ALGORITHMS FOR FINDING TUCKER PATTERNS

by

Maria Tamayo

B. Sc., Universidad de los Andes (Bogota), 2010

THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN THE

DEPARTMENT OF MATHEMATICS

FACULTY OF SCIENCE

SIMON FRASER UNIVERSITY
Spring 2013

APPROVAL

Name:	Maria Tamayo
Degree:	Master of Science
Title of thesis:	Algorithms for finding Tucker patterns
Examining Committee:	Dr. Karen Yeats Assistant Professor (Chair)
	Dr. Cedric Chauve (Senior supervisor) Associate Professor
	Dr. Tamon Stephen (Supervisor) Associate Professor
	Dr. Jonathan Jedwab (Examiner) Professor
Date Approved:	January 22 , 2013

Abstract

The Consecutive-Ones Property (C1P) in binary matrices is a combinatorial concept with applications in several area, from graph planarity testing to computational biology. Tucker patterns are families of submatrices that characterize non-C1P matrices, and thus represent natural certificates of non-C1P matrices. However, there are very few algorithmic results regarding Tucker patterns. In the present work, that is part of a systematic study of Tucker patterns, we present several algorithmic and structural results about Tucker patterns in binary matrices that do not satisfy the C1P. The results obtained are the following:

- An output-sensitive enumeration algorithm for Tucker patterns.
- A detailed study of the link between partition refinement and Tucker patterns.

To my parents, my sister and my husband!

Acknowledgments

I would like to thank my senior supervisor Dr. Cedric Chauve who gave me the opportunity to discover two new worlds: discrete mathematics and computational biology. His example and wisdom, were always encouraging and motivating. I also want to thank him for his endless corrections and suggestions, that made this thesis possible.

I also want to thank my supervisor Dr. Tamon Stephen for his guidance, knowledge and inspiration; as well as for his many corrections to this thesis.

I want to thank Brad Jones for his help with the implementation of the algorithm proposed in the last chapter, as well as for his patience and knowledge.

I would like to thank my husband for his incredible support, encouragement, guidance and enlightenment.

Also, I would like to thank my mom and my sister because I wouldn't be here if it wasn't for them. Thank you so much for your love, inspiration, company and dedication.

Finally, I want to thank all of my friends for contributing in this journey some way or another.

Contents

List of Tables

List of Figures

Part I

Background

Chapter 1

Introduction

A *binary matrix* is a matrix such that all its elements are either 0 or 1. A binary matrix has the *Consecutive Ones Property* if there exists a permutation of the columns such that all the 1's in every row are consecutive. A *valid permutation* orders the columns of a binary matrix so that all the 1's are consecutive in every row. A *consecutive ones ordering* represents an order of the columns after applying a valid permutation. Binary matrices that have the Consecutive Ones Property are called C1P matrices, and binary matrices that do not have the Consecutive Ones Property are called non-C1P matrices. The Consecutive Ones Property has many applications, where a group of objects needs to be organized consecutively. Some applications of the Consecutive Ones Property include graph theory (interval graphs, planar graphs) [22], [7], archaeology [25], radiotherapy [28] and genomics (physical mapping, paleogenomics) [32].

In this thesis, the motivation for studying the Consecutive Ones Property comes from an application in computational biology, namely ancestral genomes reconstruction. In this application, the Consecutive Ones Property gives information about the genome organization of an ancestral species.

The rest of this chapter describes the main motivation for this work, ancestral genomes reconstruction. We will end with a description of the main results and an outline for the remaining chapters.

1.1 Ancestral Genomes

The study of extinct species (Paleontology) aims at understanding the evolution of animal bodies. Since the sequencing of the human genome,[1] a genomic revolution has developed, generating the sequence of several genomes of other species. This has led to considering questions in biology from genomics. At the same time, these naturally translate into questions about genomes of extinct species [39], those leading to the field of paleogenomics [34].

Later, several techniques were developed by biologists and computational biologists to address questions in paleogenomics. Some of the techniques include cytogenetics, that rely on experimental in-vitro approaches, and bioinformatics, which relies on in-silico analysis of sequencing data [19], [21]. Recent achievements in paleogenomics have been developed such as the sequencing of the Neanderthal [16], the mammoth [18] and the bubonic plague genomes [10].

The DNA molecule degrades after a few hundred thousand years [24], which prevents sequencing from old fossils such as mammalian ancestors [14]. Nowadays, fossils are still found; for example, within the recent past, a fossil of eomaia was discovered [24]. Figure 1.1 illustrates the reconstruction of the skeleton, shape and characteristics of eomaia.

Figure 1.1: [42] Eomaia

It corresponds to an eutherian mammal dating from 125 million years ago. It is believed that eomaia was the ancestor of all placental mammals. Since the DNA molecule degrades, computational techniques are necessary to reconstruct ancient genomes. These computational techniques are based on the comparison of extant genomes, where *extant genomes* are currently existing genomes.

Some computational techniques for reconstructing ancestral genomes are well advanced, such as

[1]We will not describe in detail biology and genomic concepts, the reader can refer to [9] for more information.

computing ancient DNA sequences for single genes [27]. Other techniques are still a relatively recent problem, including ordering genes along ancestral chromosomes [12], [5]. The problem of ordering genes along a set of ancestral chromosomes can be defined in the following way: given a putative set of ancestral genes, how were they located along the chromosomes of the ancestral genome?

As mentioned above, there are several methodologies for reconstructing ancestral genomes. Some are based on parsimony in a genome evolution model [8] while others are based on methodology inspired from assembly and genome mapping techniques [38]. The former has been the subject of intense research and has lead to many interesting combinatorial problems [20]. The latter is more recent and the research behind this manuscript follows it. In the rest of this section we describe one particular methodology for reconstructing ancestral genomes that is based on the C1P [12].

The problem we are interested in is reconstructing ancestral genomes. We will see that this problem can be translated into another problem, namely, determining if a given binary matrix is C1P or not. The idea of the methodology is that, given a set of ancestral genes, we encode partial ordering information into a binary matrix, where each row describes a set of ancestral genes believed to be contiguous and each column describes a gene. Then, we ask whether the matrix has the C1P or not to use this information to find a possible order for the ancestral genome. To construct this matrix, the phylogenetic tree is used as described below.

The methodology takes as input a set of extant genomes with a phylogenetic tree describing their evolutionary relationship.

Figure 1.2 represents the phylogenetic tree of several species (this example is purely illustrative and does not represent real data). The node with the picture of eomaia represents the ancestor of all mammals of the tree. A *gene* is a sequence of nucleic acids and a *marker* is a gene with a known location on a chromosome. Each genome in the phylogenetic tree is represented as a set of sequences of markers. Each sequence represents the markers order along a single chromosome. Markers that passed down from the ancestor to the descendants are called *homologous markers*. Each marker belongs to a family of homologous markers identified with a unique label. Then, each chromosome is a sequence of labels, that represent markers. The alphabet of markers is denoted as $\mathcal{L} = \{1, \ldots, n\}$, where n is the number of labels [12]. In Figure 1.2, each marker is represented by a coloured label and each genome is represented as a set of sequences of coloured labels. In this case, the alphabet of markers is the list of different colors.

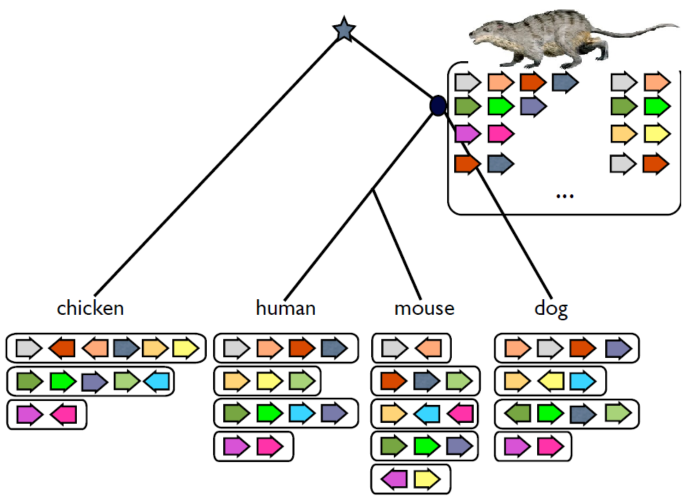
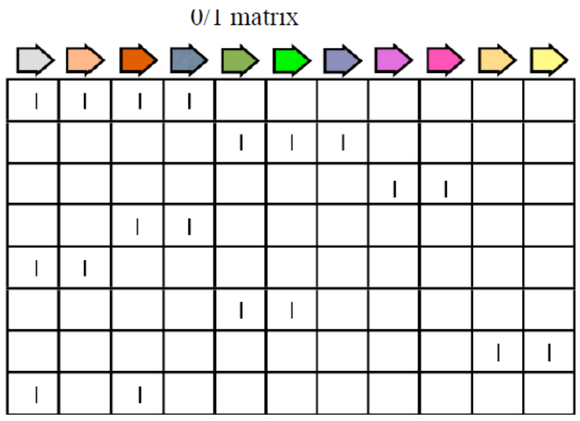


Figure 1.2: [42]Phylogenetic tree of chicken, human, mouse and dog. The ancestor that is being reconstructed is Eomaia. Each genome is represented by a set of sequences of markers, where each marker is represented by a coloured label.

The methodology for reconstructing ancestral genomes consists of two parts. First, the genomes of the extant species are used to construct *ancestral syntenies*, which are sets of markers that were possibly contiguous in the ancestor. Once the ancestral syntenies are constructed it is necessary to arrange them into a set of totally ordered subsets (that is, arrange the ancestral syntenies into sequences), that will represent one of the possible orders of the genome.

The rule that is followed to construct ancestral syntenies is the following: if a subset of the alphabet of marker labels is contiguous in at least two extant species, whose path on the phylogenetic tree goes through the ancestral node that is being reconstructed, then that set of markers could be a set of contiguous markers in the ancestral genome [12]. Then, a binary matrix M is constructed with the ancestral syntenies, where every column represents a marker and every row represents a set of contiguous markers, that is, an ancestral synteny. Therefore, there is a one in the matrix on position (i, j) if marker j is present in the ancestral synteny i.

The approach described to construct ancestral syntenies is based on a parsimony principle known as Dollo parsimony [1], that assumes that a complex character (such as a set of conserved co-localized genes) is unlikely to be gained independently. This principle is moreover backed up by theoretical study about the probability of observing groups of conserved elements in random permutations [1].



1	1	1	1							
				1	1	1				
							1	1		
		1	1							
1	1									
				1	1					
									1	1
1		1								

Figure 1.3: [42] Binary matrix$M$ constructed from the phylogenetic tree of Figure 1.2. Every column represents a marker and every row represents a set of contiguous markers.

Figure 1.3 represents the binary matrix constructed using the methodology described above for the phylogenetic tree of Figure 1.2. The sequence of markers with labels grey, orange, red and blue is an ancestral synteny because it is contiguous in species chicken, human and dog. The first row of the binary matrix M represents this ancestral synteny and the first four columns of M represent the markers that are present in this ancestral synteny.

After the binary matrix M is constructed, we analyze it. We use the C1P to organize the ancestral syntenies into possible orders of the genome. If M is C1P, then every consecutive ones ordering represents a possible order for the ancestral genome. However, if M is C1P it does not imply that all the ancestral syntenies are correct, that is, that all the sets of genes that are believed to be contiguous in the ancestor were actually contiguous. But if all the ancestral syntenies are correct, then there is a consecutive ones ordering of the markers. So, if M is non-C1P then there is at least one ancestral synteny that is not correct. Typically, when M is constructed with real data it is usually non-C1P.

This leads to the problem of transforming M into a matrix M' that is C1P. We want to construct M' so that it is as close as possible to M, in order to lose the minimum information possible. Also, because we want to understand exactly why M is non-CIP. We want to identify the rows and columns that cause M to be non-C1P. This can be done by finding all minimal obstacles that prevent M from being C1P.

In 1976, Tucker defined the minimal obstacles that prevent a binary matrix for being C1P as Tucker patterns [43]. Therefore, in order to reconstruct ancestral genomes, we want to analyze the properties of binary matrices. In particular, we want to determine if a given binary matrix is C1P or not, and in the case where is non-C1P we want to transform it into a matrix that is C1P. Therefore, finding Tucker patterns can be seen as part of a general methodology for reconstructing ancestral genomes.

1.2 Results and Plan

We use several basic concepts of Graph Theory throughout the manuscript. The reader can refer to [44] for classical notions of Graph Theory.

The remaining chapters are organized as follows: in section 2.1 we introduce binary matrices and bipartite graphs, we give a formal definition of the C1P and show the relationship between binary matrices and bipartite graphs. We follow by introducing the overlap graph and how it is related to testing the C1P. We end this section by introducing the PQ-tree and the use of it to represent all valid permutations of a C1P matrix.

In section 2.2 we discuss algorithms for deciding the C1P. We describe the notion of certifying algorithms introduced by McConnell [31]. Then, we describe previous works on deciding if a binary matrix is C1P and show how they are or are not certifying algorithms for the non-C1P. We describe the algorithms for C1P of Fulkerson and Gross [22], and of Booth and Lueker [7]. Then, we present Tucker's theorem that shows the characterization of C1P in terms of Tucker patterns, and how this turns into a certificate for the non-C1P. We follow with McConnell's certificate for the non-C1P, the incompatibility graph. Finally, we describe partition refinement, a general algorithmic technique that applies to deciding the C1P.

In chapter 3 we describe our results for partition refinement, and show how to use partition refinement for finding a certificate for the non-C1P. Also, we present an algorithm for finding a Tucker pattern of type I in a binary matrix in quadratic time.

In chapter 4 we describe our results for Tucker patterns. We describe how to find all Tucker patterns in a binary matrix in polynomial time in the size of the output. Finally, we present the results of analyzing real data with the algorithm presented above to a binary matrix that resulted from reconstructing the ancestral genome of a set of several mammalian amniotes species (mammalians, marsupials and avian species).

We analyzed the data from the set of genomes of the phylogenetic tree, consisting of the following species: Homo sapiens (humans), Pan troglodytes (chimpanzee), Pongo pygmaeus (orangutan), Macaca mulatta (monkey), Mus musculus (mouse), Rattus norvegicus (rat), Equus caballus (horse), Canis familiaris (dog), Bos taurus (cows), Monodelphis domestica (opossum), Gallus gallus (chicken), and Taeniopygia guttata (zebra finch). The results obtained are described in section 4.2.

Chapter 2

Background on the C1P

This chapter consists of two sections. The first section defines the Consecutive Ones Property and other related concepts such as binary matrices and overlap graphs. The second section presents several algorithms for deciding the C1P and certificates for matrices that do not satisfy the C1P, their complexities and their relationship with certifying algorithms.

2.1 The C1P and related concepts

We begin this section by defining binary matrices and bipartite graphs. We follow with a formal definition of the Consecutive Ones Property and overlap graphs, and show that connected components of these graphs are independent regarding the C1P. We finish by introducing PQ-trees and show that they can be used to represent all possible consecutive ones orderings of a C1P matrix.

2.1.1 Binary Matrices and Bipartite Graphs

Definition 2.1.1. Let m and n be two positive integers. An $m \times n$ *binary matrix* M is a matrix with m rows and n columns, such that all its entries are equal to 0 or 1. The *size* of a binary matrix M is the number of entries 1 in M, that is, e.

Notation 2.1.2. Let M be a $m \times n$ binary matrix. We denote by $M_{i,j}$ the value of the entry of M in row i and column j.

Notation 2.1.3. Let M be a $m \times n$ binary matrix. We denote by $R_M = \{r_1, r_2, \ldots, r_m\}$ the set of rows of M and by $C_M = \{c_1, c_2, \ldots, c_n\}$ the set of columns of M.

$M = (R_M, C_M)$ denotes the matrix with set of rows $R_M = \{r_1, r_2, \ldots, r_m\}$ and set of columns $C_M = \{c_1, c_2, \ldots, c_n\}$.

Each row r_i can be represented by a subset of $\{1, \ldots, n\}$ describing the indices of the columns c_j such that $M_{i,j} = 1$. From now on we assume that rows are represented in such a way.

Example 2.1.4. Let M be the binary matrix defined as

$$\begin{array}{c|cccc|} & c_1 & c_2 & c_3 & c_4 \\ \cline{2-5} r_1 & 1 & 1 & 0 & 0 \\ r_2 & 0 & 1 & 0 & 1 \\ r_3 & 1 & 0 & 1 & 1 \\ \cline{2-5} \end{array}$$

Then $R_M = \{r_1, r_2, r_3\} = \{\{c_1, c_2\}, \{c_2, c_4\}, \{c_1, c_3, c_4\}\}$ and $C_M = \{c_1, c_2, c_3, c_4\}$. Also, the rows of M can be represented as $R_M = \{r_1, r_2, r_3\} = \{\{1, 2\}, \{2, 4\}, \{1, 3, 4\}\}$

Definition 2.1.5. The *degree* of a binary matrix is the maximum number of entries equal to 1 in a row, denoted as Δ.

