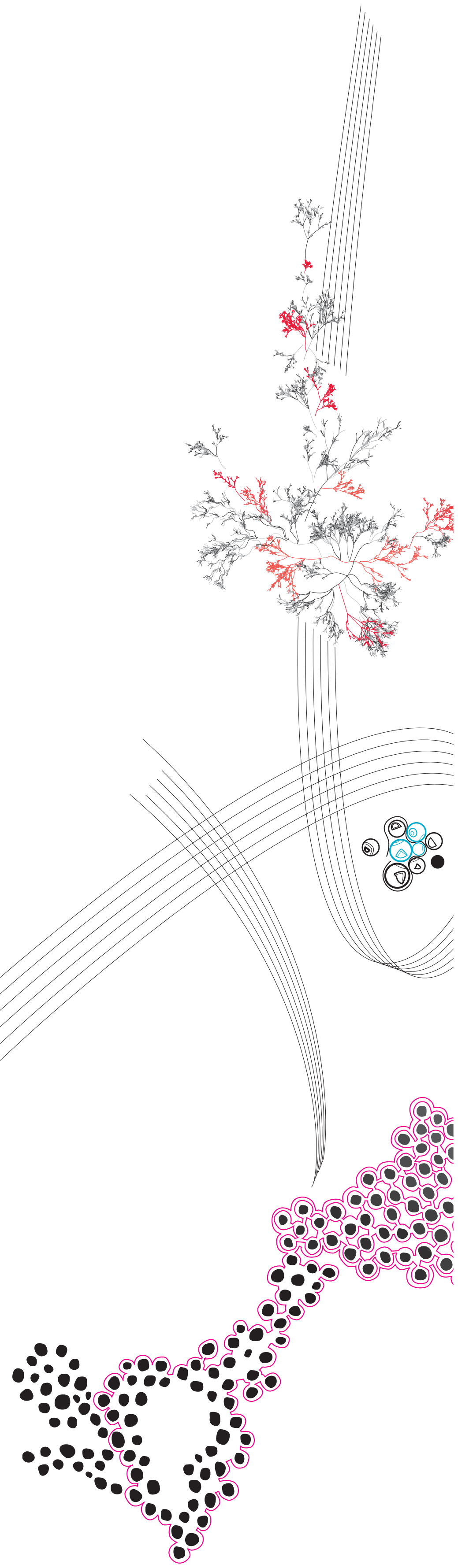
BSc Thesis Applied Mathematics



Implementation of Balanced Matrix Recognition Algorithms

Henk Kraaij

Supervisor: M. Walter

July, 2023

Department of Applied Mathematics
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Preface

I want to thank my supervisor Matthias Walter for his guidance during the process, his valuable feedback and our helpful discussions.

Implementation of Balanced Matrix Recognition Algorithms

Henk. J.L. Kraaij[*]

July, 2023

Abstract

A $\{0, \pm 1\}$ matrix is balanced if it does not contain a square submatrix with two nonzero elements per row and column in which the sum of all entries is 2 modulo 4. Zambelli [9] provided a polynomial algorithm to test balancedness of a matrix. This paper discusses the algorithm and its implementation, and compares it against a naive approach using an exponential algorithm. The running time of the polynomial algorithm is $O(n^9)$ for $\{0, 1\}$ matrices and $O(n^{11})$ for $\{0, \pm 1\}$ matrices, where n is the number of rows and columns of the input matrix.

Keywords: balanced matrix, recognition, algorithm, implementation, polynomial

1 Introduction

A $\{0, \pm 1\}$ matrix is *balanced* if it does not contain a square submatrix with two nonzero elements per row and column in which the sum of all entries is 2 modulo 4. Figure 1 contains an example of two 3 by 3 matrices. U is not balanced and B is balanced.

$$U = \begin{array}{c} \\ R_0 \\ R_1 \\ R_2 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} C_0 & C_1 & C_2 & C_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \end{array}, \quad B = \begin{array}{c} \\ R_0 \\ R_1 \\ R_2 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} C_0 & C_1 & C_2 & C_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

FIGURE 1: An unbalanced matrix U and a balanced matrix B.

We can show that the matrix U is not balanced by looking at the 3 by 3 square submatrix consisting of rows $\{R_0, R_1, R_2\}$ and columns $\{C_0, C_2, C_3\}$. Note that it has two nonzero entries in each row and column. The sum of these entries is $6 \equiv 2 \pmod 4$, so U is not balanced.

Matrix B has two square submatrices such that each row and column has two nonzero entries. One consists of the rows $\{R_1, R_2\}$ and columns $\{C_0, C_2\}$. In this submatrix the sum of the entries is $4 \equiv 0 \pmod 4$. The other submatrix consists of the rows $\{R_0, R_2\}$ and columns $\{C_0, C_3\}$. In this submatrix the sum of the entries is also $4 \equiv 0 \pmod 4$. Hence, B is balanced.

Zambelli [9] provided two polynomial time algorithms. One algorithm can check if a $\{0, \pm 1\}$ matrix is balanced with a running time of $O(n^{11})$, and the other can check if a $\{0, 1\}$ matrix is balanced with a running time of $O(n^9)$, where n is the number of rows plus the number of columns of the input matrix. These algorithms are explained in Section 3. In this paper we discuss an implementation of these polynomial algorithms, which was

[*]Email: h.j.l.kraaij@student.utwente.nl

programmed in Java. We also discuss the implementation of a naive exponential algorithm that might be faster for smaller matrices, which is explained in Section 2. In Section 4 we compare the running times of the polynomial algorithm against an exponential algorithm.

Balanced matrices are used in integer programming. They are closely related to totally unimodular matrices, which have the extra requirement that each submatrix must have determinant -1, 0 or +1. Balanced matrices do not have this requirement, they can have submatrices with determinant -2 for example. Totally unimodular matrices are a proper subset of balanced matrices [8]. Balanced matrices are of special interest in integer programming, as several polytopes arising in classical optimization problems have only integral vertices when the constraint matrix is balanced [9]. Some important applications are *generalized set packing*, *set covering* and *set partitioning* polytopes. These polytopes have only integral vertices when the matrix is balanced [2, 3].

