Bachelor Thesis

Faculty of Mathematics and Computer Sciences

Chair of Discrete Mathematics and Optimization

Computing the Varchenko Matrix and its Determinant for Partial Cubes

Birol Yazici (MN:9087931)

Supervisors: Prof. Dr. Winfried Hochstättler &

Sophia Keip, M.Sc. Mathematics

28. Dezember 2024

Contents

List of abbreviations

PC	Partial Cube
OM	Oriented Matroid
COM	Complex of Oriented Matroid
VD	Varchenko Determinant

1. Abstract

The following thesis is motivated by the paper named "The Signed Varchenko Determinant for Complexes of Oriented Matroids" by Hochstättler, Keip and Knauer who established that the determinant of the varchenko matrix for so called complexes of oriented matroids (COMs) has a nice factorization[1]. COMs represent a subset of partial cubes. Therefore the question is posed whether there are other classes of partial cubes not being COMs with a nice factorization. This work contributes to answering this question by programmatically calculating the varchenko determinant for a special class of partial cubes which can be considered as a bridge between partial cubes and complexes of oriented matroids[2].

2. Course of investigation

After putting this research question into a historic context in section 3 the so called complexes of oriented matroids are defined in chapter 4 and a graphical example is given to explain them. Then in part 5 all the theoretical requirements which are considered non-standard and essential in understanding this thesis are introduced. With this theoretical basis the research question which has been touched in the abstract can be specified exactly. In section 6 the program which solves this problem is introduced. This happens on one hand structurally in order to explain the program on a high level and clarify how it works from an input- as well as output-perspective. Then where it seems necessary complex details will be pointed out to ensure transparency. Furthermore performance considerations in setting up and running the calculation will be touched. Finally in chapter 8 the results will be represented and the posed question will be answered.

[1] See [5].

[2] The determinant of the so called varchenko matrix is also referred to as varchenko determinant.

3. Historic context

The publications of Hassler Whitney and Takeo Nakasawa[3] in the 1930s are considered as the official starting points for matroids. With the oriented matroid (OM) which is a special type of matroid the historic categorization is more difficult. However the main originators are considered to be Robert Bland, Jim Lawrence, Jon Folkman and Michel Las Vergnas. The time frame regarding the development of OMs falls in the 60s and 70s. In 1971 Graham and Pollak brought with their study of interconnection networks so called partial cubes to the academic scene[4]. In 2015 so called complexes of oriented matroids (COMs) were introduced by Bandelt, Chepoi and Knauer[5]. Knauer and Marc published then in 2019 a paper which showed a link between COMs and PCs[6]. This link is a special kind of PC a so called forbidden minor which allows to distinguish between COMs and PCs.

To complete the picture the so called varchenko matrix which has been developed in the year 1993 has to be brought up[7]. There have been a number of publications which analyze the determinant of this concept for a variety of objects. In particular Hochstättler and Welker as well as Hochstättler, Keip and Knauer specified in 2018 and 2023 this particular determinant for OMs and COMs[8]. This thesis now picks up on the special kind of PCs proposed by Knauer and Marc by calculating programmatically the determinant of the varchenko matrices for these PCs.

The depiction of the timeline shows that this whole area of matroids and related concepts is a relatively young branch of mathematics. However it proves to be an incredibly rich field which can be seen in the many alternative ways to define

[3] The contributions of Nakasawa only found more than 50 years after his death scientific appreciation, see [1]

[4] Although partial cubes are usually neither part of undergraduate- nor graduate-level courses they constitute a rich field of research with many applications, see [3] for further information.

[5] See [11]

[6] See [4]

[7] See [12]

[8] See [5] and [6]

these concepts and subsequently the many areas it is applied like combinatorics, topology, algebraic geometry, operations research and computer science to name a few.

4. COMs

In this section COMs are introduced. In order to establish the framework for COMs we agree that E represents a finite non-empty (ground) set and $\mathcal{L} \subseteq \{-, 0, +\}^E$ is defined as a set of sign vectors on E. For the vectors X, Y $\in \mathcal{L}$ the operation of composition and separation is defined as follows

Definition 1 (Composition)**.** *For X, Y* $\in \mathcal{L}$ *the operation X* $\circ$ *Y is defined by*

$$(X \circ Y)_e = Y_e \text{ if } X_e = 0 \text{ and } (X \circ Y)_e = X_e \text{ if } X_e \neq 0 \; \forall \; e \in E.$$

Definition 2 (Separation)**.** *For X, Y* $\in \mathcal{L}$ *the separator S(X,Y) is defined by*

$$S(X,Y) = \{e \in E : X_e = -Y_e \neq 0 \}.$$

With these definitions the following four axioms can be depicted.

Axiom 1 (Composition (C))**.**

$$(X \circ Y)_e \in \mathcal{L} \; \forall \; X, Y \in \mathcal{L}.$$

Axiom 2 (Symmetry (S))**.**

$$-\mathcal{L} = \{-X : X \in \mathcal{L}\} = \mathcal{L}.$$

Axiom 2 means that $\mathcal{L}$ is closed under sign reversal.

Axiom 3 (Face symmetry (FS))**.**

$$(X \circ -Y)_e \in \mathcal{L} \text{ for all } X, Y \in \mathcal{L}.$$

Axiom 4 (Strong elimination (SE))**.**

For each pair $X, Y \in \mathcal{L}$ *and for each* $e \in S(X,Y)$ *there is a* $Z \in \mathcal{L}$ *with* $Z_e = 0$ *and* $Z_f = (X \circ Y)_f$ *for all* $f \in E \backslash S(X,Y)$.

With the help of these axioms oriented matroids and complexes of oriented matroids can be defined[9].

Definition 3 (OMs and COMs)**.** *Assuming the groundset E and the system of sign vectors* $\mathcal{L}$ *then* $\mathcal{M}$ *= (E,* $\mathcal{L}$*) represents an OM if* $\mathcal{L}$ *fulfills the axioms (C),(S) and (SE). It represents a COM if* $\mathcal{L}$ *fulfills the axioms (FS) and (SE).*

An alternative way to define OMs is by demanding that $\mathcal{L}$ satisfies (FS),(SE) as well as $0 \in \mathcal{L}$ which shows that COMs are a generalization of OMs because (FS) implies (C) and 0 in combination with (FS) results in (S).
In order to better grasp a COM we will use a special case which can be visualized and is known as realizable COM. It assumes that E is an arrangement of hyperplanes of $\mathbb{R}^d$ and C is an open convex set which intersects all hyperplanes to avoid

[9] In chapter 7 these axioms are used to establish the existence of certain vectors in a COM

redundancies. By restricting the arrangement to the convex set a realizable COM results consisting of sign vectors.
Furthermore we restrict ourselves to so called simplicity which requires the axioms (N1) and (N2) to hold.

$(N1)$ $\forall\, e \in$ E: $\{\mathrm{X}_e|\ \mathrm{X} \in \mathcal{L}\} = \{+,-,0\}$;
$(N2)$ $\forall\, e \neq f \in$ E: $\{\mathrm{X}_e X_f|\ \mathrm{X} \in \mathcal{L}\} = \{+,-,0\}$.

The COM $\mathcal{M} = (\mathrm{E}, \mathcal{L})$ which satisfies (N1) and (N2) is called simple. Under this setting the sign vectors in $\mathcal{L}$ with full support represent the topes. A simple COM is determined by its tope graph[10]. Tope graphs are better known as region graphs but within the context of OMs the term tope has prevailed. In a tope graph the vertices represent regions or sign vectors which have no zero entries. A certain edge between two vertices exists if the tope only differ in exactly one sign. The sign vector in the tope graph connecting these two vertices has one and only one zero entry. This leads to the following definition.

Definition 4 (Tope graph)**.** *Given an arrangement $\mathcal{A}$ and its regions R($\mathcal{A}$) the tope graph T($\mathcal{A}$) has R($\mathcal{A}$) as the set of vertices. The set of edges constitute the pairs of adjacent regions in R($\mathcal{A}$).*

We start by assuming an arrangement of hyperplanes which can be seen in figure 1. An orientation is arbitrarily given meaning that plus and minus signs are indicating whether a point in the plane is below or above the hyperplane.

[10]See proposition 3 in [11]

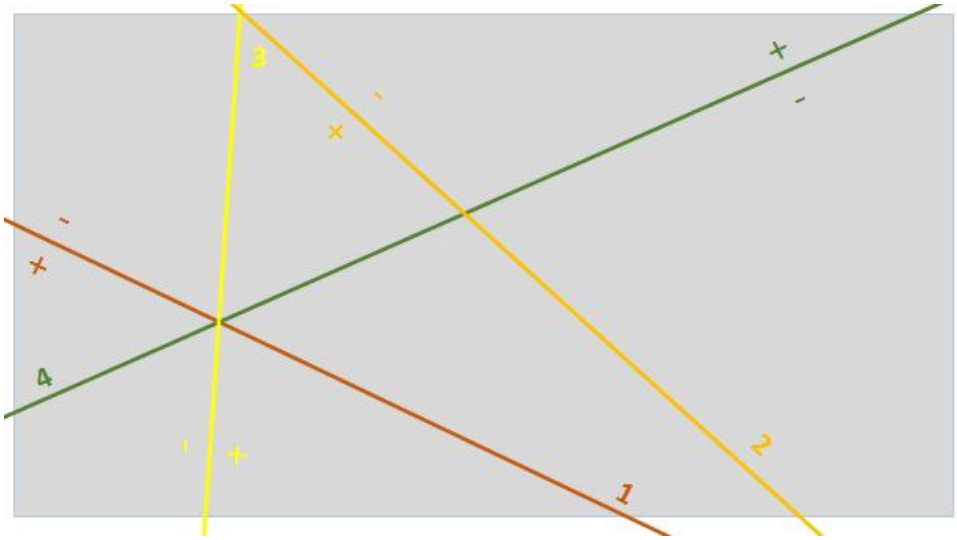

Figure 1: A hyperplane arrangement

Thus regions are formed which are characterized by a four dimensional sign vector. Each entry provides information whether a region is on the positive or negative side of a certain hyperplane.

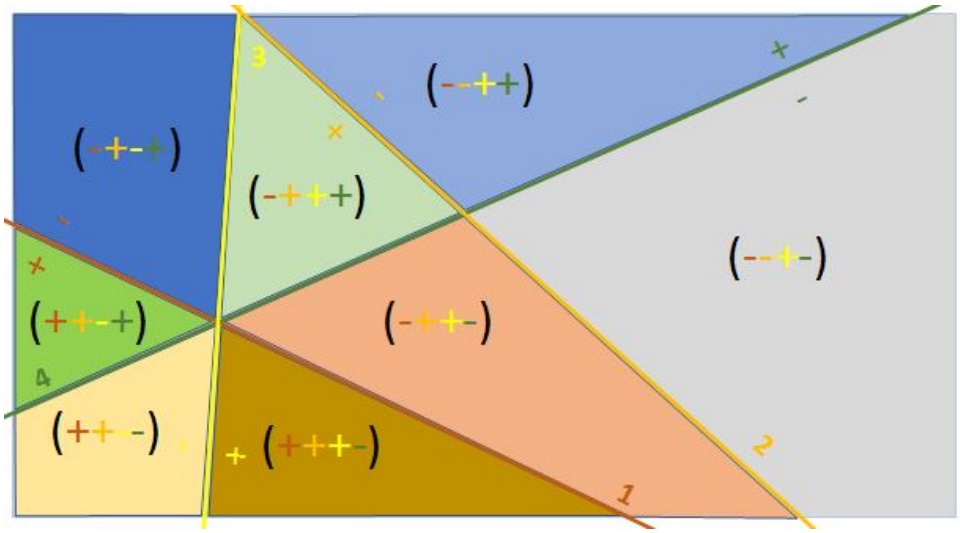

Figure 2: A COM realized by a hyperplane arrangement

The points on a hyperplane itself are represented by zero entries on the correspond-

ing sign vector. Hence intersection points of multiple hyperplanes result in sign vectors with multiple zero entries. If we interpret each region as a vertex and we connect only those with vertices respectively regions which are adjacent the tope graph in figure 3 results. The figure also includes the sign vectors representing the edges.