Definition 2.1.6. Let M be a binary matrix and r a row of M (respectively, column c). We denote by $M - r$ (respectively $M - c$) the matrix obtained by *removing* a row (respectively column) from M.

Definition 2.1.7. A *submatrix* of a binary matrix M is defined by a subset of its rows and/or of its columns.

Example 2.1.8. The matrix M' defined by $r_1 = \{1, 2\}, r_2 = \{2\}, r_3 = \{1, 3\}$ is a submatrix of the matrix M of example 2.1.4.

Definition 2.1.9. Let M be a binary matrix. The graph G_M is the *bipartite graph* with vertex set $\{R_M \cup C_M\}$ and edge set $\{(r_i, c_j) : j \in r_i\}$. That is, M is the adjacency matrix of G_M.

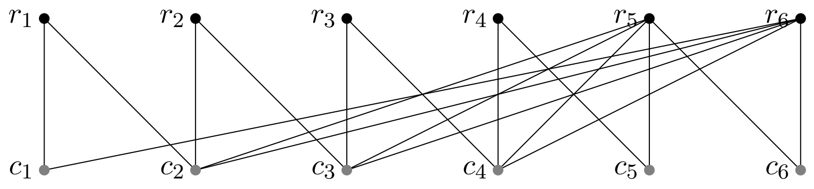
Note 2.1.10. Formally we cannot really say that M is the adjacency matrix of G_M because it should be of size $(m + n) \times (m + n)$. But it is a perfectly valid "abuse".

Notation 2.1.11. When we draw a bipartite graph, we will represent row vertices as black vertices and column vertices as grey vertices.

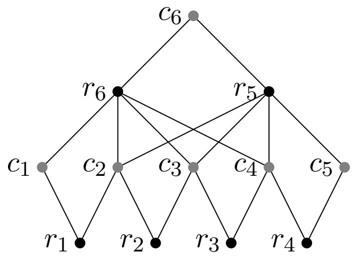
Example 2.1.12. Let M be the following binary matrix of size 6×6 and 18 entries equal to 1

$$\begin{array}{c|cccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \\ \hline
r_1 & 1 & 1 & 0 & 0 & 0 & 0 \\
r_2 & 0 & 1 & 1 & 0 & 0 & 0 \\
r_3 & 0 & 0 & 1 & 1 & 0 & 0 \\
r_4 & 0 & 0 & 0 & 1 & 1 & 0 \\
r_5 & 0 & 1 & 1 & 1 & 1 & 1 \\
r_6 & 1 & 1 & 1 & 1 & 0 & 1 \\ \hline
\end{array}$$

Then, the corresponding bipartite graph G_M of M is the following graph with 12 vertices and 18 edges



Rearranging the position of the vertices in the graph, we obtain the following drawing



Definition 2.1.13. Let M and N be binary matrices. M is *equivalent* to N if and only if M and N encode the same bipartite graph.

Example 2.1.14. Let M be the binary matrix defined by $r_1 = \{1,2\}, r_2 = \{1,3,4\}$ and $r_3 = \{1,2,3\}$. Then, M is equivalent to the binary matrix N defined by $r_1 = \{2,1\}, r_2 = \{1,3,4\}, r_3 = \{2,1,3\}$, since $G_M = G_N$.

2.1.2 The Consecutive Ones Property

The Consecutive Ones Property was introduced by Fulkerson and Gross in 1965 [22]. The motivation for studying the Consecutive Ones Property comes from deciding whether a given matrix encodes an interval graph or not. This problem itself was motivated by a biological application regarding the structure of genes [3]. We will illustrate the application with an example, taken from [3].

Consider a perfect tape recording of a piece of music. An alteration in the tape (blemished part), such as an unintentional blank space damages the tape (mutation). Given two mutant tapes it is possible to produce a perfect tape by recombination of the two tapes, only if the mutations do not intersect. If the mutations do not intersect, it is possible to replace the segment of the mutation on a tape by cutting and pasting the same segment from the other tape. Given several mutant tapes, it is possible to compare them pairwise to see if they can be recombined to form a perfect tape. Then, this information can be stored in a binary matrix, where every row and every column represents a mutant version of the tape. There is a 1 in the matrix in position (i, j) if mutant i and mutant j can be recombined to form a perfect tape, that is, if the blemished segments of mutant i and j do not intersect. Then, determining if the information is compatible with a linear order is equivalent to deciding if the constructed binary matrix is C1P.

In general, the application mentioned above looks for mutants of a portion of the genetic structure of a virus, where mutants are new genetic characters arising from a change in the DNA of the organism. The goal of this application is the following: given a large number of mutants as well as information for when the blemished portions of pairs of mutants intersect, to determine if the information is compatible with a linear order of the gene or not. The blemished portion of a mutant is a defect that is not found on the original organism.

Note 2.1.15. Without loss of generality, we assume that for the rest of this document the binary matrices that are considered do not have identical rows or columns, that they have degree at least 2 (every row has at least two entries equal to 1), and at least one entry equal to 1 per column.

Below we give a formal definition of the Consecutive Ones Property.

Definition 2.1.16. An $m \times n$ binary matrix $M = (R_M, C_M)$ has the *Consecutive Ones Property* (*C1P*) if there exists a permutation of its columns such that after ordering the columns according to this permutation, all the entries equal to 1 in each row are consecutive. Binary matrices that have the C1P are called C1P matrices, and binary matrices that do not have the C1P are called non-C1P matrices.

Example 2.1.17. The following matrix is C1P

$$\begin{array}{c|cccc|} & c_1 & c_2 & c_3 & c_4 \\ \cline{2-5} r_1 & 1 & 1 & 0 & 0 \\ r_2 & 0 & 1 & 1 & 0 \\ r_3 & 1 & 1 & 1 & 0 \\ \cline{2-5} \end{array}$$

as we can see that all the ones are consecutive in every row.

Example 2.1.18. The following matrix is C1P

$$\begin{array}{c|cccc|} & c_1 & c_2 & c_3 & c_4 \\ \cline{2-5} r_1 & 1 & 1 & 0 & 0 \\ r_2 & 1 & 0 & 1 & 1 \\ r_3 & 1 & 1 & 1 & 0 \\ \cline{2-5} \end{array}$$

as we can see that if we exchange columns c_1 and c_2,

$$\begin{array}{c|cccc|} & c_2 & c_1 & c_3 & c_4 \\ \cline{2-5} r_1 & 1 & 1 & 0 & 0 \\ r_2 & 0 & 1 & 1 & 1 \\ r_3 & 1 & 1 & 1 & 0 \\ \cline{2-5} \end{array}$$

then all the ones are consecutive in every row.

Example 2.1.19. The following matrix is non-C1P

$$\begin{array}{c|ccc|} & c_1 & c_2 & c_3 \\ \cline{2-4} r_1 & 1 & 1 & 0 \\ r_2 & 0 & 1 & 1 \\ r_3 & 1 & 0 & 1 \\ \cline{2-4} \end{array}$$

as none of the possible orders of the columns produces all the ones consecutive in every row.

Definition 2.1.20. A permutation π of the columns of M is *valid* for M if after ordering the elements of the columns of M according to this permutation, the 1s in each row are consecutive. We denote by M_π the matrix M after ordering the columns of M according to π.

Example 2.1.21. A valid permutation π for the matrix M of example 2.1.18 is $2, 1, 3, 4$ since all the 1's are consecutive after ordering the columns according to this permutation.

Definition 2.1.22. Let M be a binary matrix and π a permutation of the columns of M. A *gap* in M_π is a sequence of consecutive zeros that separates two sequences of consecutive ones in a row. The *size of the gap* is the number of consecutive zeros that separates the two sequences of consecutive ones in a row.

Example 2.1.23. In example 2.1.19 there is a gap of size 1 in row r_3, that separates the ones of that row.

Definition 2.1.24. Let M be a binary matrix. A valid permutation $(c_1, \ldots, c_n)$ defines a *consecutive ones relation* $R = \{(c_i, c_j) \mid i < j\}$.

A consecutive ones relation allow us to compare elements in a valid permutation. Then, the pair (c_i, c_j) means that c_i is to the left of c_j in the valid permutation.

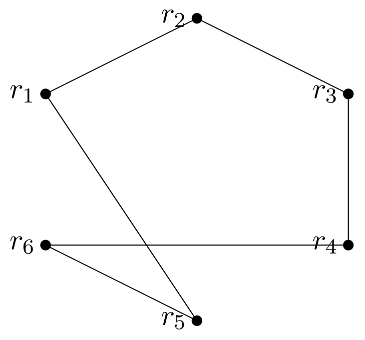
2.1.3 Overlap Graph

Definition 2.1.25. Two sets r_i and r_j are said to *overlap* if $r_i \cap r_j \neq \emptyset,\ r_i \not\subseteq r_j,\ r_j \not\subseteq r_i$.

The notion of overlap defines a relation on the rows of a binary matrix, and therefore a graph.

Definition 2.1.26. Let $M = (R_M, C_M)$ be a binary matrix. The *overlap graph* O_M has vertex set R_M and edge set $\{\{r_i, r_j\} : r_i \text{ and } r_j \text{ overlap}\}$.

Example 2.1.27. Consider the binary matrix M from example 2.1.12. The overlap graph $O(M)$ is the following graph

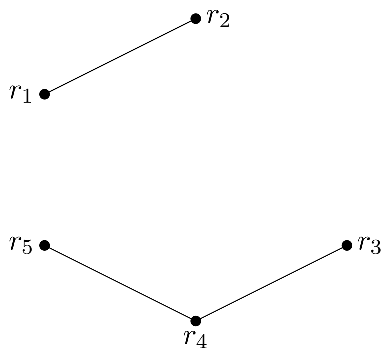


with vertices $\{r_1, r_2, r_3, r_4, r_5, r_6\}$ and edges the rows that overlap in M. For example,since r_1 overlaps r_2 and r_5 (only) in M, then r_1 is adjacent to r_2 and r_5 (only) in $O(M)$.

Example 2.1.28. Let M be the binary matrix defined by

$$\begin{array}{c|ccccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\
\hline
r_1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
r_2 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
r_3 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
r_4 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
r_5 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\hline
\end{array}$$

Consider the overlap graph $O(M)$ of M.



Then, we can see that the graph $O(M)$ has two connected components, namely $C_1 = \{r_1, r_2\}$ and $C_2 = \{r_3, r_4, r_5\}$.

Definition 2.1.29. The *support* of a set of rows $\{r_1, r_2, \ldots, r_n\}$ is the union of the columns where at least one of the rows has a 1.

Each connected component naturally defines a submatrix of M, the submatrix composed of the rows forming the component and their support. The overlap graph $O(M)$ of a binary matrix M can be useful for deciding if M is C1P. Indeed, each connected component of the overlap graph can be analyzed independently, and if every connected component is C1P, then M is C1P. This will be shown in detail in section 2.2.6.

2.2 Algorithms and Certificates for Deciding the C1P

In this section we discuss the problem of deciding if a binary matrix has the Consecutive Ones Property. We present several algorithms that have been proposed by different authors to decide if a matrix is C1P. While doing so, we follow two different paths: the first one, how to decide if a matrix is C1P; the second one, if a matrix is non-C1P to give a certificate that verifies that the answer is correct. Also, we address a related problem: if a given binary matrix is claimed to be non-C1P, how to verify that this answer is indeed correct.

We start the section by introducing certificates. Then, we show an algorithm to decide if a matrix is C1P, given by Fulkerson and Gross [22]. After, we introduce asteroidal triples and Tucker patterns. Then, we introduce the incompatibility graph and show how it is used to decide if a matrix is C1P. We follow by introducing partition refinement and how to construct PQ-trees and PQR-trees.

2.2.1 Certificates

A *certificate* is a witness that can be used to check if the solution to an algorithm is correct. A *certifying algorithm* is an algorithm that produces, with each output, a certificate or witness that the particular output is correct [31]. The concept of certifying algorithm is important, for example when dealing with complex algorithms whose implementation is hard and prone to contain bugs; the case of deciding the C1P is such an example that we will discuss below. Certificates should be able to be checked easily, that is, with a simple program, whose implementation is straightforward; and efficiently, that is, within the time and space of the initial algorithm.

An example of a certifying algorithm is the extended Euclidean algorithm for deciding the greatest common divisor of two numbers a and b. The algorithm returns $g = GCD(a,b)$ as well as a certificate: two integers x and y such that $g = ax + by$. This is a certifying algorithm since to check the correctness of the algorithm it suffices to verify that $g = ax + by$ and that g divides both a and b.

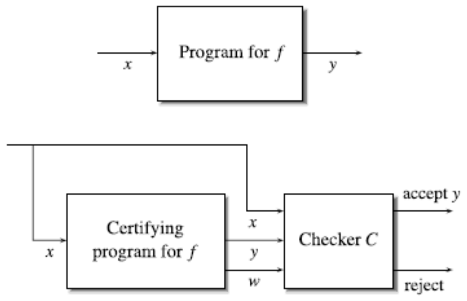


Figure 2.1: [30] The top diagram is a black box program, the bottom diagram is a certifying program

Figure 2.1 illustrates the difference between a standard and a certifying algorithm. A standard algorithm inputs x for an algorithm for computing f and outputs $y = f(x)$. There is no way to know that y is correct and that it corresponds to $f(x)$. One has to trust the algorithm. In contrast, a certifying algorithm inputs x for an algorithm for computing f and outputs y and the certificate w. Then, one can check that w proves that y is the correct result for $f(x)$.

We now consider certifying algorithms for the C1P. If a matrix is C1P, a certificate for the C1P is a valid permutation. If a matrix is non-C1P, there are no natural or intuitive certificates for the non-C1P that have been used efficiently. The first certifying algorithm for the non-C1P was introduced by McConnell in 2004 [30], and is presented in section 2.2.5.

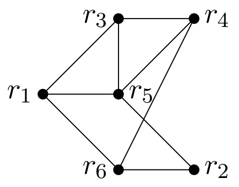
2.2.2 The Early Approaches

The first algorithm for deciding the C1P was introduced by Fulkerson and Gross in 1965 [22]. As mentioned before, Fulkerson and Gross introduced the C1P and proposed a polynomial time algorithm for deciding if a matrix is C1P. The algorithm proposed by Fulkerson and Gross attempts to build a permuted matrix by impossing inner product requirements [22].

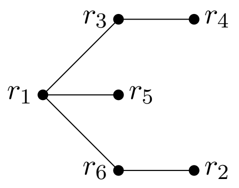
The algorithm proceeds as follows: let M be a binary matrix. Find the connected components of M and for each connected component find a spanning subtree. For each spanning subtree, apply a configuration building process that relies on calculating inner products. We illustrate this configuration building process with an example from [22]. Let M be the binary matrix

$$\begin{array}{c|cccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \\ \hline
r_1 & 1 & 0 & 1 & 0 & 1 & 1 \\
r_2 & 0 & 0 & 1 & 0 & 1 & 0 \\
r_3 & 0 & 1 & 1 & 0 & 1 & 0 \\
r_4 & 1 & 0 & 1 & 0 & 1 & 0 \\
r_5 & 1 & 0 & 0 & 1 & 1 & 1 \\
r_6 & 0 & 1 & 1 & 0 & 0 & 0 \\ \hline
\end{array}$$

with overlap graph



Consider the spanning subtree of the overlap graph



chosen arbitrarily from the overlap graph.

We will consider triples of rows r_i, r_j, r_k, for $i, j, k \in \{1, \ldots, 6\}$, such that r_i overlaps r_j and r_j overlaps r_k. Consider the triple r_3, r_1, r_5. Since M is a binary matrix, the inner product of r_1 and r_5 is $r_1 \cdot r_5 = \sum_{i=1}^{n} M_{1i}M_{5i}$ and represents the number of columns where r_1 and r_5 both contain a 1.

