but $G[v + \varepsilon]$ does not for any $\varepsilon > 0$. We call a value v safe, when $v \geq b^\star$. At each iteration, Algorithm 1 produces a safe value that it smaller than the previous one.

We make a series of observations to make v converge to $b^\star$ faster. For this, we try to make each tested safe value v as small as possible in the hopes of testing fewer of them. The first observation is that there are more than one minimum cover associated with a given maximum cardinality matching. Using the BTF (2.1), let $\mathcal{C}' = H_R \cup S_C \cup V_C$. As all nonzeros are covered by $\mathcal{C}'$ and $|\mathcal{C}'| = |H_R \cup S_R \cup V_C|$, $\mathcal{C}'$ is also a minimum cover. If we choose this cover, the sets I and J at Line 1 of Algorithm 1 become H_R and $S_C \cup V_C$, in which case, the maximum value will be sought in the submatrix $(S_R \cup V_R, H_C)$ of the BTF (2.1). This leads to the following proposition.

PROPOSITION 3.1. *Let $\mathcal{C}_1 = H_R \cup S_R \cup V_C$ and $\mathcal{C}_2 = H_R \cup S_C \cup V_C$ be two minimum covers of $G[v]$. Let v_1 and v_2 be the maximum values defined in Line 2 of Algorithm 1 for $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Then, $\min(v_1, v_2)$ is safe.*

Proof. Let v be the current value, and $v_1 < v_2$ without loss of generality. This means that v_2 is in the (V_R, S_C) block and all entries in the (V_R, H_C) of $\mathbf{A}$ are smaller than v_2. When we use v_2 in Algorithm 1, the maximum cardinality of a matching in $G[v_2]$ cannot be larger than that in $G[v]$. This is so, as the set $\mathcal{C}_1 = H_R \cup S_R \cup V_C$ still covers all edges of $G[v_2]$, including those that arise in the block (S_R, H_C). That is why the maximum cardinality of the matching is the same in $G[v]$ and $G[v_2]$, and the cover $\mathcal{C}_1$ can be used to get the next v in Line 2, which concludes the proof. ∎

Based on Proposition 3.1, one can therefore obtain the BTF form, and use the smaller of the largest uncovered element in $(S_R \cup V_R, H_C)$ and that in $(V_R, H_C \cup S_C)$. There is in fact more. The square block has a finer BTF decomposition [17], each block being a square block. For each such block, one can choose either the rows or the columns, as long as all off diagonal entries are covered. Since this can lead to combinatorially many alternatives, we do not suggest going to that direction. Instead we propose to make use of the identified two covers as much as possible for faster convergence of v to $b^\star$.

As we reason in Lemma 3.1 below, one can find a tighter upper bound on $b^\star$ depending on the deficiency of the current matching and the vertex sets of a cover.

LEMMA 3.1. *Let $\mathcal{M}$ be a maximum cardinality matching in $G[v]$ with a deficiency of k in G; $\mathbf{A}[v]$ and $\mathbf{A}$ be, respectively, the matrices associated with $G[v]$ and G; $\mathcal{C} = H_R \cup S_C \cup V_C$ be a minimum vertex cover of $G[v]$ associated with $\mathcal{M}$ when $\mathbf{A}[v]$ is permuted in a BTF (2.1); and v_k be the kth largest element in $\{w_{i,j} : i \in S_R \cup V_R, j \in H_C\}$. Then, v_k is safe.*

Proof. We first note that $|H_C| - |H_R| = k$ as all other columns are matched. Consider now the set H_C of columns. By Hall's theorem, $|H_C| \leq |\bigcup_{c \in H_C} \text{adj}(c)|$ must hold in $\mathbf{A}$ as there is a perfect matching. Among all rows in $\bigcup_{c \in H_C} \text{adj}(c)$, we have $|H_R|$ in the set H_R. Therefore there must be at least k other nonzero rows in $\mathbf{A}(S_R \cup V_R, H_C)$. As the k largest elements in $\{w_{i,j} : i \in S_R \cup V_R, j \in H_C\}$ can be in at most k rows, having less than k nonzeros in $\mathbf{A}(S_R \cup V_R, H_C)$ will not satisfy Hall's condition. Hence, v_k is safe. ∎

As before, we can identify more than one minimum cover, collect the kth largest uncovered element with respect to each, and use the minimum of those collected elements as the next v. We state this as as a corollary.

