# Tearing systems of nonlinear equations II.[☆] A practical exact algorithm

Ali Baharev[a,*], Hermann Schichl[a], Arnold Neumaier[a]

[a] *Fakultät für Mathematik, Universität Wien, Oskar-Morgenstern-Platz 1, A-1090 Wien, Austria*

## Abstract

The objective of optimal tearing is to maximize the number of variables eliminated by solving univariate equations. First, a simple algorithm is presented for automatically identifying feasible assignments: If an equation can be solved symbolically for one of its variables, and the solution is unique, explicit, and numerically stable, then, and only then, this equation-variable pair represents a feasible assignment. Tearing searches for the optimal elimination order over these numerically safe feasible assignments. We give a novel integer programming formulation of optimal tearing. Unfortunately, a high amount of permutation symmetry can cause performance problems; therefore, we developed a custom branch and bound algorithm. Based on the performance tests on the COCONUT Benchmark, we consider the proposed branch and bound algorithm practical (a) for the purposes of global optimization, and (b) in cases where systems with the same sparsity patterns are solved repeatedly and the time spent on tearing pays off.

*Keywords:* algebraic loop, diakoptics, minimum degree ordering, sparse matrix ordering, tearing

[☆] This paper is the second part of a two-part paper on tearing nonlinear systems of equations.
[*] Corresponding author
*Email address:* `ali.baharev@gmail.com` (Ali Baharev)

## 1. Introduction

**Tearing** (cf. [4, 21, 22, 38]) is the representation of a sparse system of nonlinear equations

$$f(x) = 0, \text{ where } f : \mathbb{R}^n \mapsto \mathbb{R}^m, \tag{1}$$

in a permuted form where most of the variables can be computed sequentially once a small auxiliary system has been solved. More specifically, given permutation matrices $P$ and $Q$ such that after the transformation

$$\begin{bmatrix} g \\ h \end{bmatrix} = Pf, \qquad \begin{bmatrix} y \\ z \end{bmatrix} = Qx, \tag{2}$$

$g_i(y, z) = 0$ can be rewritten in the equivalent explicit form

$$y_i = \tilde{g}_i(y_{1:i-1}, z) \tag{3}$$

using appropriate symbolic transformations. Equation (3) implies that the sparsity pattern of the Jacobian of $Pf$ is

$$J = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \text{ where } A \text{ is lower triangular,} \tag{4}$$

$J$ is therefore **bordered lower triangular**. Hereafter, we will refer to a particular choice of $P, Q, g, h, y,$ and $z$ satisfying equations (3) and (4) as an **ordering**. Given an ordering, the system of equations $f(x) = 0$ can be written as

$$\begin{aligned} g(y, z) &= 0 \\ h(y, z) &= 0. \end{aligned} \tag{5}$$

The requirement (3) that $g_i(y, z) = 0$ can be made explicit in $y_i$ essentially means $y = \bar{g}(z)$. Substituting $y$ into $h$ yields $h(\bar{g}(z), z) = 0$ or

$$H(z) = 0. \tag{6}$$

That is, the original nonlinear system of equations $f(x) = 0$ is reduced to the (usually much) smaller system $H(z) = 0$.

**Optimal tearing** is the task of finding an ordering that minimizes the border width

$$d := \dim z \tag{7}$$

of $J$. This objective is a popular choice [9, Sec. 8.4] and often results in a significant speed up, although it does not guarantee any savings in computation time in the general case. Other issues related to this objective were discussed in detail in [4], such as ignoring that (6) can become ill-conditioned or that the objective is unaware of the nonlinearities in (1). It is important to note that minimizing (7) is significantly different from the objective of the so-called fill-reducing orderings which minimize the fill-in.

In the first part [4] of this two-part paper, we surveyed the methods for performing tearing either optimally with exact methods or with greedy heuristics. The variants of tearing were also reviewed, such as tearing methods that allow in (4) forms other than bordered lower triangular form, or that solve small subsystems simultaneously, and those that have objective function different from (7). Several application areas were also discussed in [4].

## 2. Identifying feasible assignments

The task of optimal tearing is equivalent to maximizing the number of eliminated variables through assignments, see (3). If the equation $f_i(x) = 0$ of (1) can be solved symbolically for the variable $x_j$, and the solution is unique, explicit and numerically stable, then, and only then, the pair $(i, j)$ represents a **feasible assignment**. The more feasible assignments we find, the more freedom we have when searching for elimination orders. This underlines the importance of a software package (computer algebra system) for solving equations symbolically. We first discuss how we identify those rearrangements of the equations that give a unique and explicit solution in one variable, and temporarily ignore the numerical stability issues. Then, we discuss how numerically troublesome functions can be recognized. We follow a **conservative approach** in our implementation: Solutions are excluded from the feasible assignments if we fail to prove uniqueness or numerically stability, even if those solutions might in fact be unique and numerically stable.

## *2.1. Provably unique and explicit solution*

In our implementation, we use SymPy [47] and attempt to solve the individual equations of (1) for each of its variables. SymPy is a fairly mature symbolic package, and if a closed form solution exists, SymPy often finds it. We consider only those assignments as candidates for feasible assignments where the symbolic package returns exactly one explicit solution. Let us consider a few examples below.

For example, for the equation

$$x_1 - x_2 x_3 = 0 \tag{8}$$

SymPy will return

$$x_1 = x_2 x_3, x_2 = \frac{x_1}{x_3}, x_3 = \frac{x_1}{x_2}. \tag{9}$$

That is, we can make (8) explicit in any of its variables; these solutions are also unique. All three assignments are candidates for feasible assignments. They are only candidates because we have not considered numerical issues yet; issues like division by zero will be discussed in the next section.

In other cases, the solution may not be unique. For example, when solving

$$x_1^2 + 2x_1 x_2 + 1 = 0 \tag{10}$$

for $x_1$, SymPy will correctly return both

$$x_1 = -x_2 + \sqrt{x_2^2 - 1} \text{and} x_1 = -x_2 - \sqrt{x_2^2 - 1}. \tag{11}$$

Even though we could make (10) explicit in $x_1$, we still have to exclude it from the feasible assignments because there are two solutions.

A more sophisticated implementation could do the following when faced with multiple solutions as in (11). Let us *assume* that $x_1 \geq 0$ and $x_2 \geq 0$ must hold: For example, these bound constraints are also part of the input, or we deduced these lower bounds from other constraints of the input. Then, a sophisticated implementation could deduce that only $x_1 = -x_2 + \sqrt{x_2^2 - 1}$ can hold at the solution because the other formula always gives strictly negative values for $x_1$ if $x_2 \geq 0$ but we know that $x_1 \geq 0$ must hold.

Some equations do not have closed-form solutions, or the symbolic package that we use fails to produce a closed-form solution; these two cases are identical from a practical perspective. In our current implementation, such cases are never added to the feasible assignments.

As discussed in Section 1, solving implicit equations is already performed in state-of-the-art modeling systems. For the sake of this example, let us consider the case when we are stuck with the following implicit equation:

$$f_3(x_1, x_2) = 0. \tag{12}$$

One can still try to solve the equation numerically for $x_2$ and pretend that a function $g$ exists such that

$$x_2 = g(x_1) \tag{13}$$

and add $(3, 2)$ to the feasible assignments. Robust numeric methods are available for solving (12) such as the Dekker-Brent method, see [15] and [10, Ch. 4]. However, we are not aware of any modeling system that would check the uniqueness of the solution in (12); the elimination (13) is performed with the $x_1$ found by the numeric solver, and other solutions, if any, are simply ignored. We never consider implicit equations as feasible assignments in our current implementation (conservative approach).

### *2.2. Identifying numerically troublesome assignments*

In the previous section, we ignored the question of numerical stability. For example, if $x_2$ or $x_3$ happens to be 0 in (9), the corresponding formula involves division by zero. In this section, we discuss how to recognize assignments that are potentially troublesome. We again follow a conservative approach: We reject each formula that we fail to prove numerically safe (even if it is always safe to evaluate in reality).

We assume that all variables have reasonable (not big $M$) lower and upper bounds. From an engineering perspective, this requirement does not spoil the generality of the method: The variables in a typical technical system are double-bounded, although often implicitly. The physical limitations of the devices and the design typically impose minimal and maximal throughput or load of the devices; this implies bounds on the corresponding variables, either explicitly or implicitly. There are also natural physical bounds, e.g., mole fractions of chemical components must be between 0 and 1, etc.

