Engineering fast algorithms for the bottleneck matching problem

Ioannis Panagiotas, Grégoire Pichon, Somesh Singh, Bora Uçar

Engineering fast algorithms for the bottleneck matching problem

Ioannis Panagiotas
Neo4j, Sweden

Grégoire Pichon
Université Claude Bernard Lyon 1 and LIP, France
UMR5668 (CNRS, ENS de Lyon, Inria, UCBL1) France

Somesh Singh
CNRS and LIP, ENS de Lyon, France
UMR5668 (CNRS, ENS de Lyon, Inria, UCBL1) France

Bora Uçar
CNRS and LIP, ENS de Lyon, France
UMR5668 (CNRS, ENS de Lyon, Inria, UCBL1) France

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 time 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.

2012 ACM Subject Classification Theory of computation

Keywords and phrases bipartite graphs, assignment problem, matching

Related Version *Published version*: `https://doi.org/10.4230/LIPIcs.ESA.2023.87`

Supplementary Material Source code
Software: `https://gitlab.inria.fr/bora-ucar/bottled`

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 finding a maximum cardinality matching whose minimum edge weight is the maximum on bipartite graphs with edge weights. 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 motivated by the Birkhoff–von Neumann (BvN) decomposition of doubly stochastic matrices [2], which arises in practical applications [1, 4, 16, 21]. In this case, the bipartite graphs associated with matrices have equal number of vertices on both sides and contain perfect matchings. A known heuristic for the BvN decomposition [10] repeatedly calls a bottleneck matching algorithm on the bipartite graph of a dynamically changing matrix.

	A		**AP**		**S**	
matrix	MC64J2	MC64J3	MC64J2	MC64J3	MC64J2	MC64J3
atmosmodm	0.05	0.26	0.13	0.40	684.87	6487.07
CurlCurl_3	0.05	0.31	0.15	0.48	578.83	2958.07
ss	2.28	0.90	2317.46	1.73	810.10	16501.00
vas_stokes_2M	0.25	1.87	6310.44	727.57	1420.01	4571.14

Table 1 Run time, in seconds, of MC64J2 and MC64J3 on 12 problems. $\mathbf{A}$ is the original matrix, $\mathbf{AP}$ 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 $\mathbf{A}$; and $\mathbf{D}$ and $\mathbf{E}$ are positive diagonal matrices scaling $P(\mathbf{A})$ to be doubly stochastic.

The software MC64 [7, 8] is the state-of-the-art and implements two algorithms, denoted MC64J2 and MC64J3, for the bottleneck matching problem. To the best of our knowledge these are the only available codes 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, especially the newer MC64J2 [8], is well engineered. It works very well for graphs corresponding to matrices from numerical applications. However, it does 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 time 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 the 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 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 $\mathbf{A}$, for $\mathbf{AP}$ where $\mathbf{P}$ is a random permutation, and for the doubly stochastic matrix $\mathbf{S}$ which is obtained by scaling the pattern of $\mathbf{A}$ with Sinkhorn-Knopp algorithm [20].

The bipartite graphs of $\mathbf{A}$ and $\mathbf{AP}$ are the same apart from renumbering of the vertices in one part. While on $\mathbf{A}$ both MC64J2 and MC64J3 are fast, both methods suffer on $\mathbf{AP}$; the run time of MC64J2 is not acceptable for the last two instances, and that of MC64J3 is high for the last one in Table 1. The bottleneck matching problems on $\mathbf{A}$ and $\mathbf{AP}$ are essentially the same, as permuting the columns does not change the values, nor the bottleneck matching and its value. The bipartite graph of $\mathbf{S} = \mathbf{D}P(\mathbf{A})\mathbf{E}$ is the same as that of $\mathbf{A}$ with different edge weights, hence the problems are not equivalent. Now, the run time of both methods is too much for all instances. The wildly varying run time of both MC64 routines on $\mathbf{S}$ in comparison to those on $\mathbf{A}$ further 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 codes in MC64. For this purpose, we study an overlooked alternative from the literature. We make observations that pave the way for an efficient algorithm, implement and compare it against the codes from MC64. We conduct a large set of experiments to show that our approach is usually much faster than MC64 and in

addition exhibits stable and robust performance.

Section 2 gives a brief background and a summary of the known algorithms. Section 3 contains the proposed algorithm. Section 4 presents the experimental results, and Section 5 concludes the paper. The appendix contains detailed description of the experiments and further experiments.

2 Background and related work

A matrix $\mathbf{A}$ with $nnz(\mathbf{A})$ nonzero entries can be represented with a bipartite graph $G = (R \cup C, E)$ where each row of $\mathbf{A}$ corresponds to a unique vertex in R, each column of $\mathbf{A}$ corresponds to a unique vertex in C, and there is an edge (r_i, c_j) whenever $a_{ij} \neq 0$. The number m of edges in G is thus equivalent to $nnz(\mathbf{A})$. When the edges are weighted, the weight of the edge (r_i, c_j) is $|a_{ij}|$, that is the magnitude of the nonzeros of $\mathbf{A}$. For a vertex v, we use $\mathrm{adj}(v)$ 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 *perfect* if it matches all vertices. The *deficiency* of a matching $\mathcal{M}$ is the difference between the maximum cardinality of a matching and $|\mathcal{M}|$. Given a matching $\mathcal{M}$ in the graph G, 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. In a bipartite graph the maximum cardinality of a matching is equal to the minimum cardinality of a vertex cover [3, Th. 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 [18] describe algorithms to permute sparse matrices in a block upper triangular form (BTF):

$$\mathbf{A} = \begin{array}{c} \\ H_R \\ S_R \\ V_R \end{array} \begin{array}{c} \begin{array}{ccc} H_C & S_C & V_C \end{array} \\ \begin{pmatrix} \mathbf{A}_H & * & * \\ O & \mathbf{A}_S & * \\ O & O & \mathbf{A}_V \end{pmatrix} \end{array}. \tag{1}$$

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 as follows

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

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, Alg. 3.1]. As all nonzeros in the BTF (1) are confined in the 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 $\mathcal{C}$ is equal to the maximum cardinality of a matching, it 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.

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 [20]. A *permutation matrix* is a square matrix where each row/column contains 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.

Henceforth, we assume that the given bipartite graph contains perfect matchings. We comment on rectangular matrices and matrices without perfect matchings in Section 3.3.

2.1 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. Burkard et al. [3, Section 6.2.4] describe two other algorithms [12, 19], which are more theoretical.

2.1.1 Shortest-augmenting path based algorithms

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 maximum bottleneck value for any matching that matches the vertices $C' \cup \{c\}$. The process continues until a perfect matching is obtained.