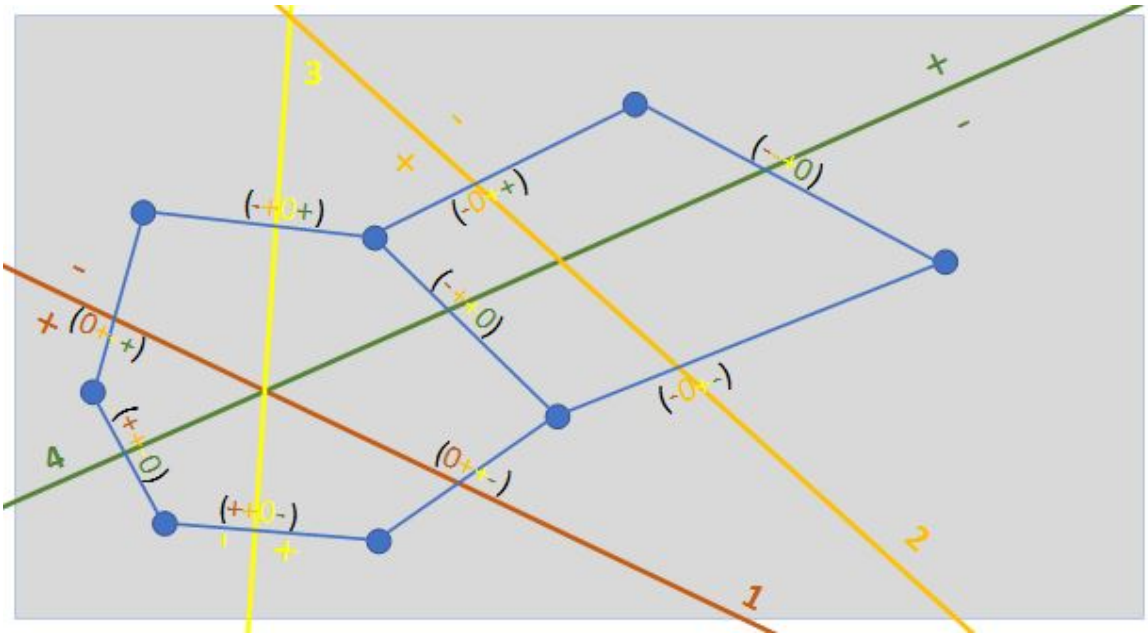

Figure 3: The tope graph derived from figure 2

Neighboring topes only differ in one entry of their respective sign vectors

5. Prerequisites and motivation

5.1. Cubes and forbidden minors

Due to its importance and recency COMs were put forward in a separate chapter. To make sure that the presented ideas and arguments can be understood going forward the following definitions and theorems are presented.

Definition 5 (Hypercube)**.** *A hypercube Q_n of dimension n is a graph G whose set of vertices consists of 2^n members. Each vertex can be represented by a n-bit binary string. Two arbitrary vertices are adjacent if their binary representation differs in exactly one bit.*

Figure 4 displays the hypercube Q_3 in a binary, graph-like and bipartite representation. Hypercubes can be defined in a large variety of ways[11]. Although all characterizations are different they are nevertheless equivalent. For this paper including the calculations and results presented later we apply the graph perspective.

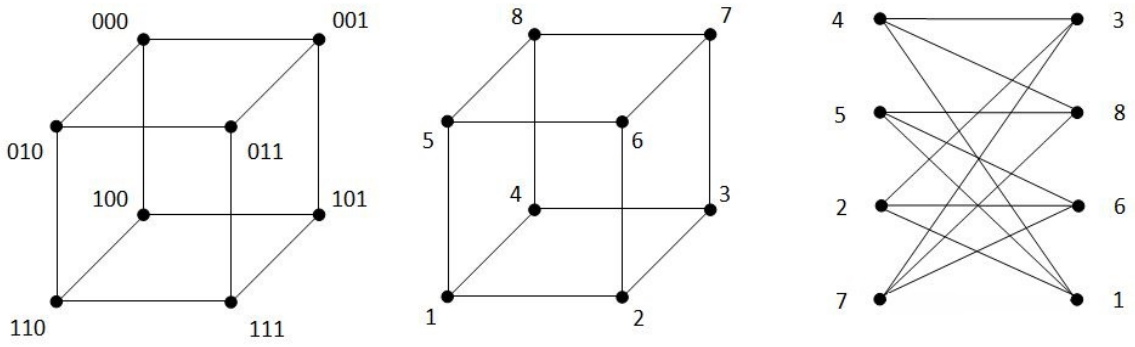

Figure 4: Binary, graph and bipartite hypercube representation

In a next step we define partial cubes.

Definition 6 (Partial Cube)**.** *A graph G is a partial cube if it can be isometrically embedded in a hypercube Q_n which means that the distance function d for G as well as Q_n upholds the condition $d_G(u, v) = d_{Q_n}(u,v)$ for any vertex pair (u,v) in G.*

Therefore a partial cube can be identified with a subgraph of a hypercube so that the distance between any pair of vertices in the PC matches the distance of that pair in the hypercube. A PC has to be a connected graph.

[11] See [2]

A way to analyze and characterize partial cubes is via a binary relation on the set of edges. Djokovic[12] and Winkler[13] proposed separately similar relations to do this. For our purposes it will be sufficient to present Winkler's approach. A graph G = (V,E) with vertices V and edges E is assumed. An edge connecting the vertices u and w is represented by the expression $\{v, w\}$. The graph theoretical distance between two vertices v, w $\in$ V is defined by $\delta(v, w)$ as the length or the number of edges of the shortest path from v to w. If such a path is not existent then $\delta(v, w) = \infty$ holds. The distance from one vertex v to itself is $\delta(v, v) = 0$.

Definition 7 (Winkler's relation)**.** *For the edges {u,v} and {x,y} the relation θ is defined as*

$$\{x, y\}\theta\{u, v\} \Leftrightarrow \delta(x, u)+\delta(y, v) \neq \delta(x, v)+\delta(y, u)$$

For partial cubes this relation fulfills the three requirements of reflexivity, transitivity and symmetry and thus qualifies as a equivalence relation. Therefore the edges of a partial cube can be decomposed into equivalence classes. The expression [e] denotes the set of edges belonging to the equivalence class to which the edge e of the Graph G is part of. The graph without the edges of a particular equivalence class [e] is describe by G\[e]. The set containing all the equivalence classes is represented by E. Figure 5 which shows the tope graph we have presented earlier with color coded edges according to its equivalence classes.

[12]See [14]
[13]See [13]

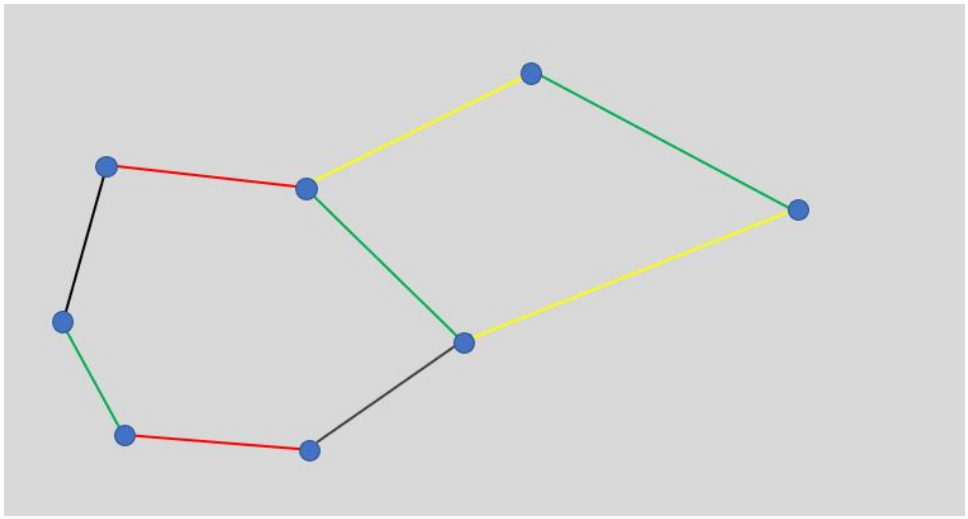

Figure 5: Tope graph with equivalent class colouring

Removing the edges of a particular class respectively color leads to two conneceted graphs which are separated. Figure 6 shows two connected but separated graphs after all the edges of the green equivalence class haven been omitted.

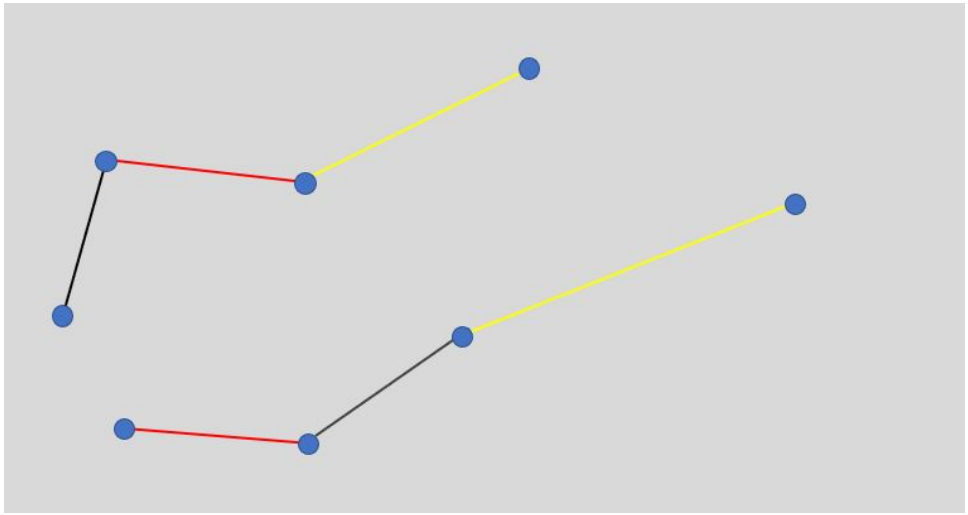

Figure 6: Deletion of the green equivalence class edges

Knauer and Tilen established the following relationship between COMs and PCs.

Theorem 1 (PC COM equivalence[14]). *For a graph G both conditions are equivalent:*

(i) G is the tope graph of a COM

(ii) G is a PC with no PC minor from the set $\mathcal{Q}^-$

A graph H is classified as a PC minor of the graph G if H results from a sequence of contractions and restrictions of G. In a contraction two vertices of a graph which are joined by one edge are merged into one new vertex and the respective edge is removed. A restriction of a graph G means that only a subgraph is taken. In the context of PC minors both operations are applied with regards to equivalence classes. For the contraction this results in the fact that a whole equivalence class of edges is contracted. For the restriction it means that after a whole equivalence class is deleted one of the two remaining subgraphs is taken. The set $\mathcal{Q}-$ is called the forbidden pc-minors which can be constructed by the following 2-step process:[15]

1) For any v $\in Q_n$ construct $Q_n^- := Q_n \setminus$ -v for n ≥ 4 ,-v being the antipode.
2) Remove from Q_n^- any subset of N(v) $\cup$ v.

N(v) are representing all the neighbors of v. We define Q_n^{-*} and $Q_n^{--}(m)$ as PCs from which only one respectively m neighbors including v have been deleted. The forbidden minors are described by

$$\mathcal{Q}^- = \{Q_n^{-*}, Q_n^{--}(m) \mid 4 \leq \text{n}; 1 \leq \text{m} \leq \text{n} \}.$$

To avoid any misunderstanding we point out that from Q_n^{-*} as well as $Q_n^{--}(m)$ the antipode -v has been removed. The forbidden minors are defined for n ≥ 4. Nevertheless we will explain this process for n = 3 because the construction can be

[14] See theorem 1.1 in [4], for the purpose of brevity and relevance a shortened version of this theorem is displayed

[15] For the details and logic behind this process we refer to [4]

clarified graphically and the number of vertices remains manageable. As an example the hypercube displayed in figure 4 with the set of vertices V = {1,2,3,4,5,6,7,8} is used. We pick v = 1 as a starting point which has as its antipode -v = 7. After removing the antipode we end up with Q_3^- shown in figure 7.

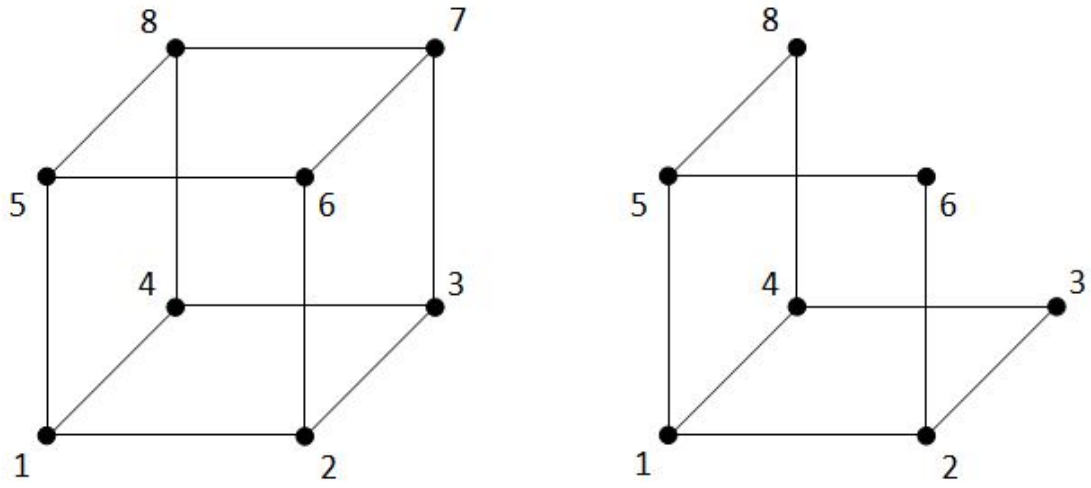

Figure 7: Q_3 and Q_3^-

The vertices which need to be removed next are N(1)={2,4,5} ∪ {1}. Figure 8 displays $Q_3^{--}(1)$ and $Q_3^{--}(2)$[16].