We used the interval arithmetic implementation available in SymPy [47] to check the numerical safety of an assignment. For the purposes of the present paper, the reader can think of **interval arithmetic** [28] as a computationally cheap way to get guaranteed (but often suboptimal) lower and upper bounds on the range of a given function over the domain of the variables. (Extended interval arithmetic can safely work with infinity, division by zero, etc., see [28, Ch. 4].) Let us look at some examples where we also give the lower and upper bounds on the variables.

*Example 1.* If we evaluate $f(x_1, x_2) = \frac{x_1 - x_2}{x_1 + x_2}$ with interval arithmetic over $x_1 \in [3,\ 9]$ and $x_2 \in [1,\ 2]$, we get $[0.0909090, 2.0]$. The true range is $[0.2,\ 0.8]$. As we can see, interval arithmetic fulfilled its contract: The range obtained from the interval arithmetic library indeed encloses the true range of the function but also overcovers it. Nevertheless, it is good enough for our purposes.

*Example 2.* If we evaluate $g(x) = \frac{1}{x^2 - x + 1}$ over $x \in [0,\ 1]$ with interval arithmetic, we get $[0.5,\ \infty]$. Again, the true range, $[1,\ \frac{4}{3}]$ is enclosed but overestimated. In this case, it is possible to get the true range with interval arithmetic if one uses the equivalent formula $\frac{4}{(2x-1)^2+3}$ for evaluating $g(x)$.

*Example 3.* Finally, an example where the evaluation fails: If we evaluate $\log(x)$ over $x \in [-1, 1]$, the interval arithmetic library throws an exception and complains that "logarithm of a negative number" was encountered.

*Rule for deciding on numerical safety.* We consider only those assignments as candidates for feasible assignments where the symbolic package returns exactly one explicit solution. When checking the numerical safety of an assignment, we try to evaluate the function on the right hand side of the assignment with interval arithmetic. The evaluation either fails with an exception, or we get back an interval $r$ as the result. We consider the assignment numerically safe if and only if the evaluation did not throw an exception and the result $r \subseteq [-M, M]$ where we set $M = 10^{15}$ as default. However we choose the variables within their bound constraints, numerically safe assignments are always safe to evaluate with ordinary floating-point arithmetic: The evaluation cannot fail due to division by zero, or due to illegal function arguments, etc.

This simple rule is admittedly not perfect. Since interval arithmetic usually overestimates the actual ranges, we can reject assignments that are in reality safe. However, accurately determining numerically safe assignments is an NP-hard problem in general [34]. In our opinion, a single function evaluation with interval arithmetic is a good compromise.

Another issue is that the above rule does not safeguard against the final system (6) becoming ill-conditioned, see [4]. Our novel proof of concept algorithms automatically mitigate this type of conditioning problem through reparameterization or redistribution [5, 7] in the context of global optimization. It is subject of future research to make these reparameterization and redistribution algorithms practical outside the field of global optimization.

## 3. Optimal tearing with integer programming

### *3.1. Background*

We recap those sections of [4] that are essential for the integer programming approach; the reader is referred to [4] for details. We construct a bipartite graph $B$ such that one of its disconnected node sets corresponds to the equations (rows), the other to the variables (columns). If a variable appears in an equation, there is an edge between the corresponding nodes, and the two nodes are not connected otherwise. We are given $F$, an arbitrary subset of edges of $B$ but with the intent of $F$ corresponding to the set of feasible assignments, see Section 2.

A matching $M$ is a subset of edges of $B$ such that at most one edge of $M$ is incident on any node. After a matching has been established, we orient $B$ as follows. We first remove those edges from $M$ that are not in $F$; the remaining edges in $M$ are directed towards variables, all other edges point towards equations. The resulting directed graph has a specific cycle structure, see [4]: It is acyclic if and only if $M$ does not involve a maximum cardinality matching on any subgraph induced by the nodes of a simple cycle.

We select a subset of $M$, called the feedback edge set, that is sufficient to reverse (make them point towards the equations too) to make the resulting acyclic. Any of the heuristics discussed in [3] is applicable. These heuristics can never fail but they may

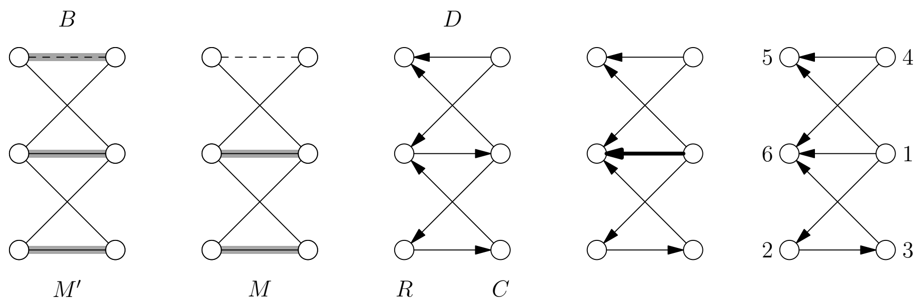

Figure 1: The steps of tearing: bipartite matching $M' \to$ matching $M$ after considering feasible assignments $\to$ orientation $\to$ feedback edge set $\to$ a possible elimination order.

produce a feedback edge set that is not of minimum cardinality; the proposed algorithm will work nevertheless.

## 3.2. *Integer programming formulation of optimal tearing*

The following integer programming formulation is used in our implementation; any feasible solution to this integer program uniquely defines a bipartite matching $M$.

$$
\begin{aligned}
\max_{y} \; & \sum_{e \in F} y_e & & \text{(find the maximum-cardinality matching)} \\
\text{s.t.} \; & \sum_{e \in E} u_{re} y_e \le 1 & \text{for each } r \in R, \quad & \text{(each row is matched at most once)} \\
& \sum_{e \in E} v_{ce} y_e \le 1 & \text{for each } c \in C, \quad & \text{(each column is matched at most once)} \\
& \sum_{e \in E} a_{se} y_e \le \frac{\ell_s}{2} - 1 & \text{for each } s \in S \quad & \text{(cycles are not allowed).}
\end{aligned}
\tag{14}
$$

Here the binary variable $y_e$ is 1 iff edge $e$ is in the matching $M$; the set $F$ is an arbitrary subset of edges of $B$, but with the intent of $F$ corresponding to the set of feasible assignments, see Sec. 2; $E$, $R$, and $C$ denote the index sets of the edges, the rows, and the columns, respectively; $u_{re}$ is 1 iff node $r$ is incident to edge $e$, and 0 otherwise; similarly, $v_{ce}$ is 1 iff node $c$ is incident to edge $e$, and 0 otherwise; $S$ is the index set of the simple cycles currently in the (incomplete) cycle matrix $A = (a_{se})$; the entry $a_{se}$ is 1 iff the edge $e$ participates in the simple cycle $s$, and 0 otherwise; $\ell_s$ is the length (the number of edges) of simple cycle $s$. The last inequality excludes maximum cardinality matchings on all subgraphs induced by simple cycles; this ensures that after orienting the bipartite graph $B$ according to the matching, the obtained directed graph $D$ is acyclic, see Sec. 3.1.

General-purpose integer programming solvers such as Gurobi [26] or SCIP [1] do not have any difficulty solving (14) as long as enumerating all simple cycles is tractable. Unfortunately, enumerating all simple cycles is typically intractable in practice; we will consider such an example in Section 7.2.

## 4. Lazy constraint generation

In practice, solving the integer program (14) directly can easily become intractable since it requires enumerating all the simple cycles of the input bipartite graph $B$. Unfortunately, even sparse graphs can have exponentially many simple cycles [42], and such graphs appear in practice, e.g., cascades (distillation columns) can realize this many simple cycles. The proposed method enumerates simple cycles in a lazy fashion, and extends the cycle matrix $A$ iteratively in the hope that only a tractable number of simple cycles has to be enumerated until a provably optimal ordering is found. The pseudo-code of the algorithm is given at Algorithm 1. The Python source code of the prototype implementation is available at [2]. The computational results will be presented in Section 7.

### *4.1. Solving a sequence of integer programs with an incomplete cycle matrix*

Let us refer to problem (14) with the complete cycle matrix as $P$, and let $\tilde{P}^{(i)}$ denote its relaxation in iteration $i$ where only a subset of simple cycles are included in the cycle matrix. One can simply start with an empty cycle matrix in $\tilde{P}^{(0)}$; more elaborate initializations are also possible.