COROLLARY 3.1. *Let $\mathcal{M}$ be a maximum cardinality matching in $G[v]$ which has a deficiency of k with respect to G; $\mathcal{C}_\ell = I_\ell \cup J_\ell$ for $\ell = 1, \ldots$ be minimum covers of $G[v]$ associated with $\mathcal{M}$, with $I_\ell \subseteq R$ and $J_\ell \subseteq C$; and $v_k^{(\ell)}$ be the kth largest element in $\{w_{i,j} : i \in R \setminus I_\ell, j \in C \setminus J_\ell\}$, and $v_k = \min_\ell\{v_k^{(\ell)}\}$. Then, v_k is safe.*

While finding many minimum covers can be helpful, finding the kth value for each could take time. Instead, we propose using the two which are readily revealed by the BTF: get the kth largest nonzero v_1 in $\mathbf{A}(S_R \cup V_R, H_C)$ and the kth largest nonzero v_2 in $\mathbf{A}(V_R, H_C \cup S_C)$ and use $v = \min(v_1, v_2)$ as the next threshold. This corresponds to applying Hall's theorem to the set H_C of columns and to the set V_R of rows, where for any threshold t larger than v, either H_C or V_R cannot satisfy the condition in Hall's theorem in $G[t]$.

3.2 Putting it all together. The proposed algorithm BOTTLED is shown in Algorithm 2. The input is a sparse matrix represented in the well-known compressed storage by columns (CSC) format. The algo-

rithm creates a compressed storage by rows (CSR) representation of the input matrix. It then sorts the nonzeros in each row in a non-increasing order of their values, similarly the nonzeros in each column are sorted. Then the threshold v is initialized as the minimum of the $2n$ nonzero values consisting of the maximum in each row and maximum in each column, which is safe as there are perfect matchings. The algorithm then updates the threshold v in a while-loop as in Algorithm 1. In the body of the while-loop there are three subroutines: MaximumCardinalityMatching, SAP, and DM-dec. These subroutines correspond to a maximum cardinality matching algorithm, a shortest augmenting path-based method to match a column, and an algorithm obtaining the row and column blocks of the BTF (2.1).

Algorithm 2: BOTTLED: The proposed bottleneck matching algorithm

Input : $\mathbf{A}$, stored by columns
Output: $\mathcal{M}$, a bottleneck perfect matching

Create a CSR representation of $\mathbf{A}$
Sort the nonzeros in each row and in each column in non-increasing order of values
1 $v \leftarrow$ min of the maximum in each row, maximum in each column
$G[v] \leftarrow (R \cup C, \emptyset)$ and $\mathcal{M} \leftarrow \emptyset$
while $|\mathcal{M}| < n$ **do**
2 for each i, release new $a_{ij} \geq v$ and for each j release new $a_{ij} \geq v$ into $G[v]$
if $|\mathcal{M}| = n - 1$ **then**
$\mathcal{M} \leftarrow \text{SAP}(G, \mathcal{M}, c)$ with the last free vertex c
else
3 $\mathcal{M}' \leftarrow$ MaximumCardinalityMatching$(G[v], \mathcal{M})$
if $|\mathcal{M}'| = n$ **then** $\mathcal{M} \leftarrow \mathcal{M}'$; break
if $|\mathcal{M}'| = |\mathcal{M}|$ **then**
4 select a vertex c
$\mathcal{M} \leftarrow \text{SAP}(G, \mathcal{M}', c)$
else
$\mathcal{M} \leftarrow \mathcal{M}'$
$\langle H_R, S_R, V_R, H_C, S_C, V_C \rangle \leftarrow$ DM-dec$(G[v], \mathcal{M})$
5 $v_1 \leftarrow k$th largest value not covered by $\mathcal{C}_1$ `/* see Prop. 3.1 for the definition of` $\mathcal{C}_1$ `*/`
6 $v_2 \leftarrow k$th largest value not covered by $\mathcal{C}_2$ `/* see Prop. 3.1 for` $\mathcal{C}_2$ `*/`
$v \leftarrow \min(v_1, v_2)$

Algorithm 2 stores the edges of $G[v]$ over the storage of $\mathbf{A}$ in the CSC and CSR formats, without explicitly building new adjacency lists. The start address of each row and column are the same as those of $\mathbf{A}$. For each row/column of $G[v]$, we keep an end-pointer which points to the smallest nonzero of $\mathbf{A}$ in that row/column that is no smaller than v. These end-pointers are initialized at Line 1 in $O(n)$ time, and incremented at Line 2 at each iteration of the while-loop. Therefore the total run time cost of building $G[v]$s is $O(\text{nnz}(\mathbf{A}))$.