1.1 Notations and definitions

In this paper, we will work with the same definitions as used by Zambelli [9]. This section contains all those definitions. It will be useful to work with the bipartite representation of the matrix. Given a $\{0,1\}$ matrix A, the *bipartite representation of* A is the bipartite graph G where the two sides of the bipartition are the sets R and C of rows and columns of A, respectively, and there is an edge between $i \in R$ and $j \in C$ if and only if $a_{ij} = 1$. An example of a $\{0,1\}$ matrix and its bipartite graph representation can be seen in Figure 2.

$$U = \begin{array}{c} \\ R_0 \\ R_1 \\ R_2 \end{array} \begin{array}{c} \begin{array}{ccc} C_0 & C_1 & C_2 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \end{array} \qquad G_U =$$

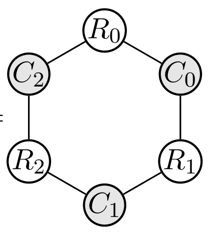


Figure 2: A $\{0,1\}$ matrix U and its bipartite representation G_U.

A $\{0,1\}$ matrix A is balanced if and only if its bipartite representation does not contain a *hole* of length 2 (mod 4) as an induced subgraph. A *hole* is a chordless cycle. We call a hole *odd* if it has length 2 (mod 4). A bipartite graph is *balanced* if it does not contain an odd hole.

$$B = \begin{array}{c} \\ R_0 \\ R_1 \\ R_2 \end{array} \begin{array}{c} \begin{array}{ccc} C_0 & C_1 & C_2 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \end{array} \qquad G_B =$$

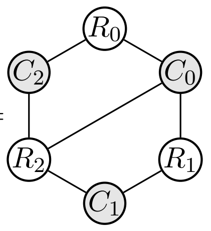


Figure 3: A $\{0,1\}$ matrix B and its bipartite representation G_B.

Figure 3 contains a bipartite graph G_B, which is the bipartite representation of matrix B. As an example, notice that G_B contains no odd hole. It does contain a cycle of length 2 (mod 4), $R_0C_0R_1C_1R_2C_2R_0$, but this cycle has a chord C_0R_2, so it is not an odd hole. Hence G_B is balanced.

For general $\{0, \pm 1\}$ matrices, we will work with their signed bipartite representation. A *signed bipartite graph* is a pair (G, σ) where G is a bipartite graph and σ is a *signing* of the edges, that is a function from $E(G)$ to $\{1,-1\}$, where $E(G)$ denotes the edge set of graph G.

Given a $\{0,\pm1\}$ matrix A, the *signed bipartitle representation of* A is the signed bipartite graph (G,σ) where G is the bipartite representation of the $\{0,1\}$ matrix underlying A and σ is defined, for each edge ij, by $\sigma = a_{ij}$. For any subgraph F of G, we define

$$\sigma(F) = \sum_{e \in E(F)} \sigma(e).$$

A $\{0,\pm1\}$ matrix A is balanced if and only if its signed bipartite representation does not contain a hole H such that $\sigma(H) \equiv 2 \pmod 4$ as an induced subgraph. We will say that such a hole is *unbalanced*, and a signed bipartite graph is *balanced* if it contains no unbalanced hole. Given a cut induced by $(S,\bar{S})$ of G (where S is a subset of the nodes of G, and $\bar{S} = V(G)\backslash S$), we define a signing σ' by

$$\sigma'(ij) = \begin{cases} \sigma(ij) & \text{if } ij \notin (S,\bar{S}) \\ -\sigma(ij) & \text{if } ij \in (S,\bar{S}) \end{cases}.$$

(G,σ) is balanced if and only if (G,σ') is balanced, since for any hole H, $\sigma(H) \equiv \sigma'(H)$ (mod 4). This is because for any signing, there is always an even number of edges in H that change their sign. Each single edge that changes sign adds or subtracts 2 from $\sigma(H)$. Since there are always an even number of edges that change their sign, $\sigma(H) \equiv \sigma'(H)$ (mod 4). We call this operation *scaling* along the cut induced by $(S,\bar{S})$.

$$A = \begin{array}{c} \\ R_0 \\ R_1 \\ R_2 \end{array} \begin{array}{c} \begin{array}{ccc} C_0 & C_1 & C_2 \end{array} \\ \begin{pmatrix} -1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \end{array}$$

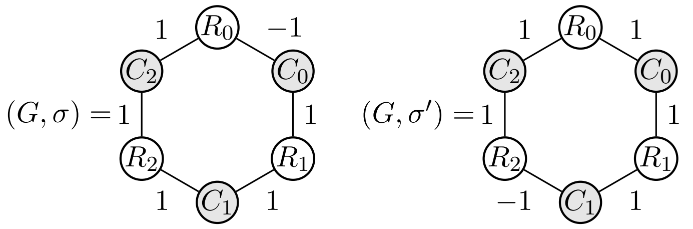


FIGURE 4: A balanced matrix A, its signed bipartite representation (G,σ), and the resulting graph (G,σ') after scaling (G,σ) along the cut induced by $(\{R_0,C_2,R_2\},\{C_0,R_1,C_1\})$.

Figure 4 contains an example of a balanced signed bipartite graph G where each edge e has the label $\sigma(e)$. Note that the hole $H = R_0C_0R_1C_1R_2C_2R_0$ contained in G has $\sigma(H) \equiv \sigma'(H) \equiv 0 \pmod 4$.

In the remainder of this paper, G will always be a bipartite graph. When we say that a graph G *contains* a graph F, we will always mean that G contains a graph isomorphic to F as an induced subgraph. Given a set X of nodes of G, we denote by $G[X]$ the subgraph of G induced by X. Given a subgraph F of G and a node x of G, we denote the set of neighbours of x in F by $N_F(x)$.

Given a path or a hole Q, we will denote by $|Q|$ the length of Q, that is the number of edges. Given a graph F and two nodes x and y of F, $d_F(x,y)$ denotes the length of the shortest path between x and y contained in F. If P is a chordless path and x and y are two nodes of P, we will denote by $P(x,y)$ the unique subpath of P between x and y. The *interior* of P is the set of all nodes of P except the endpoints of P.