The optimal solution to the relaxed problem $\tilde{P}^{(i)}$ gives the matching $M^{(i)}$; the bipartite graph is oriented according to this matching as discussed in Section 3.1. Since not all simple cycles are included in the cycle matrix, the directed graph $D^{(i)}$ obtained with the orientation is not necessarily acyclic. Therefore we need to check this.

A **topological sort** of a directed acyclic graph $G = (V, E)$ is a linear ordering of all its nodes such that if $G$ contains an edge $(u, v)$, then $u$ appears before $v$ in the ordering [13, Sec. 22.4]. The nodes in a directed graph can be arranged in a topological order if and only if the directed graph is acyclic [16, Sec. 14.8].

Topological sort succeeds if and only if $D^{(i)}$ is acyclic. If the topological sort succeeds, the algorithm has found an optimal solution to $P$ and therefore terminates.

If the topological sort fails, $D^{(i)}$ has cycles. In this case, we first create a feasible solution to $P$ as follows. We identify a feedback edge set (tear set) $T^{(i)} \subseteq M^{(i)}$ using an appropriate heuristic, see for example [3]. The proposed algorithm is guaranteed to make progress with *any* feedback edge set but the algorithm is likely to make better progress with a $T^{(i)}$ of small cardinality. Reversing the edges in $T^{(i)}$ makes the graph acyclic, see Sec. 3.1, and therefore the associated matching yields a feasible solution to $P$. We keep track of the best feasible solution to $P$ found.

After we have created a feasible solution to $P$, we improve the relaxation $\tilde{P}^{(i)}$ by adding new rows to the cycle matrix $A^{(i)}$. The directed graph $D^{(i)}$ must have at least one cycle because topological sort failed previously. The feedback edge set $T^{(i)}$ contains at least one edge of every cycle in $D^{(i)}$ by definition; therefore, there must be at least one edge $t \in T^{(i)}$ that participates in a cycle. For each edge $t \in T^{(i)}$ we compute the shortest path from the head of $t$ to the tail of $t$ with breadth-first search (BFS). Such a shortest path exists if and only if $t$ participates in a cycle; we extended this shortest path with $t$ which then gives a simple cycle (even without chords). A new row is appended to the cycle matrix per each simple cycle found. The cycle matrix $A^{(i)}$ is guaranteed to grow at least by one row by the time we finish processing all the edges in $T^{(i)}$. We then proceed with the next iteration step, starting with solving the next relaxed problem $\tilde{P}^{(i+1)}$ with this extended cycle matrix $A^{(i+1)}$. The cycle matrix is only extended as the algorithm runs; rows are never removed from it. As we will discuss it shortly, it has not been observed yet that superfluous rows would accumulate in the cycle matrix, slowing down the algorithm.

The algorithm terminates if the directed graph after the orientation becomes acyclic (as already discussed) or the objective in a relaxed problem equals the cardinality of the best known feasible solution to $P$. In both cases, the optimal solution to (14), hence the optimal ordering of (1) is found by Algorithm 1.

**Algorithm 1:** Integer programming-based algorithm with lazy constraint generation for computing the optimal tearing

```
   Input: $J$, a sparse $m \times n$ matrix; $B$, the undirected bipartite graph associated with $J$, see Sec. 3.1
   Output: A matching that maximizes the cardinality of the eliminated variables
   # $P$ denotes the integer program (14) with the complete cycle matrix of $B$
 1 Set the lower bound $\underline{z}$ and the upper bound $\bar{z}$ on the objective to 0 and $\min(m, n)$, respectively;
 2 Let $\hat{y}$ denote the best feasible solution to $P$ found at any point during the search (incumbent solution) ;
 3 Set the trivial solution $\hat{y} = 0$, realizing $\underline{z} = 0$ ;
 4 Let $A^{(i)}$ denote the incomplete cycle matrix in (14), giving the relaxed problem $\tilde{P}^{(i)}$ $(i = 0, 1, \dots)$ ;
 5 Set $A^{(0)}$ to be empty ;
 6 for $i = 0, 1, \dots$ do
 7     Solve the relaxed problem $\tilde{P}^{(i)}$; results: solution $y^{(i)}$, matching $M^{(i)}$, and objective value $z^{(i)}$ ;
       # Optional: When the integer programming solver is invoked on the line just above,
       # $\hat{y}$ can be used as a starting point
 8     Set the upper bound $\bar{z}$ to $min(\bar{z}, z^{(i)})$ ;
 9     if $\underline{z}$ equals $\bar{z}$ then
10         stop, $\hat{y}$ yields optimal tearing ;
11     Let $D^{(i)}$ denote the directed graph obtained by orienting $B$ according to $M^{(i)}$, see Sec. 3.1 ;
12     if $D^{(i)}$ can be topologically sorted then
13         stop, $y^{(i)}$ yields optimal tearing ;
14     Compute a feedback edge set (tear set) $T^{(i)} \subseteq M^{(i)}$ using an appropriate heuristic, e.g. [3] ;
       # See Sec. 3.1: $T^{(i)}$ cannot be empty as $D^{(i)}$ must have at least one cycle, and
       # reversing each edge $t \in T^{(i)}$ would make $D^{(i)}$ acyclic
15     Set those components of $y^{(i)}$ to 0 that correspond to an edge in $T^{(i)}$ ;
       # $y^{(i)}$ is now a feasible solution to $P$
16     Let $\hat{z}$ be the new objective value at $y^{(i)}$ ;
17     if $\hat{z} > \underline{z}$ then
18         Set $\underline{z}$ to $\hat{z}$ ;
19         Set $\hat{y}$ to $y^{(i)}$ ;
       # Extend the cycle matrix $A^{(i)}$ to get $A^{(i+1)}$
20     foreach $t \in T^{(i)}$ do
21         Find a shortest path $p$ from the head of $t$ to the tail of $t$ with breadth-first search (BFS) in $D^{(i)}$ ;
22         if such a path $p$ exists then
23             Turn the path $p$ into a simple cycle $s$ by adding the edge $t$ to $p$ ;
24             Add a new row $r$ to the cycle matrix corresponding to $s$ if $r$ is not already in the matrix ;
       # At this point $A^{(i+1)}$ is guaranteed to have at least one additional row compared to $A^{(i)}$
```

### *4.2. Finite termination*

The proposed algorithm must terminate in a finite number of steps. In each iteration that does not terminate the algorithm, we are guaranteed to make progress because we extend the cycle matrix by at least one row and the number of simple cycles, i.e., the maximum number of rows that the cycle matrix can have, is finite. In the worst case, all simple cycles have to be enumerated, however, we are not aware of any *challenging* graph that would trigger this worst-case (or a near worst-case) behavior. (A trivial example would be a graph with a single simple cycle: Although all simple cycles have to be enumerated, it is not a challenge.)

Even though it is not guaranteed, the gap between the lower and upper bound on the optimum can shrink in each iteration: Beside making progress by extending the cycle matrix, we may also find a better feasible solution to $P$ by solving the feedback edge set problem, and the objective of the relaxed problem may also improve as the cycle matrix grows. Focusing only on the worst-case behavior (i.e., having to enumerate all the simple cycles) is neither a realistic view of the practical performance of the algorithm nor does it reveal why the algorithm can become impractical on certain problem instances: It is the permutation symmetry that can make Algorithm 1 impractical in certain cases. By **permutation symmetry** we mean the following: Given a Hessenberg form that corresponds to a feasible solution of (14), there are typically many row permutations (bipartite matchings) that, after an appropriate column permutation, realize the same upper envelope. The cost only depends on the upper envelope.

## *4.3. Near minimal cycle matrix*

Since the algorithm only adds rows to the cycle matrix but never removes any of them, it is reasonable to ask whether superfluous rows can accumulate in the cycle matrix as the algorithm runs. Numerical experiments were carried out whose goal was to find a minimal subset of rows in the cycle matrix such that the solution to (14) with this minimal cycle matrix still gives the optimal solution to $P$.

In one set of experiments, the complete undirected bipartite graph (corresponding to the fully dense $n \times n$ matrix) was used. The complete bipartite graph admits a trivial solution, and the branch and bound algorithm of Section 6 with (18) will solve these problems instantaneously, on the root node of the search tree. However, the complete bipartite graph is quite challenging for Algorithm 1 which makes this graph appropriate for experimenting. By the time Algorithm 1 finds the optimal solution, the cycle matrix has $\binom{n}{2}^2$ rows for $n = 2 : 12$ where $n$ is the problem size; that is, the cycle matrix has $O(n^4)$ rows on termination.