The state-of-the-art implementation in MC64 [8], MC64J2, starts by computing an upper bound ω on the bottleneck value, which is the minimum of maximum in each column and row. It then computes a maximal matching on the graph containing only edge weights no smaller than ω, which is then improved by length-three augmenting paths in a preprocess step. Then, a shortest-augmenting path is sought from each free column vertex to solve the problem. MC64J2 implements an efficient adaptation of Dijkstra's algorithm to find these paths. Depending on the edge weights, the structure of the bipartite graph, or the visit order many edges and vertices may be visited while finding an augmenting path. As seen in Table 1, this can accumulate and result in very long run time.

2.1.2 Threshold-based algorithms

Let G be a weighted bipartite graph. For a value ω, let $G[\omega]$ contain only the edges of G with weight at least ω. Threshold-based algorithms find the largest ω for which $G[\omega]$ has a perfect matching. They do so by considering different values for ω, testing whether $G[\omega]$

has a perfect matching or not and tuning the next ω to a higher or lower value accordingly. MC64's implementation, MC64J3, discusses initialization and algorithmic choices to reduce the number of tests [7]. However, MC64 uses a depth-first-search algorithm to find the augmenting paths for each test which can slow it down, as seen in Table 1.

2.1.3 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 detail as ours improves upon it. Let ω be a value where $G[\omega]$ does not contain a perfect matching. Let $\mathcal{M}$ be a maximum cardinality matching in $G[\omega]$. From $\mathcal{M}$ one can define a minimum vertex cover $\mathcal{C} = H_R \cup S_R \cup V_C$ of $G[\omega]$ 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 : $\mathcal{M}$, a bottleneck perfect matching

Let ω be an upper bound on the bottleneck matching value
$\mathcal{M} \leftarrow \emptyset$
while $|\mathcal{M}| < n$ **do**
  $\mathcal{M} \leftarrow$ a maximum cardinality matching in $G[\omega]$
  if $|\mathcal{M}| < n$ **then**
1 Let $I \subseteq R$ and $J \subseteq C$ be the vertex sets of the associated cover of $G[\omega]$
2 $\omega \leftarrow \max_{i \in R \setminus I, j \in C \setminus J} w_{i,j}$ `/* in G, not G[ω] */`

The maximum cardinality matching in $G[\omega]$ can be found in $O(\sqrt{n}\,\mathrm{nnz}(\mathbf{A}[\omega]))$ 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 $O(\sqrt{n}\,\mathrm{nnz}(\mathbf{A}))$ times the number of iterations. The worst case run time for Algorithm 1 can hence be too high. This is so as a new edge, due to the reduced ω, does not mean one more edge in the maximum matching, and hence the while loop can even run for more than n iterations.

3 The proposed algorithm

Our algorithm is based on Algorithm 1 and integrates threshold techniques. 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 ω be a nonnegative value, $G[\omega]$ be as before, and the bottleneck matching value $b^\star$ be the largest ω for which $G[\omega]$ has a perfect matching. We call a value ω *safe*, when $\omega \geq b^\star$. Algorithm 1 produces decreasing safe values that converge to $b^\star$. We make a series of observations to find $b^\star$ faster. The first observation is that there are several minimum vertex covers associated with a given maximum cardinality matching. Using the BTF (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 can be different. This leads to the following proposition.

▶ **Proposition 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 $G[\omega]$ revealed by the BTF (1). Let ω_1 and ω_2 be the maximum values defined in Line 2 of Algorithm 1 for $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Then, $\min(\omega_1, \omega_2)$ is safe.*

Proof. Let ω be the current value, and $\omega_1 < \omega_2$ without loss of generality. This means that ω_2 is in $\mathbf{A}(V_R, S_C)$ and all entries in $\mathbf{A}(V_R, H_C)$ are smaller than ω_2. The maximum cardinality of a matching in $G[\omega_2]$ cannot be larger than that in $G[\omega]$, as the set $\mathcal{C}_1 = H_R \cup S_R \cup V_C$ still covers all edges of $G[\omega_2]$, including those that arise in $\mathbf{A}(S_R, H_C)$. Hence the cover $\mathcal{C}_1$ can be used to get the next ω in Line 2, which concludes the proof. ◀

Based on Proposition 1, one can use the smaller of the largest uncovered element in $(S_R \cup V_R, H_C)$ and that in $(V_R, H_C \cup S_C)$. We propose to exploit the two identified covers as much as possible for a faster convergence of ω to $b^\star$. As we reason in Lemma 2 below, one can find a tighter bound on $b^\star$, depending on the deficiency of the current matching as well as the (original) adjacencies of the vertices in the identified covers.

▶ **Lemma 2.** *Let $\mathcal{M}$ be a maximum cardinality matching in $G[\omega]$ with a deficiency of k in G; $\mathbf{A}[\omega]$ and $\mathbf{A}$ be, respectively, the matrices associated with $G[\omega]$ and G; $\mathcal{C} = H_R \cup S_C \cup V_C$ be a minimum vertex cover of $G[\omega]$ associated with $\mathcal{M}$ when $\mathbf{A}[\omega]$ is permuted in a BTF (1); Let ω_k^r be the kth largest element in $\bigcup_{i \in S_R \cup V_R} \max\{w_{i,j} : j \in H_C\}$, and ω_k^c the kth largest element in $\bigcup_{j \in H_C} \max\{w_{i,j} : i \in S_R \cup V_R\}$. Then, $\omega_k = \min(\omega_k^r, \omega_k^c)$ is safe.*

Proof. Consider first the set H_C of columns, and note that $|H_C| - |H_R| = k$ as all other columns are matched. 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 $\mathbf{A}(S_R \cup V_R, H_C)$. The element ω_k^r from $\bigcup_{i \in S_R \cup V_R} \max\{w_{i,j} : j \in H_C\}$ is safe as any value greater than that will cover less than k rows and Hall's conditions cannot be satisfied. A similar argument applies to ω_k^c by considering the set $S_R \cup V_R$ of rows. In $\mathbf{A}[\omega]$ we have $\text{adj}(S_R \cup V_R) = S_C \cup V_C$, and $|S_R \cup V_R| - |S_C \cup V_C| = k$. The element ω_k^c must be safe since we need at least k nonzero columns in $\mathbf{A}(S_R \cup V_R, H_C)$. The value ω_k is thus safe as the minimum of two safe values. ◀

One can identify several minimum covers, collect the kth largest uncovered element with respect to each, and use the minimum of the collected elements as the next ω. As finding these minimum covers can be expensive, we propose using the two which are readily revealed by the BTF. That is, we use $\omega = \min(\omega_1, \omega_2)$ where

$$\begin{aligned} \omega_1^r, \omega_1^c &= k\text{-th largest row and column maximum entries in } \mathbf{A}(S_R \cup V_R, H_C)\,, \\ \omega_1 &= \min(\omega_1^r, \omega_1^c)\,, \qquad (2) \\ \omega_2^r, \omega_2^c &= k\text{-th largest row and column maximum entries in } \mathbf{A}(V_R, H_C \cup S_C)\,, \\ \omega_2 &= \min(\omega_2^r, \omega_2^c)\,. \qquad (3) \end{aligned}$$

This corresponds to applying Hall's theorem to the sets H_C and $H_C \cup S_C$ of columns and to the sets V_R and $S_R \cup V_R$ of rows.