There is a permutation of the columns of M where either $r_1 \cdot r_5 < \min(r_1 \cdot r_3, r_3 \cdot r_5)$ or $r_1 \cdot r_5 \geq \min(r_1 \cdot r_3, r_3 \cdot r_5)$, with respective configuration C_1 or C_2, illustrated in the figures below.

$$\begin{array}{ccccccccc} & c_1 & c_6 & c_3 & c_5 & c_2 & c_4 & & \\ r_1 & 1 & 1 & 1 & 1 & & & & \\ r_3 & & & 1 & 1 & 1 & & & \\ r_5 & & & & & 1 & 1 & 1 & 1 \end{array}$$

Configuration C_1

$$\begin{array}{ccccccc} & c_4 & c_1 & c_6 & c_5 & c_3 & c_2 \\ r_1 & & 1 & 1 & 1 & 1 & \\ r_3 & & & & 1 & 1 & 1 \\ r_5 & 1 & 1 & 1 & 1 & & \end{array}$$

Configuration C_2

Configuration C_1 is built by writing $r_1 \cdot r_1$ 1's consecutively on a first row, then $r_3 \cdot r_3$ 1's consecutively on a second row such that they overlap the first row by $r_1 \cdot r_3$ 1's; and finally, writing $r_5 \cdot r_5$ 1's consecutively on a third row so that they overlap the second row by $r_3 \cdot r_5$ 1's. Configurations C_1 and C_2 differ only in the position of the third row with respect to the second row. On the left hand side of the configuration we write the row label r_i of the row that has $r_i \cdot r_i$ 1's consecutively. On top of each configuration we write the column labels of the columns whose entry is 1 on each row r_i.

To build the permuted matrix, at each step we consider a triple and obtain one of the configurations C_1 or C_2. Then, we check the inner product of the first and third rows in the configuration. If the inner product does not correspond to the one on M, then M is not C1P.

We start with the triple r_1, r_3, r_5 and build the configuration C2. We choose the configuration C_2 instead of C_1 because $r_1 \cdot r_5 \geq \min(r_1 \cdot r_3, r_3 \cdot r_5)$, which corresponds to C_2. The inner products of this configuration agree with the ones from M. Next, we process r_6, relative to r_1 and r_5, leading the configuration

$$\begin{array}{ccccccc} & c_4 & c_1 & c_6 & c_5 & c_3 & c_2 \\ r_1 & & 1 & 1 & 1 & 1 & \\ r_3 & & & & 1 & 1 & 1 \\ r_5 & 1 & 1 & 1 & 1 & & \\ r_6 & & & & & 1 & 1 \end{array}$$

The inner products of this configuration agree with the ones from M. Next, we process r_4, relative to r_1 and r_3, leading to the configuration

$$\begin{array}{ccccccc}
 & c_4 & c_1 & c_6 & c_5 & c_3 & c_2 \\
r_1 & & 1 & 1 & 1 & 1 & \\
r_3 & & & & 1 & 1 & 1 \\
r_5 & 1 & 1 & 1 & 1 & & \\
r_6 & & & & & 1 & 1 \\
r_4 & & & 1 & 1 & 1 &
\end{array}$$

Since the inner products of this configuration agree with the ones from M, we finally process r_2, relative to r_1 and r_6, leading the configuration

$$\begin{array}{ccccccc}
 & c_4 & c_1 & c_6 & c_5 & c_3 & c_2 \\
r_1 & & 1 & 1 & 1 & 1 & \\
r_3 & & & & 1 & 1 & 1 \\
r_5 & 1 & 1 & 1 & 1 & & \\
r_6 & & & & & 1 & 1 \\
r_4 & & & 1 & 1 & 1 & \\
r_2 & & & & 1 & 1 &
\end{array}$$

The matrix obtained from this configuration by filling with 0's the empty entries is a valid permutation of M that is C1P.

$$\begin{array}{c|cccccc|}
 & c_4 & c_1 & c_6 & c_5 & c_3 & c_2 \\
\hline
r_1 & 0 & 1 & 1 & 1 & 1 & 0 \\
r_2 & 0 & 0 & 0 & 1 & 1 & 1 \\
r_3 & 1 & 1 & 1 & 1 & 0 & 0 \\
r_4 & 0 & 0 & 0 & 0 & 1 & 1 \\
r_5 & 0 & 0 & 1 & 1 & 1 & 0 \\
r_6 & 0 & 0 & 0 & 1 & 1 & 0 \\
\hline
\end{array}$$

An upper bound for the number of inner products needed is given by $O(n^2)$, where n is the number of rows of M. Then, one can fit the components together using the partial order of the component graph [22].

2.2.3 Tucker Patterns and Asteroidal Triples

Tucker characterized C1P matrices in terms of five forbidden submatrices, that prevent a binary matrix from having the C1P. However, this characterization does not translate directly into an efficient algorithm for deciding the C1P.

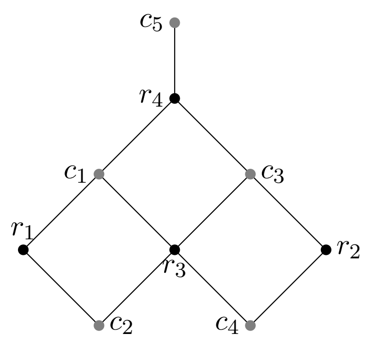
We start this subsection by giving a definition of asteroidal triples and Tucker patterns. Then, we show how Tucker patterns and asteroidal triples are used for deciding the C1P.

Definition 2.2.1. Let M be a binary matrix and G_M be the corresponding bipartite graph. Recall that G_M has vertices $G_M = (R_M, C_M)$. The *closed neighborhood* of a vertex $v \in G_M$ is defined as $N(v) = \{v\} \cup \{x \in G_M$: there is an edge between x and $v\}$.

Example 2.2.2. Let M be the following binary matrix

$$\begin{array}{c|ccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 \\
\hline
r_1 & 1 & 1 & 0 & 0 & 0 \\
r_2 & 0 & 0 & 1 & 1 & 0 \\
r_3 & 1 & 1 & 1 & 1 & 0 \\
r_4 & 1 & 0 & 1 & 0 & 1 \\
\hline
\end{array}$$

Then, the corresponding bipartite graph G_M is the following graph with 9 vertices and 11 edges



From G_M we can see that $N(c_1) = \{c_1, r_1, r_3, r_4\}$, $N(c_2) = \{c_2, r_1, r_3\}$, $N(c_3) = \{c_3, r_2, r_3, r_4\}$, $N(c_4) = \{c_4, r_2, r_3\}$ and $N(c_5) = \{c_5, r_4\}$.

Definition 2.2.3. Let M be a binary matrix with corresponding bipartite graph G_M. Three vertices $u, v, w \in C_M$ form an *C_M-asteroidal triple* (C_M-AT) if between any two of them there exists a path in G_M that does not contain a vertex from the closed neighborhood of the third vertex.

From now we assume that when using the terminology *asteroidal triple* it refers to C_M-AT, unless otherwise specified.

Example 2.2.4. Consider the binary matrix with corresponding bipartite graph from example 2.2.2. Then, the column vertices c_2, c_4, c_5 form an asteroidal triple.

Definition 2.2.5. A *filled* C_M-AT c, c', c'' is a C_M-AT together with a path between each pair of vertices of $\{c, c', c''\}$ that avoids the neighborhood of the third one.

Asteroidal triples are useful for deciding if a matrix is or is not C1P. Tucker showed that if a binary matrix has the Consecutive Ones Property, then this is equivalent to having a bipartite graph with no asteroidal triples.

Definition 2.2.6. Let M be a binary matrix with corresponding bipartite graph G_M, and $x, y \in V(G_M)$. We define the *distance* between x and y, $d(x, y)$ to be the length of the shortest path from x to y. We define the *C_M-diameter* of G_M as $\delta(G_M) = \sup\limits_{x,y \in C_M} d(x, y)$. If $x, y \in C_M$ and $d(x, y) = \delta(G_M)$, then x and y are called C_M-diameter points of G_M.

Theorem 2.2.7. [43] *A matrix M is C1P if and only if the bipartite graph G_M does not contain a C_M-AT.*

Proof. (Sketch) [43] ($\Rightarrow$) Suppose M is C1P with corresponding bipartite graph G_M and $\tilde{C}$ is a valid permutation of C. By contradiction, suppose x, y, z is an asteroidal triple of G_M, such that $x < y < z$ in $\tilde{C}$. Let $P = (c_0 = x, r_1, c_1, \ldots, r_n, c_n = z)$ be a path from x to z. We will see that y is adjacent to P. Let k be the smallest i, $0 \leq i \leq n$ such that $c_{k-1} < y < c_k$. Then, $c_k, c_{k-1} \in N(r_k)$. Since $c_{k-1} < y < c_k$ and $\tilde{C}$ is a valid permutation of C, then $y \in N(r_k)$. Then y is adjacent to P. But since x, y, z is an asteroidal triple, then there is a path between x and z that does not intersect the closed neighborhood of y, which is a contradiction. Then, G_M does not contain an asteroidal triple whose three vertices correspond to columns of M.

($\Leftarrow$) Suppose M is a binary matrix whose representing bipartite graph G_M does not contain a C_M-AT. We proceed by induction on n the number of vertices of G_M.

If $n = 1, 2$ then M is C1P.

If $n \geq 3$ suppose that for every $(n-1)$-subset S of G_M, S is C1P. A C-diameter point p and a valid permutation $\tilde{C}$ of $G_M - p$ can be chosen so that y, the right end vertex of $\tilde{C}$, is adjacent to every path from p to x, the left end vertex of $\tilde{C}$. Then, the valid permutation $\tilde{C}$ can be extended to a valid permutation of G_M in the following way: let T be the set of vertices in C which can be reached from x by a path not adjacent to y. Then, $p, y \notin T$. Let u be the rightmost vertex of T in $\tilde{C}$. Let $\tilde{C_1}$ be a valid permutation of $G_M - x$ and let v be the leftmost vertex of $G_M - x$ in $\tilde{C_1}$. Construct a new valid permutation $\tilde{C_2}$ by removing the vertices of T from $\tilde{C_1}$ and placing them with x to the left of v, in the order they had in $\tilde{C}$. Then, for $c \in \tilde{C_2}$, there are three options: $N(c) \subseteq T$, $N(c) \subseteq (\tilde{C} - T)$ or $N(c) \cap T \neq \emptyset$ and $N(c) \cap (\tilde{C} - T) \neq \emptyset$. We will see that in all of these cases $N(c)$ is consecutive in $\tilde{C_2}$. If $N(c) \subseteq T$, then $N(c)$ is consecutive in $\tilde{C_2}$ because $N(c)$ was consecutive in $\tilde{C}$. If $N(c) \subseteq (\tilde{C} - T)$, then $N(c)$ is consecutive in $\tilde{C_2}$ because $\tilde{C}$ was consecutive in $\tilde{C_1}$. If $N(c) \cap T \neq \emptyset$ and $N(c) \cap (\tilde{C} - T) \neq \emptyset$, then $N(c)$ is consecutive if $u, v \in N(c)$. Then, it remains to prove that $u, v \in N(c)$ for $c \in \tilde{C_2}$. If $N(c) \cap T \neq \emptyset$ and $N(c) \cap (\tilde{C} - T) \neq \emptyset$, then since $N(c) \cap (\tilde{C} - T) \neq \emptyset$, then $y \in N(c)$. Since v is the leftmost vertex in $\tilde{C} - T$, then $v < y$. If $N(c) \cap T \neq \emptyset$, then for $w \in T$, $w < u$. Then, we have $w < u < v < y$. Then, $w, y \in N(c)$ implies $u, v \in N(c)$. Then, $\tilde{C_2}$ is a valid permutation of C. $\square$

The rest of this subsection introduces Tucker patterns and shows the equivalence between matrices with the Consecutive Ones Property and matrices without Tucker patterns.

Figure 2.2 shows the matrices of the five Tucker patterns $M_{I_k}, M_{II_k}, M_{III_k}$, for $k \geq 1$, M_{IV}, and M_V. Figure 2.3 shows the corresponding bipartite graphs for each type of Tucker pattern. We can see that Tucker patterns M_{I_k} give chordless cycles of length at least 6, also known as *holes*, where a chord is an edge linking to non-consecutive vertices of the cycle. Tucker pattern M_{II_k} consists of a cycle with two rows that are connected to every column except for one, depending on the row. Thus, the difference between M_{I_k} and M_{II_k} is their last two rows and rightmost column. Tucker pattern M_{III_k} consists of a cycle with a row that connects to every column and three external columns that are connected to a single row, depending on the column. Thus, the difference between M_{I_k} and M_{III_k} is their last row and rightmost column. We can also see that Tucker patterns M_{I_k}, M_{II_k} and M_{III_k} are *families* of bipartite graphs, since there are many patterns for each type, each of them of different size. Instead, Tucker patterns M_{IV} and M_V are of fixed size.

$$M_{I_k} = \begin{array}{c|ccccc|} & c_1 & c_2 & c_3 & \dots & c_{k+2} \\ \hline r_1 & 1 & 1 & 0 & \dots & 0 \\ r_2 & 0 & 1 & 1 & \dots & 0 \\ \dots & & & \dots & & \\ r_{k+1} & 0 & 0 & \dots & 1 & 1 \\ r_{k+2} & 1 & 0 & \dots & 0 & 1 \\ \hline \end{array}$$

$$M_{II_k} = \begin{array}{c|cccccc|} & c_1 & c_2 & c_3 & \dots & c_{k+2} & c_{k+3} \\ \hline r_1 & 1 & 1 & 0 & \dots & 0 & 0 \\ r_2 & 0 & 1 & 1 & \dots & 0 & 0 \\ \dots & & & & \dots & & \\ r_{k+1} & 0 & 0 & \dots & 1 & 1 & 0 \\ r_{k+2} & 0 & 1 & 1 & \dots & 1 & 1 \\ r_{k+3} & 1 & 1 & \dots & 1 & 0 & 1 \\ \hline \end{array}$$

$$M_{III_k} = \begin{array}{c|cccccc|} & c_1 & c_2 & c_3 & \dots & c_{k+2} & c_{k+3} \\ \hline r_1 & 1 & 1 & 0 & \dots & 0 & 0 \\ r_2 & 0 & 1 & 1 & \dots & 0 & 0 \\ \dots & & & & \dots & & \\ r_{k+1} & 0 & 0 & \dots & 1 & 1 & 0 \\ r_{k+2} & 0 & 1 & \dots & 1 & 0 & 1 \\ \hline \end{array}$$

$$M_{IV} = \begin{array}{c|cccccc|} & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \\ \hline r_1 & 1 & 1 & 0 & 0 & 0 & 0 \\ r_2 & 0 & 0 & 1 & 1 & 0 & 0 \\ r_3 & 0 & 0 & 0 & 0 & 1 & 1 \\ r_4 & 1 & 0 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

$$M_V = \begin{array}{c|ccccc|} & c_1 & c_2 & c_3 & c_4 & c_5 \\ \hline r_1 & 1 & 1 & 0 & 0 & 0 \\ r_2 & 0 & 0 & 1 & 1 & 0 \\ r_3 & 1 & 1 & 1 & 1 & 0 \\ r_4 & 1 & 0 & 1 & 0 & 1 \\ \hline \end{array}$$

Figure 2.2: Tucker patterns M_{I_k}, M_{II_k}, M_{III_k}, M_{IV} and M_V.

Also notice that Tucker patterns are non-C1P matrices. Tucker described Tucker patterns as the "forbidden" submatrices since any matrix that contains at least one Tucker pattern is a non-C1P matrix.

Note that in figure 2.3 the vertices x, y and z represent an asteroidal triple in every Tucker pattern.

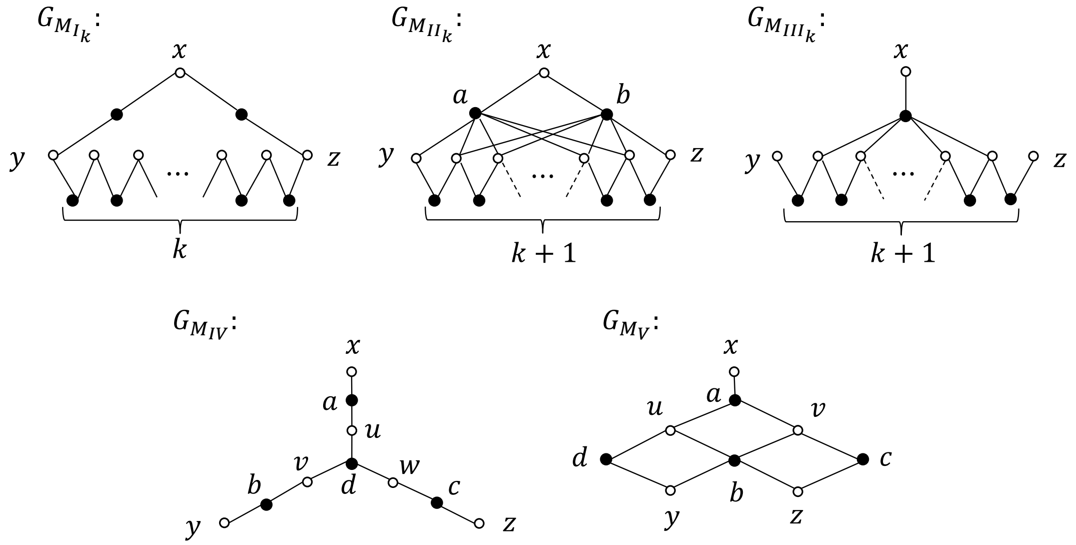


Figure 2.3: Bipartite graphs of the five Tucker patterns.