After Algorithm 1 terminated, the following additional constraint was appended to the integer program (14) to render it infeasible:

$$\sum_{k \in F} y_k \geq z^* + 1, \tag{15}$$

where $z^*$ is the objective value at the optimal solution to $P$. Then, Gurobi was invoked on this infeasible integer program to compute the Irreducible Inconsistent Subsystem (IIS). In general, an IIS is a subset of the constraints and variable bounds of the original model. If all constraints in the model except those in the IIS are removed, the model is still infeasible. However, further removing any one member of the IIS produces a feasible result. For this example, Gurobi could not find any superfluous row for $n = 2 : 8$. The computations took more than 3 hours for $n = 8$, and $n > 8$ have not been checked.

It could happen that Algorithm 1 generates a cycle matrix that cannot be reduced further but one could generate a cycle matrix with fewer rows by considering all the simple cycles in the undirected bipartite graph. The same experiment was repeated accordingly with the complete cycle matrix for $n = 2 : 6$, where it was still tractable to enumerate all the simple cycles. Gurobi again could not find a cycle matrix having fewer rows than the one that Algorithm 1 had generated.

Similar numerical experiments were carried out with those graphs in our test set where the IIS computation was tractable. Algorithm 1 never accumulated a significant number of superfluous rows for the examples tried; performance degradation due to superfluous rows in the cycle matrix never occurred.

## 5. Heuristics for ordering to lower Hessenberg form

This section introduces Algorithms 2 and 3 that form the basis of the rest of the paper. We first consider irreducible square matrices. As discussed in [4], a square matrix is irreducible if and only if the fine Dulmage-Mendelsohn decomposition [18–20, 33] generates a single block (the entire matrix itself is the block) with zero-free diagonal. The Dulmage-Mendelsohn decomposition is also referred to as block lower triangular decomposition or BLT decomposition, since it outputs a block lower triangular form if the input matrix is square and structurally nonsingular. More recent overviews of the

Dulmage-Mendelsohn decomposition method are available e.g., in [41], [17, Ch. 6], and [14, Ch. 7].

A **lower Hessenberg form** is a block lower triangular matrix but with fully dense rectangular blocks on the diagonal, rather than square blocks. Consequently, the height of the columns is nonincreasing. Since the matrix is assumed to be irreducible, the first entry in each column is either on or above the diagonal.

Let the matrix $A \in \mathbb{R}^{n \times n}$ be an irreducible matrix. In order to relate this section to tearing, the matrix $A$ of this section can be considered as the Jacobian of (1). Furthermore, it is assumed that each equation in (1) can be made explicit in any of its variables with appropriate symbolic transformations.

We closely follow the presentation of Fletcher and Hall [24] in our discussion here. Two heuristic algorithms for permuting $A$ to lower Hessenberg form are discussed. Both of these algorithms progressively remove rows and columns from $A$; the matrix that remains when some rows and columns have been removed is called the **active submatrix**, see Figures 2–3. It is called the active submatrix since it is within this submatrix where further permutations take place. The whole matrix is active when the algorithm starts, and the active submatrix is empty on termination. The indices of the removed rows and columns are assembled in the permutation vectors $\rho$ and $\kappa$, respectively, in the order they were removed; see Figure 2. The pair $\pi = (\rho, \kappa)$ will be referred to as **incomplete permutation**. The incomplete permutation unambiguously determines the active submatrix; the active submatrix unambiguously determines the removed row and column indices but not their order. For each row $i$ in the active submatrix, let $r_i(\pi)$ denote the number of nonzero entries. Similarly, let $c_j(\pi)$ be the number of nonzero entries in column $j$ of the active submatrix. Hereafter $r_i = r_i(\pi)$ and $c_j = c_j(\pi)$ will be referred to as **row count** and **column count**, respectively.

Several heuristics have been proposed to permute $A$ to one of the desirable forms (bordered block lower triangular, spiked lower triangular, lower Hessenberg form) discussed in [4], e.g., the Hellerman-Rarick family of ordering algorithms, see [23, 29, 30] and [17, Ch. 8], and the ordering algorithms of Stadtherr and Wood [44, 45]. Although there are subtle differences among these ordering algorithms, they all fit the same pattern when viewed from a sufficiently high level of abstraction [24]; they only seem to

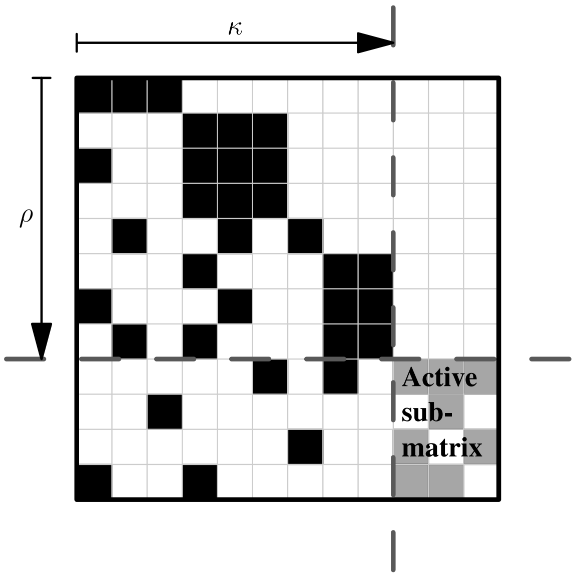

Figure 2: The reordered matrix after applying an incomplete permutation $\pi = (\rho, \kappa)$.

differ in the heuristics applied to break the ties on line 4 in Algorithm 2.

**Algorithm 2:** Heuristic for ordering to Hessenberg form [24]

```
  Input: $A$, a sparse irreducible matrix
  Output: $A$ permuted to lower Hessenberg form
1 set $A$ as the active submatrix;
2 repeat
4     find a row in the active submatrix with minimum row count;
5     put all columns which intersect this row to the left and consider them as removed;
6     update row counts in the active submatrix;
7     put all rows with zero row count to the top and consider them as removed;
8 until all rows and columns are removed;
```

We note that Algorithm 2 resembles the well-known minimum degree ordering algorithm for symmetric matrices [49, scheme 2], which in turn is a Markowitz ordering [37] applied to a symmetric problem. The reader is referred to [17, Sections 7.2 and 7.3] for a concise review of these other ordering algorithms, and to [25] for a review on the evolution of the minimum degree ordering algorithms.

Algorithm 3, the two-sided algorithm of [24], is an extension of Algorithm 2. Figure 3 shows an intermediate stage of the algorithm. The idea of working from both ends and iteratively removing all rows and columns that have a single nonzero entry already shows up in [35] as early as 1966, and later became known as forward and backward triangularization [36]. Cellier's matching rule [11, 48] also show similarities with Algorithm 3.

**Algorithm 3:** The two-sided algorithm of [24] for ordering to Hessenberg form (heuristic)

```
Input: A, a sparse irreducible matrix
Output: A permuted to lower Hessenberg form
1  set A as the active submatrix;
2  repeat
3      find either a min-row-count row or a min-column-count column, whichever has fewer entries ;
4      if a row is chosen then
5          proceed as in Algorithm 2 ;
6      if a column is chosen then
7          remove all rows which intersect this column ;
8          update column counts ;
9          remove all columns with zero column count ;
10 until all rows and columns are removed;
```

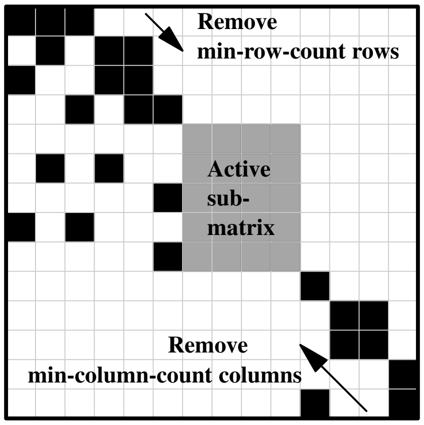

Figure 3: An intermediate stage of the two-sided algorithm for ordering to Hessenberg form.

We have discussed these algorithms here because they form the basis of the rest of the present paper:

1. In the custom branch and bound algorithm of Section 6, instead of breaking ties according to some heuristic as on line 4, all possibilities are systematically considered. The lower bounding procedure of Section 6.2, accompanying the custom branch and bound algorithm, was inspired by Algorithm 3.