3.1 Putting it all together

The proposed algorithm BOTTLED is shown in Algorithm 2. The input is a sparse matrix represented in the compressed storage by columns (CSC) format. BOTTLED creates a compressed storage by rows (CSR) representation of the input matrix. It then sorts the nonzeros in each row and each column in non-increasing order of their values. Then the threshold ω is initialized as the minimum of the $2n$ nonzero values consisting of the maximum in each row and maximum in each column. The algorithm then updates the threshold ω in a while-loop as in Algorithm 1. In 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 (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 $\omega \leftarrow$ min of the maximum in each row, maximum in each column
$G[\omega] \leftarrow (R \cup C, \emptyset)$ and $\mathcal{M} \leftarrow \emptyset$
while $|\mathcal{M}| < n$ **do**
2 for each i, release new $a_{ij} \geq \omega$ and for each j release new $a_{ij} \geq \omega$ into $G[\omega]$
if $|\mathcal{M}| = n - 1$ **then**
$\mathcal{M} \leftarrow \text{SAP}(G, \mathcal{M}, c)$ with the last free vertex c
else
3 $\mathcal{M}' \leftarrow \text{MaximumCardinalityMatching}(G[\omega], \mathcal{M})$
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
$\mathcal{M} \leftarrow \mathcal{M}'$
$\langle H_R, S_R, V_R, H_C, S_C, V_C \rangle \leftarrow \text{DM-dec}(G[\omega], \mathcal{M})$
5 $\omega \leftarrow \min(\omega_1, \omega_2)$ with ω_1 as in (2) and ω_2 as in (3)

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

In Algorithm 2, $\text{SAP}(G, \mathcal{M}, c)$ refers to the algorithm summarized in Section 2.1.1. As stated before, SAP needs the current matching to have the bottleneck value among all 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 ω 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 an augmenting path, we choose $c \in C$ whose largest edge weight not included in the current $G[\omega]$ is minimum. Matching c will lead to a reduced ω, and the reduction will hopefully be large with this choice of c (some empirical results are in the appendix of the related version). 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 $G[\omega]$.

At Line 5 of Algorithm 2, we use a binary heap with a limit k on its size. For ω_1^r, the nonzeros of each row in $S_R \cup V_R$ with value smaller than ω are visited, and the largest element is used as a key in the heap. When the heap is full, keys are added only if they are larger than the current minimum, which is then removed. The minimum of the heap is returned as

ω_1^r. Other quantities of (2) and (3) are computed similarly. All nonzeros of $\mathbf{A} - \mathbf{A}[\omega]$ in the rows $S_R \cup V_R$ and columns H_C can be visited and the heap operations can be performed in $O(\text{nnz}(\mathbf{A}(S_R \cup V_R, H_C)) + (|H_C| + |S_R \cup V_R|) \log k)$ time. A similar analysis holds for $\mathbf{A}(S_R \cup V_R, H_C)$. The run time of one iteration of the while loop is thus dominated by that of the maximum cardinality matching algorithm. We do not have an estimate on the number of iterations of the while-loop; in the empirical results in Section 4.3 the number is less than what a binary search approach would yield.

3.2 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 doubly stochastic matrix $\mathbf{A}$ 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 for $G[\omega]$, and avoid the preprocessing in Algorithm 2 at subsequent invocations.

3.3 Rectangular matrices or matrices 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, which was our motivation. 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 ω 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, with $n_R \geq n$ and 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 in a bottleneck maximum cardinality matching. We do not know of a suitable fix for this. MC64J3 on the other hand works correctly. The presented algorithm BOTTLED needs four minor modifications to handle this general case: (i) the maximum size of a matching n' should be computed at the beginning; (ii) the initialization should use the smallest of 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 Line 5, 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 is generally faster than MC64J3; 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, the run time of MC64J3 can be too large to experiment (this happens more frequently than with MC64J2). That is why we have implemented a threshold-based approach, referred here as THRESH, using the same code base of BOTTLED, and use it in our experiments instead of MC64J3. THRESH uses the initialization procedure common to MC64J3 and BOTTLED to find a large initial matching and an upper bound ω on the bottleneck value. If this matching is not maximum, then the edge weights are sorted to find the bottleneck value by a binary search between the smallest edge weight and the initial value ω. The binary search uses only the available edge weights and each time half of the edges are discarded. Our codes, written in C, are available at `https://gitlab.inria.fr/bora-ucar/bottled`; the codes used in the experiments are elsewhere [17].

The next subsection describes the data set, and Section 4.2 conducts a performance analysis of BOTTLED. Section 4.3 compares BOTTLED with MC64J2 and THRESH. Last, some experiments using BOTTLED within a BvN decomposition heuristic are discussed in Section 4.4. Appendix of the related version contains detailed information about experiments.

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 SuiteSparse [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 **A**-type instance corresponds to the bipartite graph of the original matrix with the magnitudes of the entries as edge weights. The **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 [20] algorithm, and the other three types of instances are obtained from the first three by random column permutations. We discarded the instances **A** and **AP** for 0-1 matrices; for the same set of matrices **DAE**-type and $\mathbf{D}P(\mathbf{A})\mathbf{E}$-type instances are identical, and hence we kept only one of them. We discarded the instances in which the initialization algorithm found the bottleneck value, in which case all three methods are equivalent. We report the experiments with 14 **A**-type, 18 **DAE**-type, 58 $\mathbf{D}P(\mathbf{A})\mathbf{E}$-type instances, and the same number of instances with column permutations. For each instance, each algorithm is run five times and the geometric mean of the run time is reported as that algorithm's performance on that instance; when column permutations are applied, these correspond to five different permutations. When comparing two algorithms' run time, we do not include cases where both algorithms run in less than one second (as both are very small and the difference between the algorithms is insignificant). Appendix of the related version contains all the results.

We carry out the experiments on a machine having Intel(R) Xeon(R) CPU E7-8890 v4 with a clock-speed of 2.20 GHz, and 1.5TB memory. The machine runs Debian GNU/Linux 11 (64 bit). All the codes are compiled with `GCC` version 10.2.1, with option `-O3`. We ran MatchMaker [6, 15] to verify that there were perfect matchings. We used MatchMaker with options "no cheap matching", "Push-Relabel + fairness" as the core algorithm. For the sake of completeness, we report that the maximum run time of MatchMaker for any matrix was 2.65 seconds for `vas_stokes_4M` and 28.19 for the column permuted version of the same

matrix.