Notation 2.2.8. From now, we will use x, y, z, a, b, c for the specific vertices of Tucker patterns shown in Figure 2.3.

Notation 2.2.9. We will denote by (a, P, b) the path obtained by following the path P from vertex a to vertex b.

Theorem 2.2.10. [43] *Let M be a binary matrix. G_M does not contain a C_M-AT if and only if G_M does not contain an induced subgraph that is a Tucker pattern.*

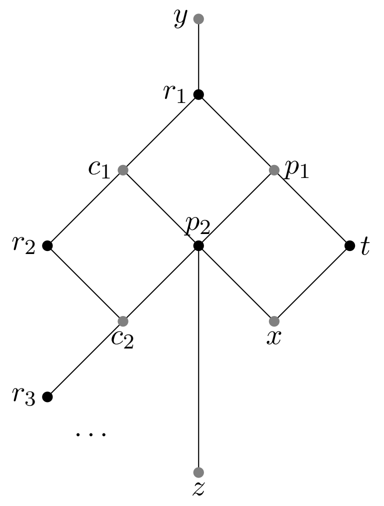


Figure 2.4: G_M contains a Tucker pattern of type V.

Proof. (Sketch) [43] ($\Rightarrow$) It is clear since Tucker patterns contain asteroidal triples.

($\Leftarrow$) Suppose G_M does not contain an induced subgraph which is a Tucker pattern. It is enough to prove the theorem for a minimal G_M, so that every proper subgraph of G_M does not contain an asteroidal triple. By contradiction, suppose x, y, x is an asteroidal triple in G_M. Let $P_{yz} = (y = c_0, r_1, c_1, r_2, c_2, \dots, r_n, c_n = z)$ be a chordless path from y to z not adjacent to x. Similarly, define P_{xy} and P_{xz}. We can assume $|P_{yz}| \geq \max\{|P_{xy}|, |P_{xz}|\}$. We can also assume that $|P_{yz}| > 2$, for otherwise $|P_{xy}| = |P_{xz}| = |P_{yz}| = 2$ and G_M would be a cycle of length 4 (G_{I_1}). Then, there are two possibilities: there is a vertex p_1 on $P_{xy} - P_{xz}$ adjacent to a vertex p_2 on $P_{xz} - P_{xy}$ or no vertex on $P_{xy} - P_{xz}$ is adjacent to a vertex on $P_{xz} - P_{xy}$. Suppose p_1 is adjacent to p_2 as stated above. We will see that G_M contains a Tucker pattern of type V as shown in figure 2.4.

Note that one of p_1, p_2 is at distance 1 from x and the other at distance 2, for otherwise there would be a Tucker pattern which is a cycle. Suppose p_2 is adjacent to x and $P_{xy} = (x, t, p_1, \dots, y)$. Also, p_2 is adjacent to z, for otherwise there would be a path from x to y not adjacent to z in $G_M - t$. Note that p_1 is adjacent to P_{yz}, or else p_1, y, z form an asteroidal triple in $G_M - x$. But, p_1 has to be adjacent to r_1, otherwise there is a path from x to z not adjacent to y in $G_M - p_2$. Then, there is a cycle $(p_1, r_1, P_{yz}, c_i, p_2)$, unless p_2 is adjacent to c_i, $0 \leq i \leq n$, where the vertices $x, t, p_1, p_2, y, r_1, c_1, r_2, c_2$ form a Tucker pattern of type V.

Now, suppose that no vertex on $P_{xy} - P_{xz}$ is adjacent to a vertex on $P_{xz} - P_{xy}$. Let w be the common vertex of the paths P_{xy} and P_{xz} that is farthest from x. Let p_1 and p_2 be the next vertices after w on P_{xy} and P_{xz}, respectively. Then, there are three possibilities: none, one or both of p_1 and p_2 are on P_{yz}. If none of p_1, p_2 is on P_{yz}, we will see that G_M is a Tucker pattern of type II_n. Note that w is not adjacent to P_{yz} because otherwise there is either a path from x to y in $G_M - p_1$ or a path from x to z in $G_M - p_2$. Suppose p_1, p_2 are adjacent to a common vertex t, for otherwise they are part of a Tucker pattern which is a cycle. Note that $t = c_k$, because if $t = r_k$ we get a contradiction, $|P_{yz}| = 2 < |P_{yz}|$. Also, p_1 is adjacent to y, for if not there is a path from x to z not adjacent to y in $G_M - p_2$. Similarly, p_2 is adjacent to z. Now, p_1 and p_2 are adjacent to consecutive c_i's because if not there is a cycle (p_1, c_j, P_{yz}, c_m), and similarly for p_2. Then, G_M is a Tucker pattern of type II. If only one of p_1 and p_2 is on P_{yz}, assume p_2 is on P_{yz}. We can assume that p_1, p_2 are adjacent to a common vertex t. Since p_2 is on P_{yz}, then $w \neq x$. Also, w is adjacent to x, if not there is a path from y to z in $G_M - t$. Since x is a column vector, then p_2 is a column vector. Then, there is a path from x to y not adjacent to z in $G_M - p_1$. If both p_1 and p_2 are on P_{yz}, then, by minimality of G_M we have $P_{xy} = (x, P_{xy}, p_1, P_{yz}, y)$ and similarly for P_{xz}. Then, there are two possibilities: $P_{yz} = (y, P_{xy}, w, P_{xz}, z)$ or not. If P_{yz} is as stated above, then G_M is a Tucker pattern of type III_1 or IV. If P_{yz} is not as stated above, then similarly to the case where only p_2 is on P_{yz}, we can assume that p_1, p_2 are adjacent to a common vertex t on P_{yz} and that w is adjacent to x. Then, p_1 and p_2 are column vertices. Then, $p_1 = q_1$ because if not x, q_1, z are an asteroidal triple in $G_M - y$. Similarly, $p_2 = q_{n-1}$. Also, w is adjacent to q_i, $1 \leq i \leq n-1$, for otherwise there is Tucker pattern which is a cycle. Therefore, G_M is a Tucker pattern of type III_n. □

Now we can state Tucker's main theorem.

Theorem 2.2.11. [43] *A matrix M is C1P if and only if the corresponding bipartite graph G_M does not contain an induced subgraph which is a Tucker pattern.*

Proof. This follows from Theorems 2.2.7 and 2.2.10. □

In terms of deciding the C1P, we can see that finding Tucker patterns M_{IV} and M_V in a binary matrix is "easy" since they are of fixed size. In principle, an exhaustive search is enough to find Tucker patterns M_{IV} and M_V, in time $O(m^4n^6)$ and $O(m^4n^5)$, respectively. Variations of exhaustive search techniques will improve the complexities. In Chapter 4, we propose one such variation of an exhaustive search technique. On the contrary, finding Tucker patterns M_{I_k}, M_{II_k} and M_{III_k} is not trivial. For instance, to find all Tucker patterns M_{I_k} we need to find all chordless cycles in a graph. One way to do so is to find all cycles in a graph and check if they are chordless or not, but finding all cycles in a graph is not an easy problem (see [37], [40], [2]). (Tucker patterns M_{II_k} and M_{III_k} are also not easy to find as they are modifications of cycles in a graph.)

We can see that asteroidal triples and Tucker patterns are natural certificates for non-C1P, but there are few algorithms for producing them [13]. Tucker's proofs are not constructive: they do not show how to construct either Tucker patterns or asteroidal triples.

2.2.4 Finding Tucker Patterns

In this section, we describe two algorithms for finding Tucker patterns in a binary matrix. The first algorithm was introduced in 2012 by Dom et al. [13] and the second algorithm was introduced in 2010 by Blin et al. [6]. The algorithm of Dom et al. [13] takes as input a binary matrix M and gives as output a submatrix M' that contains a Tucker pattern. The algorithm relies on Theorem 2.2.10 and searches for an asteroidal triple on the given matrix M. We will denote by $G_M = (V, E)$ the bipartite graph.

The idea of the algorithm of [13] is the following: for every triple of column vertices x, y and z in G_M compute the length of the shortest paths between x and y, x and z, and y and z that does not intersect the closed neighborhood of the third vertex, respectively. If all three paths exist for a triple x, y and z, then such a triple is an asteroidal triple. Choose such an asteroidal triple x, y and z such that the sum of the lengths of the paths is minimum. For such x, y and z, consider the submatrix defined by the row and column vertices that belong to the paths between x and y, x and z, and y and z. That submatrix contains an asteroidal triple and therefore is a Tucker pattern.

Note that the authors of [13] are interested in finding a Tucker pattern with minimum size (minimum number of vertices in the corresponding bipartite graph), rather than finding a minimal Tucker pattern (one that does not contain any other Tucker pattern). This notion is based on the idea of finding a Tucker pattern from an asteroidal triple where the sum of the shortest paths between every pair of vertices in the triple is minimum. The authors of [13] were interested in finding minimum Tucker patterns because their goal was to solve three applications of the C1P that involve deleting the minimum number of columns, rows and entries, respectively, to transform a non-C1P matrix into a C1P matrix. This can be achieved by finding minimum Tucker patterns and deleting them from the matrix.

Note that the submatrix returned by this algorithm does not have to be of minimum size, because there are vertices that can belong to several of the shortest paths. Then, the sum of the lengths of the paths is not necessarily the same as the number of vertices in the union of the three paths. Dom et al. proved in [13] that the algorithm returns a submatrix of minimum size only for Tucker patterns M_{I_k} and M_{II_k}. This algorithm has complexity $O(\Delta mn^2 + n^3)$, where m is the number of rows in M, n is the number of columns in M and Δ is the degree of M.

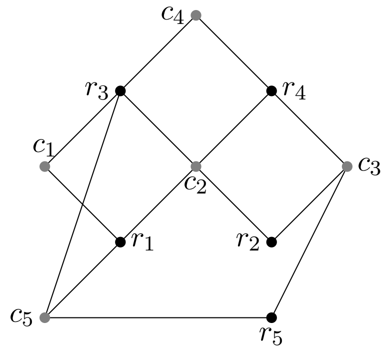
For finding Tucker pattern M_{III_k} the idea is to take advantage of the similarity between patterns M_{I_k} and M_{III_k}. As mentioned before, the bipartite graph of pattern M_{I_k} is a hole and the matrices of patterns M_{I_k} and M_{III_k} are different only in their last row and rightmost column. By complementing the rightmost column of M_{III_k} we obtain a hole with an extra column. Then, the problem of finding M_{III_k} can be reduced to the problem of finding a minimum-size hole. This algorithm runs in time $O(\Delta^3 m^3 n + \Delta^2 m^2 n^2)$. For finding Tucker patterns M_{IV} and M_V, Dom et al. propose exhaustive searches that have complexities $O(\Delta^3 m^2 n^3)$ and $O(\Delta^4 m^2 n)$, respectively.

Combining the complexities of the different algorithms, Dom et al. propose algorithms that can find a Tucker pattern of minimum size in a given binary matrix M, with m rows, n columns and at most Δ entries equal to 1 in each row, in time $O(\Delta^3 m^2 n(m + n^2))$.

Example 2.2.12. Let M be the following binary matrix

$$\begin{array}{c|ccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 \\
\hline
r_1 & 1 & 1 & 0 & 0 & 1 \\
r_2 & 0 & 1 & 1 & 0 & 1 \\
r_3 & 1 & 1 & 0 & 1 & 1 \\
r_4 & 0 & 1 & 1 & 1 & 0 \\
r_5 & 0 & 0 & 1 & 0 & 1 \\
\hline
\end{array}$$

with bipartite graph G_M



There are seven asteroidal triples in G_M: five from the cycles $\{r_1, c_2, r_2, c_3, r_5, c_5\}$, $\{r_1, c_2, r_4, c_3, r_5, c_5\}$, $\{r_2, c_2, r_3, c_3, r_5, c_5\}$, $\{r_3, c_2, r_4, c_3, r_5, c_5\}$, $\{r_3, c_3, r_4, c_4, r_5, c_5\}$ and two from the two Tucker patterns of type II, $\{r_1, r_2, r_3, r_4, c_1, c_2, c_3, c_4\}$ and $\{r_1, r_2, r_3, r_4, c_2, c_3, c_4, c_5\}$. The sum of the shortest paths between every pair of vertices for the five cycles is 9 and the same sum is 11 for the two Tucker patterns of type II. Therefore, there are five minimum Tucker patterns, namely, the five cycles.

We now illustrate the algorithms proposed by Blin et al. [6]. These algorithms take as input a binary matrix M, with m rows, n columns and at most Δ entries equal to 1 in each row, and give as output a Tucker pattern of minimum size of type M_{I_k}, M_{II_k}, M_{III_k}, M_{IV} and M_V, respectively. The algorithms rely on finding shortest paths and two graph pruning techniques that the authors call clean and anticlean. The cleaning technique of a vertex x consists in removing the neighborhood $N(x)$ of x, that is, $clean(x) = G_M(M)[V \setminus N(x)]$. The anticleaning technique of a vertex x consists in removing all vertices that are not neighbors of x and are not the same type of vertex as x (row or column vertex).

The idea of the first two algorithms for finding a Tucker pattern M_{I_k} and M_{II_k}, respectively, is the following: enumerate all possible sets of vertices $\{x, y, z, a, b\}$ (see Figure 2.3) and use the clean and anticlean techniques to remove the edges of the neighbors of x, a and b. Then, find the shortest path between y and z. These algorithms will find a Tucker pattern M_{I_k} and M_{II_k}, respectively, in time $O(m^2\Delta^3(n + \Delta m))$.

The third algorithm is similar to the first two, but differs in that it only enumerates the sets of four vertices $\{x, y, z, a\}$. Then, it follows the same approach as the other algorithms, that is, it uses the clean and anticlean techniques to remove the edges of the neighbors of x and a. Then, it finds the shortest path between y and z. This algorithm will find a Tucker pattern M_{III_k} in time $O(m\Delta n^2(n + \Delta m))$. For finding Tucker patterns M_{IV} and M_V Blin et al. propose exhaustive searches that have complexities $O(\Delta m^2 n^3)$ and $O(\Delta^4 m^2 n)$, respectively.

Combining the complexities of the different algorithms, Blin et al. propose an algorithm to find a minimum Tucker pattern on a binary matrix with m rows, n columns and at most Δ entries equal to 1 in each row, in time $O(\Delta^3 m^2(m\Delta + n^3))$.

In Chapter 4 we propose an algorithm for finding a Tucker pattern in a binary matrix that improves the complexity of the algorithms of Dom et al. and Blin et al. The proposed algorithm resembles the Blin et al. algorithm for finding Tucker patterns $G_{M_{II_k}}$ and $G_{M_{III_k}}$, and it differs mainly in that the algorithm of Blin et al. removes the edges of the neighbors of $\{x, a, b\}$ (see Figure 2.3) to find a path between y and z while the algorithm we propose splits column and row vertices adjacent to a and b into two sets, and finds a path between y and z, that alternates between vertices of these two sets. The proposed algorithm has quadratic complexity and will allow us to use Tucker patterns as a certificate for the non-C1P. However, unlike these algorithms, we do not output a Tucker pattern with a minimum number of vertices.

2.2.5 The Incompatibility Graph

In this section we define the incompatibility graph introduced by McConnell in 2004 [30], when he introduced the first certifying algorithm for deciding the C1P. Then, we describe McConnell's use of the incompatibility graph as a certificate for the non-C1P.