2. Beside inspiration, Algorithm 2 is also used in the branch and bound algorithm of Section 6 to find a good feasible at the root node.
3. Although the integer programming approach of Section 4 does not require it, both Algorithm 2 and 3 can be used for generating a good initial point.

*Rectangular and reducible matrices in lower Hessenberg form.* This is our extension of the lower Hessenberg forms to the under- and over-determined cases. $A$ is assumed to be a full rank $m \times n$ matrix.

If $m \geq n$, we order $A$ into the form $\begin{bmatrix} A_T \\ A_B \end{bmatrix}$ where $A_T \in \mathbb{R}^{n \times n}$ is a structurally nonsingular lower Hessenberg form. If $m \leq n$, we order $A$ into the form $[A_L \; A_R]$ where $A_R \in \mathbb{R}^{m \times m}$ is a structurally nonsingular lower Hessenberg form. Note that neither $A_T$ nor $A_R$ is required to be irreducible; they are only required to have structural full rank. See [4] at the Dulmage-Mendelsohn decomposition on finding the structural rank of a matrix. As we showed in [4], decomposing these matrices further into smaller irreducible matrices could result in more guessed variables in tearing.

We can obtain such lower Hessenberg forms as follows. If $m > n$, we run Algorithm 3 but we always choose rows; if $m < n$, we run Algorithm 3 but we always choose columns.

## 6. Optimal tearing by a custom branch and bound algorithm

The proposed algorithm is similar to Algorithm 2 but instead of breaking ties according to some heuristic as on line 4 of Algorithm 2, the proposed algorithm considers all possibilities in a branch and bound algorithm. The idea is certainly not new, see e.g. [12].

*Simplifying assumptions.* Let $A \in \mathbb{R}^{m \times n}$ denote the Jacobian of the system (1). (It is a deviation from the notations of Section 1 regarding $A$.) It is assumed in this section that $A$ has full structural column rank if $m \geq n$, and $A$ has full structural row rank if $m \leq n$. In short, $A$ is structurally a full rank matrix. See [4] at the Dulmage-Mendelsohn decomposition how this assumption can be checked. For the sake of simplicity, it is assumed in this section that each equation in (1) can be made explicit in any of its variables with appropriate symbolic transformations.

## *6.1. The proposed branch and bound algorithm*

The reader is referred to Section 5 regarding the definition of the active submatrix, permutation $\pi = (\rho, \kappa)$, row count $r_i = r_i(\pi)$ and column count $c_j = c_j(\pi)$. We say that we **eliminate a row** if we solve the corresponding equation in (1) for one of its variables and then remove the row and all intersecting columns from the active submatrix. The **cost of eliminating row** $i$, or simply the **row cost** of row $i$, is the number of variables that still need to be guessed when we eliminate row $i$. In terms of the row count, the cost of row $i$ is $\max(0,\ r_i - 1)$: According to our assumption, each equation can be symbolically transformed into an assignment to any of its remaining unknowns, meaning that at most $r_i - 1$ variables need to be guessed when row $i$ is eliminated. The **cost**

$$z = z(\rho) \tag{16}$$

**of an incomplete or complete permutation** $\pi = (\rho, \kappa)$ is the sum of all row costs when the rows are eliminated one-by-one along the upper envelope, in the order determined by $\rho$. If the permutation is incomplete, the elimination (and the summation of the row costs) stops at the end of $\rho$, and the active submatrix remains. The proposed algorithm seeks a minimum cost permutation $\pi$ that brings $A$ into lower Hessenberg form.

A **lower bound on the cost** of the best possible complete permutation that still may be achievable given an incomplete permutation $(\rho, \kappa)$ is

$$\underset{\wedge}{z}(\rho) = z(\rho) + \min_i(\max(0,\ r_i - 1)), \tag{17}$$

since $z(\rho)$ has already been spent, and at least $\min_i(\max(0,\ r_i - 1))$ guesses has to be made in order to continue the elimination. This lower bound is also sharp in the following sense: It can happen that all the other remaining rows can be iteratively eliminated at zero cost after having eliminated the minimum cost row.

*Distinguishing features.* We now give the specific details that make the proposed method a *custom* branch and bound algorithm.

1. The search tree is traversed in depth-first search order.
2. The best-first search rule is applied when branching; the score of a node is $\underset{\wedge}{z}(\rho)$, as defined by (17). The lowest score node is explored first, breaking ties arbitrarily.

An efficient implementation of the best-first search is possible with a min-priority queue [13, Sec. 6.5].

3. An initial feasible solution is computed with Algorithm 2 before the branch and bound search starts. A lower bound on the optimal cost elimination is computed based on the rules of Section 6.2 (run only once).
4. When a new complete permutation is found (a leaf node is reached by the branch and bound search), a procedure is run with this complete permutation to improve the lower bound on the optimal cost. This procedure will be discussed in Section 6.3.
5. The algorithm keeps tracks of the trailing submatrices that have already been fully discovered during the depth-first search. Whenever we encounter an active submatrix that has already been discovered, we just retrieve its optimal ordering and cost from the bookkeeping.
6. The 'back-track rule' of Hernandez & Sargent [31] was applied to discard entire subtrees of the branch and bound search tree by excluding sequences that cannot possibly produce strictly lower cost solutions than the already found ones. The reader is referred to [31] for further details.
7. The bipartite graph, corresponding to the sparse matrix to be ordered, can become disconnected. Whenever this happens, the connected components are processed independently.

The source code of the Python prototype implementation is available at [2]. The computational results will be presented in Section 7.

## *6.2. Lower bounds based on the minimum row and column counts*

The input of the lower bound deducing algorithm is an $m \times n$ matrix $A$. Let $z^*$ denote the cost of the optimal ordering of $A$. In our numerical experience, the following approaches proved to be helpful:

- lower bounds based on the minimum row and column counts,
- relaxation by partitioning the columns of $A$ (called column slice relaxation).

This section describes the former, the next section will address the latter approach.

We start with the case when $m \geq n$. In any ordering, the cost of eliminating the first row cannot be less than the minimum of all the row costs in the entire matrix $A$, that is, we have the following lower bound on $z^*$:

$$z^* \geq \min_i r_i - 1, \text{ where } i \in \text{row indices of } A, \text{ and } m \geq n. \tag{18}$$

A simple consequence of (18) is that a square nonsingular matrix $A$ ($m = n$) cannot be put into lower triangular form if the minimum row count is at least 2. As far as we can tell, basically the same idea appears in [12]

Let us now focus on the case when $m \leq n$. We have to guess at least $n-m$ variables in any permutation: A variable is either guessed or assigned, and we can perform at most as many assignments as there are equations. Since each guessed variable costs 1 according to our definition of the cost function, we want to assign to as many variables as possible using our $m$ equations. Therefore, we form a nonsingular $m \times m$ submatrix of $A$, and our goal is to find a cost optimal lower Hessenberg form in this submatrix. The remaining part of $A$ is an $m \times (n-m)$ matrix; it only contains variables that have to be guessed anyway because we have used up all our equations to perform eliminations in the other square submatrix. That is, we form a partition of $A$ as follows: $A = [A_L \, A_R]$ where $A_R$ is the sought cost optimal $m \times m$ lower Hessenberg form, and $A_L$ contains the left over columns that have to be guessed because we have no remaining equations to form assignments to those variables. We can think of it as running the two-sided algorithm of Fletcher and Hall [24] but we always choose columns so that $A_R$ will be a lower Hessenberg form. See also Figure 3 and Algorithm 3.

If the last column of the full rank, Hessenberg form matrix $A_R$ has $c_\ell$ entries in an arbitrary ordering of $A$, i.e., the column count of the last column is $c_\ell$, the elimination cost associated with $A_R$ is at least $c_\ell - 1$. The proof is given in the Appendix. The minimum column count over all the columns in $A$ (and not just in $A_R$) is an obvious a lower bound on $c_\ell$:

$$c_\ell \geq \min_j c_j, \text{ where } j \in \text{column indices of } A. \tag{19}$$

We can now give a lower bound on $z^*$:

$$z^* \geq \min_j c_j - 1 + (n-m), \text{ where } j \in \text{column indices of } A, \text{ and } m \leq n. \tag{20}$$

Here, the first term on the right-hand side is the lower bound on the elimination cost in $A_R$, and the second term accounts for the variables that we have to guess in $A_L$ regardless of $A_R$.