4.2 Performance analysis of BOTTLED

We first analyze the percentage of the total run time of BOTTLED spent in the preprocessing step—creating the CSR representation or sorting the entries. Table 2 summarizes the results for six different problem instances, where BOTTLED took more than one second for **A**-, **DAE**- and/or **D**P(**A**)**E**-type instances. This table presents the geometric mean of the percentage of the time spent creating the CSR representation and sorting (with respect to the total time of BOTTLED). The row "tTime" contains the geometric mean of the run time 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 **D**P(**A**)**PE** is obtained by taking the geometric mean of ratio of the run time of BOTTLED on the 33 **D**P(**A**)**PE** instances to that on the 33 **D**P(**A**)**E** instances.

As we can see from Table 2, the average run time of BOTTLED on any of the six problem instance types is below 10 seconds. A first observation is that for all instances, the two preprocessing steps account for a non-negligible part of the total run time. At two extremes, they take 7% and 37% of the total time in **D**P(**A**)**E** and **DAPE** instances, respectively. We further observe that the absolute run time of CSR and sorting increases for the permuted instances. The percentage of the total time spent in CSR also increases for the permuted instances while that of sorting either increases or remains the same. If the CSR representation is available, its creation can be skipped and one can reduce the run time considerably. As discussed in Section 3.2, 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	**AP**	**DAE**	**DAPE**	**D**P(**A**)**E**	**D**P(**A**)**PE**
CSR	5%	13%	7%	25%	4%	17%
sort	5%	5%	10%	12%	3%	3%
tTime	6.54(s)	1.28	3.43(s)	1.02	3.40(s)	1.28

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 time of BOTTLED on the instances **A**, **DAE** and **D**P(**A**)**E** in seconds, and the ratios of the others to their counterparts.

matrix	instance-type	tTime	CSR	sort
vas_stokes_4M	**A**	14.81	6.70	3.17
	AP	32.36	20.13	5.51
	DAE	23.44	6.93	3.66
	DAPE	52.99	19.93	5.57
vas_stokes_2M	**D**P(**A**)**E**	11.67	2.96	2.01
	DP(**A**)**PE**	19.30	8.81	2.57

Table 3 Breakdown of the total time (tTime) of BOTTLED in seconds.

Another observation from Table 2 is that the run time of BOTTLED consistently increases for the permuted instances. To put this into perspective, we present the run time of BOTTLED on a few instances in Table 3. We see that the increase in run time for the permuted matrices can be attributed, in part, to the creation of the CSR. For example, for vas_stokes_4M, excluding CSR and sort times from the total time yields 4.94 and 6.72 seconds for the while loop for **A**- and **AP**-type respectively. A similar calculation for the pairs **D**P(**A**)**E** and **D**P(**A**)**PE** for vas_stokes_2M shows that the while loop takes 6.7 and 7.92 seconds, respectively. In the instance pairs **DAE** and **DAPE** for vas_stokes_4M, the increase in the CSR time is still a contributing factor. From these two tables, we conclude that BOTTLED's preprocessing takes up a significant portion of the total run time.

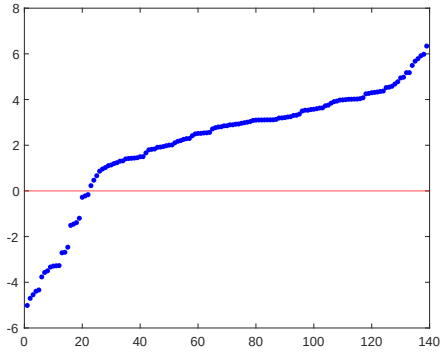
We have also investigated the number of iterations of the while-loop of BOTTLED in different types of instances to see how stable and robust it is with respect to random column permutations. The number of iterations of the while loop were almost always the same for

all 90 instances; in a few cases there was a difference of one in the number of iterations. The small changes in the number of iterations confirm the robustness of BOTTLED.

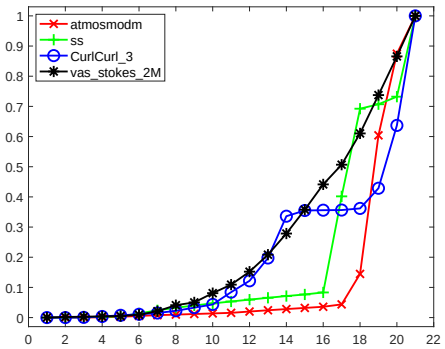
4.3 Comparison of different algorithms

We compared BOTTLED against MC64J2 and THRESH. We first provide a broad overview of the comparisons in Table 4, which lists the geometric mean of the run time of the three methods as well as their maximum run time in different problem instance types (all instances). As can be seen, BOTTLED exhibits the best performance overall, despite being slightly slower on average for **A**-, **DAE**-, **DAPE**- instance types than MC64J2—the margin is not large enough for MC64J2 to make up for the performance loss on the other instance types. Furthermore, BOTTLED's maximum run time is consistently smaller than those of both MC64J2 and THRESH. In contrast, the maximum run time of MC64J2 is prohibitive in all but the **A** and **DAE**-type instances. While THRESH avoids the prohibitive run time of its equivalent MC64J3 (see Table 1) by using a better cardinality matching algorithm [15], its maximum run time is still noticeably larger than that of BOTTLED in all but two cases.

We now look more into the performance of BOTTLED versus that of MC64J2. Figure 1a shows the natural logarithm (in the y-axis) of the ratio of the run time of MC64J2 to that of BOTTLED for the instances on which either method has a run time greater than 1 second. Here, MC64J2 is faster than BOTTLED on 22 instances, and BOTTLED is faster on the remaining 117 instances. In all 139 instances, the geometric mean of the ratio of the run time of MC64J2 to that of BOTTLED is 8.5, confirming that BOTTLED is faster than MC64J2 in average.



(a) The natural logarithm of the ratio of the run time of MC64J2 to that of BOTTLED in the y-axis in sorted order on 139 problem instances in the x-axis.



(b) The run time behavior of MC64J2 on four **D**P(**A**)**E**-type instances measured at 20 uniform steps (x-axis), normalized to the total time (y-axis).

Figure 1 Performance comparison between MC64J2 and BOTTLED, and investigation on MC64J2.

In order to put these numbers into a perspective with the run time, we present Table 5. Table 5 lists six matrices in which the ratio of the run time of MC64J2 to that of BOTTLED was the smallest for certain instance types with or without permutation. It next lists six matrices with different instance types in which the ratio of the run time of MC64J2 to that of BOTTLED was the largest for the original matrices and three for the permuted ones (three others were already in the list). Here we see that MC64J2 can have equivalently good performance on both the **A**- and **AP**-type instances (see the first block in Table 5). Still, there were some **AP** instances where MC64J2 ran prohibitively long (see the last two rows with ss and vas_stokes_4M). MC64J2 also struggled on several **D**P(**A**)**E** and **D**P(**A**)**PE** instances as seen in the second block. The prohibitively high run time of MC64J2 highlight

		A	**AP**	**DAE**	**DAPE**	**DP(A)E**	**DP(A)PE**
geomean	MC64J2	0.49	6.23	0.16	0.74	12.47	20.16
	THRESH	2.58	3.80	1.09	1.54	2.24	3.16
	BOTTLED	1.23	2.38	0.44	0.81	1.23	2.07
maximum	MC64J2	123.04	18054.00	31.42	9139.63	1419.99	6813.20
	THRESH	43.55	64.90	59.42	91.74	63.51	77.57
	BOTTLED	14.52	31.81	23.22	51.55	58.64	76.20

Table 4 The geometric mean and the maximum of the run time of the three methods on six different instance types.