A path P and a node m are said to be *close* if m belongs to or has a neighbour in the interior of P. Two paths P_1 and P_2 are said to be *close* to each other if there is a node in the interior of P_1 that is close to P_2.

The following two graphs will play an important role in the remainder of the paper, because these graphs always imply the existence of an unbalanced hole.

1.1.1 3-path configurations

Given two nonadjacent nodes x and y in distinct sides of the bipartition, a *3-path configuration* between x and y is a graph consisting of three chordless paths P_1, P_2, P_3 between x and y such that, for every $1 \leq i < j \leq 3$, no node in the interior of P_i belongs to or has a neighbour in the interior of P_j. We say that P_1, P_2, P_3 form a 3-path configuration.

An example of a graph containing a 3-path configuration can be seen in Figure 5. The three dashed lines in this graph represent chordless paths between a_i and b_i for $i \in \{1, 2, 3\}$ such that each pair of these paths is not close to each other. Note that x and y must be nonadjacent, so all of the paths forming the 3-path configuration must at least have a length equal to 3.

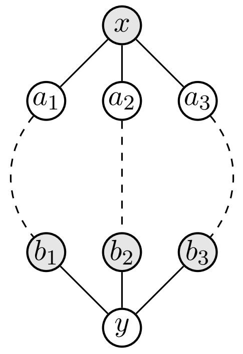


FIGURE 5: A 3-path configuration. The dashed lines indicate potentially longer paths.

1.1.2 Wheels

A *wheel* consists of a hole H and a node v outside H with at least 3 distinct neighbours in H, and is denoted by (H, v). A wheel (H, v) for which v has k neighbours in H is said to be a *k-wheel.* A *sector* of (H, v) is a maximal subpath of H with no neighbours of v in its interior. The *spokes* of (H, v) are the edges of G of the form uv where u is a neighbour of v in H. (H, v) is an *odd wheel* if it is a wheel and v has an odd number of neighbours in H.

Figure 6 contains an example of a 5-wheel (H, v). Note that because it is a 5-wheel, there are exactly 5 spokes, i.e. there are no other neighbours of v in the hole H.

The importance of 3-path configurations and odd wheels is explained by the next proposition.

Proposition 1.1. *If G is a bipartite graph containing a 3-path configuration or an odd wheel, then (G, σ) is not balanced for any signing σ.*

The proof for this proposition can be found in [9].

2 Naive recognition algorithm

To get a good estimate of the speed of the implemented polynomial algorithm, we will compare it to a naive algorithm. This naive algorithm checks for balancedness by iterating

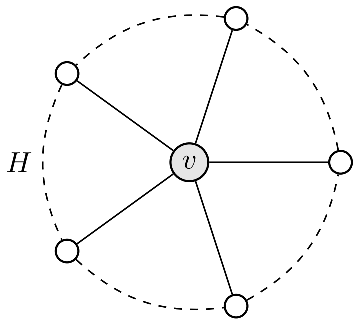


FIGURE 6: A 5-wheel. The dashed lines indicate potentially longer paths.

over each square submatrix. In Algorithm 1 we see an overview of the naive algorithm. This algorithm runs in exponential time, since we have to iterate over almost all combinations of rows and columns and the number of combinations grows exponentially. Note that R is the set of rows and C is the set of columns of A.

Algorithm 1 Naive recognition algorithm

```
Input: Matrix $A$
Output: True if the matrix is balanced, otherwise false
1: for all $2 \leq size \leq min(|R|, |C|)$ do
2:     for all $X \subseteq R$ with $|X| = size$ do
3:         for all $Y \subseteq C$ with $|Y| = size$ do
4:             if $A_{X,Y}$ has two nonzero entries per row and column then
5:                 Count the sum $s$ of the entries of $A_{X,Y}$.
6:                 if $s = 2 \pmod 4$ then
7:                     return false
8: return true
```

Proposition 2.1. *Algorithm 1 correctly determines if an m-by-n matrix A is balanced. The running time of the algorithm is* $O(2^{m+n})$.

I implemented this algorithm by recursively iterating over all the subsets of R and C.

There are some optimizations that were implemented to make this algorithm faster. First, there is a pre-processing step, in which we remove all the rows and columns with fewer than two nonzero entries. We can do this because any square submatrix that includes these rows/columns would not correspond to an unbalanced hole.

We can do something similar before line 3 of the algorithm. When we have a selection of rows, we can skip over all the columns that do not have exactly two nonzero entries in $A_{X,C}$.

Another optimization that was implemented is that in step 3, when we make Y, we can be smart about which columns we include. Before we select any column, we make a list *rowCount* in which we can count the number of nonzero entries in each row with the current selection of columns. When we select a column, we check that there are no rows that would get more than two nonzero entries in their row if we would include it. If the inclusion of the column would cause one of the rows to have more than two nonzero entries, we skip the column.

We also make sure that each column we add makes the count of one of the rows go from 0 to 1. This makes sure that we do not get a situation where the counts of the rows are all either 2 or 0, which would mean that we have found a hole that is smaller than the current size we are looking at. We can skip this because the algorithm will have checked the hole in a previous iteration. This is illustrated by the following example.

Figure 7 contains an example of a matrix A that we could be iterating over. Let us assume that we are currently looking at the submatrices of size 4, and we have $X = \{R_0, R_1, R_2, R_3\}$ and $Y = \{C_0, C_1\}$. We need to select a third column. We can not select C_2 because R_0 already has two nonzero entries. We can also not select C_3 because then we would get the hole $A_{\{R_0,R_1,R_2\},\{C_0,C_1,C_2\}}$ in our submatrix, which we have already checked when we iterated over the submatrices of size 3. C_4 is the only column left, but we need two more columns so we can also skip C_4.

$$rowCount = \begin{array}{c} R_0 \\ R_1 \\ R_2 \\ R_3 \end{array}\begin{pmatrix} 2 \\ 1 \\ 1 \\ 0 \end{pmatrix} \qquad A = \begin{array}{c|ccccc} & C_0 & C_1 & C_2 & C_3 & C_4 \\ \hline R_0 & -1 & 1 & 1 & 0 & 0 \\ R_1 & 1 & 0 & 0 & 1 & 1 \\ R_2 & 0 & 1 & 0 & 1 & 0 \\ R_3 & 0 & 0 & 1 & 0 & 1 \end{array}$$

FIGURE 7: A matrix A and the *rowCount* list that keeps track of the number of nonzero entries in the selected rows $\{R_0, R_1, R_2, R_3\}$ with the current selection of columns $\{C_0, C_1\}$.