The inequality (18) gives a sharp lower bound in the sense that after removing the row with the minimum count, it can happen that all the remaining rows can be iteratively eliminated at zero cost in $A$. The same holds for (20): After removing the minimum cost column in $A$, it is possible that the active submatrix in $A_R$ can be permuted to lower triangular form (has zero cost). Besides being sharp, (18) and (20) are available at no additional implementation effort and in constant time: An efficient implementation of the algorithm has to keep track of the minimum row and/or column count anyway, e.g., with a min-heap data structure [13, Ch. 6].

## *6.3. Column slice relaxation*

Let $\tilde{A}$ denote an arbitrarily chosen column slice of $A$. The columns in $\tilde{A}$ must be either guessed or eliminated , and $\tilde{A}$ has all the rows that can possibly used for elimination. By computing the optimal elimination cost for $\tilde{A}$, we lower bound the elimination cost for the columns in $\tilde{A}$ in the optimal cost elimination of $A$: The columns not in $\tilde{A}$ only impose further constraints on the elimination order but those constraints are ignored (relaxation). We get a lower bound on the optimal elimination cost of $A$ if the columns of $A$ are partitioned, and the optimal elimination cost for each column slice is computed and then accumulated. In practice, it is usually not worth computing the optimal elimination cost for the slices; therefore, we only compute a lower bound in each slice, e.g. as discussed in Section 6.2.

Although any column partition of $A$ can be used for relaxation, the usefulness of the deduced lower bound greatly depends on the choice of the column partition. In our numerical experience, the following blocking procedure gives useful results on certain problem instances. We walk along the upper envelope of the Hessenberg form: We always step either to the right or downwards, and we always move as long as we can before having to change the direction. We **partition** the matrix both horizontally and vertically whenever we are about to step more than one to the right. In other words: We partition the ordered matrix right before those places where non-zero cost row eliminations happen. This gives the partition shown in Figure 4. This partitioning

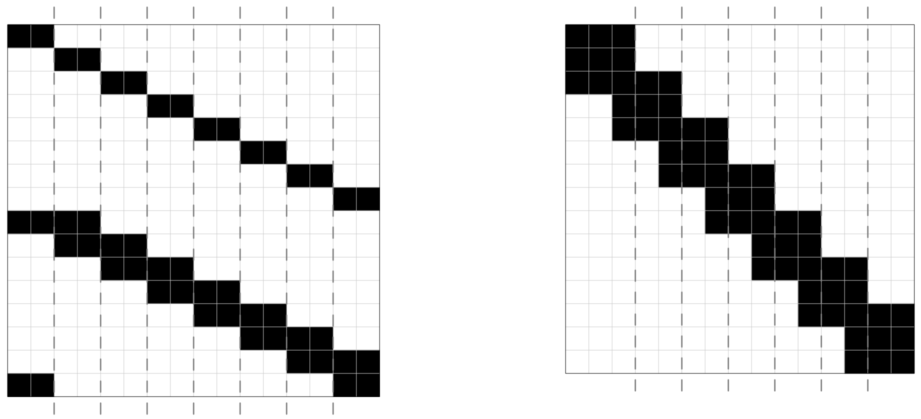

Figure 4: Partitioning the Hessenberg form at the places where variables had to be guessed.

technique proved to be useful for the problem shown on the left of Figure 4, whereas it does not produce useful lower bounds for the problem shown on the right of Figure 4. The pattern on the left would be a very challenging pattern for the algorithm without this lower bounding procedure; however, the algorithm proves optimality immediately on the root node with this column slice relaxation.

## 7. Computational results

The computations were carried out with the following hardware and software configuration. Processor: Intel(R) Core(TM) i5-3320M CPU at 2.60GHz; operating system: Ubuntu 14.04.3 LTS with 3.13.0-67-generic kernel; the state-of-the-art integer programming solver Gurobi [26] was called through its API from Python 2.7.6.; the graph library `NetworkX` [27] 1.9.1 was used.

### *7.1. Checking correctness with brute-force and randomized testing*

The algorithms proposed in Sections 4 and 6 were first cross-checked as follows. All bipartite graphs with bipartite node set of cardinality $n = 1 \dots 6$ were generated with `nauty` [39]. The cost of the optimal tearing was computed for each graph with both algorithms, then cross-checked for equality. A similar cross-checking was carried out with random bipartite graphs of random size (bipartite node sets of cardinality

$7 \ldots 30$) and varying sparsity, generated with `networkx` [8, 27]. The algorithms agreed in all tested cases.

## *7.2. Ordering the steady-state model of a distillation column*

The bipartite graph corresponding to the steady-state model equations of the distillation column of [32] (with $N = 50$ stages) has 1350 nodes in each bipartite node set, 3419 edges out of which 2902 are considered as feasible assignments. The undirected graph has more the $10^7$ simple cycles (possibly several orders of magnitude more); enumerating all of them is not tractable in reasonable time. The algorithm of Section 4 has nevertheless no difficulty finding and proving the optimal ordering in 2.5 seconds. The optimal tearing has cost 53 and the final cycle matrix had 285 rows when the algorithm terminated. As this example shows, size is not an appropriate measure of the difficulty.

## *7.3. Performance on the COCONUT Benchmark*

*The need for highly structured matrices.* It is not tractable to perform a brute-force search for matrices that trigger poor performance: The search space is already too large for $n = 7$. The randomized tests revealed that the integer programming approach of Section 4 can become impractical (too slow) if the graph is dense and the cardinality of at least one of the two bipartite node sets exceeds 12. See also Section 4.3 where tests with full graphs were carried out. However, apart from this, even the randomized tests did not prove to be useful for finding matrices that lead to poor performance of the algorithms. Hand-crafted, highly structured matrices, such as the one on the right of Figure 4, cause significantly worse performance in the custom branch and bound algorithm than any of the 10 000 randomly generated matrices of the same size.

In order to find highly structured sparsity patterns that are difficult to order optimally, a series of experiments were carried out with the COCONUT Benchmark [43], a collection of test problems. Since the present paper focuses on systems of equations, and the COCONUT Benchmark consists of optimization problems, compromises had to be made. It will be discussed in the corresponding paragraphs how an appropriate subset of the COCONUT Benchmark was selected.

*Initial row order: running the algorithm 12 times.* It has been observed with highly structured matrices that changing the initial order of the rows of the input matrix can lead to significant variation in performance. To avoid such biases, each matrix has been ordered 12 times in our tests: Each matrix was ordered starting with the original row order (1.), then with the reverse of the original row order (2.), and with 10 random row permutations (3–12). We consider a problem solved if the custom branch and bound algorithm can prove optimality in 10 seconds in *all* 12 cases; we consider the tearing suboptimal otherwise. It is important to emphasize that even if the algorithm fails to prove optimality, it delivers a reasonable ordering together with a rigorous lower bound on the cost of the optimal tearing. We now describe the experiments in details.

*Ordering the Jacobian of the equality constraints.* A subset of the COCONUT Benchmark was selected in this experiment, consisting of 316 problems, where (a) the problems do not have any inequality constraints, (b) the problems have at least 8 equality constraints, (c) the Jacobian of the constraints is not full. The rationale for these requirements is as follows. Extending the proposed method to inequality constraints will be the subject of another paper; therefore, we excluded all problems having inequalities. The reason for the size requirement (b) is that below this size, one can easily enumerate and evaluate all the possibilities in a brute-force fashion. Adding these problems to the performance test does not add any value since neither of the proposed algorithms have any difficulty solving them. As for requirement (c), the proposed custom branch and bound algorithm solves the case of the full matrix immediately on the root node; moreover, there is nothing to be gained with tearing if the Jacobian is full. The results are summarized in Table 1; the corresponding subset of the COCONUT Benchmark and the Python source code for reproducing the computations are available at [2].