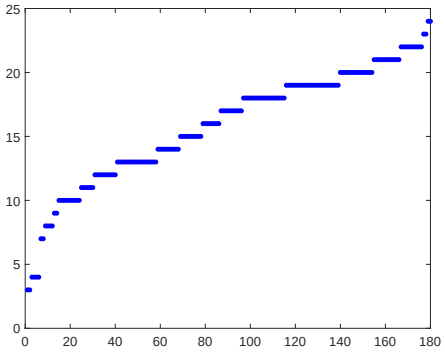
the issue in the approaches based on the shortest augmenting paths: one may visit many edges and vertices to find augmenting paths at different stages during the execution. As such a behavior is also seen for DFS-based-cardinality matching algorithms [6], it cannot be attributed on the particular implementation of the shortest path algorithm in MC64J2.

Figure 1b shows the run time behavior of MC64J2 on the four **D**P(**A**)**E**-type instances of Table 1. The total number of augmentations is divided by 20 to obtain a step size, the run time is measured after each step and normalized by the total time. As this figure shows, in some instances the augmentations took about the same time all throughout, whereas in others later augmentations took more time. As the greedy matching approach of MC64J2 does not always find a maximum matching, it can needlessly result in many additional augmentations. Those augmentations may lead to large run time, not solely because of their number. For example, in the **D**P(**A**)**E**-type instance of CurlCurl_3, a square matrix with $n = 1219574$ rows and nnz $= 13544618$ nonzeros, there are 20040 augmentations with a total run time of about 600 seconds, even though the maximum cardinality matching on $G[\omega_0]$ has a deficiency of one. Obviously, detecting this would lead to much better run time. On the other hand, for the **D**P(**A**)**E**-type instance of rajat31, a square martrix with $n = 4690002$ rows and nnz $= 20316253$ nonzeros, MC64J2 needs 1562500 augmentations (its greedy approach finds a maximum cardinality matching initially), and the whole run time is 48.39 seconds. As rajat31 is much bigger than CurlCurl_3 and needs more augmentations, its shorter run time attests that the process of augmenting one-by-one can take large time due to instance specific properties. It hence cannot constitute a reliable method for the bottleneck matching problem in general. In passing we note that MC64J2 is well-engineered and is faster than using our own SAP implementation for the augmentations.

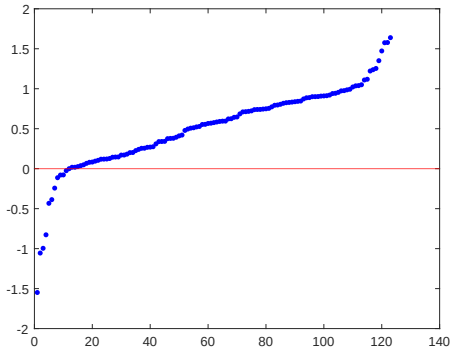
We compare now BOTTLED and THRESH. In total, BOTTLED beat THRESH in 165 instances out of 180. The largest differences (in seconds) in favor of THRESH were in the **A**- and **AP**-type instances of c-73b and c-73. THRESH obtained a run time of 2.88 and 3.48 seconds on c-73b, and 6.35 and 5.11 seconds on c-73. On these instances BOTTLED was relatively close to THRESH (see Table 5)—the largest difference is 10.67 seconds on the **A**-type instance of c-73b. Since both methods utilize the same core matching algorithm, the superiority of BOTTLED over THRESH should come from doing fewer iterations. Figure 2a supports this reasoning by plotting the difference between the number of iterations of THRESH and that of BOTTLED in nondecreasing order. As seen in this figure, THRESH's number of iterations is always larger than BOTTLED's; the theoretical observations of Section 3 translate to practical gains. We compute the ratio of the run time of THRESH to that of BOTTLED for the instances on which either THRESH or BOTTLED has a run time greater than 1 second (123 instances), and present the natural logarithm of this ratio in the y-axis in Figure 2b. As seen in this figure, BOTTLED fares better than THRESH in the majority of cases.

matrix	type	MC64J2	BOTTLED	type	MC64J2	BOTTLED
c-73b	**A**	0.09	13.55	**AP**	0.10	9.42
c-73	**A**	0.19	14.52	**AP**	0.34	14.68
dielFilterV3real	**A**	0.33	11.72	**AP**	0.81	21.48
boyd2	**D**P(**A**)**E**	0.03	3.31	**D**P(**A**)**PE**	0.07	1.84
boyd2	**DAE**	0.03	2.43	**DAPE**	0.10	1.50
dielFilterV3clx	**A**	0.12	3.97	**AP**	0.49	7.19
atmosmodd	**D**P(**A**)**E**	555.76	7.01	**D**P(**A**)**PE**	1757.81	12.11
vas_stokes_1M	**D**P(**A**)**E**	452.36	4.20	**D**P(**A**)**PE**	2427.71	7.42
vas_stokes_2M	**D**P(**A**)**E**	1419.99	11.93	**D**P(**A**)**PE**	6813.20	18.37
ss	**D**P(**A**)**E**	824.91	4.66	**D**P(**A**)**PE**	2621.10	10.79
CurlCurl_3	**D**P(**A**)**E**	610.16	1.54	**D**P(**A**)**PE**	211.56	3.80
atmosmodj	**D**P(**A**)**E**	565.60	7.27	**D**P(**A**)**PE**	1683.32	11.97
vas_stokes_4M	**DAE**	0.83	23.22	**DAPE**	9139.63	51.55
ss	**A**	2.25	2.65	**AP**	2532.09	8.65
vas_stokes_4M	**A**	0.53	14.23	**AP**	18054.00	31.81

Table 5 The run time of MC64J2 and BOTTLED on selected instances, where their ratios were the lowest or the highest in different problem instance types or the permuted versions.



(a) The number of iterations of THRESH minus that of BOTTLED in the y-axis in sorted order on all 180 instances.



(b) The natural logarithm of the ratio of the run time of THRESH to that of BOTTLED in the y-axis in sorted order on 123 instances in the x-axis.

Figure 2 Performance comparison between THRESH and BOTTLED.