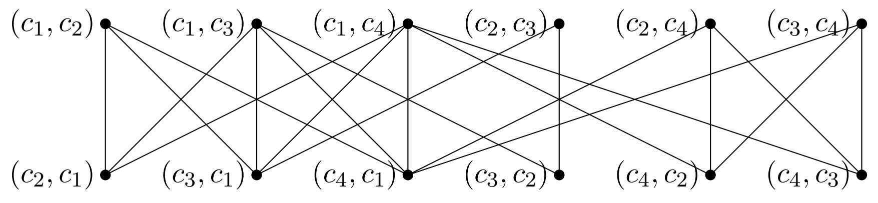
Definition 2.2.13. Let $M = (R_M, C_M)$ be an $m \times n$ binary matrix. The *incompatibility graph* of M is an undirected graph $I_M = (V, E)$, with vertex set $V = \{(c_i, c_j) \mid i, j = 1, \ldots, n, i \neq j\}$. Two vertices (c_i, c_j) and (c_j, c_k) are adjacent if at least one of the following holds:

1. $c_i = c_k$
2. There exists a row r_l in M such that $M_{li}, M_{lk} = 1$ but $M_{lj} = 0$

Example 2.2.14. Let M be the following binary matrix

$$
\begin{array}{c|cccc|}
 & c_1 & c_2 & c_3 & c_4 \\
\hline
r_1 & 1 & 1 & 0 & 0 \\
r_2 & 0 & 1 & 1 & 1 \\
r_3 & 1 & 1 & 1 & 0 \\
\hline
\end{array}
$$

The incompatibility graph I_M of M is the following graph,



whose vertices are all the possible combinations of two columns. The first type of edges connect opposite vertices, for example (c_1, c_2) and (c_2, c_1). This type of edges are incompatible in the sense that if one of them holds, then the other is not possible. For example, if (c_1, c_2) is true, then c_1 is to the left of c_2 in a valid permutation. Then, it is not possible to have c_2 to the left of c_1 on the same valid permutation. The second type of edges connect pairs whose corresponding vertices on M are not consecutive ones in a row of M. For example, (c_3, c_1) and (c_1, c_2) is an incompatible edge since c_3, c_1, c_2 are not consecutive ones in row r_2.

The edges of the incompatibility graph are incompatible edges in the sense that they can not appear at the same time in a valid permutation. If M is C1P, then G_M should not have incompatible pairs. If M is bipartite and C1P, then there is a well-defined partition which we can use to make a top-half and a bottom-half. Then, the top-half of the vertices in the incompatible graph, that represents a valid permutation, should be an independent set. The bottom-half of the vertices in the incompatibility graph represents the reverse of the valid permutation, which is also a valid permutation, and should also be an independent set. Then, if M has the Consecutive Ones Property the incompatibility graph must be bipartite. McConnell used this observation to give the following result for the non-C1P.

Theorem 2.2.15. [30] *Let M be a binary matrix. M is C1P if and only if the incompatibility graph I_M is bipartite.*

In [30], McConnell claimed a bound of $n + 2$ for the smallest odd cycle contained in the incompatibility graph of a non-C1P matrix, where n is the number of columns in the matrix. In 2011 [29], this bound was corrected to $n+3$ when n is odd.

McConnell's algorithm is a certifying algorithm for the non-C1P because it provides a certificate, which are odd cycles of the incompatibility graph. It is an efficient algorithm because McConnell also claimed that the incompatibility graph can be computed in linear time/space using partition refinement. However, McConnell does not provide all the details for finding an odd cycle in the incompatibility graph. Since the incompatibility graph is an auxiliary object, it is not as natural as a certificate as Tucker patterns or asteroidal triples.

2.2.6 Partition Refinement

In this subsection we introduce the notion of partition refinement. Also, we give a relationship between C1P and partition refinement and show how to use partition refinement to decide the C1P.

Theorem 2.2.16. [22] *A binary matrix M is C1P if and only if every connected component of its overlap graph $O(M)$ is C1P.*

Proof. ($\Rightarrow$) Suppose there exists one connected component $C = \{r_{i_1}, \dots, r_{i_k}\}$ of $O(M)$ that is non-C1P. Consider the support of C, $\{c_{j_1}, \dots, c_{j_l}\}$. Since C is non-C1P, then any permutation π of the support C induces a gap (a non-empty sequence of consecutive zeros that separates two sequences of consecutive ones) in C_π. Then, adding rows and columns to C cannot close the gap. That is, if we add columns to C, any permutation π' of the columns of this augmented matrix $M' = (R', C')$ will contain as a subpermutation a permutation of the columns of C that will induce a gap on some row in $C'_{\pi'}$. Adding rows will obviously not change this property. Therefore, M is non-C1P.

($\Leftarrow$) Suppose that every connected component of M is C1P. If $O(M)$ has only one connected component, then M is C1P. If $O(M)$ has more than one connected component, then they are disjoint and the support of the union of the connected components is the support of M. Then, a valid permutation for M can be constructed combining valid permutations on the support of each connected component. □

Note 2.2.17. Without loss of generality, from now on we assume that the overlap matrix of all the binary matrices considered have only one connected component, unless stated otherwise. This assumption has no algorithmic complexity cost since computing the connected components of $O(M)$ can be done in linear time $O(e)$ [15].

Partition refinement is a very powerful tool that has many applications such as automaton minimization, string sorting, and modular decomposition [23].

Definition 2.2.18. A *partition* $\mathcal{P}$ of a set E is a set of disjoint subsets of E, $\{E_1, \dots E_k\}$ whose union is exactly E.

Definition 2.2.19. *Refining* a partition $\mathcal{P}$ with a pivot set S consists of replacing each class $E \in \mathcal{P}$ by two classes $E_a = E \cap S$ and $E_b = E \setminus S$, such that $E_a, E_b \neq \emptyset$.

Definitions 2.2.18 and 2.2.19 are definitions of a set of numbers.

A partition can be refined in linear time $O(e)$ as described in [23]. Note that in general a partition is an unordered set. Whenever a partition is refined, the new class E_a can be inserted either to the right or the left of E_b. It depends on the application that is being considered to establish restrictions on the refinement process. For some applications, it might be required to embed a partition into a combinatorial structure, including a possibly higher algorithmic cost, or the need of a more advanced algorithm to maintain the linear time complexity. This will be the case for deciding the C1P, where valid permutations will be defined from a total order of the parts of a partition of the columns of a matrix.

Definition 2.2.20. Let $R = \{r_1, ..., r_n\}$ be a set of rows. R is *connected* if its overlap graph has a single connected component.

From now on, when we use the term connected set of rows we refer to connected set of rows in the sense of the overlap graph, unless otherwise stated.

Testing the C1P of a connected matrix can be done using partition refinement. In order to do so, it is required to order the rows of the matrix in a total order and to process them, as successive pivots, according to this order. Not any order is satisfactory, as intuitively, it does not make sense to refine a partition by a pivot that does not intersect with this partition. [1] We formalize below a family of orders that avoid this issue.

[1] [36] This condition can however be somewhat relaxed, although in a limited way.

Definition 2.2.21. A total order $r_1, \dots, r_k$ of a connected set of rows R is *overlap-consistent* if, for any $i = 2, \dots, k$, r_i overlaps at least one of $r_1, \dots, r_{i-1}$.

The condition for being overlap-consistent is equivalent to stating that the rows are ordered according to an arbitrary walk in the overlap graph of R. In [30], the fact that the order of processing the rows comes from a depth-first walk in the overlap graph is central to obtain a linear time complexity.

Definition 2.2.22. Let $R = \{r_1, ..., r_n\}$ be a connected set of rows and S its support. An *ordered partition* for R is an ordered set $(P_1 = \{S_1, R_1\}, ..., P_k = \{S_k, R_k\})$ such that the S_i are disjoint subsets of S and the R_i are subsets of R, and that satisfies the following properties:

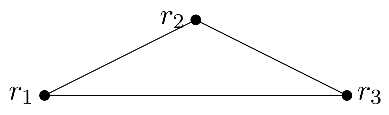
1. The union of the S_i is equal to S.
2. The union of the R_i is equal to R.
3. If r_i appears in $P_{i_1}, \dots, P_{i_l}$, then r_i is the union of $S_{i_1}, \dots, S_{i_l}$.

Note that definition 2.2.22 does not assume that R is C1P.

Example 2.2.23. This example illustrates how to use partition refinement to decide the C1P. Let M be the following connected set of rows

$$\begin{array}{c|ccc|} & c_1 & c_2 & c_3 \\ \hline r_1 & 1 & 1 & 0 \\ r_2 & 0 & 1 & 1 \\ r_3 & 1 & 0 & 1 \\ \hline \end{array}$$

To construct the partition refinement of M we first calculate the overlap graph $O(M)$. The following graph illustrates the overlap graph $O(M)$



Then, we process rows of M according to a depth-first order traversal of a spanning tree of O(M), chosen arbitrarily. In this case, we will follow the order r_1, r_2, r_3. We process the first row and write

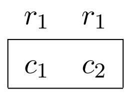


Figure 2.5: An ordered partition of the support of the processed columns with a single part.

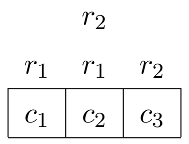


Figure 2.6: An ordered partition of the support of the processed columns with three parts.

horizontally the columns that belong to that row and above each column we write the rows that it belongs to. Then, we obtain the following partition $P = (P_1 = \{c_1, c_2, r_1\})$

Then, we refine the partition using row r_2 (see Figure 2.6). Now c_2 will be separated from c_1 because r_1 and r_2 overlap and their intersection is c_2. We obtain the following partition refinement $P = (P_1 = \{c_1, r_1\}, P_2 = \{c_2, r_1, r_2\}, P_3 = \{c_3, r_2\})$

Finally, we refine the partition with row r_3 (see Figure 2.7). We can see that when refining the partition with r_3 the columns that belong to r_3 are not consecutive in the partition, there is a gap between c_1 and c_3.

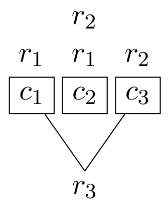


Figure 2.7: A failed partition refinement of the support of the processed columns with three parts.

Definition 2.2.24. Let $P_1, \dots, P_k$ be an ordered partition of a connected set of rows $R = \{r_1, \dots, r_n\}$. $\{P_1, \dots, P_k\}$ is said to be a *C1P ordered partition* if it satisfies the following properties:

1. The union of the S_i is equal to S.
2. The union of the R_i is equal to R.
3. Each r_i appears in a consecutive set of P_j.
4. If r_i appears in $P_a, \dots, P_b$, then r_i is the union of $S_a, \dots, S_b$.

We can see that partition refinement can be used to decide if a matrix is C1P. If the matrix is C1P, the partition refinement returns a partition of the columns of the matrix, where the order of the columns in the partition gives a valid permutation. A successful partition refinement returns a C1P ordered partition. The case when the matrix is non-C1P will be discussed at the end of the section.

Definition 2.2.25. Let $P_1, \dots, P_k$ be a C1P ordered partition of a connected set of rows $M = \{r_1, \dots, r_n\}$. This C1P ordered partition is *canonical* if it satisfies the additional property that

5. No two P_i and P_j are such that $R_i = R_j$.

The theorem below follows immediately from the definition of C1P ordered partition. It is mostly a rewriting, using the terminology on partition refinement we introduced, of the classical encoding of all valid permutations of a C1P matrix using a PQ-tree.

Theorem 2.2.26. *Let M be a binary matrix. M is C1P if and only if there exists a C1P ordered partition of M. Moreover, if M is C1P, then there is a unique ordered partition of its columns that is canonical and C1P.*

Note that we can extend to a general matrix M if $O(M)$ has multiple components using Theorem 2.2.16. We now state an important property of the unique canonical C1P ordered partition of a connected C1P matrix, that will be crucial in the design of a data structure to encode all valid permutations of a C1P matrix: the PQ-tree that we describe in a subsequent section.

Definition 2.2.27. Let $\mathcal{P}$ be a an ordered partition of a set S. Then a permutation π on S is *valid* for $\mathcal{P}$ if the following implication holds for every pair of elements x, y of S: if x and y are respectively in parts P_i and P_j of $\mathcal{P}$, with $i < j$, then x appears before y in π.

Property 2.2.28. A permutation π of the columns of a connected C1P matrix M is valid for M if and only if either π or its mirror is valid for its unique canonical ordered partition $\mathcal{P}$.

The property follows immediately from the fact that the parts of the canonical ordered partition $\mathcal{P}$ are defined in terms of intersections of the different rows of M. We illustrate this property with the following example.

Example 2.2.29. Let M be the following C1P matrix

$$\begin{array}{c|cccccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 \\ \hline
r_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
r_2 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
r_3 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
r_4 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline
\end{array}$$

Then, by applying partition refinement to M we obtain the canonical C1P ordered partition $P = (P_1 = \{c_1, r_1\}, P_2 = \{c_2, r_1, r_2\}, P_3 = \{c_3, c_4, r_1, r_2, r_3\}, P_4 = \{c_5, c_6, r_2, r_3, r_4\}, P_5 = \{c_7, r_3, r_4\}, P_6 = \{c_8, r_4\})$

$$\begin{array}{cccccc}
 & & r_3 & r_4 & & \\
 & r_2 & r_2 & r_3 & r_4 & \\
r_1 & r_1 & r_1 & r_2 & r_3 & r_4 \\
\boxed{c_1} & \boxed{c_2} & \boxed{c_3, c_4} & \boxed{c_5, c_6} & \boxed{c_7} & \boxed{c_8}
\end{array}$$

where every block represents a set of columns that correspond to the S_i, for $1 \leq i \leq 6$. And every R_i, for $1 \leq i \leq 6$, is represented as the set of rows above the corresponding S_i for the partition.

We can use the partition to obtain all valid permutations of M, by permuting the columns that belong to the same part. So, we can permute c_3, c_4 as well as c_5, c_6. Then, the valid permutations of M are: $\pi_1 = c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8$, $\pi_2 = c_1, c_2, c_4, c_3, c_5, c_6, c_7, c_8$, $\pi_3 = c_1, c_2, c_3, c_4, c_6, c_5, c_7, c_8$, $\pi_4 = c_1, c_2, c_4, c_3, c_6, c_5, c_7, c_8$ and their mirrors, $\pi_5 = c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1$, $\pi_6 = c_8, c_7, c_6, c_5, c_3, c_4, c_2, c_1$, $\pi_7 = c_8, c_7, c_5, c_6, c_4, c_3, c_2, c_1$, $\pi_8 = c_8, c_7, c_5, c_6, c_3, c_4, c_2, c_1$.

Notation 2.2.30. From now on we use indistinguishably the terms *partition refinement* and *C1P canonical ordered partition refinement*, to refer to C1P canonical ordered partition refinement of a C1P matrix.

Definition 2.2.31. Let $R = \{r_1, \ldots, r_n\}$ be a connected set of rows that is not C1P but such that $R' = \{r_1, \ldots, r_{n-1}\}$ is C1P. Let S be the support of R. The *mapping* of r_k onto the partition refinement of R' $\{P_1 = \{S_1, R_1\}, P_2 = \{S_2, R_2\}, \ldots, P_k = \{S_k, R_k\}\}$, is the ordered set $(Q_0 = \{0, C_0\}, Q_1 = \{j_1, C_1\}, \ldots, Q_p = \{j_p, C_p\})$ such that

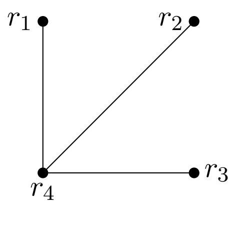
1. $p \leq k$
2. for $i = 1, \ldots, p$, $C_i \subseteq S_{j_i}$
3. $r_k = \cup_i C_i$, where rows are being represented by their supporting columns
4. $1 \leq j_1 < j_2 < \cdots < j_p \leq k$

In the definition above, the C_is represent the intersection of r_n with the parts of the partition refinement of R', but for C_0 it represents the elements of r_n that do not appear in any of these parts (i.e. the part of the support that is specific to r_n).

Example 2.2.32. Let M be a binary matrix defined as

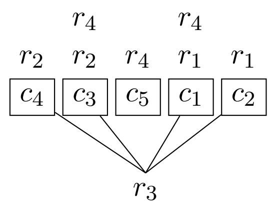
$$\begin{array}{c|ccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 \\
\hline
r_1 & 1 & 1 & 0 & 0 & 0 \\
r_2 & 0 & 0 & 1 & 1 & 0 \\
r_3 & 1 & 1 & 1 & 1 & 0 \\
r_4 & 1 & 0 & 1 & 0 & 1 \\
\hline
\end{array}$$

with bipartite graph



To calculate the partition refinement of M we process the rows in order r_2, r_4, r_1, r_3.

Then the mapping of r_3 onto the partition refinement of $R' = \{r_1, r_2, r_4\}$ is the ordered set $(Q_0 = \{0, \emptyset\}, \{1, c_4\}, \{2, c_3\}, \{4, c_1\}, \{5, c_2\})$, as illustrated below.



where every block represents a set of columns that correspond to the S_i, for $1 \leq i \leq 5$, every R_i, for $1 \leq i \leq 5$, is represented as the set of rows above the corresponding S_i for the partition, and the mapping of r_3 is represented below the partition.

Recall that a gap is a sequence of consecutive zeros that separates two sequences of consecutive ones in a row.

Now, we give a definition of a gap that involves the notation of partition refinement, and will be used to characterize a non-C1P matrix through the notion of failed partition refinement.