*Ordering the Jacobian of the first-order optimality conditions.* Solving the first-order optimality conditions involves solving a system of equations [40, Ch. 12.3]. In this experiment, the structural sparsity pattern of this system was ordered:

$$K = \begin{bmatrix} H & J^T \\ J & I \end{bmatrix}, \tag{21}$$

Table 1: Five-number summary of the problem size distributions when ordering the Jacobian of the equality constraints with a time-limit of 10 seconds. Optimal means that the branch and bound algorithm of Section 6 could prove optimality of the tearing found in all 12 runs; the result is considered suboptimal otherwise. See the text for details.

| | Number of rows | | | | | |
|---|---|---|---|---|---|---|
| | min | lower quartile | median | upper quartile | max | count |
| All problems | 8 | 19 | 160 | 2000 | 14000 | 316 |
| Optimal | 8 | 11 | 55 | 990 | 13800 | 225 |
| Suboptimal | 25 | 237 | 1024 | 3375 | 14000 | 91 |

where $H \in \mathbb{R}^{n\times n}$ is the Lagrangian Hessian, and $J \in \mathbb{R}^{m\times n}$ is the Jacobian of the equality constraints. Note that $K$ is symmetric but neither of the algorithms exploit it.

Similarly to the previous tests, a subset of the COCONUT Benchmark was selected such that (a) the problems do not have any inequality constraints, (b) the corresponding $K$ has at least 8 rows (and therefore at least 8 columns), (c) neither $H$ nor $J$ is full, (d) $H$ is not empty, (e) $K$ has full structural rank. There are 376 such problems. The results are summarized in Table 2; the corresponding subset of the COCONUT Benchmark and the Python source code for reproducing the computations are available at [2].

Table 2: Five-number summary of the problem size distributions when ordering the Jacobian of the first-order optimality conditions (see Equation (21)) with a time-limit of 10 seconds. Optimal means that the branch and bound algorithm of Section 6 could prove optimality of the tearing found in all 12 runs; the result is considered suboptimal otherwise. See the text for details.

| | Number of rows of $K$ | | | | | |
|---|---|---|---|---|---|---|
| | min | lower quartile | median | upper quartile | max | count |
| All problems | 8 | 18 | 176 | 2509 | 33997 | 376 |
| Optimal | 8 | 12 | 18 | 29 | 30002 | 183 |
| Suboptimal | 29 | 351 | 2048 | 10399 | 33997 | 193 |

## 8. Conclusions

Two exact algorithms for tearing were proposed in the present paper: (i) an algorithm based on integer programming, and (ii) a custom branch and bound algorithm. The integer programming based approach has no difficulty finding and proving the optimal ordering of the steady-state model equations of the distillation column of [32] with $N = 50$ stages corresponding to a matrix with 1350 rows and 1350 columns (Sec. 7.2). Despite this success, the integer programming approach tends to be impractical for *dense* matrices if the number of rows exceeds 12. The cause of the inefficiency is permutation symmetry: Given a Hessenberg form that corresponds to a feasible solution of the integer program (14), there are typically many row permutations that, after an appropriate column permutation, realize the same upper envelope, but the cost only depends on the upper envelope. One could dynamically add further constraints to the integer program (14) to break this permutation symmetry (similarly to how the cycle matrix is extended). However, we decided to follow another approach to mitigate this issue.

While the integer programming approach solves the problem of tearing indirectly (the solution of (14) only gives a bipartite matching), the custom branch and bound algorithm tackles tearing by constructing the Hessenberg form directly. This more natural formulation and the full control over the branch and bound search facilitates improvements such as organizing the search (best-first and depth-first search), better relaxations that yield better lower bounds (Sections 6.2 and 6.3), avoiding repeated work by memoization (keeping track of the explored submatrices), exploiting independent subproblems (the bipartite graph becoming disconnected), implementing custom exclusion rules (the back-track rule of [31]), etc. Based on the performance on the COCONUT Benchmark, see Section 7.3, we consider the proposed branch and bound algorithm practical for the purposes of global optimization [5–7], and in cases where systems with the same sparsity pattern are solved repeatedly and the time spent on tearing pays off. Even if the algorithm fails to prove optimality within the user-defined time limit, it delivers a reasonable ordering together with a rigorous lower bound on the cost of the optimal tearing.

*Future work.* The proposed branch and bound algorithm can probably be improved further with a more sophisticated implementation. Examples include the followings. Extending the two-sided algorithm of Fletcher & Hall [24], and putting it into a branch and bound context would most likely lead to significant improvements both in speed and robustness. The two-sided algorithm could also be used to improve the lower bound. Another example for improvements is a simplifier that transforms the problem into an equivalent problem that is easier to solve. The branch and bound algorithm of Section 6 does not have any simplifier. Simplifications are, e.g., removing full rows and full columns, removing duplicate rows and duplicate columns (duplicate: have the same sparsity pattern), removing dominated rows and columns, etc. After solving the simplified problem, one then has to reconstruct the solution to the original problem. These simplifications, together with the exclusion rule of [31], *enforce partial order* among the rows, and therefore attenuate the harmful effects of permutation symmetry.

While analyzing the sparsity patterns where the proposed method failed to prove optimality in 10 seconds, it become obvious that a robust implementation also needs *strong global information* derived from properties of the entire graph. The details of how such global information can be derived and used are another subject for future research. In any case, patterns like the one on the right of Figure 4, which currently triggers poor performance as the size is increased, should be solved immediately on the root node if global information is used.

As discussed in [4], the current objective of tearing has issues. Future research should target better objective functions. An appealing candidate is to decompose the problem into smaller subproblems while minimizing the largest subproblem size [46]. Finally, independently of all the prospective research directions listed so far, improving the currently very conservative method of finding feasible assignments (c.f. Section 2) would give more freedom for any future tearing algorithm, hence it would lead to potentially better orderings.

**Acknowledgement**

The research was funded by the Austrian Science Fund (FWF): P27891-N32. In addition, support by the Austrian Research Promotion Agency (FFG) under project number 846920 is gratefully acknowledged.

## Appendix A The upper envelope and its relation to the row and column counts

In a square lower Hessenberg form, the first nonzero entries of the columns form the **upper envelope**. We say that we **walk the envelope**, when we walk from the top left corner to the bottom right corner along the (upper) envelope as follows: We always step either to the right or downwards, and we always move as long as we can before having to change the direction. The row count $r_i$ is the number of steps we make to the right on the top of row $i$. Now we walk the envelope in reverse, that is, from the bottom right corner to the top left corner, then the column count $c_j$ is the number of steps that we make upwards immediately before column $j$. See also Figure 5.

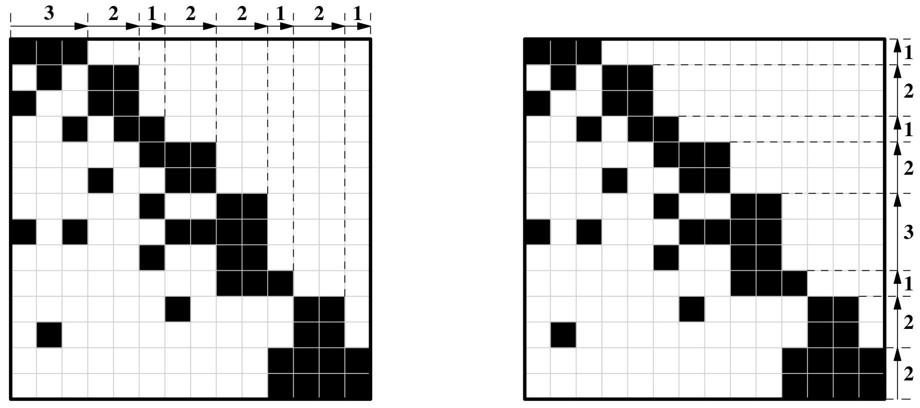

Figure 5: Left: Walking the upper envelope from the top left corner to the bottom right corner gives the row counts. Right: Walking from the bottom right corner to the top left corner gives the column counts.

## Appendix B Lower bound on the minimum cost ordering

For the full rank matrix $A \in \mathbb{R}^{n \times n}$ we derive the following inequality:

$$z^* \geq \min_j c_j - 1, \text{ where } j \in \text{column indices of } A; \tag{22}$$

$z^*$ denotes the optimal cost ordering of $A$ to lower Hessenberg form, and the cost defined as in Section 6; $c_j$ is the column count of column $j$, see Section 5. Furthermore, we define $x^+$ as

$$x^+ = \max(0, x).$$

Since $A$ is a square matrix, we have

$$\sum_i r_i = \sum_j c_j = n, \tag{23}$$

see also Appendix A.