4.4 Inside a BvN decomposition method

Table 6 presents the run time of the BvN decomposition method on the **DAE**- and **D**P(**A**)**E**-type instances of four matrices (obtained with 2500 scaling iterations). We run the BvN decomposition method until 50 permutation matrices or a coefficient of 0.92 are obtained. As each permutation matrix is obtained by a call to BOTTLED, we show their number in the column "num.perm". Further, the table also presents the time for the initial preprocessing of BOTTLED, and the maximum time taken in a call to BOTTLED subsequently for obtaining a permutation matrix. We observe that the maximum run time of BOTTLED, in an iteration, multiplied by the number of calls is at least 2.92 (for **DAE** of ss) and up to 10.67 (for **D**P(**A**)**E** of ss) times the total BvN time. This suggests that the run time of the subsequent bottleneck matching calls reduces appreciably, and BOTTLED works well inside the decomposition method by avoiding the preprocessing.

matrix	instance	BOTTLED initialization	longest run time of BOTTLED in an iteration	BvN num.perm	BvN time
atmosmodm	**DAE**	0.22	58.73	46	452.14
	DP(**A**)**E**	0.21	14.43	50	200.80
CurlCurl_3	**DAE**	0.35	17.30	50	183.43
	DP(**A**)**E**	0.34	4.33	50	68.45
ss	**DAE**	1.15	13.37	50	228.92
	DP(**A**)**E**	1.46	39.77	50	186.31
vas_stokes_2M	**DAE**	5.04	4.74	50	78.32
	DP(**A**)**E**	5.10	4.87	50	74.36

Table 6 Run time, in seconds, of BOTTLED and the BvN decomposition heuristic, along with the number of permutation matrices in the decomposition.

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 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 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 remains 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 established the suitability of the proposed algorithm for this purpose.

Currently the proposed approach resorts to an augmenting-path-based method in few corner cases and only when there are perfect matchings. We plan to explore the possibility to use them more effectively, along with potential data reduction rules.

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. doi:http://dx.doi.org/10.1137/S0895479899358443.

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. URL: http://www.sciencedirect.com/science/article/pii/S0024379518302568, doi:https://doi.org/10.1016/j.laa.2018.05.017.

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. URL: http://www.sciencedirect.com/science/article/pii/S0024379516001257, doi:http://dx.doi.org/10.1016/j.laa.2016.02.023.

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. arXiv:http://jlms.oxfordjournals.org/cgi/reprint/s1-10/37/26.pdf.

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 I. Panagiotas, G. Pichon, S. Singh, and B. Uçar. Bottled: Fast algorithms for the bottleneck matching problem, April 2023. doi:10.5281/zenodo.7871464.

18 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. URL: http://doi.acm.org/10.1145/98267.98287, doi:10.1145/98267.98287.

19 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.

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

21 V. Valls, G. Iosifidis, and L. Tassiulas. Birkhoff's decomposition revisited: Sparse scheduling for high-speed circuit switches. *IEEE/ACM Transactions on Networking*, 29(6):2399–2412, 2021. doi:10.1109/TNET.2021.3088327.

A Description of the appendix

A detailed description of the data set is given in Appendix B. This same section also contains the whole set of experiments with MC64J2, THRESH, and BOTTLED on all the data set. Appendix C contains an investigation of the parts of the proposed BOTTLED algorithm, where we present results to assess the effects of using the shortest-augmenting path methods.

B Detailed tables

We give comprehensive results for the presented experiments. We start with the properties of matrices used in experiments in Table 7 for the **A**-type problem instances, in Table 8 for the **DAE**-type problem instances, and in Table 10 for the **D**P(**A**)**E**-type problem instances. Recall that there are 14 **A**-type, 18 **DAE**-type, 58 **D**P(**A**)**E**-type instances, and the same number of instances with column permutations, as we discarded the instances in which the initialization algorithm found the bottleneck value. In these three tables, we give the number of rows/columns and the number of nonzeros of the matrices; some are repeated in different tables for the sake of convenience. We then give information about the graph $G[\omega_0]$, where ω_0 is the value obtained in the initialization (the minimum of maximum in each row and in each column): (i) the number of edges in $G[\omega_0]$; (ii) the deficiency obtained by the fast greedy algorithm of MC64J2 under the header "def.$_{\text{MC64J2}}$"; and (iii) the difference between n and the maximum cardinality of a matching in $G[\omega_0]$. The following three tables Table 11, Table 12 and Table 13 give the run time of MC64J2, THRESH, and BOTTLED on, respectively, (i) **A**- and **AP**-type instances; (ii) **DAE**- and **DAPE**-type instances; and (iii) **D**P(**A**)**E**- and **D**P(**A**)**PE**-type instances.

			$G[\omega_0]$		
matrix	n	nnz	#Edges	def.$_{\text{MC64J2}}$	def.
boyd2	466316	1500397	1220614	93252	93252
c-73	169422	1279274	701684	3440	3410
c-73b	169422	1279274	409370	3004	3004
cont-300	180895	988195	896402	453	299
d_pretok	182730	1641672	638067	52425	52419
dielFilterV3clx	420408	32886208	18823657	7	5
dielFilterV3real	1102824	89306020	54707197	35	18
Goodwin_095	100037	3226066	2705970	94	94
Goodwin_127	178437	5778545	4848818	126	126
Hamrle3	1447360	5514242	3664040	214288	202281
PR02R	161070	8185136	3410413	481	11
ss	1652680	34753577	9288800	24964	24964
turon_m	189924	1690876	546760	72826	72786
vas_stokes_4M	4382246	131577616	94445792	4	4

Table 7 $n \times n$ matrices with nnz nonzeros, the number of edges in the graph $G[\omega_0]$, where ω_0 is the first bottleneck value of the **A**-type instances, the deficiency " def.$_{\text{MC64J2}}$" of the initialization of MC64J2, and the deficiency "def." of the maximum cardinality of a matching in $G[\omega_0]$ with respect to that of G.

C Investigating the effect of resorting to SAP in BOTTLED

We 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 2 of the **A**-type, 8 of the **DAE**-type, and 16 of the **D**P(**A**)**E**-type instances. We present results with six instances in Table 9, where we give the

matrix	n	nnz	$G[\omega_0]$ #Edges	$G[\omega_0]$ def.$_{\text{MC64J2}}$	$G[\omega_0]$ def.
analytics	303813	2006126	824483	20	3
boyd2	466316	1500397	1278564	10	2
c-73	169422	1279274	882004	383	1
crashbasis	160000	1750416	163658	2	2
d_pretok	182730	1641672	299061	4103	3917
Hamrle3	1447360	5514242	2569988	18499	17456
kim2	456976	11330020	1687449	2643	2628
lung2	109460	492564	110229	2	2
mac_econ_fwd500	206500	1273389	469290	11094	4413
mc2depi	525825	2100225	656454	1	1
PR02R	161070	8185136	514760	966	59
RM07R	381689	37464962	1706225	4231	7
stomach	213360	3021648	220920	35	15
torso1	116158	8516500	1023697	1423	1024
torso3	259156	4429042	263564	1	1
TSOPF_FS_b39_c30	120216	3121160	390053	825	9
turon_m	189924	1690876	399669	781	474
vas_stokes_4M	4382246	131577616	18666723	172	170