[16]Please be aware the case for n=3 is not defined, it is just used to exemplify the process.

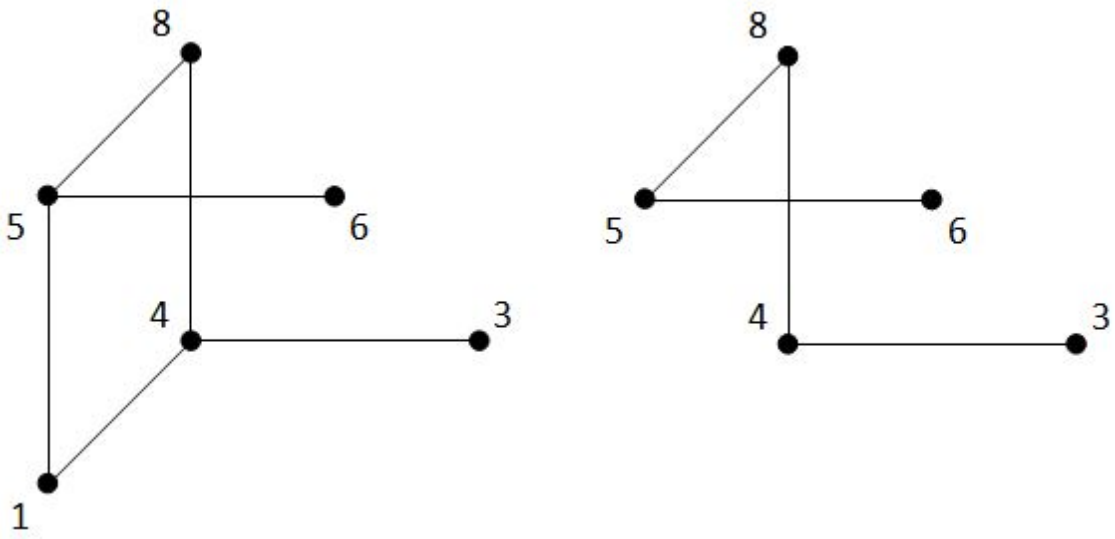

Figure 8: $Q_3^{--}(1)$ and $Q_3^{--}(2)$

5.2. Varchenko matrix

The last central concept which needs to be specified is the varchenko matrix which was initially defined for a hyperplane arrangement[17]. Although for the scope of this thesis the concept has to be applied to partial cubes we explain it in its original form by giving an example first. Figure 9 displays an arrangement of 3 hyperplanes H_i for i = 1...3 which creates the regions R_j for j = 1...7.

[17]See [12]

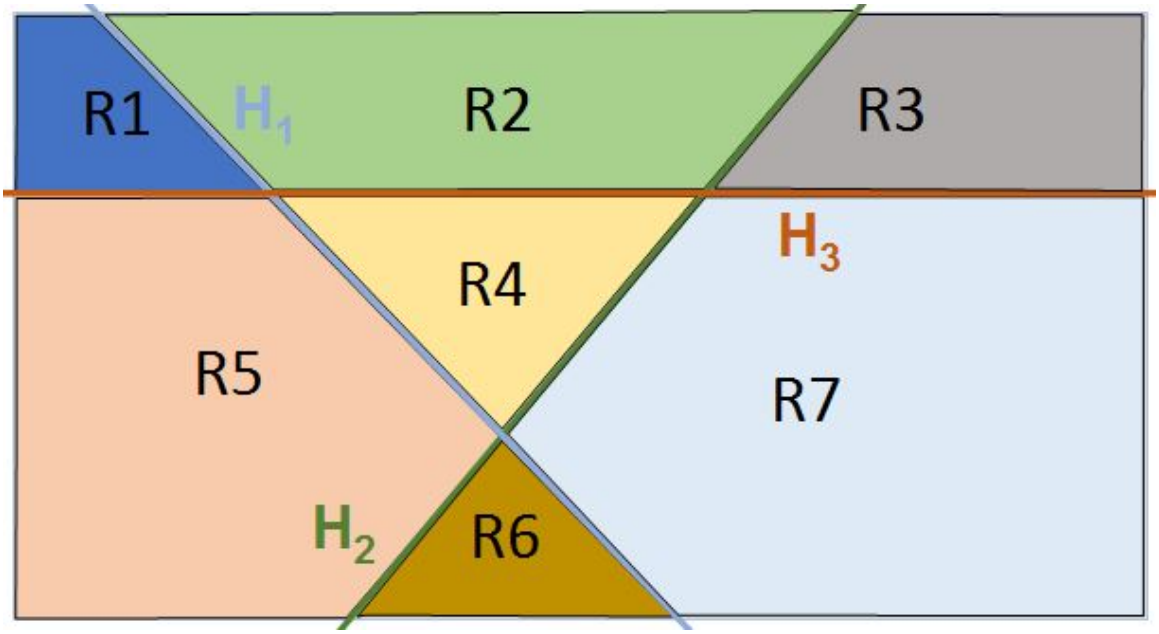

Figure 9: Arrangement of three hyperplanes H_1, H_2 and H_3

The given arrangement leads to a 7×7 Varchenko Matrix $\mathcal{V}$ displayed below[18].

$$\mathcal{V} = \begin{pmatrix} 1 & a_1 & a_1a_2 & a_1a_3 & a_3 & a_2a_3 & a_1a_2a_3 \\ a_1 & 1 & a_2 & a_3 & a_1a_3 & a_1a_2a_3 & a_2a_3 \\ a_1a_2 & a_2 & 1 & a_2a_3 & a_1a_2a_3 & a_1a_3 & a_3 \\ a_1a_3 & a_3 & a_2a_3 & 1 & a_1 & a_1a_2 & a_2 \\ a_3 & a_1a_3 & a_1a_2a_3 & a_1 & 1 & a_2 & a_1a_2 \\ a_2a_3 & a_1a_2a_3 & a_1a_3 & a_1a_2 & a_2 & 1 & a_1 \\ a_1a_2a_3 & a_2a_3 & a_3 & a_2 & a_1a_2 & a_1 & 1 \end{pmatrix}$$

Each cell (j,k) of this matrix provides information on which of the three hyper-

[18]The example is taken from [7]

planes separates region j from region k. With each hyperplane H_i the variable a_i is associated. For example the cell (1,3) with the value a_1a_2 means that the regions R_1 and R_3 are separated by the hyperplanes H_1 as well as H_2. H_3 is not a separating hyperplane meaning those regions are on the same side of that hyperplane. In this case the separator S(R,R') from 2 matches the requirements for establishing the varchenko matrix meaning it represents now the set of hyerplanes separating region R from region R'. For a given hyperplane arrangement $\mathcal{A}$ a variable a_H is assigned for each H $\in \mathcal{A}$. The expression $\mathcal{V} = \mathcal{V}(\mathcal{A})$ where the rows and columns correspond to the regions R($\mathcal{A}$) defines the varchenko matrix by

$$\mathcal{V}_{RR'} = \prod_{H \in S(R,R')} \mathrm{a}_H.$$

With regard to PCs the varchenko matrix provides information which vertices are separated or disconnected from each other after removing a whole equivalence class of edges. The separator S requires generalization and expects as arguments no longer regions but vertices. It also returns no longer hyerplanes but a set of equivalences classes. An example detailing the calculation and providing intuition within the partial cube context is given in chapter 6.3.2. We proceed by giving the following definition

Definition 8 (Varchenko matrix for a partial cube)**.** *For a given partial cube graph G a variable $a_{[e]}$ is assigned for each equivalene class $[e] \in E$. The expression $\mathcal{V} = \mathcal{V}(\mathcal{G})$ where the rows and columns correspond to the vertices of the graph G is defined as the vachenko matrix by*

$$\mathcal{V}_{ij} = \prod_{[e] \in S(i,j)} a_{[e]}.$$

After having defined the relevant concepts as a basis we can move on to the theorems from which the hypothesis examined in this thesis are derived. In their 2022 paper titled "The signed varchenko determinant for complexes of oriented matroids" Hochstättler, Keip and Knauer established the following result [19]

Theorem 2 (Determinant of the varchenko matrix for COMS)**.** *Let V be the varchenko matrix of the COM $(\mathcal{E}, \mathcal{L})$ with ground set $\mathcal{E}$. Then the equation*

$$det(\mathcal{V}) = \prod_{Y \in \mathcal{L}} (1 - c(Y)^2)^{b_Y}$$

holds where c(Y) := $\prod_{e \in z(Y)} x_e$ and b_Y are nonnegative integers.

The expression c(Y) is the product of the zeroset variables of Y. For further details we refer to the mentioned paper[20]. The important finding is that for the COMs a nice factorization exists.

5.3. Motivation

The goal of this thesis follows up on the question whether there is a forbidden minor which was developed by Tilen and Knauer[21] whose varchenko matrix factorizes as nice as they do for COMs. Based on the current research we can make the statement that if there is a COM it must have a nice factorization. So the motivation or hope is to find at least 1 non-COM or partial cube with a nice factorization in order to falsify the reverse implication of the previous theorem. The exact scope of this work is to determine the varchenko determinant for all

[19] See theorem 2.9 in [1]
[20] See [4]
[21] See [5]

forbiden minors in the dimensions 4 and 5. This task is achieved programmatically by developing code which takes the respective partial cubes derived from Q_4 and Q_5 as input in the form of graphs. It means the program expects that for every forbidden minor a graph G = (V,E) is specified by the vertices V, the edges E and an adjacency list. The expected outputs are the factorized polynomials of the varchenko determinant for Q_4^{-*}, $Q_4^{--}(1)$, $Q_4^{--}(2)$, $Q_4^{--}(3)$, $Q_4^{--}(4)$ and Q_5^{-*}, $Q_5^{--}(1)$, $Q_5^{--}(2)$, $Q_5^{--}(3)$, $Q_5^{--}(4)$, $Q_5^{--}(5)$.

6. Overview and analysis of code

6.1. Implementation

The task of creating the varchenko matrices was realized in the programming language python on the basis of the streamlined editor visual studio code from Microsoft. The main programmes which achieve this outcome are stored in the two files *Calc_VM.py* and *PartialCube.py*. Each file contains a number of functions which call or use functions which themselves are stored in other *.py*-files. Only the content of *Calc_VM.py* and *PartialCube.py* is available in the appendix because the other referenced *.py*-files are publicly available[22]. With the goal of minimizing the existence of bugs a number of tests have been performed. This happened with graphs where the equivalence class structure as well as the varchenko matrix can be deducted by visualizing the graph.

The calculation of the determinants of the respective matrices is realized via maple. Originally the varchenko matrix creation and determinant calculation were executed via python. However the determinant calculation was so time consuming in python in particular for the minors of dimension n=5 that a switch to maple was necessary. The maple code is also provided in the appendix.

[22]The files are available on [9]

6.2. Input-output structure

Running the programme *Calc_VM.py* creates the desired varchenko matrix. It expects as input an adjacency list for a graph G in curly brackets. The following example shows the adjacency list for the graph defined in figure 10:

G ={0:[1,7],1:[2,0],2:[1,3],3:[2,4],4:[3,5,7],5:[4,6],6:[5,7],7:[0,4,6]}.