With these notations, the cost $z$ of a given permutation to lower Hessenberg form is

$$z = \sum_i (r_i - 1)^+ \tag{24}$$

according to our definition. Since

$$(r_i - 1)^+ \geq r_i - 1 \tag{25}$$

it follows that

$$z = \sum_i (r_i - 1)^+ \geq \left[\sum_i (r_i - 1)\right]^+ = \left[\left(\sum_i r_i\right) - n)\right]^+. \tag{26}$$

From (23) we have

$$\left[\left(\sum_i r_i\right) - n)\right]^+ = \left[\left(\sum_j c_j\right) - n)\right]^+. \tag{27}$$

Let

$$\underline{c} = \min_j c_j, \tag{28}$$

and with $\underline{c}$ we can continue as follows:

$$\left[\left(\sum_j c_j\right) - n)\right]^+ \geq \left[n\left(\min_j c_j\right) - n\right]^+ = (n\underline{c} - n)^+ = n(\underline{c} - 1)^+ \geq \underline{c} - 1. \tag{29}$$

To summarize (24)–(29):

$$z \geq \underline{c} - 1 = \min_j c_j - 1; \tag{30}$$

that is, we have proved (22).

## References

[1] T. Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, July 2009.

[2] A. Baharev, 2015. Exact and heuristic methods for tearing. URL `https://sdopt-tearing.readthedocs.org`.

[3] A. Baharev, H. Schichl, and A. Neumaier. An exact method for the minimum feedback arc set problem. Submitted, 2015.

[4] A. Baharev, H. Schichl, and A. Neumaier. Tearing systems of nonlinear equations – I. A review. Submitted, 2016.

[5] A. Baharev, F. Domes, and A. Neumaier. Sampling solutions of sparse nonlinear systems. Submitted, 2015. URL `http://www.mat.univie.ac.at/~neum/ms/maniSol.pdf`.

[6] A. Baharev, L. Kolev, and E. Rév. Computing multiple steady states in homogeneous azeotropic and ideal two-product distillation. *AIChE Journal*, 57:1485–1495, 2011.

[7] A. Baharev and A. Neumaier. A globally convergent method for finding all steady-state solutions of distillation columns. *AIChE J.*, 60:410–414, 2014.

[8] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Phys. Rev. E*, 71:036113, 2005.

[9] L.T. Biegler, I.E. Grossmann, and A.W. Westerberg. *Systematic Methods of Chemical Process Design*. Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[10] R.P. Brent. *Algorithms for Minimization without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[11] F.E. Cellier and E. Kofman. *Continuous system simulation*. Springer Science & Business Media, 2006.

[12] J.H. Christensen. The structuring of process optimization. *AIChE Journal*, 16(2): 177–184, 1970.

[13] T.H. Cormen, C. Stein, R.L. Rivest, and C.E. Leiserson. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, USA, 3rd ed., 2009.

[14] T.A. Davis. Direct methods for sparse linear systems. In N.J. Higham, editor, *Fundamentals of algorithms*. Philadelphia, USA: SIAM, 2006.

[15] T.J. Dekker. Finding a zero by means of successive linear interpolation. In B. Dejon and P. Henrici, editors, *Constructive aspects of the fundamental theorem of algebra*, pp. 37–51. London: Wiley-Interscience, 1969.

[16] N. Deo. *Graph theory with applications to engineering and computer science*. Prentice-Hall, Inc., Englewood Cliffs, NJ, USA, 1974.

[17] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1986.

[18] A.L. Dulmage and N.S. Mendelsohn. Coverings of bipartite graphs. *Can. J. Math.*, 10: 517–534, 1958.

[19] A.L. Dulmage and N.S. Mendelsohn. A structure theory of bipartite graphs of finite exterior dimension. *Trans. Royal Society of Canada. Sec. 3.*, 53:1–13, 1959.

[20] A.L. Dulmage and N.S. Mendelsohn. Two Algorithms for Bipartite Graphs. *J. Soc. Ind. Appl. Math.*, 11:183–194, 1963.

[21] H. Elmqvist and M. Otter. Methods for tearing systems of equations in object-oriented modeling. In *Proceedings ESM'94, European Simulation Multiconference, Barcelona, Spain, June 1–3*, pp. 326–332, 1994.

[22] H. Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund University, Sweden, May 1978.

[23] A.M. Erisman, R.G. Grimes, J.G. Lewis, and W.G.J. Poole. A structurally stable modification of Hellerman-Rarick's $P^4$ algorithm for reordering unsymmetric sparse matrices. *SIAM J. Numer. Anal.*, 22:369–385, 1985.

[24] R. Fletcher and J.A.J. Hall. Ordering algorithms for irreducible sparse linear systems. *Annals of Operations Research*, 43:15–32, 1993.

[25] A. George and J.W.H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[26] Gurobi. Gurobi Optimizer Version 6.0. Houston, Texas: Gurobi Optimization, Inc., May 2015. (software program). `http://www.gurobi.com`, 2014.

[27] A.A. Hagberg, D.A. Schult, and P.J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pp. 11–15, Pasadena, CA USA, August 2008.

[28] E. Hansen and G.W. Walster. *Global Optimization Using Interval Analysis*. New York, NY: Marcel Dekker, Inc., 2nd ed., 2003.

[29] E. Hellerman and D.C. Rarick. Reinversion with preassigned pivot procedure. *Math. Programming*, 1:195–216, 1971.

[30] E. Hellerman and D.C. Rarick. The partitioned preassigned pivot procedure ($P^4$). In D.J. Rose and R.A. Willoughby, editors, *Sparse Matrices and their Applications*, The IBM Research Symposia Series, pp. 67–76. Springer US, 1972.

[31] R. Hernandez and R.W.H. Sargent. A new algorithm for process flowsheeting. *Computers & Chemical Engineering*, 3(14):363–371, 1979.

[32] E. Jacobsen and S. Skogestad. Multiple steady states in ideal two-product distillation. *AIChE Journal*, 37:499–511, 1991.

[33] D.M. Johnson, A.L. Dulmage, and N.S. Mendelsohn. Connectivity and reducibility of graphs. *Can. J. Math*, 14:529–539, 1962.

[34] V. Kreinovich and R.B. Kearfott. Beyond Convex? Global Optimization is Feasible Only for Convex Objective Functions: A Theorem. *Journal of Global Optimization*, 33(4): 617–624, 2005.

[35] W. Lee, J.H. Christensen, and D.F. Rudd. Design variable selection to simplify process calculations. *AIChE Journal*, 12(6):1104–1115, 1966.

[36] T.D. Lin and R.S.H. Mah. Hierarchical partition-a new optimal pivoting algorithm. *Mathematical Programming*, 12(1):260–278, 1977.

[37] H.M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3(3):255–269, 1957.

[38] S.E. Mattsson, M. Otter, and H. Elmqvist. Modelica hybrid modeling and efficient simulation. In *Decision and Control, 1999. Proceedings of the 38th IEEE Conference on*, vol. 4, pp. 3502–3507, 1999.

[39] B.D. McKay and A. Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014.

[40] J. Nocedal and S.J. Wright. *Numerical Optimization*. Springer, New York, USA, second ed., 2006.

[41] A. Pothen and C.J. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw.*, 16:303–324, 1990.

[42] B. Schwikowski and E. Speckenmeyer. On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, 117(13):253–265, 2002.

[43] O. Shcherbina, A. Neumaier, D. Sam-Haroud, X.H. Vu, and T.V. Nguyen. Benchmarking global optimization and constraint satisfaction codes. In C. Bliek, C. Jermann, and A. Neumaier, editors, *Global Optimization and Constraint Satisfaction*, vol. 2861 of *Lecture Notes in Computer Science*, pp. 211–222. Springer Berlin Heidelberg, 2003. URL `http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html`.

[44] M.A. Stadtherr and E.S. Wood. Sparse matrix methods for equation-based chemical process flowsheeting–I: Reordering phase. *Computers & Chemical Engineering*, 8(1): 9–18, 1984.

[45] M.A. Stadtherr and E.S. Wood. Sparse matrix methods for equation-based chemical process flowsheeting–II: Numerical Phase. *Computers & Chemical Engineering*, 8(1): 19–33, 1984.

[46] D.V. Steward. Partitioning and tearing systems of equations. *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, 2(2):345–365, 1965.

[47] SymPy Development Team. *SymPy: Python library for symbolic mathematics*, 2014. URL `http://www.sympy.org`.

[48] P. Täuber, L. Ochel, W. Braun, and B. Bachmann. Practical realization and adaptation of Cellier's tearing method. In *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, pp. 11–19, New York, NY, USA, 2014. ACM.

[49] W.F. Tinney and J.W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55:1801–1809, 1967.