Table 8 $n \times n$ matrices with nnz nonzeros, the number of edges in the graph $G[\omega_0]$, where ω_0 is the first bottleneck value of the **DAE**-type instances, the deficiency " def.$_{\text{MC64J2}}$" of the initialization of MC64J2, and the deficiency "def." of the maximum cardinality of a matching in $G[\omega_0]$ with respect to that of G.

number of iterations of the while-loop when SAP is not called (no aug.), when the vertex selection is not applied within SAP (no.vs, a vertex with the largest value is selected instead), and with the presented version of BOTTLED. As seen in Table 9, resorting to SAP in general helps, and the vertex selection heuristic makes BOTTLED more robust; for example in c-73 not using SAP results in 63 additional iterations, and in lung2 not selecting the vertex with the minimum value results in 46 additional iterations.

matrix	instance	no aug.	with aug. no.vs	with aug. BOTTLED
c-73b	**A**	38	10	10
c-73	**DAE**	66	3	3
Hamrle3	**D**P(**A**)**E**	22	11	11
lung2	**D**P(**A**)**E**	15	55	9
PR02R	**DAE**	19	14	16
cont-300	**D**P(**A**)**E**	8	10	10

Table 9 The number of iterations of the while loop, without augmenting paths, with augmenting paths but without vertex selection heuristic, and the presented version of BOTTLED.

			$G[\omega_0]$		
matrix	n	nnz	#Edges	def.$_{\text{MC64J2}}$	def.
af_0_k101	503625	17550675	12175825	7939	6520
af_1_k101	503625	17550675	12175825	7939	6520
af_2_k101	503625	17550675	12175825	7939	6520
af_3_k101	503625	17550675	12175825	7939	6520
af_4_k101	503625	17550675	12175825	7939	6520
af_5_k101	503625	17550675	12175825	7939	6520
analytics	303813	2006126	1575964	665	1
atmosmodd	1270432	8814880	3469408	60277	59496
atmosmodj	1270432	8814880	3469408	60277	59496
atmosmodl	1489752	10319760	2836248	86407	85376
atmosmodm	1489752	10319760	2836248	86407	85376
BenElechi1	245874	13150496	9301880	7097	4470
boyd2	466316	1500397	1313873	10	2
Chevron3	381381	3413113	2849503	2695	2672
Chevron4	711450	6376412	5586092	3741	3718
circuit5M	5558326	59524291	44231380	1448	3
CO	221119	7666057	7620031	4566	1
cont-300	180895	988195	813457	2239	1134
crashbasis	160000	1750416	1292128	1490	1456
CurlCurl_3	1219574	13544618	12157436	20040	1
d_pretok	182730	1641672	516597	75581	75581
dc1	116835	766396	485343	1	1
dc2	116835	766396	485343	1	1
dc3	116835	766396	485343	1	1
ecology1	1000000	4996000	4623256	3947	3944
ecology2	999999	4995991	4616887	4067	4064
FEM_3D_thermal2	147900	3489300	1890900	1205	1200
Ga10As10H30	113081	6115633	3692549	2026	2
Ga19As19H42	133123	8884839	4261426	1792	5
Ge99H100	112985	8451395	3395058	4606	67
Goodwin_095	100037	3226066	1001631	8502	8210
Goodwin_127	178437	5778545	1841347	15260	14988
Hamrle3	1447360	5514242	2581382	123188	122750
Hardesty1	938905	12143314	10661294	6394	4922
hcircuit	105676	513072	506361	1	1
iChem_Jacobian	274087	4137369	1716099	96864	88302
kim2	456976	11330020	8963433	3974	2604
Lin	256000	1766400	541648	15933	15576
lung2	109460	492564	267598	887	176
mac_econ_fwd500	206500	1273389	702581	12009	1005
majorbasis	160000	1750416	1292128	1490	1456
mc2depi	525825	2100225	1939125	3023	3016
parabolic_fem	525825	3674625	3288209	8798	2996
PR02R	161070	8185136	3596918	38349	37621
rajat31	4690002	20316253	7815002	1562500	1562500
RM07R	381689	37464962	35794471	5406	1555
Si41Ge41H72	185639	15011265	5791343	4724	41
ss	1652680	34753577	21786247	32086	1
ss1	205282	845089	590921	7862	3175
stomach	213360	3021648	1712993	42033	41776
t2em	921632	4590832	4149720	4512	4512
tmt_unsym	917825	4584801	4278035	2986	2599
torso1	116158	8516500	1840272	1799	1791
torso2	115967	1033473	781379	3861	2618
turon_m	189924	1690876	534458	77704	77704
vas_stokes_1M	1090664	34767207	23969241	3175	6
vas_stokes_2M	2146677	65129037	47869836	5686	71
xenon2	157464	3866688	1945788	28313	25849

Table 10 $n \times n$ matrices with nnz nonzeros, the number of edges in the graph $G[\omega_0]$, where ω_0 is the first bottleneck value of the $\mathbf{D}P(\mathbf{A})\mathbf{E}$-type instances, the deficiency " def.$_{\text{MC64J2}}$" of the initialization of MC64J2, and the deficiency "def." of the maximum cardinality of a matching in $G[\omega_0]$ with respect to that of G.

	A-type			**AP**-type		
matrix	MC64J2	THRESH	BOTTLED	MC64J2	THRESH	BOTTLED
boyd2	0.02	0.25	0.08	0.11	0.59	0.28
c-73	0.19	6.35	14.52	0.34	5.11	14.68
c-73b	0.09	2.88	13.55	0.10	3.48	9.42
cont-300	1.51	0.31	0.10	3.89	0.45	0.16
d_pretok	7.34	0.46	0.38	3.91	0.70	0.58
dielFilterV3clx	0.12	9.66	3.97	0.49	13.03	7.19
dielFilterV3real	0.33	28.04	11.72	0.81	38.78	21.48
Goodwin_095	0.02	0.61	0.12	4.76	0.90	0.29
Goodwin_127	0.03	1.09	0.25	21.65	2.01	0.75
Hamrle3	123.04	7.85	1.62	13.86	14.34	5.31
PR02R	3.59	2.85	0.84	42.93	3.63	1.73
ss	2.25	10.23	2.65	2532.09	17.17	8.65
turon_m	0.89	0.49	0.41	2.10	0.75	0.61
vas_stokes_4M	0.53	43.55	14.23	18054.00	64.90	31.81
geomean	0.49	2.58	1.23	6.23	3.80	2.38

Table 11 Run time (in seconds) of MC64J2, THRESH, and BOTTLED on all **A**- and **AP**-type instances. The geometric mean of the run time of MC64J2, THRESH, and BOTTLED over all **A**- and **AP**-type instances are 1.74 3.13 and 1.71, respectively.