In Algorithm 2, $\text{SAP}(G, \mathcal{M}, c)$ refers to the algorithm summarized in Section 2.2.2. As stated before, SAP needs the current matching to have the bottleneck value among all those matchings covering the same set of column vertices. The approach outlined in Algorithm 1 produces such matchings, that is why, at any point, one can resort to SAP. We invoke SAP in two cases: (i) when the deficiency is one; (ii) when an update of v did not yield an increase in the cardinality of the current matching. The first case is straightforward. For the second case, we apply a simple heuristic to help the algorithm converge faster. As any free column vertex can be the start of of an augmenting path, we choose $c \in C$ whose largest edge weight not included in the current $G[v]$ is minimum. The search for augmenting paths works on G, not on $G[v]$. Matching c will lead to a reduced v, and the reduction will hopefully be large with this choice of c (empirical results in Section 4.2).

3.3 Further discussions. The most common algorithms for the maximum cardinality matching problem take an initial matching as input and augment it. This is very suitable at Line 3 of Algorithm 2, as we have a maximum cardinality matching on a graph, we add new edges, and then ask for a maximum cardinality matching in the new graph. We have used the code-base of MatchMaker [6, 15] to implement this step in the implicit representation of $G[v]$.

To find the kth largest element at Lines 5 and 6 of Algorithm 2, we use a binary heap with a limit k on its size, where the values of the nonzeros are the keys. We insert nonzeros of $\mathbf{A}(S_R \cup V_R, H_C)$ into this heap by visiting the columns in H_C one by one. For each visited column, we keep inserting nonzeros, which are smaller than the current v, into the heap until the nonzero to be inserted is less than the minimum value in the heap (when the heap has k elements), or as long as there are less than k elements in the heap. In the worst case, we insert all nonzeros of $\mathbf{A}(S_R \cup V_R, H_C)$ into the heap, with the time complexity $O(\text{nnz}(\mathbf{A}(S_R \cup V_R, H_C)) \log k + |H_C|)$. A similar analysis holds for $\mathbf{A}(S_R \cup V_R, H_C)$. Therefore the run time of one iteration of the while loop is dominated by the run time of the maximum cardinality matching algorithm. We do not have an estimate on the number of iterations of the while loop (empirical results in Section 4.2).

3.3.1 In the context of a BvN decomposition method. The Birkhoff–von Neumann (BvN) theorem [2] states that a doubly stochastic matrix $\mathbf{A}$ can be written as $\mathbf{A} = \sum_{i=1}^{\ell} \alpha_i \mathbf{P}_i$ where each $\mathbf{P}_i$ is a permutation matrix, and α_is are positive coefficients summing up to one. Such a decomposition is not unique, and the

problem of finding a decomposition with the smallest ℓ value is NP-Complete [10].

One heuristic [10] for obtaining a BvN decomposition of a double stochastic matrix $\mathbf{A}$ with a small number ℓ of permutations matrices works as follows. It finds the value b of a bottleneck matching whose pattern is the permutation matrix $\mathbf{P}$, replaces $\mathbf{A}$ with $\mathbf{A} - b\mathbf{P}$, and continues until a zero matrix is obtained. A property of this heuristic is that the successive bottleneck values are in a non-increasing order [9]. Our bottleneck matching algorithm is very fitting in this case. One can create the CSR representation and sort each row and column once. Then, executing the while loop of Algorithm 2 will obtain a bottleneck matching for the current matrix. Once b and $\mathbf{P}$ are obtained, replacing $\mathbf{A}$ with $\mathbf{A}-b\mathbf{P}$ can be done by subtracting b from each matched entry and updating that entry's position in the sorted list of both rows and columns in overall $O(n + \text{nnz}(\mathbf{A}))$ time. While doing so, one can update the end-pointers used fo $G[v]$, and avoid preprocessing in Algorithm 2 at subsequent invocations.

3.3.2 Bipartite graphs without perfect matchings. The case in which there are perfect matchings in the given bipartite graph is common in applications where the bipartite graphs correspond to sparse matrices. This is especially so in the BvN decomposition. Nonetheless MC64's J2 and J3 work for cases in which one part of the bipartite graph has more vertices than the other, where the smaller side can be perfectly matched. This corresponds to $n_R \times n$ matrices for $n_R > n$ that have column-perfect matchings. Our algorithm can handle this case either by initializing v at Line 1 using only the column values, or by using those and only the nth maximum of the set of n_R maximum entries, one from in each row.