3 Polynomial recognition algorithm

In this section we will discuss how the polynomial time algorithm can check if a matrix is balanced. The algorithm consists out of four smaller algorithms. The first step in the algorithm is to change the matrix to the bipartite representation.

We then start by checking if the graph contains a 3-path configuration. From Proposition 1.1 we know that if this is the case, we are guaranteed to have an unbalanced hole in the graph. In Section 3.1 we will discuss this algorithm in more detail. If we find a 3-path configuration, we can output that the graph is unbalanced and stop.

If the graph does not contain a 3-path configuration, we check if the graph contains a *detectable 3-wheel*. This is a 3-wheel where one of the sectors has length 2. See Figure 8 for an example of a detectable 3-wheel. The sector highlighted in blue has length 2, and since there are 3 spokes, the hole H and v form a detectable 3-wheel.

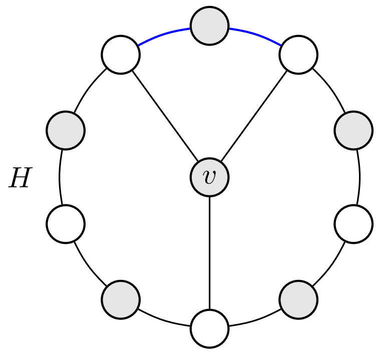


FIGURE 8: A detectable 3-wheel.

A detectable 3-wheel is an odd wheel, so from Proposition 1.1 it follows that if the graph contains a detectable 3-wheel, it contains an unbalanced hole. Section 3.2 will discuss this algorithm in more detail. If we find a detectable 3-wheel, we can output that the graph is unbalanced and stop.

The third algorithm has the goal of creating a set of so called *cleaners*. We will come back to the definitions when we explain this algorithm in Section 3.3. In short, it is guaranteed that at least one of the cleaners will remove the middle node of an odd wheel, if there is one in the graph.

The fourth algorithm iterates over all the cleaners, and in each iteration we try to find an unbalanced hole in the graph. We do this by iterating over every triple of nodes in the graph and finding the shortest paths between them. These paths will form a hole, which we can check for balancedness. The third and fourth algorithms can be combined for signed graphs, which we will discuss in Section 3.3.

3.1 3-path configuration recognition algorithm

In this section we will discuss Algorithm 2.2 from Zambelli [9]. The overall running time of this algorithm is $O(|V(G)|^9)$. The way this algorithm works is that we try to find three paths that have the same endpoints, and that are not close to eachother. If we can find three paths like that, they form a 3-path configuration. Figure 9 shows the general structure of a 3-path configuration, and it uses the same labels that are also used in Algorithm 2. Note that the figure only shows the 3-path configuration in a graph. Also note that the dashed lines form potentially longer chordless paths between a_i and b_i for $i = 1, 2, 3$ and that they are all not close to each other. We can see the path $P_1(m) = xa_1P_1'(m)mP_1''(m)b_1y$ in blue, where $P_1'(m)$ is the shortest path between a_1 and m, and $P_1''(m)$ is the shortest path between b_1 and m. One other note is that in the node m can be in either partition of the bipartite graph.

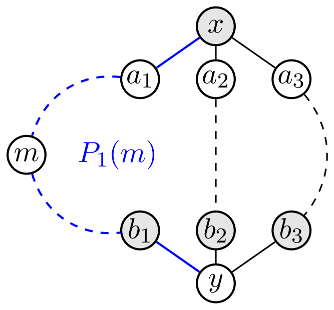


FIGURE 9: Structure of a 3-path configuration.

In Algorithm 2 we will show the algorithm that can find such a 3-path configuration in polynomial time. In the implementation we used Breadth First Search (BFS) to calculate the shortest path between two nodes. In Figure 10 we can see a graph and two of the 3-path configurations that were found by the algorithm.

Algorithm 2 Determine if a graph contains a 3-path configuration [9].

```
Input: Graph $G$
Output: True if the graph contains a 3-path configuration, otherwise false For each 6-tuple
    $a_1, a_2, a_3, b_1, b_2, b_3$ such that:
      • $a_1, a_2, a_3 \in R$ and $b_1, b_2, b_3 \in C$
      • $a_i$ is nonadjacent to $b_j$ for every $i \neq j$
      • there exist nonadjacent $x$ and $y$ such that $x$ is adjacent to $a_1, a_2, a_3$ and $y$ is
        adjacent to $b_1, b_2, b_3$;
    do the following:
1: For $i = 1, 2, 3$, compute the set $X(i)$ of nodes that are not adjacent to any of $x, y, a_j, b_j$
   for $j \neq i$.
2: For $i = 1, 2, 3$, for every node $m \in X(i)$, compute the shortest paths $P'_i(m)$ between
   $a_i$ and $m$ in $G[X(i) \cup a_i]$ and $P''_i(m)$ between $b_i$ and $m$ in $G[X(i) \cup b_i]$ (if they exist).
3: For $i = 1, 2, 3$, for every node $m \in X(i) \cup a_i$, define $P_i(m)$ as follows: if $a_i$ is adjacent
   to $b_i$, then $P_i(a_i) = xa_ib_iy$ and $P_i(m)$ is undefined for every $m \in X(i)$; else $P_i(a_i)$ is
   undefined and for every $m \in X(i)$ satisfying the following
      • $P'_i(m)$ and $P''_i(m)$ both exist
      • $P'_i(m)$ is not close to $P''_i(m)$          ▷ We ensure that $P_i(m)$ is a chordless path.
   let $P_i(m) = xa_iP'_i(m)mP''_i(m)b_iy$, else $P_i(m)$ is undefined.
4: For every $m \in X(i) \cup a_i$ such that $P_i(m)$ is defined, compute the set $Y_i(m)$ of nodes
   that are not close to $P_i(m)$.
5: For every $1 \leq i < j \leq 3$, and for every $m_i \in X(i) \cup a_i$ and every $m_j \in X(j) \cup a_j$, verify
   that the interior of $P_j(m_j)$ is contained in $Y_i(m_i)$. If this is the case, we say that the
   pair $m_i, m_j$ is $(i, j)$-good.          ▷ We ensure that $P_i(m_i)$ and $P_j(m_j)$ are not close.
6: Verify if there exists a triple $m_1, m_2, m_3$ such that $m_i \in X(i) \cup a_i$ for $i = 1, 2, 3$, and such
   that $m_i, m_j$ is $(i, j)$-good for every $1 \leq i < j \leq 3$. If such a triple exists, return true
   and also output the 3-path configuration $P_1(m_1), P_2(m_2), P_3(m_3)$. Otherwise return
   false.
```