	DAE-type			**DAPE**-type		
matrix	MC64J2	THRESH	BOTTLED	MC64J2	THRESH	BOTTLED
analytics	0.01	0.42	0.16	0.06	0.52	0.21
boyd2	0.03	2.37	2.43	0.10	1.63	1.50
c-73	0.08	0.30	0.12	0.15	0.38	0.15
crashbasis	0.01	0.27	0.06	0.02	0.54	0.24
d_pretok	0.09	0.43	0.35	0.40	0.60	0.60
Hamrle3	0.10	2.64	0.95	0.46	4.26	2.08
kim2	31.42	4.79	1.81	37.35	7.18	3.76
lung2	0.01	0.08	0.02	0.01	0.12	0.04
mac_econ_fwd500	1.52	0.75	0.64	3.94	1.15	0.97
mc2depi	0.02	0.35	0.10	0.06	0.74	0.31
PR02R	1.80	2.49	1.43	6.70	3.13	2.15
RM07R	16.45	11.25	3.98	43.19	14.65	6.89
stomach	0.01	0.66	0.13	0.03	0.92	0.37
torso1	10.76	1.94	0.85	25.14	2.37	1.15
torso3	0.02	1.03	0.20	0.05	1.61	0.70
TSOPF_FS_b39_c30	0.66	0.59	0.24	0.21	0.84	0.43
turon_m	0.01	0.39	0.21	0.73	0.59	0.38
vas_stokes_4M	0.83	59.42	23.22	9139.63	91.74	51.55
geomean	0.16	1.09	0.44	0.74	1.54	0.81

Table 12 Run time (in seconds) of MC64J2, THRESH, and BOTTLED on all **DAE**- and **DAPE**-type instances. The geometric mean of the run time of MC64J2, THRESH, and BOTTLED over all **DAE**- and **DAPE**-type instances are 0.35, 1.30 and 0.59 respectively.

	$\mathbf{D}P(\mathbf{A})\mathbf{E}$-type			$\mathbf{D}P(\mathbf{A})\mathbf{PE}$-type		
matrix	MC64J2	THRESH	BOTTLED	MC64J2	THRESH	BOTTLED
af_0_k101	79.91	8.87	3.56	418.20	13.52	7.68
af_1_k101	80.76	8.92	3.59	426.10	14.00	7.72
af_2_k101	84.49	9.38	3.81	447.46	13.94	8.02
af_3_k101	81.85	9.03	3.67	425.69	13.43	7.92
af_4_k101	80.37	8.90	3.59	428.32	13.66	7.73
af_5_k101	80.95	8.95	3.63	447.20	14.13	7.88
analytics	0.11	0.43	0.18	0.15	0.53	0.23
atmosmodd	555.76	9.18	7.01	1757.81	13.67	12.11
atmosmodj	565.60	9.38	7.27	1683.32	14.20	11.97
atmosmodl	621.46	13.25	8.70	1580.69	20.32	16.21
atmosmodm	699.66	14.14	9.38	1768.57	22.60	18.84
BenElechi1	31.45	2.92	2.53	20.36	4.68	4.56
boyd2	0.03	3.37	3.31	0.07	2.09	1.84
Chevron3	8.38	2.60	1.12	45.28	3.49	1.65
Chevron4	22.24	5.11	1.99	150.66	6.84	4.12
circuit5M	4.99	63.51	58.64	57.68	77.57	76.20
CO	45.63	1.88	0.62	0.36	2.49	1.19
cont-300	3.17	0.54	0.35	3.93	0.74	0.53
crashbasis	4.82	0.73	0.26	14.12	0.90	0.40
CurlCurl_3	610.16	4.34	1.54	211.56	7.08	3.80
d_pretok	8.36	1.11	1.20	6.33	1.34	1.50
dc1	0.01	0.14	0.06	0.04	0.19	0.10
dc2	0.01	0.14	0.06	0.04	0.19	0.10
dc3	0.01	0.14	0.06	0.03	0.19	0.10
ecology1	43.10	3.65	1.92	14.36	5.55	3.44
ecology2	66.06	2.54	1.93	24.35	4.13	3.57
FEM_3D_thermal2	0.39	0.83	0.29	15.53	1.26	0.57
Ga10As10H30	18.84	1.46	0.55	0.36	1.82	0.75
Ga19As19H42	30.42	2.39	0.92	0.98	2.73	1.22
Ge99H100	25.74	2.29	1.00	15.39	2.55	1.21
Goodwin_095	3.72	0.95	0.83	8.53	1.33	1.18
Goodwin_127	6.28	2.16	2.08	50.78	2.95	2.81
Hamrle3	0.29	1.74	1.24	0.81	4.42	3.24
Hardesty1	198.57	6.98	4.78	14.75	11.30	9.23
hcircuit	0.01	0.07	0.02	0.02	0.11	0.04
iChem_Jacobian	75.27	1.70	1.51	99.14	2.48	2.32
kim2	112.82	5.64	1.61	345.17	8.74	3.75
Lin	34.97	0.92	0.93	64.27	1.21	1.09
lung2	0.01	0.12	0.07	0.02	0.18	0.11
mac_econ_fwd500	3.89	0.65	0.30	4.58	0.89	0.48
majorbasis	4.69	0.74	0.26	15.00	0.97	0.42
mc2depi	5.70	1.32	1.08	27.13	1.57	1.57
parabolic_fem	57.21	1.66	1.12	39.10	2.09	1.60
PR02R	37.99	2.48	1.92	61.01	3.22	2.93
rajat31	48.39	11.00	7.83	55.91	26.56	28.76
RM07R	259.28	9.13	5.56	468.04	12.83	8.77
Si41Ge41H72	65.22	4.32	1.72	6.51	5.07	2.34
ss	824.91	10.77	4.66	2621.10	18.19	10.79
ss1	5.74	0.45	0.36	5.33	0.62	0.44
stomach	1.45	1.16	0.24	11.23	1.69	0.49
t2em	53.66	3.52	2.11	89.95	5.57	4.81
tmt_unsym	47.59	3.28	4.83	106.41	5.57	8.59
torso1	10.62	1.45	0.64	17.61	1.77	0.87
torso2	0.45	0.24	0.19	1.19	0.38	0.32
turon_m	6.22	0.60	0.71	3.75	0.80	1.02
vas_stokes_1M	452.36	12.00	4.20	2427.71	15.56	7.42
vas_stokes_2M	1419.99	26.46	11.93	6813.20	34.32	18.37
xenon2	6.66	1.14	0.81	4.01	1.54	1.21
geomean	12.47	2.24	1.23	20.16	3.16	2.07

Table 13 Run time (in seconds) of MC64J2, THRESH, and BOTTLED on all $\mathbf{D}P(\mathbf{A})\mathbf{E}$- and $\mathbf{D}P(\mathbf{A})\mathbf{PE}$-type instances. The geometric mean of the run time of MC64J2, THRESH, and BOTTLED over all $\mathbf{D}P(\mathbf{A})\mathbf{E}$- and $\mathbf{D}P(\mathbf{A})\mathbf{PE}$-type instances are 15.86, 2.66 and 1.59 respectively.