Consider now the most general case corresponding to $n_R \times n$ matrices for $n_R \geq n$, without column perfect matchings. In this case, MC64J2 returns a maximum cardinality matching, without necessarily finding the correct bottleneck value. That is so because not all vertices from which the shortest augmenting paths are sought can be matched by a bottleneck matching. We do not have an easy fix for this. MC64J3 on the other hand works correctly. The proposed algorithm handles this case with four minor modifications: (i) one needs to precompute the maximum size of a matching n' at the beginning; (ii) the initialization should use the n'th maximum of n maximum entries, one from each column, and the n'th maximum of the n_R maximum entries, one from each row; (iii) at Lines 5 and 6, the deficiency is $k = n' - |\mathcal{M}|$; and (iv) the shortest augmenting path method should not be used.

4 Experiments

We observed in a preliminary set of experiments that MC64J2 (MC64 with shortest augmenting paths) is generally faster than MC64J3 (MC64 with thresholding); the geometric mean of the ratios of the run time of the latter to that of the former was 4.6. As already highlighted in Table 1, in a number of cases, the run time of MC64J3 can be too large to experiment, and this happens more frequently than with MC64J2. That is why we investigate the performance of BOTTLED in comparison to MC64J2. Our codes, written in C, are available at `https://doi.org/10.5281/zenodo.7534918`, which additionally includes the original codes of MC64 (in Fortran) and a wrapper to them.

The next subsection describes the data set, and Section 4.2 investigates the algorithmic choices of BOTTLED. Section 4.3 compares BOTTLED with MC64J2 where order of magnitude differences in their run time are reported. Last, some experiments of using BOTTLED within a BvN decomposition heuristic are discussed in Section 4.4.

4.1 Data set and measurements. We have experimented with all square matrices with perfect matchings, at least 100,000 rows, less than 250,000,000 nonzeros, and with no explicit zeros from the SuiteSparse Collection [5]. There were 113 such matrices at the time of experimentation. From each matrix, we created six types of problem instances, which are denoted as $\mathbf{A}, \mathbf{DAE}, \mathbf{D}P(\mathbf{A})\mathbf{E}, \mathbf{AP}, \mathbf{DAPE}$, and $\mathbf{D}P(\mathbf{A})\mathbf{PE}$. The $\mathbf{A}$-type instance corresponds to the bipartite graph of the original matrix with the magnitudes of the entries as edge weights. The $\mathbf{DAE}$-type and $\mathbf{D}P(\mathbf{A})\mathbf{E}$-type instances correspond, respectively, to the scaled version of the matrices and their patterns with 20 iterations of the Sinkhorn–Knopp [19] algorithm, and the other three types of instances are obtained from the first three by random column permutations. We discarded the instances $\mathbf{A}$ and $\mathbf{AP}$ for 0-1 matrices; for the same set of matrices $\mathbf{DAE}$-type and $\mathbf{D}P(\mathbf{A})\mathbf{E}$-type instances are the same, and hence we kept only one of them. We thus report experiments based on 638 instances. For each instance, each algorithm is run five times and the geometric mean of the run times is reported as that algorithm's performance on that instance; when column permutations are applied, these correspond to five different permutations.

We ran the maximum cardinality matching algorithm from MatchMaker [6, 15] to verify that there were perfect matchings. We report the observed run times for the sake of completeness. The maximum run time for all $\mathbf{A}$-type instances was 0.28 seconds for vas_stokes_4M. The largest two run times for $\mathbf{AP}$-type instances were

97.49 and 8.44 seconds, respectively, for circuit5M and rajat31.

4.2 Performance analysis of BOTTLED. We first start by analyzing how much of the run time of BOTTLED is spent in the preprocessing step of creating the CSR representation or sorting the entries, and how much of it is spent in the while loop of BOTTLED.

Table 2 provides a summary of results for six different problem instances, where BOTTLED took more than one second for **A**-and/or **D**P(**A**)**E**-type instances. This table presents the geometric mean of the percentage of the time spent in creating the CSR representation and in sorting (with respect to the total time of BOTTLED). The row "tTime" contains the geometric mean of the run times of BOTTLED on the instances **A**, **DAE** and **D**P(**A**)**E** in seconds, and for the instances **AP**, **DAPE** and **D**P(**A**)**PE** it contains the geometric mean of the ratio of the run time of BOTTLED on the permuted instances to the original ones. For example, the run time under the column **DAPE** is obtained by taking the geometric mean of ratio of the run time of BOTTLED on 113 **DAPE** instances to that on 113 **DAE** instances.