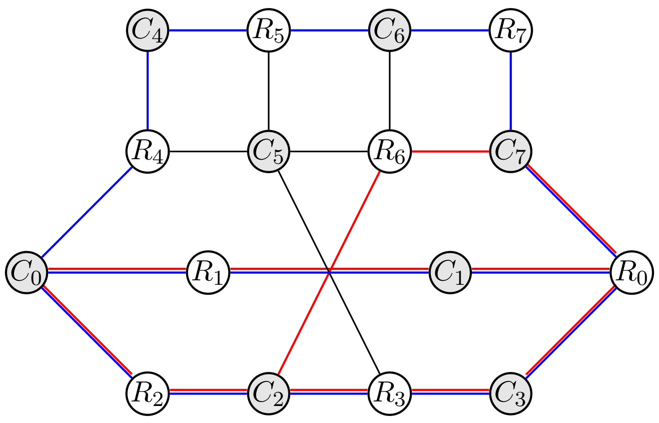


FIGURE 10: Example of a graph containing at least two 3-path configurations, one displayed in blue and one in red.

3.2 Detectable 3-wheel recognition algorithm

In this section we will discuss the Algorithm 3.2 from Zambelli [9]. This algorithm has a polynomial running time of $O(|V(G)|^9)$.

The algorithm is Algorithm 2. Figure 11 illustrates how the algorithm tries to find the detectable 3-wheel. We iterate over every 7-tuple of nodes in this specific configuration. Each greyed out edge with a cross through the middle represents that the nodes at both endpoints must nonadjacent.

During the implementation of this algorithm, I noticed two small errors in Algorithm 3.2 from Zambelli [9]. One was already corrected in a later version of the detectable 3-wheel detection algorithm, in Lemma 5 of [5]. It was missing the v in step 4 of Algorithm 2.

One other correction I made is the fifth point on the list before step one. If we look at Figure 9, we can see that u_1 and v_2 must indeed not be adjacent, or else the hole produced by the algorithm would actually not be a hole. Similarly u_2 must also not be adjacent v_1.

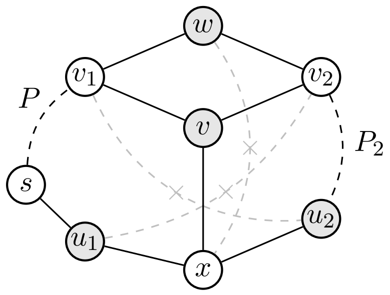


FIGURE 11: Structure of a detectable 3-wheel.

3.3 Finding the smallest unbalanced hole

The following definitions are from Zambelli [9]. Let H be an unbalanced hole. We say that a vertex $x \in V(G)\backslash V(H)$ is major for H if $N_H(x)$ is not contained in a subpath of H of length 2. Note that if we have a wheel (H_v, v), the v is major for H_v. Let $M(H)$ be the set of major vertices for H. We say that H is *clean* if $M(H) = \emptyset$. A set $X \subseteq V(G)\backslash V(H)$ is a *cleaner* for H if $M(H) \subseteq X$ (i.e. if H is clean in $G\backslash X$). A signed bipartite graph (G, σ) is *clean* if G is either balanced or it contains a clean *smallest* unbalanced hole. An unbalanced hole is smallest if it has minimum length among all unbalanced holes.

Algorithm 3 Determine if a graph contains a detectable 3-wheel [9].

```
Input: Graph $G$ containing no 3-path configuration
Output: True if the graph contains a detectable 3-wheel, otherwise false
  For each $u_1, u_2, v, v_1, v_2, w, s$ such that:
    • $v$ and $w$ are both adjacent to $v_1$ and $v_2$
    • there exists a node $x$ such that $x$ is adjacent to $v, u_1, u_2$ but not to $w$
    • $s$ is adjacent to $u_1$
    • either $s = v_1$ or no node in $\{u_2, v, v_2, x, w\}$ is coincident with or adjacent to $s$.
    • $u_1$ is not adjacent to $v_2$, and $u_2$ is not adjacent to $v_1$.
  do the following:
1: Compute the set $X$ of nodes that do not belong to or have a neighbour in $\{u_2, x, v, v_2, w\}$.
2: Compute the shortest path $P$, if one exists, between $v_1$ and $s$ in $G[X \cup v_1]$. If no such path exists, select a different 7-tuple.
3: Verify that the only neighbour of $u_1$ in $P$ is $s$, if this is the case let $P_1 = v_1 P s u_1$, otherwise select a different 7-tuple.
4: Compute the set $Y$ of all nodes that do not belong to or have a neighbour in $P_1 \cup \{v, w, x\}$.
5: Compute, if one exists, a chordless path $P_2$ between $u_2$ and $v_2$ with interior contained in $Y$. If $P_2$ exists, then let $H = w v_1 P_1 u_1 x u_2 P_2 v_2 w$, return true and also output the detectable 3-wheel $(H, v)$. Otherwise return false.
```

I implemented Algorithms 4.4 and 4.6 from Zambelli [9]. These algorithms create polynomially many cleaner sets for unsigned and signed graphs respectively. By Theorem 4.2 and Lemma 4.5 in Zambelli [9] it follows that at least one of these cleaner sets must be a cleaner for the smallest unbalanced hole in the (un)signed graph.

By iterating over these cleaner sets, and applying them to the graph by removing the vertices in the cleaner set from the graph, we can start to look for the smallest unbalanced hole.