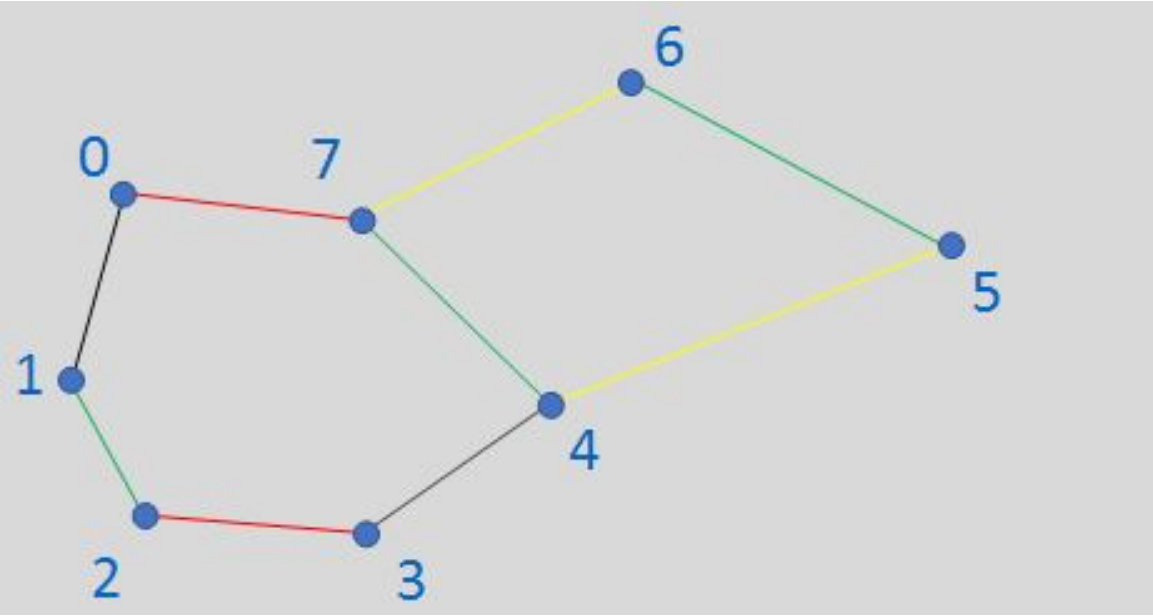

Figure 10: A graph with eight vertices and four equivialence classes

The adjacency lists have to be generated manually. An automated or programmatic creation process was not pursued. Since the forbidden minor structure is limited to n <= 5 such an undertaking cannot be justified. However the manual process is error-prone. In order to minimize mistakes in creating the graphs for the forbidden minors first a binary vertex structure was established. Then cross checks were applied on the neighbours which had to meet the conditions regarding the sum of digits and the identity of n-1 digits for each neighbour. After those

checks have been passed the vertices were renamed with numbers starting from zero and then the adjacency list is created.

Once $Calc\_VM.py$ is run it firstly checks if the graph provided as input is a partial cube. If this is the case the program produces as output a comma separated file named VarchenkoMatrix.txt. The creation of the determinant can be activated by removing the uncomment signs which were put in place due to the mentioned efficiency considerations. The variables linked to the varchenko matrix calculation are represented by the symbols x_i and separated by the star sign "*". The cell (i,j) of the respective varchenko matrix can be found in the i-th line of the txt-file after the (j-1)-th comma. The creation of the varchenko matrix is fully automated. This means that only the graph needs to be provided as input and the programme by itself defines the correct size of the varchenko matrix and the number of variables. In a next step the determinants are calculated by running the two maple programmes named $Calc\_Det\_PC\_4.mv$ or $Calc\_Det\_PC\_5.mv$ for the dimensions n = 4 and n = 5. As inputs the txt-files are expected to be named $Qn\_star.txt$ or $Q\_n\_m.txt$ in accordance with the naming convention of the forbidden minors. With n standing for the dimension and m for the number of neighbors being removed. Hence the txt-files containing the varchenko matrices generated in the previous step need to be manually renamed. The factorized polynomials calculated in maple are returned right away.

6.3. Programm details

After explaining on a high level how the programme works the relevant details will be lined out with the help of an example. In a first step the python code will be explained by splitting it into two blocks with regards to content. The second step deals with explaining the maple programmes respectively the determinant calculation.

6.3.1. Block1: Establishing the equivalence classes in python

The code for determining the equivalence classes uses functionality from a public python library which has been provided by David Eppstein. The library resulted from a paper which introduced an algorithm that recognizes partial cubes in quadratic time[23]. The code of Eppstein had to be modified by extending the function

"def PartialCubeEdgeLabeling(G)" into "def PartialCubeEdgeECs(G)".

For any given graph G being a partial cube it structures and creates the equivalence classes and its members in a python dictionary. The function itself is also wrapped or used in a variety of functions as a building block. Just returning the equivalence classes is done with the function "def EquiClassPCReps(G)". The function "def CreateEcVarDict(ecs)" creates given a set of equivalence classes ecs a dictionary in which each equivalence class is assigned a variable x_i. The three functions outlined are the most important ones in understanding the process of creating the varchenko matrix. Therefore the use of these functions will be shown by an example. We are assuming the graph G displayed in figure 11 is assigned the following adjacency list
G ={0:[1,7],1:[2,0],2:[1,3],3:[2,4],4:[3,5,7],5:[4,6],6:[5,7],7:[0,4,6]}.

[23]See [9] and [10]

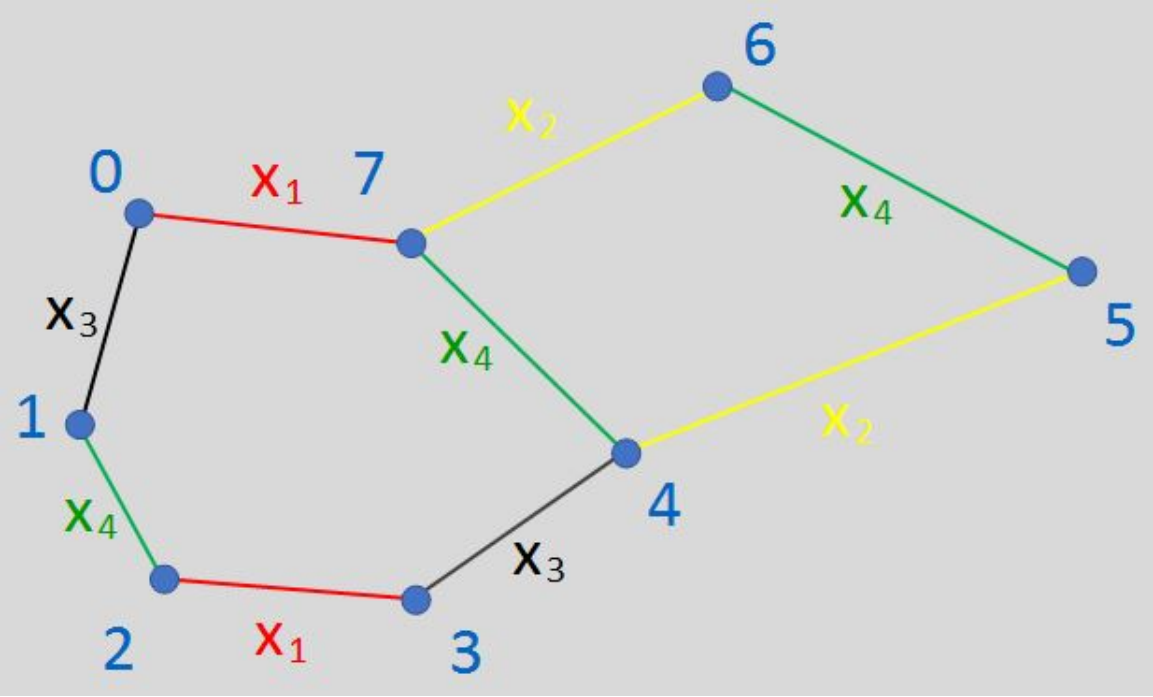

Figure 11: A graph with eight vertices and four equivialence classes

Then the code for calling those three functions looks like [24]

ecs_mem_dict = PartialCubeEdgeECs(G)

ecs_mem_dict =

{**(3, 4)**: {(0, 1), (3, 4)}, **(2, 3)**: {(2, 3), (0, 7)}, **(4, 7)**: {(1, 2), (4, 7), (5, 6)}, **(6, 7)**: {(6, 7), (4, 5)}}

ecs = EquiClassPCReps(G)

ecs = {(2, 3), (6, 7), (3, 4), (4, 7)}

ecDict = CreateEcVarDict(ecs)

ecDict = {(2, 3): x1, (6, 7): x2, (3, 4): x3, (4, 7): x4} .

[24] For a better overview the result of each function call is presented right away also the equivalence class representatives in the dictionary *esc_mem_dict* have been marked bold.

6.3.2. Block 2: Creating the varchenko matrix in python

The varchenko matrix for a graph G is built with the help of the function "def buildVarMat(G)" which uses the three previously stated functions . The process to build the matrix is based on a double loop which iterates trough each cell below the diagonal of the corresponding varchenko matrix. For each vertex pair or respectively cell (i,j) all the edges of each equivalence class are deleted from the graph G. Then the programme analyzes whether vertex i is still connected to vertex j in the new disconnected graph. For all the separating equivalence classes the respective variable is incorporated into the product that represents the value in cell (i,j)[25]. Figure 12 graphically explains the process for the cell (0,1).

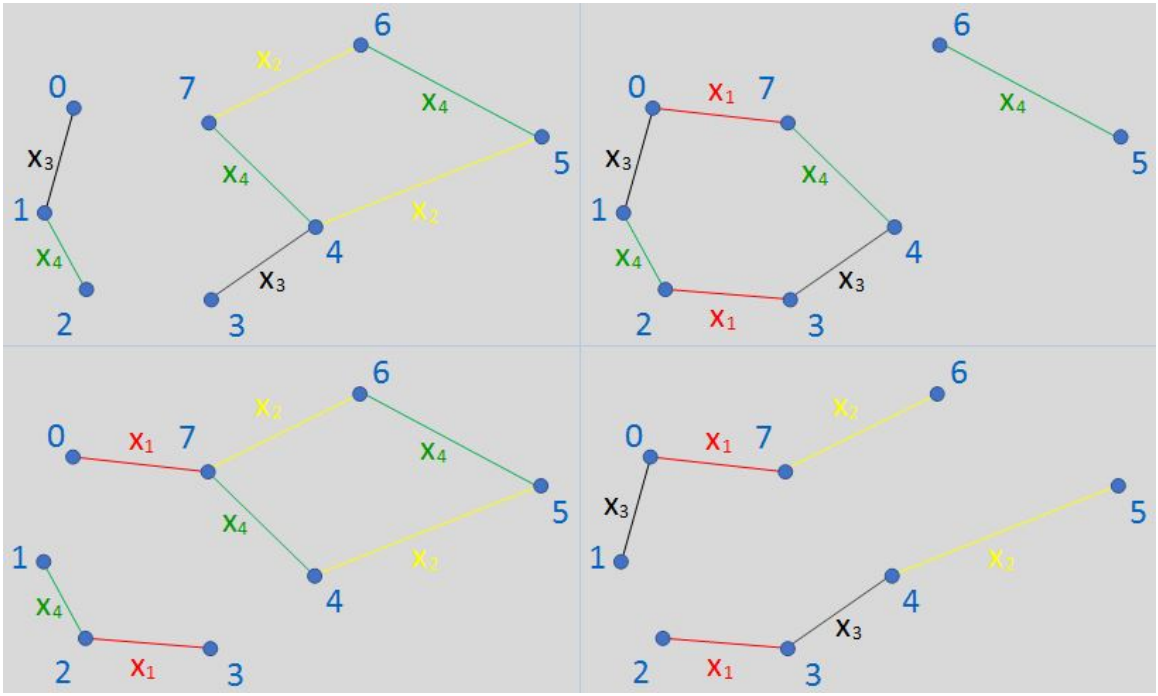

Figure 12: Removing all the edges for each of the four equivalence classes

[25]The rows and columns correspond to the vertices whose naming starts from 0 meaning that cell (0,0) represents the cell in the first row and first column of the matrix.

It can be seen that all the edges for each one of the four equivalence class have been removed. Only in the case when the black or the x_3 equivalence class has been removed there exists no path that connects the vertices 0 and 1[26]. Thus x_3 is the only separating equivalence class and the cell (0,1) in the varchenko matrix displayed below is assigned the value of x_3.

$$\mathcal{V} = \begin{pmatrix} 1 & x_3 & x_3x_4 & x_1x_3x_4 & x_1x_4 & x_1x_2x_4 & x_1x_2 & x_1 \\ x_3 & 1 & x_4 & x_1x_4 & x_1x_3x_4 & x_1x_2x_3x_4 & x_1x_2x_3 & x_1x_3 \\ x_3x_4 & x_4 & 1 & x_1 & x_1x_3 & x_1x_2x_3 & x_1x_2x_3x_4 & x_1x_3x_4 \\ x_1x_3x_4 & x_1x_4 & x_1 & 1 & x_3 & x_2x_3 & x_2x_3x_4 & x_3x_4 \\ x_1x_4 & x_1x_3x_4 & x_1x_3 & x_3 & 1 & x_2 & x_2x_4 & x_4 \\ x_1x_2x_4 & x_1x_2x_3x_4 & x_1x_2x_3 & x_2x_3 & x_2 & 1 & x_4 & x_2x_4 \\ x_1x_2 & x_1x_2x_3 & x_1x_2x_3x_4 & x_2x_3x_4 & x_2x_4 & x_4 & 1 & x_2 \\ x_1 & x_1x_3 & x_1x_3x_4 & x_3x_4 & x_4 & x_2x_4 & x_2 & 1 \end{pmatrix}$$

6.4. Determinant calculation

The calculation for all the forbidden minors is standardized. The code displayed below shows the calculation for $Q_5^{--}(1)$.