As we can see from Table 2, the average run time of BOTTLED on any of the six instances for the selected matrices is below 10 seconds. A first major observation is that for all instances, the two preprocessing steps account for a significant part of the total run time. In the least important case they take 48% of the whole time (for the **A** instances) and in the most important case 84% (for the **DAPE** instances). We further observe that the absolute run time of CSR and sorting increase for the permuted instances. The percentage of the total time spent in CSR also increases for the permuted instances whereas that of sorting decreases. For example, creating the CSR takes 0.84 seconds for **A** and 3.02 seconds for **AP**. This causes the percentage of total time spent in CSR to increase from 22% to 41% as we move from **A** to **AP**. On the other hand, sorting takes ∼1 second for **A**-type and 1.18 seconds for **AP**-type instances. Now, the percentage of total time spent sorting drops from 26% for **A** to 16% for **AP**. If the CSR representation is available, its creation can be skipped and one can reduce run time considerably. As discussed in Section 3.3.1, for the targeted BvN application BOTTLED can skip not only the CSR creation, but also the sorting phase across different runs of the algorithm.

A second observation from Table 2 is that the run time of BOTTLED nearly doubles for the permuted instances. To put this into perspective, we present the absolute run time of BOTTLED on the six problem instances associated with two matrices in Table 3. We see

	A	**AP**	**DAE**	**DAPE**	**D**P(**A**)**E**	**D**P(**A**)**PE**
CSR	22%	41%	35%	60%	24%	47%
sort	26%	16%	41%	24%	27%	17%
tTime	3.80s	1.94	3.61s	2.06	5.50s	1.81

Table 2: Percentage of the total time spent in creating a CSR matrix and sorting the nonzeros in BOTTLED; "tTime": the geometric mean of the run times of BOTTLED on the instances **A**, **DAE** and **D**P(**A**)**E** in seconds, and the ratios of the others to their counterparts.

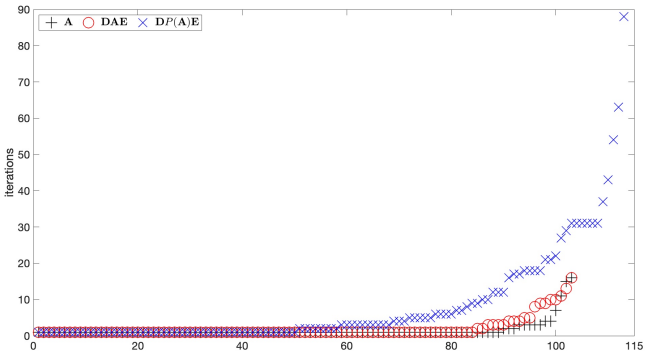

Figure 1: The number of iterations of the while loop of BOTTLED. 113 instances of type **D**P(**A**)**E** and 103 instances of other types.

that the increase in run time for the permuted matrices comes from creating the CSR; for vas_stokes_4M, for example, excluding CSR and sort times from the total time yields 5.36 and 6.74 seconds for **A** and **AP** respectively. A similar calculation for the pairs **D**P(**A**)**E** and **D**P(**A**)**PE** shows that the while loop takes 7.00 and 5.68 seconds. In the instance pairs, **DAE** and **DAPE**, the increases in the while loop's time are more pronounced but they are still dominated by the increases in the CSR time.

From the Tables 2 and 3, we conclude that BOTTLED's preprocessing takes nearly as much time as the while loop, if not more.

We next investigate the number of iterations of the while-loop of BOTTLED in different types of instances to see how stable and robust it is. In **A**- and **AP**-type instances of a given matrix, the number of iterations of the while loop were always the same for all 103 matrices (10 of them are 0-1). In **DAE**- and **DAPE**-type instances of a given matrix, the number of iterations were the same for 99 (out of 103) instances, in one they differed by two, and in the others by one. In the **D**P(**A**)**E**- and **D**P(**A**)**PE**-type instances, the number of iterations were the same for 84 (out of 113) instances, differed by one in 13, by two in 11, by three in 2 instances; the other