Figure 12 shows the structure of Algorithm 6.2 from Zambelli [9]. This algorithm combines the search for cleaner sets and unbalanced holes for signed graphs. We will not go into the details behind this algorithm too much. The algorithm has a polynomial running time of $O(|V(G)|^{11})$. In Figure 12 we see two chordless paths $v_0v_1v_2v_3$ and $u_0u_1u_2u_3$. The following lemma is due to Conforti, Conruéjols, Kapoor and Vušković [4]:

Lemma 3.1. *(Lemma 4.5 in [9]) Let (G, σ) be a signed graph. Let G be the smallest unbalanced hole of (G, σ). Then there exist two edges u_1u_2 and v_1v_2 of H such that every major node for H is adjacent to one of u_1, u_2, v_1, v_2.*

Algorithm 6.2 from Zambelli [9] then provides us with the final step where we use Lemma 3.1 to find the smallest unbalanced hole. We are looking for the structure as we see in Figure 12. The input is a signed graph containing no 3-path configuration or detectable 3-wheel. We iterate over all the chordless paths $v_0v_1v_2v_3$ and $u_0u_1u_2u_3$. Then we remove all the neighbouring vertices of the endpoints of the edges u_1u_2 and v_1v_2. Next we iterate over each node x, and we find the shortest path Q from x to v_0, and we find the shortest path Q' from x to v_2. If these exist, we verify that the Q and Q' are not close to each other, and that they are not close to v_1. If this is the case let $H = v_0v_1v_2Q'xQv_0$,

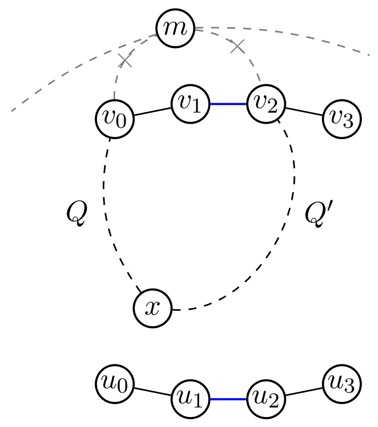


FIGURE 12: Structure of Algorithm 6.2 [9].

and if $\sigma(H) = 2 \pmod 4$, then H is an unbalanced hole. If we do not find any unbalanced hole, G is balanced.

It is important that we are able to remove the nodes that are major for H, because otherwise the major node could make a 'shortcut' in the hole, which would mean that the shortest paths algorithms would not find the hole anymore.

Also note that in Figure 12 both Q and Q' do not go through $u_0u_1u_2u_3$, but it is possible that $u_0u_1u_2u_3$ is part of either path.

For unsigned matrices, the algorithm is a bit simplified which brings the running time down to $O(|V(G)|^9)$. The algorithm can be found in Zambelli [9] (Algorithm 5.2).

4 Computational study

To get an idea of how well the implementations perform, I ran tests on a dataset of randomly generated balanced matrices. More specifically, I used network matrices [8], since these can easily be randomly generated, and they are always balanced (network matrices are totally unimodular). Sections 4.1 and 4.2 explain how the random network matrices were generated.

4.1 Generating random balanced $\{0, \pm 1\}$ matrices

The algorithms were tested on network matrices. Network matrices are a prominent class of totally unimodular matrices. Every totally unimodular matrix is also balanced, so we can use these network matrices to run performance tests on the algorithms. The matrices were generated using a randomly generated tree T. We construct T by starting from a root node. For each node v we add to T, we randomly choose a node u of the already connected nodes from a uniform distribution, and we add a directed edge between them. The direction of the edge is randomly chosen, there is equal chance for the edge to be either u, v or v, u. Next we add edges that are not in the tree. We choose two nodes, s and t, and we add a directed edge s, t between them.

The network matrix rows correspond to the edges in initial tree T, and the columns correspond to the edges in $E \backslash T$. For each edge $s, t \in E \backslash T$, we calculate the path P from s to t in T. If P goes over an edge in T forward, the entry in the row corresponding to that edge and column corresponding to s, t is 1. If it goes over an edge backwards, the entry is -1. And if P does not go over an edge, the entry is 0. Figure 13 contains an example of a network matrix and the graph it was generated from.

$$N = \begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 2 & 0 & 1 & -1 \\ 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 \\ 5 & 0 & -1 & 1 \end{array}$$

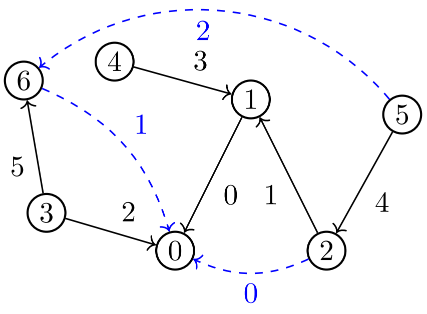


Figure 13: An example of a network matrix N, and the directed graph it was generated from, with T in black and the edges $E \backslash T$ in blue.

4.2 Generating random balanced $\{0, 1\}$ matrices

To generate random balanced $\{0, 1\}$ matrices we make some changes to the algorithm for generating random balanced $\{0, \pm 1\}$ matrices in section 4.1. When constructing T, instead of choosing the direction of each edge randomly, we now make sure that the directions of the edges always point in the direction of a sink node, which can be reached via the directed edges from all nodes. Before we add each edge s, t that that is not in the tree, we need to ensure that there exists a path in T from s to t. The resulting matrix will only have $\{0, 1\}$ entries.

4.3 Test Results

The tests were performed on a Lenovo Thinkpad P51, with a Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, with 16GB memory and without parallelization. The implementations can be found on my gitlab page [7].

Each algorithm was tested with 50 randomly generated network matrices. These generation algorithms were implemented as part of the CMR [1]. For the $\{0, \pm 1\}$ polynomial algorithm and the exponential algorithm I used $\{0, \pm 1\}$ network matrices, and for the $\{0, 1\}$ polynomial algorithm I used $\{0, 1\}$ network matrices. The running time of the $\{0, \pm 1\}$ polynomial algorithm for $n \geq 16$ was already long, so for those values of n I took the average running time of 10 matrices instead of 50. At $n \geq 19$ the running time became very long even for a single matrix. I decided to not include any results from those bigger matrices, because the sample size would become too small. In Figure 14 we can see the results of these tests.