Q_5_1 := ImportMatrix("$path\_of\_file/VM\_5\_1.txt$", delimiter = ",");

Q_5_1 := map(t -> if(type(t, string), parse(t), t), Q_5_1);

Q_5_1 := GaussianElimination(Q_5_1);

factor(Determinant(Q_5_1));

In the first two lines the renamed file whose creation was described in the previous

[26]See lower left corner of figure 12

section is read into maple and adjusted so it can be processed correctly [27]. Then the Gaussian Elimination algorithm is applied which transforms the matrix into a triagonal structure. This step is necessary because otherwise the calculation of the determinant would require too much time. Finally the polynomial is factorized and returned.

6.5. Scalability

Setting up the adjacency lists manually and ensuring their correctness is very time-consuming. For each minor of dimension 5 the whole calculation took around 30 minutes which has to be considered as a lower bound since the number of vertices grows exponentially with the dimension size. Therefore automating the creation of the input files for the forbidden minors will result in significant time gains. Furthermore running all the calculations in python in contrast with the current python and maple solution would approximately save 3 minutes per minor due to renaming and loading processes. However an efficient python-only solution is currently not achievable. As the libary in python responsible for the symbolic calculations does not possess a Gaussian algorithm which allows to transform the varchenko matrix in a lower or upper triangular form. Examining the PC minors for the dimension 6 can be considered a borderline case but for all higher dimensions the analysis of the varchenko determinant cannot be recommended without the proposed improvements. An upper limit for the developed programme to work are partial cubes with 50 equivalence classes. For more than 50 equivalence classes manual adjustments in the code are necessary. [28]

[27] path_of_file stands for the location of the file
[28] See the python code in A

7. Calculations

Below the determinants for the varchenko matrices of the forbidden minors are listed using the abbreviation VD. The results are displayed as they were produced in maple.

$\underline{\mathbf{VD}(Q_4^{-*})}$ =

$-(x_1-1)^6(x_1+1)^6(x_2-1)^6(x_2+1)^6(x_3-1)^6(x_3+1)^6(x_4-1)^6$

$(x_4+1)^6(x_1x_3x_4-1)(x_1x_3x_4+1)$

$\underline{\mathbf{VD}(Q_4^{--}(1))}$ =

$(x_1-1)^6(x_1+1)^6(x_2-1)^5(x_2+1)^5(x_3-1)^5(x_3+1)^5(x_4-1)^5$

$(x_4+1)^5(x_2x_3x_4-1)(x_2x_3x_4+1)$

$\underline{\mathbf{VD}(Q_4^{--}(2))}$ =

$(x_1-1)^5(x_1+1)^5(x_2-1)^5(x_2+1)^5(x_3-1)^4(x_3+1)^4(x_4-1)^4(x_4+1)^4$

$(x_1^2x_2^2x_3^2x_4^2-x_1^2x_3^2x_4^2-x_2^2x_3^2x_4^2+1)$

$\underline{\mathbf{VD}(Q_4^{--}(3))}$ =

$-(x_1-1)^4(x_1+1)^4(x_2-1)^4(x_2+1)^4(x_3-1)^4(x_3+1)^4(x_4-1)^3(x_4+1)^3$

$(2x_1^2x_2^2x_3^2x_4^2-x_1^2x_2^2x_4^2-x_1^2x_3^2x_4^2-x_2^2x_3^2x_4^2+1)$

$\underline{\mathbf{VD}(Q_4^{--}(4))}$ =

$(x_1-1)^3(x_1+1)^3(x_2-1)^3(x_2+1)^3(x_3-1)^3(x_3+1)^3(x_4-1)^3(x_4+1)^3$

$(3x_1^2x_2^2x_3^2x_4^2-x_1^2x_2^2x_3^2-x_1^2x_2^2x_4^2-x_1^2x_3^2x_4^2-x_2^2x_3^2x_4^2+1)$

The minors for the dimension n = 5 are presented below.

$\underline{\mathbf{VD}(Q_5^{-*})}$ =

$-(x_1-1)^{14}(x_1+1)^{14}(x_2-1)^{14}(x_2+1)^{14}(x_3-1)^{14}(x_3+1)^{14}(x_4-1)^{14}(x_4+1)^{14}$

$(x_5-1)^{14}(x_5+1)^{14}(x_2x_3x_4x_5-1)(x_2x_3x_4x_5+1)$

$\underline{\mathbf{VD}(Q_5^{--}(1))} =$

-$(x_1-1)^{13}(x_1+1)^{13}(x_2-1)^{13}(x_2+1)^{13}(x_3-1)^{14}(x_3+1)^{14}(x_4-1)^{13}(x_4+1)^{13}$
$(x_5-1)^{13}(x_5+1)^{13}(x_1x_2x_4x_5-1)(x_1x_2x_4x_5+1)$

$\underline{\mathbf{VD}(Q_5^{--}(2))} =$

$(x_1-1)^{13}(x_1+1)^{13}(x_2-1)^{12}(x_2+1)^{12}(x_3-1)^{13}(x_3+1)^{13}(x_4-1)^{12}(x_4+1)^{12}$
$(x_5-1)^{12}(x_5+1)^{12}(x_1^2x_2^2x_3^2x_4^2x_5^2-x_1^2x_2^2x_4^2x_5^2-x_2^2x_3^2x_4^2x_5^2+1)$

$\underline{\mathbf{VD}(Q_5^{--}(3))} =$

$(x_1-1)^{12}(x_1+1)^{12}(x_2-1)^{12}(x_2+1)^{12}(x_3-1)^{12}(x_3+1)^{12}(x_4-1)^{11}(x_4+1)^{11}$
$(x_5-1)^{11}(x_5+1)^{11}(2x_1^2x_2^2x_3^2x_4^2x_5^2-x_1^2x_2^2x_4^2x_5^2-x_1^2x_3^2x_4^2x_5^2-x_2^2x_3^2x_4^2x_5^2+1)$

$\underline{\mathbf{VD}(Q_5^{--}(4))} =$

$(x_1-1)^{11}(x_1+1)^{11}(x_2-1)^{11}(x_2+1)^{11}(x_3-1)^{11}(x_3+1)^{11}(x_4-1)^{11}(x_4+1)^{11}$
$(x_5-1)^{10}(x_5+1)^{10}(3x_1^2x_2^2x_3^2x_4^2x_5^2-x_1^2x_2^2x_3^2x_5^2-x_1^2x_2^2x_4^2x_5^2-x_1^2x_3^2x_4^2x_5^2-x_2^2x_3^2x_4^2x_5^2+1)$

$\underline{\mathbf{VD}(Q_5^{--}(5))} =$

$(x_1-1)^{10}(x_1+1)^{10}(x_2-1)^{10}(x_2+1)^{10}(x_3-1)^{10}(x_3+1)^{10}(x_4-1)^{10}(x_4+1)^{10}$
$(x_5-1)^{10}(x_5+1)^{10}(4x_1^2x_2^2x_3^2x_4^2x_5^2-x_1^2x_2^2x_3^2x_4^2-x_1^2x_2^2x_3^2x_5^2-x_1^2x_2^2x_4^2x_5^2-x_1^2x_3^2x_4^2x_5^2-$
$x_2^2x_3^2x_4^2x_5^2+1)$

With a few trivial reorganizations it can be seen that some determinants of the minors have a factorization which is consistent with theorem 2. For example after applying some basic substitutions the minor Q_5^{-*} can be written as

$\underline{\mathbf{VD}(Q_5^{-*})} \quad =$

(1 - x_1^2)$^{14}(1-x_2^2)^{14}(1-x_3^2)^{14}(1-x_4^2)^{14}(1-x_5^2)^{14}(1-x_2^2x_3^2x_4^2x_5^2)$

This also applies for $Q_4^{--}(1)$ or $Q_5^{--}(1)$. For the sake of self inclusion, completeness and in order to make the mentioned concepts from the previous chapters more tangible it will be shown why the partial cube Q_5^{-*} is not a COM. In the figure below Q_5^{-*} is displayed using different partial cube representations.

	binary					+1/-1					+ / -				
Vectors	Digit1	Digit2	Digit3	Digit4	Digit5	Entry1	Entry2	Entry3	Entry4	Entry5	Entry1	Entry2	Entry3	Entry4	Entry5
0	0	1	1	1	1	-1	+1	+1	+1	+1	-	+	+	+	+
1	1	0	1	1	1	+1	-1	+1	+1	+1	+	-	+	+	+
2	1	1	1	0	1	+1	+1	+1	-1	+1	+	+	+	-	+
3	1	1	1	1	0	+1	+1	+1	+1	-1	+	+	+	+	-
4	1	1	0	1	1	+1	+1	-1	+1	+1	+	+	-	+	+
5	0	0	0	1	1	-1	-1	-1	+1	+1	-	-	-	+	+
6	0	0	1	0	1	-1	-1	+1	-1	+1	-	-	+	-	+
7	0	0	1	1	0	-1	-1	+1	+1	-1	-	-	+	+	-
8	0	0	1	1	1	-1	-1	+1	+1	+1	-	-	+	+	+
9	0	1	0	0	1	-1	+1	-1	-1	+1	-	+	-	-	+
10	0	1	0	1	1	-1	+1	-1	+1	+1	-	+	-	+	+
11	0	1	0	1	0	-1	+1	-1	+1	-1	-	+	-	+	-
12	0	1	1	0	1	-1	+1	+1	-1	+1	-	+	+	-	+
13	0	1	1	0	0	-1	+1	+1	-1	-1	-	+	+	-	-
14	0	1	1	1	0	-1	+1	+1	+1	-1	-	+	+	+	-
15	1	0	1	0	0	+1	-1	+1	-1	-1	+	-	+	-	-
16	1	0	1	0	1	+1	-1	+1	-1	+1	+	-	+	-	+
17	1	1	0	0	0	+1	+1	-1	-1	-1	+	+	-	-	-
18	1	0	1	1	0	+1	-1	+1	+1	-1	+	-	+	+	-
19	1	0	0	0	1	+1	-1	-1	-1	+1	+	-	-	-	+
20	1	0	0	1	0	+1	-1	-1	+1	-1	+	-	-	+	-
21	1	0	0	1	1	+1	-1	-1	+1	+1	+	-	-	+	+
22	1	1	1	0	0	+1	+1	+1	-1	-1	+	+	+	-	-
23	1	1	0	0	1	+1	+1	-1	-1	+1	+	+	-	-	+
24	1	1	0	1	0	+1	+1	-1	+1	-1	+	+	-	+	-
25	0	0	0	0	1	-1	-1	-1	-1	+1	-	-	-	-	+
26	0	0	0	1	0	-1	-1	-1	+1	-1	-	-	-	+	-
27	0	0	1	0	0	-1	-1	+1	-1	-1	-	-	+	-	-
28	0	1	0	0	0	-1	+1	-1	-1	-1	-	+	-	-	-
29	0	0	0	0	0	-1	-1	-1	-1	-1	-	-	-	-	-
30	1	0	0	0	0	+1	-1	-1	-1	-1	+	-	-	-	-
31	1	1	1	1	1	+1	+1	+1	+1	+1	+	+	+	+	+

Figure 13: Different partial cube representations

The vectors missing from Q_5^{-*} are the vectors 30 and 31 which are marked in blue. Following the terminology from chapter 5 the vector 29 can be considered v, the vector 31 corresponds to the antipode -v and the vector 30 is the so called first neighbor. Our reasoning will be based on the "+1/0/-1" representation. Firstly we outline the overall strategy. As a starting point we are assuming that Q_5^{-*} qualifies as a COM and hence the axioms of face symmetry (FS) and strong elimination (SE) apply. Then with the help of those axioms the existence of certain vectors

is established which in turn can be used to imply the existence of a vector which is by definition excluded from Q_5^{-*}. Hence the assumption of Q_5^{-*} being a COM cannot be maintained.

From the following vector pairs in figure 14 it can be inferred on the basis of SE that the vectors 32, 33, 34 and 35 are part of the assumed COM.

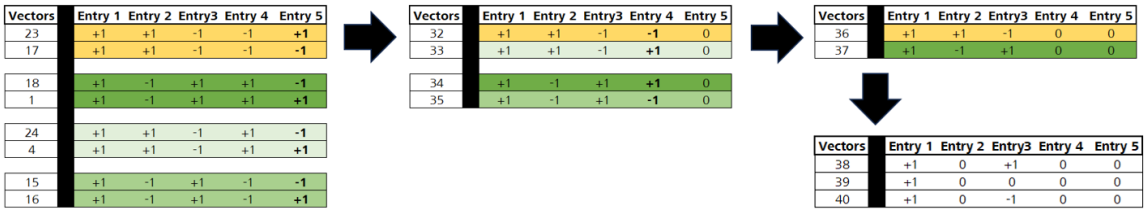

Figure 14: Establishing vectors with the help of the COM axioms

The entries which have opposing signs are displayed in bold. Based on the same argument we can deduct the presence of the vectors 36 and 37. There are now two entries with different signs. SE guarantees the existence of one vector from the vectors 38, 39 or 40. It can be easily seen that if the vectors 38 and 39 are composed with vector 1 the vector 31 is part of Q5-*. Composing vector 40 with vector 29 results also in a contradiction because of number 30. Therefore Q_5^{-*} cannot be a COM.