	A			**AP**			**DAE**			**DAPE**			**DP(A)E**			**DP(A)PE**		
matrix	tTime	CSR	sort	tTime	CSR	sort	tTime	CSR	sort	tTime	CSR	sort	tTime	CSR	sort	tTime	CSR	sort
circuit5M	3.11	1.42	1.17	8.88	6.30	1.55	3.10	1.42	1.24	9.23	6.53	1.74	65.28	1.42	1.17	90.92	6.75	1.67
vas_stokes_4M	16.32	6.96	4.00	32.31	19.47	6.10	27.41	6.68	4.13	60.95	20.10	6.26	19.12	7.40	4.72	30.75	19.34	5.73

Table 3: Breakdown of the total time of BOTTLED. tTime is the total time, CSR and sort are the time for the two preprocessing steps, in seconds.

matrix	instance	no.aug.	no.vs	BOTTLED
BenElechi1	**DP(A)E**	55	47	47
c-73b	**A**	133	19	16
Hamrle3	**DP(A)E**	28	17	17
lung2	**DP(A)E**	17	56	10
PR02R	**DAE**	22	15	16

Table 4: The number of iterations of the while loop, without augmenting paths, without vertex selection heuristic, and the presented version of BOTTLED.

three differed by 6, 8, and 9 iterations. These small changes in the number of iterations confirm the robustness of BOTTLED. In passing, we present the number of iterations of BOTTLED's while-loop on **A**-, **DAE**-, and **DP(A)E**-type instances in Figure 1 sorted in increasing order. The figure shows that in many cases, only one iteration of the while-loop is performed, and that the instances of type **DP(A)E** are usually harder, requiring more iterations.

We now briefly investigate the effect of resorting to the SAP subroutine, and the effects of the vertex selection heuristic (Line 4 of Algorithm 2). By design, BOTTLED resorts to SAP on rare cases; indeed it takes place only in 8 of the **DAE**-type and 24 of the **DP(A)E**-type instances. We present results with five instances in Table 4, where we give the number of iterations of the while-loop when SAP is not called (no.aug), when the vertex selection is not applied (no.vs; a vertex with larger value is selected instead), and with the presented version of BOTTLED. As seen in Table 4, resorting to SAP in general helps, and the vertex selection heuristic makes BOTTLED more robust; for example in `c-37b` not using SAP results in over a hundred additional iterations, whereas in `lung2` not relying on the vertex selection heuristic leads to 46 extra iterations.

4.3 Comparisons with MC64. We have compared BOTTLED with MC64J2 on all 638 instances. We have observed that across all problem instances for which either MC64J2 or BOTTLED has a run time greater than 1 second, on average BOTTLED takes 63% of MC64J2's run time. We now take a closer look at the performance of MC64J2 and BOTTLED. Table 5 presents a comparison of their run time on six types of instances associated with a selected set of matrices. MC64J2 takes its longest run time for at least one of the instance types on the matrices selected. We also include the five problem instances for which BOTTLED had its longest time—these are marked with ∗.

We observe that BOTTLED is faster than MC64J2 in a majority of the cases; even in cases on which BOTTLED had its longest run times. When BOTTLED is slower than MC64J2, the absolute difference in their respective run time is never too large—the maximum difference is around 59 seconds for **DP(A)E**-type instance of circuit5M. Furthermore, BOTTLED consistently significantly outperforms MC64J2 on the problem instances for which MC64J2 obtains its longest run time. For example, on the **DP(A)PE**-type instance of vas_stokes_4M, MC64J2's run time is 23074.50 seconds, while BOTTLED completes in 30.75 seconds. Similarly, on the **DP(A)PE**-type instance of cage15, MC64J2 runs in 14345.60 seconds, while BOTTLED takes only 22 seconds. As we see from the table, BOTTLED always completes in under 100 seconds with its largest run time across all problem instances being around 91 seconds. This is in stark contrast to the run times of MC64J2, which can vary considerably depending on the matrix and the instance—in our test-suite, its largest run time reaches as high as 23074.50 seconds. In light of this discussion, we conclude that the performance of BOTTLED is superior to and more reliable than MC64J2's.