We can extrapolate the results by approximating the $\{0, \pm 1\}$ and $\{0, 1\}$ polynomial algorithms by fitting polynomials of degree 11 and 9 respectively using the least squares method. Similarly the exponential algorithm can be approximated by an exponential function. Figure 15 contains these approximations.

The algorithms were also tested on another set of various balanced matrices stemming from the integer programming benchmark instances from MIPLIB [6]. The results of these tests are shown in Table 1. The $\{0, \pm 1\}$ matrices were checked by the $\{0, \pm 1\}$ polynomial algorithm and the $\{0, 1\}$ matrix by the $\{0, 1\}$ algorithm.

In Figure 14 we see that for the random network matrices the exponential algorithm is faster for smaller matrices. In Figure 15 we see that for random $\{0, 1\}$ network matrices, the $\{0, 1\}$ polynomial algorithm is slower than the exponential algorithm for matrices of sizes of at most 62. For $\{0, \pm 1\}$ network matrices, the $\{0, \pm 1\}$ polynomial algorithm is

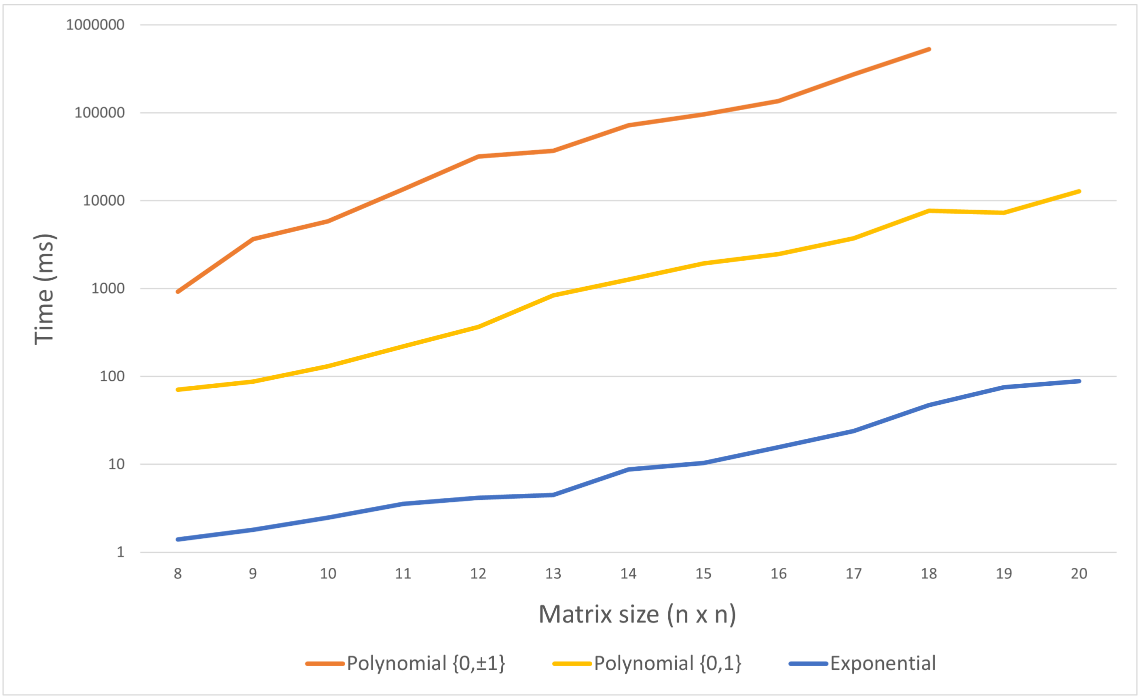


Figure 14: Average results of tests on random network matrices.

Table 1: Test results of several other balanced matrices.

Matrix	Size	Entries	Polynomial time (ms)	Exponential time (ms)
supportcase18.dense	30×30	$\{0, 1\}$	48338	> 2200000
30n20b8.dense	27×22	$\{0, \pm 1\}$	7241	185763
neos-3046615.dense	18×32	$\{0, \pm 1\}$	67030	3307
neos-1445765.dense	10×12	$\{0, \pm 1\}$	1411	24

slower than the exponential algorithm for matrices of sizes of at most 106. Interestingly the point at which the $\{0, \pm 1\}$ polynomial algorithm gets faster is once both algorithms take about 10^{15} ms, which is over 30000 years.

In Table 1 we see that for the *supportcase18.dense* and *30n20b8.dense* matrices, the polynomial algorithm is faster than the exponential algorithm. These matrices are relatively small. This shows that the polynomial algorithm has the potential to be faster for smaller matrices even though the exponential algorithm is generally faster on randomly generated network matrices.