8. Summary, results and conclusion

In a first step an historical overview has been given which was followed by introducing all the relevant concepts for this thesis. These concepts have then been explained mainly by graphic examples. After this theoretical groundwork has been done the central question of this thesis has been formulated. Then the code how

this task will be solved progammatically has been outlined. Finally the solution which contained the nice factorizations of $Q_4^{--}(1)$, Q_5^{-*} and $Q_5^{--}(1)$ has been presented. Therefore the question whether there exists a non-COM being a partial cube possessing a nice factorization can be positively answered. This allows to falsify the reverse implication of theorem 2. This means from an object having a nice factorization you cannot deduct on its structural property. Still there could be subclasses of partial cubes not being COMs whose varchenko determinant also factorize nicely. Finding or disproving such a factorization theorem or establishing a framework which provides guidelines to narrow it down is up to further research.

Appendices

A. Python Code: Calc_VM.py

```
"""Calc_VM.py
B. Yazici 2024
"""
import DFS
import PartialCube
import copy
from StrongConnectivity import StronglyConnectedComponents
import sympy
from sympy import *
from itertools import combinations

#calculates number of vertices in graph
def NumOfVert(G):
    return len(G)
```

```
#calculates number of equivalence classes in graph
def AmtEquiClassPC(G):
    ver_dict_ec = PartialCube.PartialCubeEdgeLabeling(G)
    coll_ec = set()
    for ver1 in ver_dict_ec:
        for ver2 in ver_dict_ec[ver1]:
            ec = ver_dict_ec[ver1][ver2]
            coll_ec.add(Arrange2EleTuple(ec))
    return(len(coll_ec))

def EquiClassPCReps(G):
    ver_dict_ec = PartialCube.PartialCubeEdgeECs(G)
    return set(ver_dict_ec.keys())

#prints for each vertex to which equivalence classes it is linked
def VerWithVerEC(G):
    ver_dict_ec = PartialCube.PartialCubeEdgeLabeling(G)
    coll_ec = set()
    for ver1 in ver_dict_ec:
```

```
        for ver2 in ver_dict_ec[ver1]:
            ec = ver_dict_ec[ver1][ver2]
            print(ver1,Arrange2EleTuple(ec))

#returns for each vertex to which "ec side" it is linked to (a,b) or (b,a)
def VerWithEC_unarr(G):
    ver_dict_ec = PartialCube.PartialCubeEdgeLabeling(G)
    dict_ec = {}
    coll_ec = set()
    for ver1 in ver_dict_ec:
        coll_ec = set()
        for ver2 in ver_dict_ec[ver1]:
            ec = ver_dict_ec[ver1][ver2]
            coll_ec.add(ec)
        dict_ec[ver1]=coll_ec
    return dict_ec

#returns for each vertex the equivalence class irrespective of side
def VerWithEC(G):
```

```
    ver_dict_ec = PartialCube.PartialCubeEdgeLabeling(G)
    dict_ec = {}
    coll_ec = set()
    for ver1 in ver_dict_ec:
        coll_ec = set()
        for ver2 in ver_dict_ec[ver1]:
            ec = ver_dict_ec[ver1][ver2]
            coll_ec.add(Arrange2EleTuple(ec))
        dict_ec[ver1]=coll_ec
    return dict_ec


#returns for (a,b) a tuple (a,b) if a>b else (b,a)
def Arrange2EleTuple(tup):
    if tup[0] > tup[1]:
        return (tup[1],tup[0])
    else:
        return tup


#returns a set of equivalence classes an arranges them, no distinction between (a,b) or (b,a)
```

```
def CreateSetECArr(set_EC):
    set_a_EC = set()
    for ec in set_EC:
        set_a_EC.add(Arrange2EleTuple(ec))
    return set_a_EC


#returns a dictionary where each equivalence class is linked to x1,x2....
def CreateEcVarDict(coll_ec):
    i = 1
    dict_ec ={}
    for ec in coll_ec:
        dict_ec[ec] = var_dict[i]
        i = i + 1
    return dict_ec


#returns a graph G whose "edge" is removed
def removeEdgeGraph(G,edge):
    G_new = copy.deepcopy(G)
    G_new[edge[1]].remove(edge[0])
```

```
    G_new[edge[0]].remove(edge[1])
    return G_new

#checks if Graph G is connected
def isConnectedGraph(G):
    L = list(StronglyConnectedComponents(G))
    if len(L) != 1:
        return False
    else:
        return True

#returns all the edges from Graph G
def getEdgesGraph(G):
    edges = set()
    for ver in G:
        for g in G[ver]:
            edges.add(Arrange2EleTuple((ver,g)))
    return edges
```

```
#checks for the vertices v and w if they are still connected after all the edges from the equivalence class are removed
def getVMcell(G,v,w,ecsEdges,ecDict):
    vmc = ""
    for ec in ecsEdges:
        G_new = copy.deepcopy(G)
        for edge in ecsEdges[ec]:
            G_new = removeEdgeGraph(G_new,edge)
        if not DFS.reachable(G_new,v,w):
            if vmc == "":
                vmc = ecDict[ec]
            else:
                vmc = vmc * ecDict[ec]
    return vmc

#returns a square matrix of nov x nov
def sqrMat(nov):
    lst = [1]*nov
    M = Matrix([lst])
    for i in range(0,nov-1):
```

```
            M=M.row_insert(1,Matrix([lst]))
        return M

    #returns the Varchenko Matrix for Graph G
    def buildVarMat(G):
        ecs = EquiClassPCReps(G)
        ecDict = CreateEcVarDict(ecs)
        ecsEdges = PartialCube.PartialCubeEdgeECs(G)
        nov = NumOfVert(G)
        M = sqrMat(nov)

        for i in range(0,nov):
            for j in range(0,i):
                z = getVMcell(G,i,j,ecsEdges,ecDict)
                M[(i)*nov+j]=z
                M[(j)*nov+i]=z

        return M
```

```
#saves Matrix in local directory as comma separated txt file
def saveMatrix(M):
    n = shape(M)[1]
    i = 1
    with open('VarchenkoMatrix.txt', 'w') as f:
        for line in M:
            if i % n != 0:
                inp = str(line) + str(",")
                f.write(inp)
            else:
                inp = str(line) + str("\n")
                f.write(inp)
            i = i+1

#saves Determinant
def saveDet(D):
    with open('Det.txt', 'w') as f:
        f.write(str(D))
```

```
#arranges edges (a,b) to (b,a) if b>a so edges belonging to ec (a,b) and (b,a) are put in the class (a,b)
def arrangeDictionary(inp_dict):
    out_dict = {}
    for t in inp_dict:
        if Arrange2EleTuple(t) not in out_dict:
            out_dict[Arrange2EleTuple(t)]=inp_dict[t]
        else:
            out_dict[Arrange2EleTuple(t)]=out_dict[Arrange2EleTuple(t)].union(inp_dict[t])
    return out_dict


#Start: Verzeichnis globaler Parameter/Variable----------------------------------
x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x19,x20,x21,x22,x23,x24,x25,x26,x27,x28,x29,x30,x31,x32,x33,x34,
x35,x36,x37,x38,x39,x40,x41,x42,x43,x44,x45,x46,x47,x48,x49,x50
= sympy.symbols('x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x19,x20,x21,x22,x23,x24,x25,x26,x27,x28,x29
,x30,x31,x32,x33,x34,x35,x36,x37,x38,x39,x40,x41,x42,x43,x44,x45,x46,x47,x48,x49,x50')

var_dict = {1:x1,2:x2,3:x3,4:x4,5:x5,6:x6,7:x7,8:x8,9:x9,10:x10,11:x11,12:x12,13:x13,14:x14,
15:x15,16:x16,17:x17,18:x18,19:x19,20:x20,21:x21,22:x22,23:x23,24:x14,25:x25,26:x26,27:x27,
28:x28,29:x29,30:x30,31:x31,32:x32,33:x13,34:x34,35:x35,36:x36,37:x37,38:x38,39:x39,40:x40,
```

41:x41,42:x42,43:x43,44:x44,45:x45,46:x46,47:x47,48:x48,49:x49,50:x50}

#Ende: Verzeichnis globaler Parameter Variable————————————————————

print("————————-Start————————-")

#Testgraphs:

#G = {0:[1,2,4],1:[0,3,5],2:[0,3],3:[1,2,6],4:[0,5],5:[4,1,6],6:[5,3]}

#G = {0:[1,2,4],1:[0,3,5],2:[0,3],3:[1,2,6],4:[0,5],5:[4,1,6],6:[5,3,7],7:[6]}

#G = {0:[2,3],1:[3,4],2:[0],3:[0,1],4:[1]}

#G = {0:[1,5,6],1:[0,2],2:[1,3,7],3:[2,4],4:[3,5,8],5:[0,4],6:[0,9],7:[2,9],8:[4,9],9:[6,7,8]}

#G = {0:[1,5],1:[0,2],2:[1,3],3:[2,4,6,10],4:[3,5,7],5:[0,4],6:[3,7,9],7:[4,6,8],8:[7,9],9:[6,8,10],10:[3,9]}

#G = {0:[1,2,4,6] ,1:[0,3,5] ,2:[0,3,7] ,3:[1,2,8] ,4:[0,5,9] ,5:[1,4,10] ,6:[0,7,9] ,7:[2,6,8,11] ,8:[3,7,12] ,9:[4,6,10,11] ,10:[5,9,12] ,11:[7,9,12] ,12:[8,10,11] }

#G = {0:[1,2,4,7] ,1:[0,3,5] ,2:[0,3,8] ,3:[1,2,6,9] ,4:[0,5,10] ,5:[1,4,6,11] ,6:[3,5,13] ,7:[0,8,10] ,8:[2,7,9,12] ,9:[3,8,13] ,10:[4,7,11,12] ,11:[5,10,13] ,12:[8,10,13],13:[6,9,11,12] }

#G = {0:[4,5,6] ,1:[4,5,7] ,2:[4,6,7] ,3:[5,6,7] ,4:[0,1,2] ,5:[0,1,3] ,6:[0,2,3] ,7:[1,2,3]}

#G = {0:[1,2] ,1:[0,3] ,2:[0,4] ,3:[1,5] ,4:[2,5,6] ,5:[3,4,7] ,6:[4,7] ,7:[5,6]}

#G = {0:[1,2,4,7],1:[0,3,8],2:[0,3,5],3:[1,2,6],4:[0,10],5:[2,6,7,9],6:[3,5,8],7:[0,5,8,10],
8:[1,6,7,11] ,9:[5,10], 10:[4,7,9,11],11:[8,10] }

#G = {0:[1,4] ,1:[0,2,5] ,2:[1,3,6] ,3:[2,7] ,4:[0,5,8] ,5:[1,4,6] ,6:[2,5,7] ,7:[3,6,9] ,8:[4,9] ,9:[7,8] }

#G = {0:[1,2,5] ,1:[0,3,7] ,2:[0,3,4] ,3:[1,2,6,8] ,4:[2,5,8,9] ,5:[0,4,11] ,6:[3,7,10] ,7:[1,6,12] ,8:[3,4,10] ,9:[4,10,11] ,10:[6,8,9,12] ,11:[5,9,12] ,12:[7,10,11] }

#G = {0:[1],1:[0]}

#Forbidden Minors of Dim 4:

#G_4_4 = {0:[1,5,6],1:[0,2],2:[1,3,7],3:[2,4],4:[3,5,8],5:[0,4],6:[0,9],7:[2,9],8:[4,9],9:[6,7,8]}

#G_4_3 = {0:[1,5,6],1:[0,2],2:[1,3,7],3:[2,4],4:[3,5,8],5:[0,4,10],6:[0,9,10],7:[2,9],8:[4,9,10],9:[6,7,8],10:[5,6,8]}

#G_4_2 = {0:[1,5,6],1:[0,2],2:[1,3,7],3:[2,4,11],4:[3,5,8],5:[0,4,10],6:[0,9,10],7:[2,9,11],8:[4,9,10,11],9:[6,7,8],10:[5,6,8],11:[3,7,8]}

#G_4_1 = {0:[1,5,6],1:[0,2,12],2:[1,3,7],3:[2,4,11],4:[3,5,8],5:[0,4,10],6:[0,9,10,12],7:[2,9,11,12],8:[4,9,10,11],9:[6,7,8],10:[5,6,8],11:[3,7,8],12:[1,6,7]}