4.4 Inside a BvN decomposition method. Table 6 presents the run time of the BvN decomposition method on the **DAE**- and **DP(A)E**-type instances of four matrices. We run the BvN decomposition method until either the sum of the obtained coefficients is close enough to one, or until 50 permutation matrices are obtained—for the converged cases the minimum sum of the coefficients was 0.92. As each permutation matrix is obtained by a call to BOTTLED, we show their number in the column "num.perm". The table presents also the total time of BOTTLED on the instances of type **DAE** and **DP(A)E**, repeated from Table 5 for convenience. We observe that the run time of BOTTLED call multiplied by the number of calls is at least 2.07, on **DP(A)E** of ss, and at most 9.84, on **DAE** of vas_stokes_4M, times the total BvN time. This suggests a significant drop in run time for the subsequent bottleneck matching calls,

matrix	instance	MC64J2	BOTTLED
Bump_2911	**A**	0.56	9.63
	AP	1.22	27.17
	DAE	0.53	9.08
	DAPE	1.70	27.04
	DP(**A**)**E**	9705.93	8.34
	DP(**A**)**PE**	3.24	28.64
cage15	**A**	0.67	10.34
	AP	1.97	20.36
	DAE	0.67	11.33
	DAPE	2.40	20.82
	DP(**A**)**E**	914.74	13.31
	DP(**A**)**PE**	14345.60	22.00
circuit5M	**A**	0.30	3.11
	AP	29.69	8.88
	DAE	0.34	3.10
	DAPE	16.22	9.23
	DP(**A**)**E**	6.07	65.28 ∗
	DP(**A**)**PE**	78.56	90.92 ∗
CurlCurl_4	**A**	0.12	0.83
	AP	0.37	5.04
	DAE	0.12	0.91
	DAPE	0.38	5.05
	DP(**A**)**E**	2055.55	2.58
	DP(**A**)**PE**	866.57	8.35
dielFilterV3real	**A**	0.38	12.58
	AP	0.85	22.57
	DAE	0.38	9.72
	DAPE	0.76	19.35
	DP(**A**)**E**	1170.60	13.24
	DP(**A**)**PE**	144.14	22.10
d_pretok	**A**	9.67	0.53
	AP	6.97	0.85
	DAE	0.12	0.36
	DAPE	0.53	0.64
	DP(**A**)**E**	10.85	2.19
	DP(**A**)**PE**	7.28	2.32
Hamrle3	**A**	136.11	1.90
	AP	15.39	6.52
	DAE	0.11	1.36
	DAPE	0.65	3.16
	DP(**A**)**E**	0.30	1.66
	DP(**A**)**PE**	0.94	4.05
iChem_Jacobian	**A**	27.49	0.14
	AP	149.35	0.60
	DAE	0.02	0.15
	DAPE	0.06	0.78
	DP(**A**)**E**	105.30	3.22
	DP(**A**)**PE**	121.55	4.51
kim2	**A**	55.12	0.59
	AP	103.63	1.96
	DAE	40.32	2.71
	DAPE	47.62	5.05
	DP(**A**)**E**	125.72	2.46
	DP(**A**)**PE**	416.31	5.32

matrix	instance	MC64J2	BOTTLED
mac_econ_fwd500	**A**	0.60	0.13
	AP	1.50	0.42
	DAE	2.05	0.73
	DAPE	6.72	1.10
	DP(**A**)**E**	5.26	0.95
	DP(**A**)**PE**	7.34	1.45
PR02R	**A**	4.48	0.93
	AP	52.07	2.03
	DAE	1.98	1.26
	DAPE	7.60	2.30
	DP(**A**)**E**	39.83	9.73
	DP(**A**)**PE**	64.47	12.74
RM07R	**A**	0.81	4.01
	AP	289.88	6.86
	DAE	19.32	4.67
	DAPE	48.57	8.02
	DP(**A**)**E**	277.83	38.62 ∗
	DP(**A**)**PE**	517.90	53.20 ∗
ss	**A**	3.07	2.73
	AP	2503.86	8.84
	DAE	0.17	1.43
	DAPE	0.56	6.90
	DP(**A**)**E**	865.30	5.81
	DP(**A**)**PE**	2886.73	11.85
torso1	**A**	5.70	0.96
	AP	5.41	1.21
	DAE	13.82	4.65
	DAPE	33.96	6.32
	DP(**A**)**E**	11.23	2.85
	DP(**A**)**PE**	21.50	3.75
vas_stokes_1M	**A**	0.18	3.73
	AP	2041.20	7.16
	DAE	0.18	3.21
	DAPE	2.17	7.19
	DP(**A**)**E**	501.26	4.67
	DP(**A**)**PE**	2510.33	6.87
vas_stokes_2M	**A**	0.32	7.23
	AP	6657.39	14.37
	DAE	0.32	6.10
	DAPE	4.99	13.85
	DP(**A**)**E**	1570.88	14.29
	DP(**A**)**PE**	7749.24	24.10
vas_stokes_4M	**A**	0.65	16.32
	AP	18510.20	32.31
	DAE	0.90	27.41
	DAPE	9645.50	60.95 ∗
	DP(**A**)**E**	1833.95	19.12
	DP(**A**)**PE**	23074.50	30.75