Definition 2.2.33. Let $R = \{r_1, \ldots, r_n\}$ be a connected set of rows such that $R' = \{r_1, \ldots, r_{n-1}\}$ is connected and C1P. Let $\mathcal{P} = (P_1 = \{S_1, R_1\}, \ldots, P_k = \{S_k, R_k\})$ be a partition refinement of R' and $\mathcal{Q} = (Q_0 = \{0, C_0\}, Q_1 = \{j_1, C_1\}, \ldots, Q_p = \{j_p, C_p\})$ be the mapping of r_n onto $\mathcal{P}$.

$\mathcal{P}$ and $\mathcal{Q}$ define an *inside gap* $\{a, b\}$ if

1. $a = j_m, b = j_n,$ for some m, n such that $1 \leq m < n \leq p$
2. There exists $S_q \in \mathcal{P}$ such that $1 \leq q \leq k$ and $a < q < b$ and there exists $c \in S_q, c \notin r_n$.

$\mathcal{P}$ and $\mathcal{Q}$ define an *outside gap* $\{0, a\}$ (resp. $\{b, 0\}$; $\{0, 1, k\}$) if

1. $C_0 \neq \emptyset$,
2. $a = j_1 > 1$ and $j_p = k$ and $S_k \subseteq r_n$ (resp. $b = j_p < k$ and $j_1 = 1$ and $S_1 \subseteq r_n$; $j_1 = 1$ and $j_p = k$ and there exists $c \in S_1, c \notin r_n$ and $c' \in S_k, c' \notin r_n$).

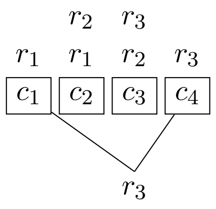
Together, $\mathcal{P}$ and $\mathcal{Q}$ are called a *failed partition refinement* if they define at least one gap.

An inside gap $\{a, b\}$ is *minimal* if there exists $0 \leq i \leq p-1$ such that $j_i = a$ and $j_{i+1} = b$.

Example 2.2.34. Let M be the following connected set of rows that is non-C1P

$$\begin{array}{c|cccc|}
 & c_1 & c_2 & c_3 & c_4 \\
\hline
r_1 & 1 & 1 & 0 & 0 \\
r_2 & 0 & 1 & 1 & 0 \\
r_3 & 0 & 0 & 1 & 1 \\
r_4 & 1 & 0 & 0 & 1 \\
\hline
\end{array}$$

Then, a failed partition refinement of M is the partition $(P_1 = \{c_1, r_1\}, P_2 = \{c_2, r_1, r_2\}, P_3 = \{c_3, r_2, r_3\}, P_4 = \{c_4, r_3\})$ and the ordered set $(Q_0 = \{0, \emptyset\}, Q_1 = \{1, c_1\}, Q_2 = \{4, c_4\})$, and can be seen below



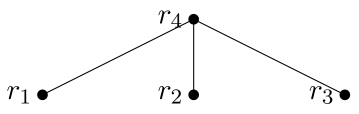
where the gap is the inside gap $\{1, 4\}$.

Example 2.2.34 is an example of a failed partition refinement with an inside gap. The following example illustrates the case where a failed partition refinement has an outside gap.

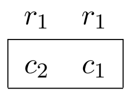
Example 2.2.35. Let M be the following binary matrix

$$\begin{array}{c|cccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \\
\hline
r_1 & 1 & 1 & 0 & 0 & 0 & 0 \\
r_2 & 0 & 0 & 1 & 1 & 0 & 0 \\
r_3 & 0 & 0 & 0 & 0 & 1 & 1 \\
r_4 & 1 & 0 & 1 & 0 & 1 & 0 \\
\hline
\end{array}$$

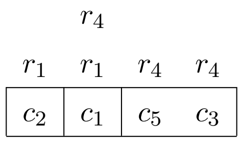
The overlap graph $O(M)$ of M is illustrated in the following drawing



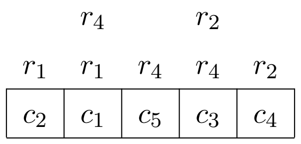
Using a spanning tree of $O(M)$ we can consider the order r_1, r_4, r_2, r_3 to process the rows of M. First, we process row r_1 obtaining



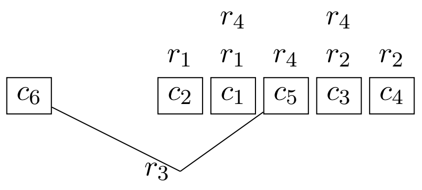
Then, we refine the partition using row r_4, obtaining



We continue refining the partition with r_2, obtaining the following drawing



Finally, we refine the partition with row r_3, obtaining a failed partition refinement. The partition refinement of M is the partition $(P_1 = \{c_2, r_1\}, P_2 = \{c_1, r_1, r_4\}, P_3 = \{c_5, r_4\}, P_4 = \{c_3, r_2, r_4\}, P_5 = \{c_4, r_2\}$, and the ordered set $(Q_0 = \{0, c_6\}, Q_1 = \{3, c_5\})$ and can be seen using the following drawing



where the outside gap is $\{0, 3\}$.

Remark 2.2.36 (Partition refinement as a certificate.)**.** Note that the structure of a partition refinement could be seen as a certificate for the property of being C1P, by verifying that it satisfies the five properties of canonical C1P ordered partition refinement; and for the non-C1P, by verifying that it satisfies the four properties of failed partition refinement. In the following we describe how this can be achieved. Formally, we would need to describe precise data structures for the encoding of partition refinement, but we stay at a high abstraction level as standard structures such as linked lists and arrays are sufficient to implement the strategies we outline in linear time.

To verify that a partition is a canonical C1P ordered partition we need to verify that the union of S_i is equal to S, which can be obviously done by a simple scan of all parts of the partition. Similarly, checking that the union of R_i is equal to R (point 2 of the definition) and that every row is completely included in the partition (point 4) and appears in a consecutive set of parts (point 3) can be done by a simple left-to-right scan of the partition. These four steps can be implemented in $O(e)$ time. Point 5, that is specific to the canonical aspect, can be checked by looking at the labels of consecutive parts, which again is elementary and can be achieved in $O(e)$ time, as the total number of labels is in $O(e)$, provided the relative order of every pair of rows is the same in every label that contains them both, which can be easily ensured during the construction of the partition.

To verify that a partition is a failed partition refinement we need to check that R' is a canonical partition refinement, plus additional properties related to r_n and the fact that it creates a gap in a failed partition refinement. This can be done by finding the mapping of r_n onto the partition refinement of R', then looking at the two different kinds of gaps:

- if $C_0 \neq \emptyset$, then we need only to look at the inclusion of parts P_1 and P_k into r_n to search for an outside gap,
- otherwise, we can look for an inside gap by looking at consecutive parts of the mapping Q.

In both cases, once the mapping of r_n is given, this reduces to a single scan of the parts of $\mathcal{P}$, and thus is elementary and can be done in $O(e)$ time.

It is however fair to remark that, although this approach of certifying the C1P/non-C1P satisfies the formal definition of a certificate, as it involves simple and efficient algorithms and data structures, it can not compare with the simpler and elegant certificate given by odd cycles of the incompatibility graph.

2.2.7 PQ-trees and PQR-trees

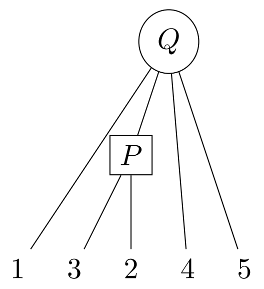
Another algorithm for deciding the C1P is based on a data structure called a PQ-tree. In 1975, Booth and Lueker [7] introduced the PQ-tree, a rooted tree with two kinds of nodes, P-nodes and Q-nodes. PQ-trees describe non-empty sets of permutations. PQ-trees encode in linear space possibly an exponential number of permutations. In the context of the C1P, this data structure has the following fundamental property: if a binary matrix is C1P, then the set of all its valid permutations can be represented by a PQ-tree.

Booth and Lueker [7] used PQ-trees for the first algorithm for deciding the C1P in linear time. Their algorithm can decide if a binary matrix is C1P in time $O(m + n + e)$, where m is the number of rows, n is the number of columns and e is the number of entries equal to 1 in the binary matrix. When the algorithm succeeds, a PQ-tree is returned. Otherwise, when the binary matrix is non-C1P, the algorithm fails and no information is returned.

Definition 2.2.37. Let $N = \{1, \ldots, n\}$ be a finite set (referred to as the ground set of elements). A *PQ-tree* is an ordered rooted tree with two kinds of internal nodes, namely P-nodes and Q-nodes, such that the children of a Q-node are linearly ordered and the children of a P-node are unordered. The leaves are uniquely labeled by the elements of this ground set (no two leaves have the same label).

Definition 2.2.38. The *frontier* of a PQ-tree is the permutation of its ground set obtained by reading the leaves from left to right.

Example 2.2.39. Let $N = \{1, 2, 3, 4, 5\}$ be a ground set. A PQ-tree T of N is the following rooted tree

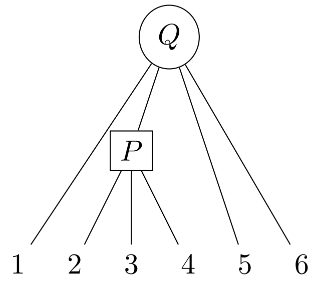


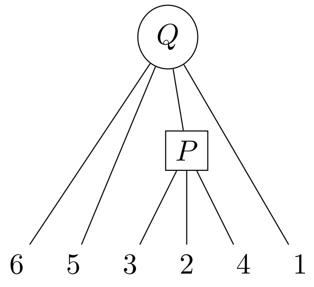
and the frontier of T is $\pi = \{1, 3, 2, 4, 5\}$.

Definition 2.2.40. Let T be a PQ-tree. The *allowed transformations* of T are changing the order of the children of a P-node and reversing the children of a Q-node.

Definition 2.2.41. Let T_1 and T_2 be two different PQ-trees on the same ground set N. T_1 and T_2 are *equivalent* if one can transform one into the other by a sequence of allowed transformations.

Example 2.2.42. Let T_1 and T_2 be the following PQ-trees on $N = \{1, 2, 3, 4, 5, 6\}$





Then, T_1 and T_2 are equivalent since we can transform T_2 into T_1 by changing the order of the P-node from $3, 2, 4$ to $4, 3, 2$ and then reversing the order of the Q-node.

The equivalence relation defined above determines equivalence classes of PQ-trees. Equivalence classes of PQ-trees define sets of permutations of the ground set, by taking the frontiers of the trees in the equivalence class.

PQ-trees are also defined on binary matrices by taking as ground set the set of columns of the matrix. We will see how to use PQ-trees in deciding the C1P.

Theorem 2.2.43. [7] *Let M be a C1P binary matrix. Then the set of all its valid permutations is encoded by a PQ-tree T_M that can be computed in $O(n + m + e)$ time.*

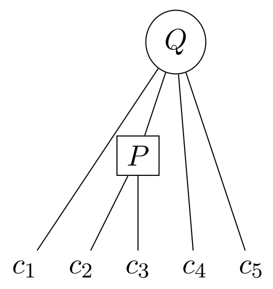
PQ-trees can be used to describe the set of all permutations of a binary matrix M that are valid permutations.

Definition 2.2.44. Let M be a C1P matrix M with PQ-tree T_M. The set of permutations encoded by T_M, that is, the set of all possible orderings of the leaves of T_M, is the set of all possible *consecutive ones orderings* of T_M and is denoted $\rho(T_M)$.

Example 2.2.45. Let M be the binary matrix

$$\begin{array}{c|ccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 \\
\hline
r_1 & 1 & 1 & 1 & 1 & 1 \\
r_2 & 0 & 1 & 1 & 1 & 0 \\
r_3 & 0 & 0 & 0 & 1 & 1 \\
r_4 & 0 & 1 & 1 & 1 & 0 \\
\hline
\end{array}$$

The PQ-tree of M is the rooted tree T_M



From the PQ-tree we can see all the consecutive ones orderings of M by using all possible allowed transformations: c_1, c_2, c_3, c_4, c_5; c_1, c_3, c_2, c_4, c_5; c_5, c_4, c_3, c_2, c_1; and c_5, c_4, c_2, c_3, c_1.

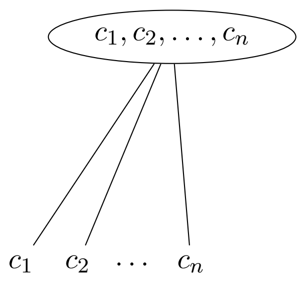


Figure 2.8: Universal PQ-tree

Proposition 2.2.46. *Let M be a C1P matrix. Then, there exists a PQ-tree T_M such that π is a valid permutation for M if and only if $\pi \in \rho(T_M)$.*

The principle of the algorithm of Booth and Lueker [7] is as follows: start from the universal PQ-tree T (Figure 2.8) encoding the set P of all permutations of the ground set. Now process rows of M in an arbitrary order. For each row r, the current PQ-tree T_M is refined to encode only the subset of P that contains r as an interval.

If at some point, this refinement leads to an empty set of permutations (no permutation of P contains r as an interval) then M is not C1P. Otherwise, the resulting PQ-tree is T_M. In the paper of Booth and Lueker, refinements are done using a rewriting approach based on a finite set of tree templates, that is considered as very difficult to implement properly.

We describe an alternative way to construct a PQ-tree due to McConnell [30]. Let M be a C1P matrix. Consider $O(M)$, the overlap graph of M. $O(M)$ is a set of connected components that defines the inclusion tree I_M, where every node in the tree is a set of rows of M. The PQ-tree T_M is a refinement of I_M. To construct the PQ-tree of M we do partition refinement for every connected component. For every connected component a new Q-node is inserted in T_M, together with the corresponding columns that belong to that connected component. After refining each connected component a P-node is added to the tree for every part of the partition that contains at least two columns. Then, every node S has a label in the following way:

$$S = \begin{cases} Q\text{-node} & \text{if } S \text{ is a connected component with at least three children} \\ P\text{-node} & \text{if } S \text{ is not a } Q\text{-node and } S \text{ has at least two children} \end{cases}$$

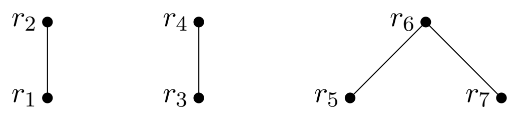
All the possible leaf orders of the PQ-tree can be obtained by changing the order of P-nodes and

Q-nodes. Since the children of a Q-node are linearly ordered, then we have only two possible orders of the children, namely the given one and its reverse. Since the children of a P-node are unordered, then we can rearrange the children in any way, obtaining a different order for each rearrangement. Then all the possible leaf orders of the PQ-tree can be obtained by rearranging the order of the P-nodes and reversing the order of the Q-nodes. These orders are all the possible consecutive ones orderings from a matrix M.

Example 2.2.47. Let M be the following C1P matrix

$$\begin{array}{c|ccccccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \\ \hline
r_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
r_2 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
r_3 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
r_4 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
r_5 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
r_6 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
r_7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline
\end{array}$$

Consider the overlap graph $O(M)$ of M,



that consists of three connected components, $C_1 = \{r_1, r_2\}$, $C_2 = \{r_3, r_4\}$, and $C_3 = \{r_5, r_6, r_7\}$. Applying partition refinement to every component, we obtain the following partitions. For C_1 we obtain

$$\begin{array}{|ccc|c|ccccc|}
 \multicolumn{3}{c}{} & \multicolumn{1}{c}{r_2} & \multicolumn{5}{c}{} \\
 \multicolumn{1}{c}{} & \multicolumn{1}{c}{r_1} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{r_1} & \multicolumn{2}{c}{} & \multicolumn{1}{c}{r_2} & \multicolumn{2}{c}{} \\ \hline
c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \\ \hline
\end{array} \tag{2.1}$$

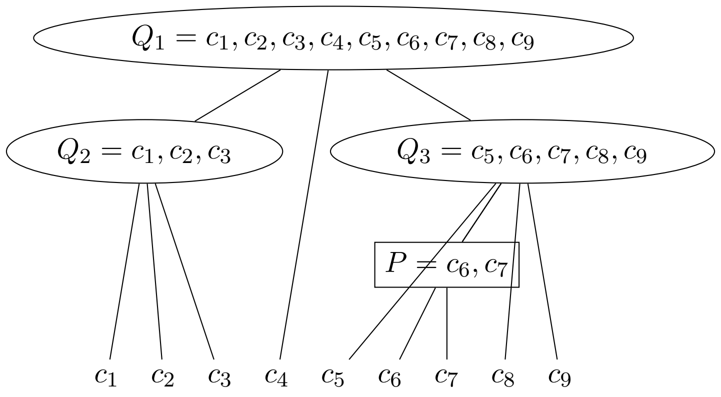
for C_2 we obtain

$$\begin{array}{ccc} & r_4 & \\ r_3 & r_3 & r_4 \\ \hline \multicolumn{1}{|c|}{c_1} & \multicolumn{1}{c|}{c_2} & \multicolumn{1}{c|}{c_3} \\ \hline \end{array} \tag{2.2}$$

for C_3 we obtain

$$\begin{array}{cccc} & & r_7 & \\ & r_6 & r_6 & r_7 \\ r_6 & r_5 & r_5 & r_5 \\ \hline \multicolumn{1}{|c|}{c_5} & \multicolumn{1}{c|}{c_6 \quad c_7} & \multicolumn{1}{c|}{c_8} & \multicolumn{1}{c|}{c_9} \\ \hline \end{array} \tag{2.3}$$

Then, the PQ-tree of M is defined by the partition refinement of all the connected components C_1, C_2, and C_3, and has the following form


$Q_1 = c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9$
$Q_2 = c_1, c_2, c_3$
$Q_3 = c_5, c_6, c_7, c_8, c_9$
$P = c_6, c_7$
c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9

where the Q-nodes are represented as circular nodes, P-nodes are represented as rectangular nodes and leaves are represented without anything around them.