#G_4_star = {0:[1,5,6],1:[0,2,12],2:[1,3,7],3:[2,4,11],4:[3,5,8],5:[0,4,10],6:[0,9,10,12],7:[2,9,11,12],8:[4,9,10,11],9:[6,7,8],10:[5,6,8,13],11:[3,7,8,13],12:[1,6,7,13],13:[10,11,12]}

#Forbidden Minors of Dim 5:

#G_5_star = {0:[8,10,12,14],1:[8,16,18,21],2:[12,16,22,23],3:[14,18,22,24],4:[10,21,23,24],5:[8,10,21,25,26],6:[8,12,16,25,27],7:[8,14,18,26,27],8:[0,1,5,6,7],9:[10,12,23,25,28],10:[0,4,5,9,11],11:[10,14,24,26,28],12:[0,2,6,9,13],13:[12,14,22,27,28],14:[0,3,7,11,13],15:[16,18,22,27],16:[1,2,6,15,19],17:[22,23,24,28],18:[1,3,7,15,20],19:[16,21,23,25],20:[18,21,24,26],21:[1,4,5,19,20],22:[2,3,13,15,17],23:[2,4,9,17,19],24:[3,4,11,17,20],25:[5,6,9,19,29],26:[5,7,11,20,29],27:[6,7,13,15,29],28:[9,11,13,17,29],29:[25,26,27,28]}

#G_5_1 = {0:[8,10,12,14],1:[8,16,18,21],2:[12,16,22,23],3:[14,18,22,24],4:[10,21,23,24],5:[8,10,21,25,26],6:[8,12,16,25,27],
7:[8,14,18,26,27],8:[0,1,5,6,7],9:[10,12,23,25,28],10:[0,4,5,9,11],11:[10,14,24,26,28],12:[0,2,6,9,13],13:[12,14,22,27,28],14:[0,3,7,11,13],
15:[16,18,22,27],16:[1,2,6,15,19],17:[22,23,24,28],18:[1,3,7,15,20],19:[16,21,23,25],20:[18,21,24,26],21:[1,4,5,19,20],22:[2,3,13,15,17],
23:[2,4,9,17,19],24:[3,4,11,17,20],25:[5,6,9,19],26:[5,7,11,20],27:[6,7,13,15],28:[9,11,13,17]}
#G_5_2 = {0:[8,10,12,14],1:[8,16,18,21],2:[12,16,22,23],3:[14,18,22,24],4:[10,21,23,24],5:[8,10,21,25,26],6:[8,12,16,25,27],
7:[8,14,18,26,27],8:[0,1,5,6,7],9:[10,12,23,25],10:[0,4,5,9,11],11:[10,14,24,26],12:[0,2,6,9,13],13:[12,14,22,27],14:[0,3,7,11,13],
15:[16,18,22,27],16:[1,2,6,15,19],17:[22,23,24],18:[1,3,7,15,20],19:[16,21,23,25],20:[18,21,24,26],21:[1,4,5,19,20],22:[2,3,13,15,17],
23:[2,4,9,17,19],24:[3,4,11,17,20],25:[5,6,9,19],26:[5,7,11,20],27:[6,7,13,15]}
#G_5_3 = {0:[8,10,12,14],1:[8,16,18,21],2:[12,16,22,23],3:[14,18,22,24],4:[10,21,23,24],5:[8,10,21,25,26],6:[8,12,16,25],
7:[8,14,18,26],8:[0,1,5,6,7],9:[10,12,23,25],10:[0,4,5,9,11],11:[10,14,24,26],12:[0,2,6,9,13],13:[12,14,22],14:[0,3,7,11,13],15:[16,18,22]
,16:[1,2,6,15,19],17:[22,23,24],18:[1,3,7,15,20],19:[16,21,23,25],20:[18,21,24,26],21:[1,4,5,19,20],22:[2,3,13,15,17],23:[2,4,9,17,19],
24:[3,4,11,17,20],25:[5,6,9,19],26:[5,7,11,20]}
#G_5_4 = {0:[8,10,12,14],1:[8,16,18,21],2:[12,16,22,23],3:[14,18,22,24],4:[10,21,23,24],5:[8,10,21,25],6:[8,12,16,25],7:[8,14,18],
8:[0,1,5,6,7],9:[10,12,23,25],10:[0,4,5,9,11],11:[10,14,24],12:[0,2,6,9,13],13:[12,14,22],14:[0,3,7,11,13],15:[16,18,22],16:[1,2,6,15,19],
17:[22,23,24],18:[1,3,7,15,20],19:[16,21,23,25],20:[18,21,24],21:[1,4,5,19,20],22:[2,3,13,15,17],23:[2,4,9,17,19],24:[3,4,11,17,20],
25:[5,6,9,19]}
#G_5_5 = {0:[8,10,12,14],1:[8,16,18,21],2:[12,16,22,23],3:[14,18,22,24],4:[10,21,23,24],5:[8,10,21],6:[8,12,16],7:[8,14,18],
8:[0,1,5,6,7],9:[10,12,23],10:[0,4,5,9,11],11:[10,14,24],12:[0,2,6,9,13],13:[12,14,22],14:[0,3,7,11,13],15:[16,18,22],16:[1,2,6,15,19],
17:[22,23,24],18:[1,3,7,15,20],19:[16,21,23],20:[18,21,24],21:[1,4,5,19,20],22:[2,3,13,15,17],23:[2,4,9,17,19],24:[3,4,11,17,20]}

```
if not PartialCube.isPartialCube(G):
    print("Der Graph G ist kein Partial Cube")
else:
    print("G ist ein Partial Cube")
    print("G hat die folgende Äquivalenzklassen mit Bezeichner Vertreter xi")
    ecs = EquiClassPCReps(G)
    print(CreateEcVarDict(ecs))
    print("G hat die folgenden Äquivalenzklassen inkl. den Klassenmitgliedern:")
    print(PartialCube.PartialCubeEdgeECs(G))
    print("Die Varchenko Matrix VM von G lautet:")
    M = buildVarMat(G)
    sympy.pretty_print(M)
    saveMatrix(M)
    print("VM wurde als VarchenkoMatrix.txt gespeichert")
print("————————————-End————————————-")
```

B. Python Code: PartialCube.py

```
"""PartialCube.py

Test whether a graph is an isometric subgraph of a hypercube.

D. Eppstein, September 2005, rewritten May 2007 per arxiv:0705.1025. incl modifications are by B. Yazici """

import BFS
import Medium
from Bipartite import isBipartite
from UnionFind import UnionFind
from StrongConnectivity import StronglyConnectedComponents
from Graphs import isUndirected
import unittest
from itertools import combinations

def PartialCubeEdgeLabeling(G):
    """
    Label edges of G by their equivalence classes in a partial cube structure.
```