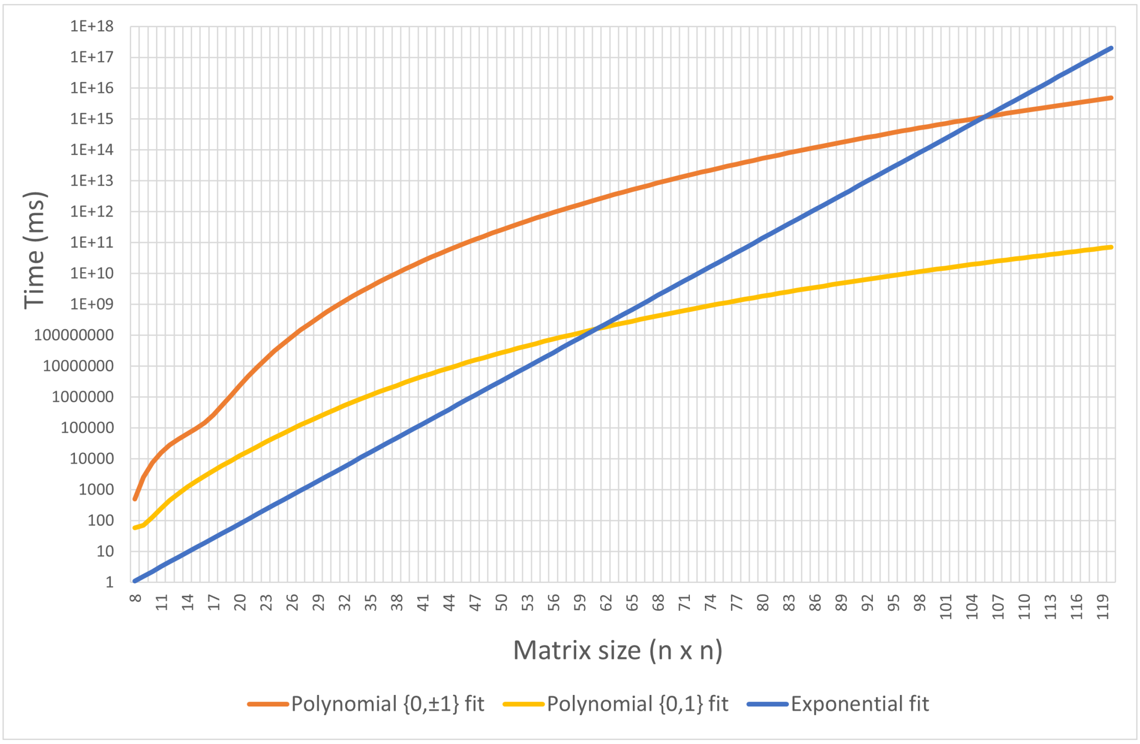


Figure 15: Extrapolated fit of the test results on random network matrices.

5 Conclusions

From the results we see that for the randomly generated network matrices, the exponential algorithm is faster on reasonable time scales. We also saw that there exist matrices for which the polynomial algorithm is faster.

An example of another type of matrix for which the polynomial algorithm is faster is the matrix where the bipartite representation is a large hole. The exponential algorithm has to go over all the combinations of rows and columns while the polynomial algorithm will quickly find that the graph contains no 3-path configurations and detectable 3-wheels. Next it would have a relatively small amount of paths to iterate over in the final step of the algorithm, because a hole does not contain a lot of edges.

It is important to note that all the matrices used in testing were balanced matrices, so all the algorithms would never terminate early since there were no unbalanced holes to find. When one would use the algorithm to check for any matrix if it is balanced or not, the running time could be much lower if the matrix turns out to be unbalanced.

The exponential algorithm does use some pre-processing optimizations that give it an advantage over the polynomial algorithms, which does not make use of any pre-processing. The exponential algorithm ignores all the rows and columns with less than two nonzero entries before iterating over all submatrices, which is very effective for the random network matrices. These matrices often have multiple rows and columns with less than two nonzero entries.

A similar pre-processing technique for the polynomial algorithms would be to repeatedly remove the leaves (vertices of degree 1) of the bipartite representation of the matrix until there are no leaves left. This is an effective pre-processing step since removing leaves are never part of a hole, and it makes each step of the algorithm faster since there are less vertices to iterate over. This would not change time complexity for the polynomial algorithms.

I ran some small tests to compare the running times and the size of the cleaner set of the $\{0, \pm 1\}$ polynomial algorithm on three different matrices, in their original form and with their leaves removed. Table 2 shows how the running time and size of the cleaner set decreases for three different random network matrices by removing the leaves.

Table 2: Comparison of the $\{0, \pm 1\}$ polynomial algorithm with and without the leave removal step.

Matrix	Size	Running time (ms)	Cleaner set size
network-10-10-1.dense	10×10	4134	37401
network-10-10-1.dense (without leaves)	8×8	3375	33670
network-15-15-1.dense	15×15	35736	183315
network-15-15-1.dense (without leaves)	11×13	20608	162165
network-20-20-1.dense	20×20	1194965	5201925
network-20-20-1.dense (without leaves)	14×19	763815	4516515

The polynomial algorithm running time could also be improved by implementing it in a faster programming language than Java, such as C. And there are some changes that could be made to the implementation that could speed it up. Currently we use the JGraphT package to use their graph datastructures. In the current implementation the vertices are stored as a String. This makes it easier to debug the code, but vertex comparisons take longer. It would be faster to change the vertices to integers instead of String objects.

There is potential for a small optimization in the 3-path configuration algorithm 2. In step 3 we check if a_i is adjacent to b_i and if this is the case we can ignore what we calculated in step 2, because all the $P_i(m)$ for $m \neq a_i$ would be undefined. So it would make more sense to check this before doing step to, so we can potentially skip part of the algorithm in some cases.

Some testing can also be done in the future on transposing the matrix in when using the exponential algorithm. Perhaps there is some way to estimate if the algorithm would run faster if the matrix was transposed.

Future research can be done by exploring other pre-processing techniques. One idea is to compress long sections of connected vertices of degree 2. Making these chains of vertices smaller could potentially speed up all the BFS algorithms that are used for finding the shortest paths in the polynomial algorithms.

References

[1] CMR: Combinatorial Matrix Recognition Library · https://discopt.github.io/cmr/.

[2] Claude Berge. Balanced matrices. *Mathematical Programming*, 2(1):19–31, February 1972.

[3] Michele Conforti and Gérard Cornuéjols. Balanced 0, ±1-matrices, bicoloring and total dual integrality. *Mathematical Programming*, 71(3):249–258, December 1995.

[4] Michele Conforti, Gérard Cornuéjols, Ajai Kapoor, and Kristina Vušković. Balanced 0, ± 1 Matrices II. Recognition Algorithm. *Journal of Combinatorial Theory, Series B*, 81(2):275–306, March 2001.

[5] Michele Conforti, Gérard Cornuéjols, and Kristina Vušković. Balanced matrices. *Discrete Mathematics*, 306(19):2411–2437, October 2006.

[6] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3):443–490, September 2021.

[7] Henk Kraaij. s2006022 / BalancedMatrixRecognition · GitLab · https://gitlab.utwente.nl/s2006022/balancedmatrixrecognition, May 2023.

[8] Alexander Schrijver. Theory of Linear and Integer Programming. *John Wiley & Sons, Inc.*, 1986.

[9] Giacomo Zambelli. A polynomial recognition algorithm for balanced matrices. *Journal of Combinatorial Theory, Series B*, 95(1):49–67, 2005.