We now describe a generalization of the PQ-tree that was discovered independently by Meidanis, Porto and Telles [33] and McConnell [30]. The motivation stems from the observation that when a binary matrix is non-C1P, the reason can be often located in a small set of rows and/or columns, and thus some, possibly large, submatrices might be C1P. The initial approach of Booth and Lueker produces an empty set of permutation for any non-C1P matrix, thus hiding this possibly interesting structure (especially for applications such as genomics). Therefore, the idea was to introduce a third kind of node, R-nodes, that will identify non-C1P submatrices.

Given a binary matrix M, a unique PQR-tree T_M can be computed as follows: if a connected component of $O(M)$ is non-C1P, then the corresponding node of I_M is labeled by R.

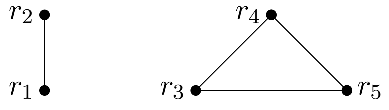
The PQR-tree can be constructed similarly to the PQ-tree, introducing an R-node whenever the 1's cannot be consecutive. We have the following result that allows us to use PQR-trees in deciding the C1P.

Theorem 2.2.48. [33], [30] *A binary matrix M is C1P if and only if T_M does not contain any R-node.*

Example 2.2.49. Consider the binary matrix M,

$$\begin{array}{c|ccccccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \\ \hline
r_1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
r_2 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
r_3 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
r_4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
r_5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \hline
\end{array}$$

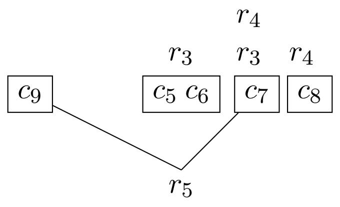
with overlap graph $O(M)$



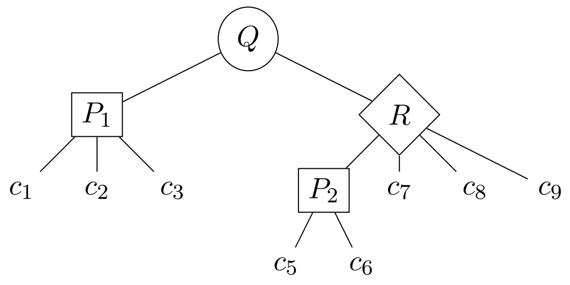
$O(M)$ has two components: $C_1 = \{r_1, r_2\}$ and $C_2 = \{c_3, c_4, c_5\}$. For C_1 we obtain the partition refinement

$$\begin{array}{|ccc|c|ccccc|}
\multicolumn{3}{c}{} & \multicolumn{1}{c}{r_2} & \multicolumn{5}{c}{} \\
\multicolumn{1}{c}{} & \multicolumn{1}{c}{r_1} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{r_1} & \multicolumn{2}{c}{} & \multicolumn{1}{c}{r_2} & \multicolumn{2}{c}{} \\ \hline
c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 & c_9 \\ \hline
\end{array} \tag{2.4}$$

and for C_2 we obtain the failed partition refinement



Since r_5 is not consecutive, the R-node contains the columns that belong to r_5. The corresponding PQR-tree is shown below.



where $Q = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9\}$, $P_1 = \{c_1, c_2, c_3\}$, $P_2 = \{c_5, c_6\}$ and $R = \{c_5, c_6, c_7, c_8, c_9\}$.

In the example above, we can see that since there is an R-node the matrix is non-C1P. Although M is non-C1P, some information can still be extracted from the PQR-tree, namely that the conflict is being produced by the columns in the R-node. In this case, the conflict is produced by columns c_7, c_8, c_9 together with one of c_5 or c_6. In the case where a PQR-tree has no R-nodes, then, we know that the matrix is C1P.

Theorem 2.2.50. [30] *Computing the PQR-tree T_M of a binary matrix can be done in $O(n+m+e)$ time and space.*

Note that an R-node is not a certificate, as it is a structure obtained by partition refinement. Also, an R-node identifies a submatrix (rows and columns) that contains a Tucker pattern, but it does not provide any hint as to which rows and columns form this Tucker pattern. It is mostly useful to narrow down the matrix of a non-C1P matrix that contains such structures.

Part II

New Results

Chapter 3

Structural and Algorithmic Results on Partition Refinement

In this chapter we investigate the use of the structure of partition refinement to find a certificate of non-C1P. We begin this chapter by describing the structure of failed partition refinement for each type of Tucker pattern. Then, we show how to use this structure to extract an asteroidal triple of a single Tucker pattern. We also show how to use it to extract a Tucker pattern of type I, that is, a chordless cycle of length at least 6.

The motivation for the research described in this chapter is to design a linear time and space algorithm to find an asteroidal triple or a Tucker pattern in a non-C1P matrix, as the only certificate known so far that can be extracted in linear time, the odd cycles of the incompatibility graph [30], is not known to be any of these natural obstructions to the C1P. From a methodological point of view, our focus is on using the combinatorial structure obtained when applying partition refinement to the analyzed matrix. Although we did not fully succeed in our search for such an algorithm, we present a series of results that advance toward this goal.

3.1 Matrices that are Tucker Patterns

In this section we consider the case when the binary matrix M is a Tucker pattern. The questions we address are the following: can the combinatorial structure obtained using partition refinement help to decide which family of Tucker patterns M belongs to and can it help to identify quickly the three vertices x, y, z that define an asteroidal triple[1]. We provide positive answers to both questions.

We focus solely on the three vertices x, y, z as, once they are given, checking that the remaining edges define an asteroidal triple is easy, by looking for paths joining the three pairs of vertices. This can be done in linear time by a simple depth-first search, thus satisfying the requirement of certifying algorithms of an easy check of the validity of the certificate.

3.1.1 Simple algorithms

First, it is however important to remind that the two questions we address can be answered easily without relying on the notion of partition refinement, as we describe now.

Deciding if a binary matrix M is a Tucker pattern can be done in the following way: if M is a M_{IV} or M_V, then it can be checked easily since it has fixed size. If M is M_{I_k}, then all rows have degree 2 so it is the incidence matrix of a graph. We can test in linear time whether that graph is a cycle. If M is M_{II_k} or M_{III_k} then we can extract all rows of degree 2, make a graph from them, check that it is a path and then check the remaining rows satisfy the structure of patterns M_{II_k} or M_{III_k}. All these simple rules can obviously be checked in $O(e)$ time, since we are using a sparse representation of the matrix, that is, assuming that the rows are given as sets of columns.

Next, we can extract an asteroidal triple easily, as the above approach also produces a total order for rows and columns of the current matrix that matches the one given by the classical presentation of Tucker patterns (see Chapter 2), and thus allows us to locate immediately vertices x, y, z from M.

In the next section, we look at using the additional information provided by partition refinement.

3.1.2 Algorithms based on Partition Refinement

In this section we describe the structure of failed partition refinement for every Tucker pattern. Since we are assuming that M is a Tucker pattern, then applying partition refinement to M fails only when the last row is processed, so this structure consists of a refined partition augmented by a row whose elements are not consecutive in this partition.

[1]We remind here that by asteroidal triple, we mean C_M-AT as defined in Chapter 2.

Despite the fact that there can be an exponential number of orders in which the rows can be processed so that they are overlap-consistent, an exhaustive analysis does not require to examine them all. Indeed, as the order for processing rows of a C1P matrix does not impact the resulting partition of its columns, so the only factor determining the possible structures associated to a given pattern is the last processed row.

The results for the exhaustive analysis of processing partition refinement in every possible overlap-consistent order are shown in Figures 3.1 to 3.5. These Figures illustrate the failed partition refinement for each type of Tucker pattern, failing with every possible row. On the left column we indicate the last row that is being processed, that is, the row where the partition fails. On the right column we illustrate the failed partition refinement: every block represents the columns that belong to one of the parts of the partition, with the corresponding rows that belong to that part indicated above the block. The capital letters represent a part of the partition that is being repeated with different indices, where the indices are indicated in the square brackets around that part. Below the blocks is the last row r that is being processed and the lines joining r to the parts indicate to which parts the elements of r belong. The labels of rows and columns are the ones given in Figure2.3, that presents the five families of Tucker patterns.

Gaps are indicated in the partition in the following way: an *inside* gap is a part P_i, for some $1 \leq i \leq n$, that it is not joined to r and such that there are two parts P_j and P_k that are joined to r and $j < i$ and $i < k$. An *outside* gap can be seen in the partition as an inside gap with the additional property that P_j or P_k does not contain any rows above the block.

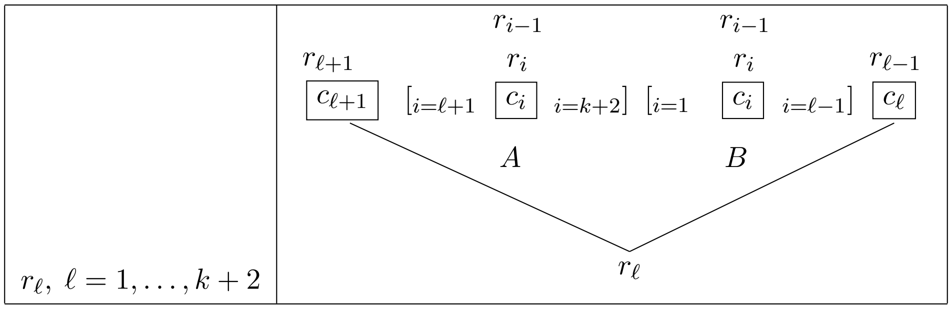


Figure 3.1: Failed partition refinement for M_{I_k}.

Note that for M_{IV} and M_V, r_4 can not be the last row in the partition because there is no order with r_4 as last row that satisfies the notion of overlap-consistent order.

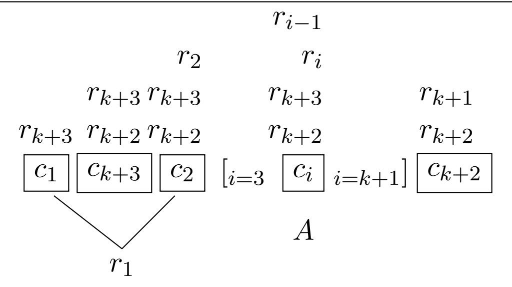
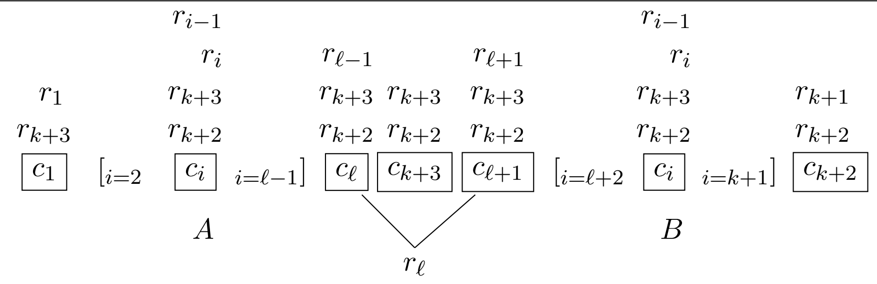
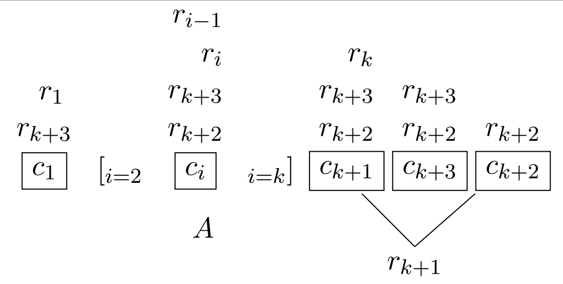
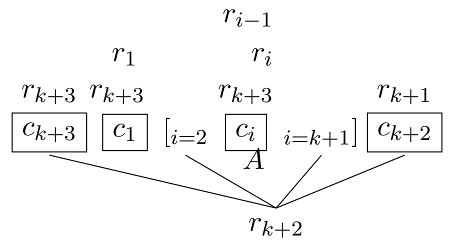
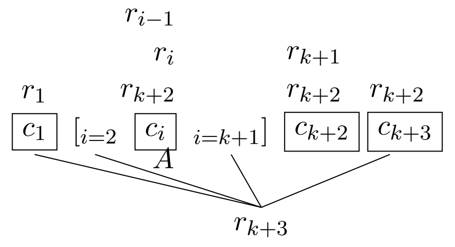
r_1	 r_{i-1} r_2 r_i r_{k+3} r_{k+3} r_{k+3} r_{k+1} r_{k+3} r_{k+2} r_{k+2} r_{k+2} r_{k+2} c_1 c_{k+3} c_2 $\big[_{i=3}$ c_i $_{i=k+1}\big]$ c_{k+2} A r_1
$r_\ell,\ \ell = 2, \ldots, k$	 r_{i-1} r_{i-1} r_i $r_{\ell-1}$ $r_{\ell+1}$ r_i r_1 r_{k+3} r_{k+3} r_{k+3} r_{k+3} r_{k+3} r_{k+1} r_{k+3} r_{k+2} r_{k+2} r_{k+2} r_{k+2} r_{k+2} r_{k+2} c_1 $\big[_{i=2}$ c_i $_{i=\ell-1}\big]$ c_ℓ c_{k+3} $c_{\ell+1}$ $\big[_{i=\ell+2}$ c_i $_{i=k+1}\big]$ c_{k+2} A B r_ℓ
r_{k+1}	 r_{i-1} r_i r_k r_1 r_{k+3} r_{k+3} r_{k+3} r_{k+3} r_{k+2} r_{k+2} r_{k+2} r_{k+2} c_1 $\big[_{i=2}$ c_i $_{i=k}\big]$ c_{k+1} c_{k+3} c_{k+2} A r_{k+1}
r_{k+2}	 r_{i-1} r_1 r_i r_{k+3} r_{k+3} r_{k+3} r_{k+1} c_{k+3} c_1 $\big[_{i=2}$ c_i $_{i=k+1}\big]$ c_{k+2} A r_{k+2}
r_{k+3}	 r_{i-1} r_i r_{k+1} r_1 r_{k+2} r_{k+2} r_{k+2} c_1 $\big[_{i=2}$ c_i $_{i=k+1}\big]$ c_{k+2} c_{k+3} A r_{k+3}

Figure 3.2: Failed partition refinement of M_{II_k} for various choices of final row processed.