We follow the algorithm of arxiv:0705.1025, in which a number of equivalence classes equal to the maximum degree of G can be found simultaneously by a single breadth first search, using bitvectors. However, in order to avoid deep recursions (problematic in Python) we use a union-find data structure to keep track of edge identifications discovered so far. That is, we repeatedly contract our initial graph, maintaining as we do the property that G[v][w] points to a union-find set representing edges in the original graph that have been contracted to the single edge v-w.
"""


```
# Some simple sanity checks
if not isUndirected(G):
    raise Medium.MediumError("graph is not undirected")
L = list(StronglyConnectedComponents(G))
if len(L) != 1:
    raise Medium.MediumError("graph is not connected")

# Set up data structures for algorithm:
# - UF: union find data structure representing known edge equivalences
# - CG: contracted graph at current stage of algorithm
```

```
# - LL: limit on number of remaining available labels
UF = UnionFind()
CG = {v:{w:(v,w) for w in G[v]} for v in G}
NL = len(CG)-1

# Initial sanity check: are there few enough edges?
# Needed so that we don't try to use union-find on a dense
# graph and incur superquadratic runtimes.
n = len(CG)
m = sum([len(CG[v]) for v in CG])
if 1«(m//n) > n:
    raise Medium.MediumError("graph has too many edges")

# Main contraction loop in place of the original algorithm's recursion
while len(CG) > 1:
    if not isBipartite(CG):
        raise Medium.MediumError("graph is not bipartite")

    # Find max degree vertex in G, and update label limit
```

```
deg,root = max([(len(CG[v]),v) for v in CG])
if deg > NL:
    raise Medium.MediumError("graph has too many equivalence classes")
NL -= deg

# Set up bitvectors on vertices
bitvec = {v:0 for v in CG}
neighbors = {}
i = 0
for neighbor in CG[root]:
    bitvec[neighbor] = 1«i
    neighbors[1«i] = neighbor
    i += 1

# Breadth first search to propagate bitvectors to the rest of the graph
for LG in BFS.BreadthFirstLevels(CG,root):
    for v in LG:
        for w in LG[v]:
            bitvec[w] |= bitvec[v]
```

```
# Make graph of labeled edges and union them together
labeled = {v:set() for v in CG}
for v in CG:
    for w in CG[v]:
        diff = bitvec[v]∧bitvec[w]
        if not diff or bitvec[w] & ∼ bitvec[v] == 0:
            continue # zero edge or wrong direction
        if diff not in neighbors:
            raise Medium.MediumError("multiply-labeled edge")
        neighbor = neighbors[diff]
        UF.union(CG[v][w],CG[root][neighbor])
        UF.union(CG[w][v],CG[neighbor][root])
        labeled[v].add(w)
        labeled[w].add(v)

# Map vertices to components of labeled-edge graph
component = {}
compnum = 0
```

```
for SCC in StronglyConnectedComponents(labeled):
    for v in SCC:
        component[v] = compnum
    compnum += 1

# generate new compressed subgraph
NG = {i:{} for i in range(compnum)}
for v in CG:
    for w in CG[v]:
        if bitvec[v] == bitvec[w]:
            vi = component[v]
            wi = component[w]
            if vi == wi:
                raise Medium.MediumError("self-loop in contracted graph")
            if wi in NG[vi]:
                UF.union(NG[vi][wi],CG[v][w])
            else:
                NG[vi][wi] = CG[v][w]
CG = NG
```

```
    # Here with all edge equivalence classes represented by UF.
    # Turn them into a labeled graph and return it.
    return {v:{w:UF[v,w] for w in G[v]} for v in G}


#modified by BYAZ in order to return the library of eqclasses and the corresponding edges
def PartialCubeEdgeECs(G):
    """
    Label edges of G by their equivalence classes in a partial cube structure.
    We follow the algorithm of arxiv:0705.1025, in which a number of equivalence classes equal to the maximum
    degree of G can be found simultaneously by a single breadth first search, using bitvectors. However, in order
    to avoid deep recursions (problematic in Python) we use a union-find data structure to keep track of edge
    identifications discovered so far. That is, we repeatedly contract our initial graph, maintaining as we do the
    property that G[v][w] points to a union-find set representing edges in the original graph that have been
    contracted to the single edge v-w.
    """

    # Some simple sanity checks
    if not isUndirected(G):
```

```
        raise Medium.MediumError("graph is not undirected")
    L = list(StronglyConnectedComponents(G))
    if len(L) != 1:
        raise Medium.MediumError("graph is not connected")

    # Set up data structures for algorithm:
    # - UF: union find data structure representing known edge equivalences
    # - CG: contracted graph at current stage of algorithm
    # - LL: limit on number of remaining available labels
    UF = UnionFind()
    CG = {v:{w:(v,w) for w in G[v]} for v in G}
    NL = len(CG)-1

    # Initial sanity check: are there few enough edges?
    # Needed so that we don't try to use union-find on a dense
    # graph and incur superquadratic runtimes.
    n = len(CG)
    m = sum([len(CG[v]) for v in CG])
    if 1<<(m//n) > n:
```

```
        raise Medium.MediumError("graph has too many edges")

    # Main contraction loop in place of the original algorithm's recursion
    while len(CG) > 1:
        if not isBipartite(CG):
            raise Medium.MediumError("graph is not bipartite")

        # Find max degree vertex in G, and update label limit
        deg,root = max([(len(CG[v]),v) for v in CG])
        if deg > NL:
            raise Medium.MediumError("graph has too many equivalence classes")
        NL -= deg

        # Set up bitvectors on vertices
        bitvec = {v:0 for v in CG}
        neighbors = {}
        i = 0
        for neighbor in CG[root]:
            bitvec[neighbor] = 1«i
```

```
        neighbors[1«i] = neighbor
        i += 1

    # Breadth first search to propagate bitvectors to the rest of the graph
    for LG in BFS.BreadthFirstLevels(CG,root):
        for v in LG:
            for w in LG[v]:
                bitvec[w] |= bitvec[v]

    # Make graph of labeled edges and union them together
    labeled = {v:set() for v in CG}
    for v in CG:
        for w in CG[v]:
            diff = bitvec[v]∧bitvec[w]
            if not diff or bitvec[w] & ∼ bitvec[v] == 0:
                continue # zero edge or wrong direction
            if diff not in neighbors:
                raise Medium.MediumError("multiply-labeled edge")
            neighbor = neighbors[diff]
```

```
                UF.union(CG[v][w],CG[root][neighbor])
                UF.union(CG[w][v],CG[neighbor][root])
                labeled[v].add(w)
                labeled[w].add(v)

# Map vertices to components of labeled-edge graph
component = {}
compnum = 0
for SCC in StronglyConnectedComponents(labeled):
    for v in SCC:
        component[v] = compnum
    compnum += 1

# generate new compressed subgraph
NG = {i:{} for i in range(compnum)}
for v in CG:
    for w in CG[v]:
        if bitvec[v] == bitvec[w]:
            vi = component[v]
```

```
                wi = component[w]
                if vi == wi:
                    raise Medium.MediumError("self-loop in contracted graph")
                if wi in NG[vi]:
                    UF.union(NG[vi][wi],CG[v][w])
                else:
                    NG[vi][wi] = CG[v][w]

        CG = NG

# Here with all edge equivalence classes represented by UF.
# Turn them into a labeled graph and return it.
edges = getEdgesGraph(G)
result = {}

for edge in edges:
    temp = set()
    #temp.add(UF[edge[0],edge[1]])
    #result[edge]=temp
```

```
            if UF[edge[0],edge[1]] in result.keys():
                result[UF[edge[0],edge[1]]].add(edge)
            else:
                temp.add(edge)
                result[UF[edge[0],edge[1]]]=temp

        return ComRedEc(arrangeDictionary(result))

    #by BYAZ
    #returns for (a,b) a tuple (a,b) if a>b else (b,a)
    def Arrange2EleTuple(tup):
        if tup[0] > tup[1]:
            return (tup[1],tup[0])
        else:
            return tup

    #by BYAZ
    #returns all the edges from Graph G
```

```
def getEdgesGraph(G):
    edges = set()
    for ver in G:
        for g in G[ver]:
            edges.add(Arrange2EleTuple((ver,g)))
    return edges


def MediumForPartialCube(G):
    """
    Find a medium corresponding to the partial cube G.
    Raises MediumError if G is not a partial cube.
    Uses the O(n^2) time algorithm of arxiv:0705.1025.
    """
    L = PartialCubeEdgeLabeling(G)
    M = Medium.LabeledGraphMedium(L)
    Medium.RoutingTable(M) # verification step per arxiv:0705.1025
    return M


def PartialCubeLabeling(G):
```

```
    """Return vertex labels with Hamming distance = graph distance."""
    return Medium.HypercubeEmbedding(MediumForPartialCube(G))

def isPartialCube(G):
    """Test whether the given graph is a partial cube."""
    try:
        MediumForPartialCube(G)
        return True
    except Medium.MediumError:
        return False

#by BYAZ
#removes duplicate ecs out of dictonary
def arrangeDictionary(inp_dict):
    out_dict = {}
    for t in inp_dict:
        if Arrange2EleTuple(t) not in out_dict:
            out_dict[Arrange2EleTuple(t)]=inp_dict[t]
        else:
```

```
                out_dict[Arrange2EleTuple(t)]=out_dict[Arrange2EleTuple(t)].union(inp_dict[t])
    return out_dict

#by BYAZ
#gets dictionary and unions ecs which are not disjoint
def ComRedEc(ecsrep):
    for ec in ecsrep:
        ecsrep[ec].add(ec)

    for key1, key2 in combinations(ecsrep.keys(), r = 2):
        if not(ecsrep[key1].isdisjoint(ecsrep[key2])):
            ecsrep[key1]=ecsrep[key1].union(ecsrep[key2])
            ecsrep[key2]=ecsrep[key2].union(ecsrep[key1])

    fr = {}
    for ec in ecsrep:
        if ecsrep[ec] not in fr.values():
            fr[ec]=ecsrep[ec]
    return fr
```

```
def PartialCubeECTest(G):
    """
    Label edges of G by their equivalence classes in a partial cube structure.

    We follow the algorithm of arxiv:0705.1025, in which a number of equivalence classes equal to the maximum
    degree of G can be found simultaneously by a single breadth first search, using bitvectors. However, in
    order to avoid deep recursions (problematic in Python) we use a union-find data structure to keep track of
    edge identifications discovered so far. That is, we repeatedly contract our initial graph,maintaining as we do
     the property that G[v][w] points to a union-find set representing edges in the original graph that have
    been contracted to the single edge v-w.
    """

    # Some simple sanity checks
    if not isUndirected(G):
        raise Medium.MediumError("graph is not undirected")
    L = list(StronglyConnectedComponents(G))
    if len(L) != 1:
        raise Medium.MediumError("graph is not connected")
```

```
# Set up data structures for algorithm:
# - UF: union find data structure representing known edge equivalences
# - CG: contracted graph at current stage of algorithm
# - LL: limit on number of remaining available labels
UF = UnionFind()
CG = {v:{w:(v,w) for w in G[v]} for v in G}
NL = len(CG)-1

# Initial sanity check: are there few enough edges?
# Needed so that we don't try to use union-find on a dense
# graph and incur superquadratic runtimes.
n = len(CG)
m = sum([len(CG[v]) for v in CG])
if 1«(m//n) > n:
    raise Medium.MediumError("graph has too many edges")

# Main contraction loop in place of the original algorithm's recursion
while len(CG) > 1:
    if not isBipartite(CG):
```

```
        raise Medium.MediumError("graph is not bipartite")

    # Find max degree vertex in G, and update label limit
    deg,root = max([(len(CG[v]),v) for v in CG])
    if deg > NL:
        raise Medium.MediumError("graph has too many equivalence classes")
    NL -= deg

    # Set up bitvectors on vertices
    bitvec = {v:0 for v in CG}
    neighbors = {}
    i = 0
    for neighbor in CG[root]:
        bitvec[neighbor] = 1«i
        neighbors[1«i] = neighbor
        i += 1

    # Breadth first search to propagate bitvectors to the rest of the graph
    for LG in BFS.BreadthFirstLevels(CG,root):
```

```
        for v in LG:
            for w in LG[v]:
                bitvec[w] |= bitvec[v]

    # Make graph of labeled edges and union them together
    labeled = {v:set() for v in CG}
    for v in CG:
        for w in CG[v]:
            diff = bitvec[v]∧bitvec[w]
            if not diff or bitvec[w] & ∼ bitvec[v] == 0:
                continue # zero edge or wrong direction
            if diff not in neighbors:
                raise Medium.MediumError("multiply-labeled edge")
            neighbor = neighbors[diff]
            UF.union(CG[v][w],CG[root][neighbor])
            UF.union(CG[w][v],CG[neighbor][root])
            labeled[v].add(w)
            labeled[w].add(v)
```

```
# Map vertices to components of labeled-edge graph
component = {}
compnum = 0
for SCC in StronglyConnectedComponents(labeled):
    for v in SCC:
        component[v] = compnum
    compnum += 1

# generate new compressed subgraph
NG = {i:{} for i in range(compnum)}
for v in CG:
    for w in CG[v]:
        if bitvec[v] == bitvec[w]:
            vi = component[v]
            wi = component[w]
            if vi == wi:
                raise Medium.MediumError("self-loop in contracted graph")
            if wi in NG[vi]:
                UF.union(NG[vi][wi],CG[v][w])
```

```
                else:
                    NG[vi][wi] = CG[v][w]

    CG = NG

# Here with all edge equivalence classes represented by UF.
# Turn them into a labeled graph and return it.
edges = getEdgesGraph(G)
for e in edges:
    print(e,UF[e[0],e[1]])
print("————————")
for e in edges:
    print(e,UF[e[1],e[0]])

# edges = getEdgesGraph(G)
# result = {}

# for edge in edges:
#       temp = set()
```

```
#       #temp.add(UF[edge[0],edge[1]])
#       #result[edge]=temp

#       if UF[edge[0],edge[1]] in result.keys():
#           result[UF[edge[0],edge[1]]].add(edge)
#       else:
#           temp.add(edge)
#           result[UF[edge[0],edge[1]]]=temp

# print(result)
# return arrangeDictionary(result)
```

C. Maple Code

```
with(LinearAlgebra);
Q_4_star := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_4_0.txt", delimiter = ",");
Q_4_star := map(t -> `if`(type(t, string), parse(t), t), X_4_0);
factor(Determinant(X_4_0));

Q_4_1 := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_4_1.txt", delimiter = ",");
Q_4_1 := map(t -> `if`(type(t, string), parse(t), t), X_4_1);
factor(Determinant(X_4_1));

Q_4_2 := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_4_2.txt", delimiter = ",");
Q_4_2 := map(t -> `if`(type(t, string), parse(t), t), X_4_2);
factor(Determinant(X_4_2));

Q_4_3 := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_4_3.txt", delimiter = ",");
Q_4_3 := map(t -> `if`(type(t, string), parse(t), t), X_4_3);
factor(Determinant(X_4_3));

Q_4_4 := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_4_4.txt", delimiter = ",");
```

```
Q_4_4 := map(t -> `if`(type(t, string), parse(t), t), X_4_4);
factor(Determinant(X_4_4));

Q_5_star := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_5_star.txt", delimiter = ",");
Q_5_star := map(t -> `if`(type(t, string), parse(t), t), Q_5_star);
Q_5_star := GaussianElimination(Q_5_star);
factor(Determinant(Q_5_star));

Q_5_1 := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_5_1.txt", delimiter = ",");
Q_5_1 := map(t -> `if`(type(t, string), parse(t), t), Q_5_1);
Q_5_1 := GaussianElimination(Q_5_1);
factor(Determinant(Q_5_1));

Q_5_2 := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_5_2.txt", delimiter = ",");
Q_5_2 := map(t -> `if`(type(t, string), parse(t), t), Q_5_2);
Q_5_2 := GaussianElimination(Q_5_2);
factor(Determinant(Q_5_2));

Q_5_3 := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_5_3.txt", delimiter = ",");
```

```
Q_5_3 := map(t -> `if`(type(t, string), parse(t), t), Q_5_3);
Q_5_3 := GaussianElimination(Q_5_3);
factor(Determinant(Q_5_3));

Q_5_4 := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_5_4.txt", delimiter = ",");
Q_5_4 := map(t -> `if`(type(t, string), parse(t), t), Q_5_4);
Q_5_4 := GaussianElimination(Q_5_4);
factor(Determinant(Q_5_4));

Q_5_5 := ImportMatrix("C:\Users\Administrator\Desktop\PC\VM_5_5.txt", delimiter = ",");
Q_5_5 := map(t -> `if`(type(t, string), parse(t), t), Q_5_5);
Q_5_5 := GaussianElimination(Q_5_5);
factor(Determinant(Q_5_5));
```

References

[1] Hirokazu Nishimura, Susumu Kuroda: *A Lost Mathematician, Takeo Nakasawa The Forgotten Father of Matroid Theory.* Birkhäuser, c2009

[2] P. Gregor: *Characterizations of hypercubes - a survey.* https://ktiml.mff.cuni.cz/ gregor/hypercube/hypercube-course.htm, Lecture 4

[3] K. Ovchinnikov, *Graphs and Cubes.* Universitext, Springer

[4] K. Knauer, T. Marc, *On Tope Graphs of Complexes of Oriented Matroids.* Discrete & Computational Geometry, 2020 - Springer

[5] W. Hochstättler, S. Keip, K. Knauer, *The Signed Varchenko Determinant for Complexes of Oriented Matroids.* https://arxiv.org/abs/2211.13986

[6] W. Hochstättler, V. Welker, *The Varchenko de-*

terminant for oriented matroids. https: Mathematische Zeitschrift, 2019 - Springer

[7] R. Stanley, *An Introduction to Hyperplane Arrangements.* IAS/Park City Mathematical Series, volume 14. American Mathematical Society, Providence, RI, 2004

[8] D. Eppstein, J. Falmagne, S. Ovchinnikov, *Media Theory.* Interdisciplinary Applied Mathematics, Springer

[9] D. Eppstein, *Python library.* https://ics.uci.edu/ eppstein/PADS/

[10] D. Eppstein, *Recognizing partial cubes in quadratic time.* https://arxiv.org/abs/0705.1025

[11] H. Bandelt, V. Chepoi, K. Knauer, *COMs: Complexes of oriented matroids.* Journal of Combinatorial Theory, Series A

[12] A. Varchenko, *Bilinear form of real configuration of hyperplanes.* Adv. Math. 97(1) (1993), 110–144

[13] P.M. Winkler, *Isometric embedding in products of complete graphs.* Discrete Applied Mathematics 7 (2) (1984), 221–225

[14] D.Z. Djokovic, *Distance-preserving subgraphs of hypercubes.* Journal of Combinatorial Theory, Series B 14 7 (1973), 263–267