Table 5: Run time, in seconds, of MC64J2 and BOTTLED on six problem instances of 17 matrices.

matrix	instance	BOTTLED	BvN	
			num.perm	time
Bump_2911	**DAE**	9.08	14	20.66
	DP(**A**)**E**	8.34	50	179.88
cage15	**DAE**	11.33	13	20.30
	DP(**A**)**E**	13.31	50	217.76
ss	**DAE**	1.43	13	6.57
	DP(**A**)**E**	5.81	24	67.19
vas_stokes_4M	**DAE**	27.41	41	114.22
	DP(**A**)**E**	19.12	50	256.87

Table 6: Run time, in seconds, of BOTTLED and BvN, and the number of permutation matrices obtained.

which is in large part thanks to the synergy of the decomposition method and BOTTLED, as the preprocessing step is not repeated.

5 Conclusion

We have investigated the problem of finding a maximum bottleneck matching in bipartite graphs. Existing implementations for the problem suffer from unpredictable run time that can often get prohibitively large, i.e., requiring thousands or even tens of thousands of seconds to complete. We have proposed a new algorithm called BOTTLED that converts an inefficient, duality-based approach into an efficient one through a series of theoretical findings. Experimental results show that BOTTLED is almost always faster than the state of the art methods. Furthermore, its run time is reliable and always remain within reasonable time limits. We have also explored its use inside a heuristic for the Birkhoff–von Neumann decomposition of doubly stochastic matrices and experimentally confirmed the suitability of the proposed algorithm for this purpose.

We plan to investigate methods to improve the performance of the proposed algorithm. In particular, it should be possible to consider more nonzeros than the kth largest uncovered edge allows. One can decrease the threshold beyond the kth largest uncovered edge until there are at least k rows in the $(S_R \cup V_R, H_C)$. This requires some extra bookkeeping, which we plan to explore.

References

[1] M. Benzi and B. Uçar. Preconditioning techniques based on the Birkhoff–von Neumann decomposition. *Computational Methods in Applied Mathematics*, 17:201–215, 2017.

[2] G. Birkhoff. Tres observaciones sobre el algebra lineal. *Univ. Nac. Tucumán Rev. Ser. A*, (5):147–150, 1946.

[3] R. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. SIAM, Philadelphia, PA, USA, 2009.

[4] C. Chang, W. Chen, and H. Huang. On service guarantees for input-buffered crossbar switches: A capacity decomposition approach by Birkhoff and von Neumann. In *Quality of Service, 1999. IWQoS '99. 1999 Seventh International Workshop on*, pages 79–86, 1999.

[5] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[6] 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.

[7] I. S. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 20(4):889–901, 1999.

[8] I. S. Duff and J. Koster. On algorithms for permuting large entries to the diagonal of a sparse matrix. *SIAM Journal on Matrix Analysis and Applications*, 22:973–996, 2001.

[9] F. Dufossé, K. Kaya, I. Panagiotas, and B. Uçar. Further notes on Birkhoff–von Neumann decomposition of doubly stochastic matrices. *Linear Algebra and its Applications*, 554:68–78, 2018.

[10] F. Dufossé and B. Uçar. Notes on Birkhoff–von Neumann decomposition of doubly stochastic matrices. *Linear Algebra and its Applications*, 497:108–115, 2016.

[11] A. L. Dulmage and N. S. Mendelsohn. Coverings of bipartite graphs. *Canadian Journal of Mathematics*, 10:517–534, 1958.

[12] H. N. Gabow and R. E. Tarjan. Algorithms for two bottleneck optimization problems. *J. Algorithms*, 9(3):411–417, 1988. Hopcroft-Karp algorithm can be used as an approximation algorithm p415.

[13] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10(37):26–30, 1935.

[14] 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.

[15] 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.

[16] J. Kulkarni, E. Lee, and M. Singh. Minimum Birkhoff-von Neumann decomposition. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 343–354. Springer, 2017.

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

[18] A. P. Punnen and K. Nair. Improved complexity bound for the maximum cardinality bottleneck bipartite matching problem. *Discret. Appl. Math.*, 55(1):91–93, 1994.

[19] R. Sinkhorn and P. Knopp. Concerning nonnegative matrices and doubly stochastic matrices. *Pacific J. Math.*, 21:343–348, 1967.