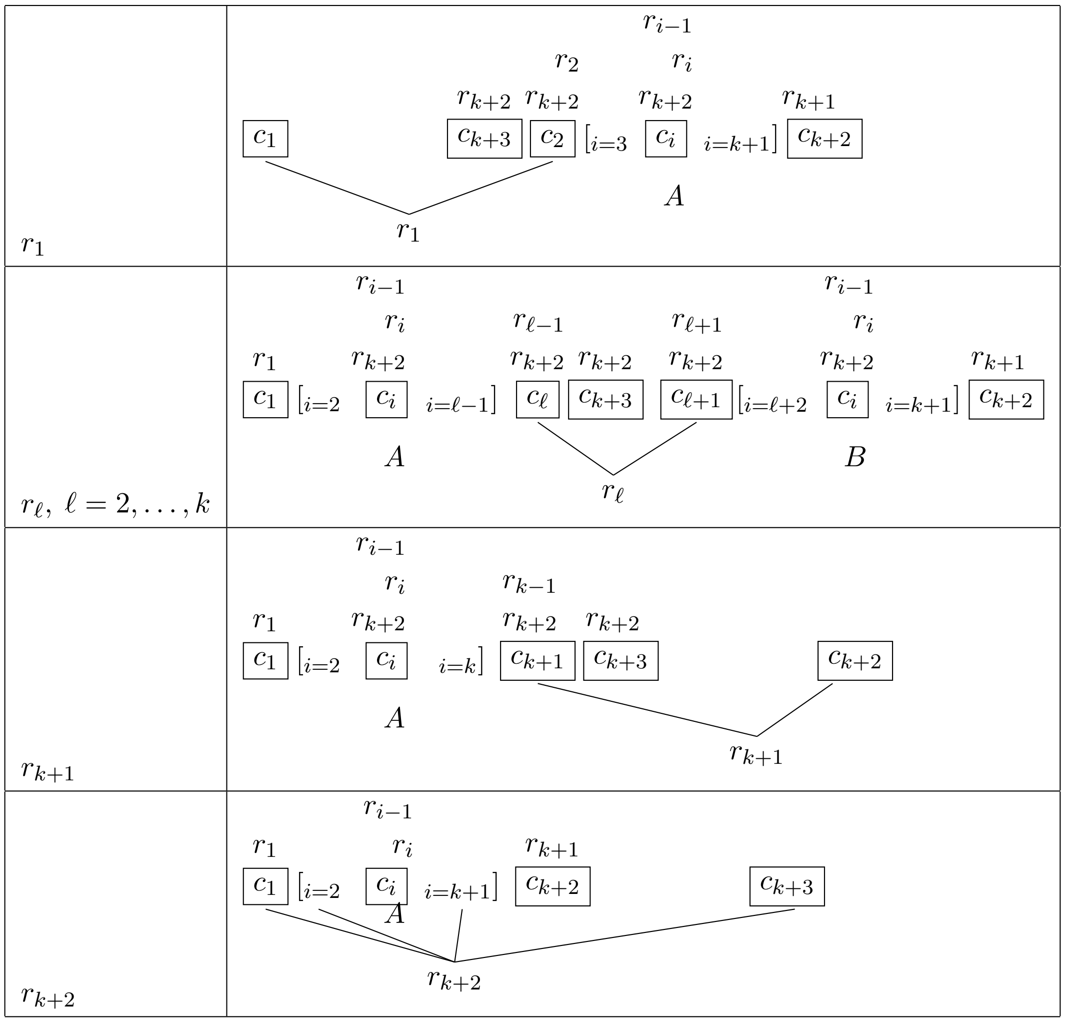
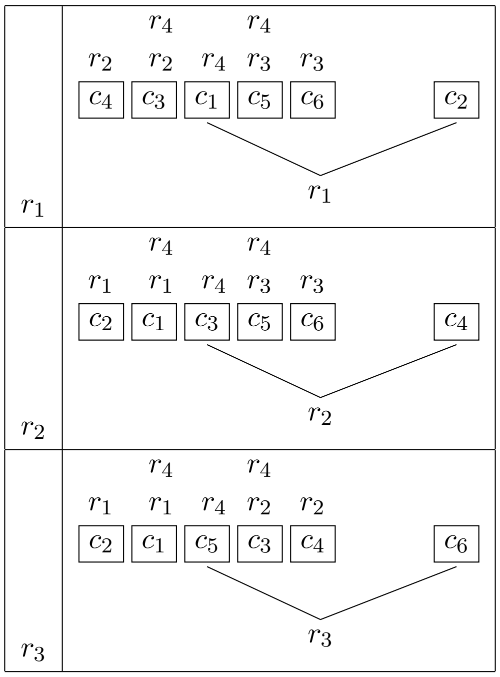

r_1
r_{i-1}
r_2
r_i
r_{k+2} r_{k+2} r_{k+2} r_{k+1}
$\boxed{c_1}$ $\boxed{c_{k+3}}$ $\boxed{c_2}$ $\big[_{i=3}$ $\boxed{c_i}$ $_{i=k+1}\big]$ $\boxed{c_{k+2}}$
A
r_1
$r_\ell,\ \ell = 2,\ldots,k$
r_{i-1} r_{i-1}
r_i $r_{\ell-1}$ $r_{\ell+1}$ r_i
r_1 r_{k+2} r_{k+2} r_{k+2} r_{k+2} r_{k+2} r_{k+1}
$\boxed{c_1}$ $\big[_{i=2}$ $\boxed{c_i}$ $_{i=\ell-1}\big]$ $\boxed{c_\ell}$ $\boxed{c_{k+3}}$ $\boxed{c_{\ell+1}}$ $\big[_{i=\ell+2}$ $\boxed{c_i}$ $_{i=k+1}\big]$ $\boxed{c_{k+2}}$
A B
r_ℓ
r_{k+1}
r_{i-1}
r_i r_{k-1}
r_1 r_{k+2} r_{k+2} r_{k+2}
$\boxed{c_1}$ $\big[_{i=2}$ $\boxed{c_i}$ $_{i=k}\big]$ $\boxed{c_{k+1}}$ $\boxed{c_{k+3}}$ $\boxed{c_{k+2}}$
A
r_{k+1}
r_{k+2}
r_{i-1}
r_1 r_i r_{k+1}
$\boxed{c_1}$ $\big[_{i=2}$ $\boxed{c_i}$ $_{i=k+1}\big]$ $\boxed{c_{k+2}}$ $\boxed{c_{k+3}}$
A
r_{k+2}

Figure 3.3: Failed partition refinement of M_{III_k} for various choices of final row processed.



r_1
r_4 r_4
r_2 r_2 r_4 r_3 r_3
c_4 c_3 c_1 c_5 c_6 c_2
r_1
r_2
r_4 r_4
r_1 r_1 r_4 r_3 r_3
c_2 c_1 c_3 c_5 c_6 c_4
r_2
r_3
r_4 r_4
r_1 r_1 r_4 r_2 r_2
c_2 c_1 c_5 c_3 c_4 c_6
r_3

Figure 3.4: Failed partition refinement of M_{IV}.

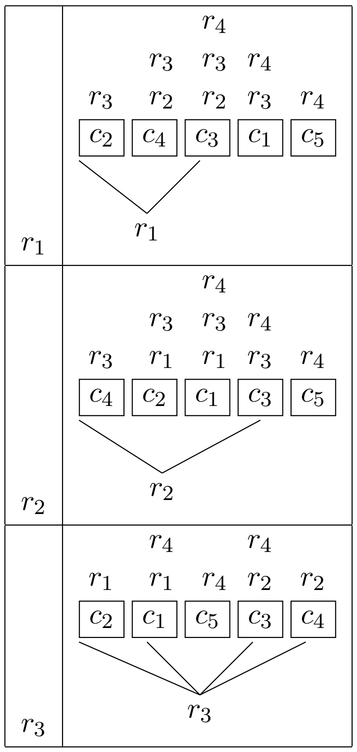

r_4
r_3 r_3 r_4
r_3 r_2 r_2 r_3 r_4
c_2 c_4 c_3 c_1 c_5
r_1
r_1
r_4
r_3 r_3 r_4
r_3 r_1 r_1 r_3 r_4
c_4 c_2 c_1 c_3 c_5
r_2
r_2
r_4 r_4
r_1 r_1 r_4 r_2 r_2
c_2 c_1 c_5 c_3 c_4
r_3
r_3

Figure 3.5: Failed partition refinement of M_V.

We now formalize a few observations.

Property 3.1.1. The failed partition refinements of all Tucker patterns, M_{I_k}, M_{II_k}, M_{III_k}, M_{IV} and M_V have only one gap.

Property 3.1.2. The failed partition refinements of Tucker patterns M_{I_k}, M_{II_k} and M_V have only an inside gap. The failed partition refinements of Tucker pattern M_{IV} have only an outside gap. The failed partition refinements of Tucker pattern M_{III_k} have both inside and outside gaps.

Property 3.1.3. The failed partition refinements of Tucker pattern M_{I_k} have a gap of length k. The failed partition refinements of Tucker patterns M_{II_k} and M_{III_k} have a gap of length one. The failed partition refinements of Tucker pattern M_{IV} have a gap of length 2. The failed partition refinements of Tucker pattern M_V have gaps of length 1 or 2.

Now we show how to find an asteroidal triple using failed partition refinement for a single Tucker pattern, or more precisely, to identify quickly the vertices x, y, z of such a triple.

Proposition 3.1.4. *Let M be a Tucker pattern. If partition refinement fails at row r, then the rightmost and leftmost columns of the failed partition refinement together with a column inside the gap define an asteroidal triple.*

Proof. For M_{I_k}, since Tucker patterns of type I are chordless cycles, every threesome of columns forms an asteroidal triple. In particular, the rightmost and leftmost columns of partition refinement together with any column inside the gap, will define an asteroidal triple.

For M_{II_k}, columns c_1, c_{k+2}, c_{k+3} define an asteroidal triple. From Figure 3.2 we can see that these columns correspond to the rightmost column, the leftmost column of the failed partition refinement and the column inside the gap.

For M_{III_k}, columns c_1, c_{k+2}, c_{k+3} define an asteroidal triple. From Figure 3.3 we can see that these columns correspond to the rightmost column, the leftmost column of the failed partition refinement and the column inside the gap.

For M_{IV}, columns c_2, c_4, c_6 define an asteroidal triple. From Figure 3.4 we can see that these columns correspond to the rightmost column, the leftmost column of the failed partition refinement and the column inside the gap.

For M_V, columns c_2, c_4, c_5 define an asteroidal triple. From Figure 3.5 we can see that these columns correspond to the rightmost column, the leftmost column of the failed partition refinement and the column inside the gap. □

We tried to generalize Proposition 3.1.4 to general matrices that are non-C1P, hoping to use the structure of failed partition refinement to extract an asteroidal triple from it. It turns out that in general, the result is not true as can be seen with the example below. First, we state the question we considered.

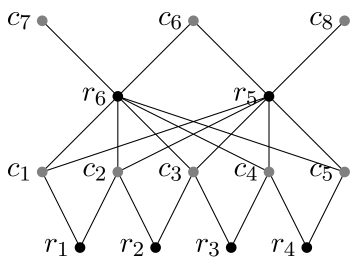
Question 3.1.5. *Let M be a binary matrix that is non-C1P. If partition refinement fails at row r, do the rightmost and leftmost columns of the partition refinement together with a column inside the gap define an asteroidal triple?*

The answer is that this is not true in general. We can see this with the following example:

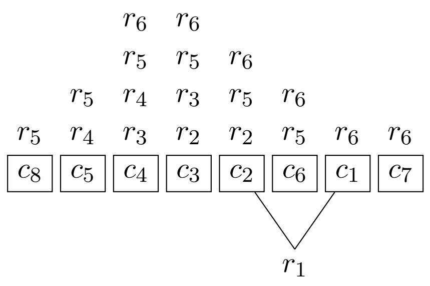
Example 3.1.6. Let M be the following binary matrix

$$\begin{array}{c|cccccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 & c_8 \\
\hline
r_1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
r_2 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
r_3 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
r_4 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
r_5 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
r_6 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
\hline
\end{array}$$

Then, the corresponding bipartite graph is



with one asteroidal triple c_1, c_5, c_6. And, the partition refinement using the order of the rows $r_5, r_6, r_4, r_3, r_2, r_1$ is


r_6 r_6
r_5 r_5 r_6
r_5 r_4 r_3 r_5 r_6
r_5 r_4 r_3 r_2 r_2 r_5 r_6 r_6
c_8 c_5 c_4 c_3 c_2 c_6 c_1 c_7
r_1

Then, the rightmost column of the failed partition refinement is c_7, the leftmost column of the failed partition refinement is c_8 and the column inside the gap is c_6. But c_7, c_8 and c_6 do not form an asteroidal triple of M since there is no path from c_7 to c_8 that does not contain r_6 and r_5 and $N(c_6) = \{c_6, r_5, r_6\}$.

Due to the fact that there is a finite number of well defined configurations for Tucker patterns, and that any non-C1P matrix contains such a pattern, using the structure of partition refinement seems at first to offer a promising way to attack the problem of finding Tucker patterns or asteroidal triples. Example 3.1.6, that is built on a simple extension of pattern M_{II}, shows however that the property of having a column within a gap as part of an asteroidal triple seems to hold, which we formalize in the following conjecture.

Conjecture 3.1.7. *Let M be a binary matrix that is non-C1P. If partition refinement fails at row r, then there exists at least one C_M-AT (x, y, z) such that one of the columns x, y, z is within a gap of the failed partition refinement.*

In the next section, we rely on this assumed property and explore the problem of finding a cycle using a gap of the failed partition refinement.

3.2 Extracting a cycle from a non-C1P binary matrix

In this section, we consider the general case where M is a binary matrix that is non-C1P and we present an algorithm that uses partition refinement to find, if possible, a Tucker pattern of type I (chordless cycle) in non-C1P binary matrices.

In this section, by *cycle* we mean a chordless cycle of length at least 6 in G_M, as these are the ones that coincide with definition of a Tucker pattern M_{I_k}, where a chord is an edge that links two non-consecutive vertices of the cycle. As we focus on using the structure obtained when the partition refinement technique fails, we assume without loss of generality that M is a connected non-C1P matrix with n rows $\{r_1, \ldots, r_n\}$ such that $\{r_1, \ldots, r_{n-1}\}$ is C1P.

The problem we consider is precisely the following: given a connected non-C1P binary matrix M, with rows $\{r_1, \ldots, r_n\}$ and such that $\{r_1, \ldots, r_{n-1}\}$ is C1P, decide if M contain a chordless cycle of length at least 6, and if it does, compute such a cycle. We consider two cases:

- **Problem EC-MIN**, where M contains a single gap (and is thus a minimal gap),
- **Problem EC-GENERAL** where gaps of M are not restricted.

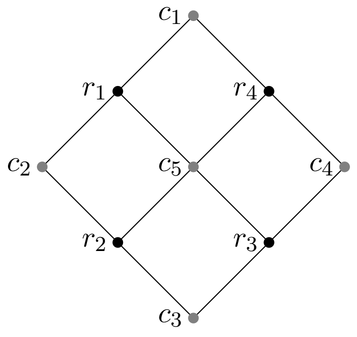
The motivation for studying this precise problem is twofold. First it deals with the, a priori, simplest family of Tucker patterns, as a chordless cycle is a very simple combinatorial structure. Second, there already exists algorithms for answering this problem in general binary matrices/bipartite graphs, that do not rely on partition refinement, and we are interested in looking at the gain we can obtain when using partition refinement. The existing algorithms are motivated by the recognition of *chordal bipartite* graphs (see [26], and especially references in Section 4), namely bipartite graphs without a chordless cycle of length at least 6. Extracting a chordless cycle of length at least 6 is thus a certificate for non chordal bipartite graphs. It can be done, in arbitrary bipartite graphs, using the notion of *doubly lexicographic ordering* in time $O((n+m)^2)$ or $O(n+m+e\log(n+m))$.

Notation 3.2.1. Let M be a non-C1P matrix with failed partition refinement $\{\mathcal{P} = (P_1 = \{S_1, R_1\}, \ldots, P_k = \{S_k, R_k\}), \mathcal{Q} = (Q_0 = \{0, C_0\}, Q_1 = \{j_1, C_1\}, \ldots, Q_p = \{j_p, C_p\})\}$. For a row r of $\{r_1, \ldots, r_{n-1}\}$, we denote by $le(r)$ the leftmost part of $\mathcal{P}$ whose label contains r and by $ri(r)$ the rightmost part of $\mathcal{P}$ whose label contains r.

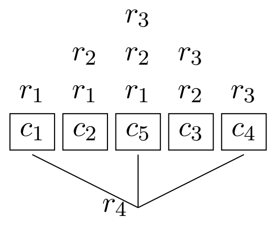
Example 3.2.2. Let M be the following matrix

$$\begin{array}{c|ccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 \\
\hline
r_1 & 1 & 1 & 0 & 0 & 1 \\
r_2 & 0 & 1 & 1 & 0 & 1 \\
r_3 & 0 & 0 & 1 & 1 & 1 \\
r_4 & 1 & 0 & 0 & 1 & 1 \\
\hline
\end{array}$$

with corresponding bipartite graph



Then, the failed partition refinement is $\{\mathcal{P} = (P_1 = \{c_1, r_1\}, P_2 = \{c_2, r_1, r_2\}, P_3 = \{c_5, r_1, r_2, r_3\}, P_4 = \{c_3, r_2, r_3\}, P_5 = \{c_4, r_3\}), \mathcal{Q} = (Q_0 = \{0, \emptyset\}, Q_1 = \{1, c_1\}, Q_2 = \{3, c_5\}), Q_3 = \{5, c_4\}\}$ and has the following structure



Then $le(r_2) = 2$ and $ri(r_2) = 4$.

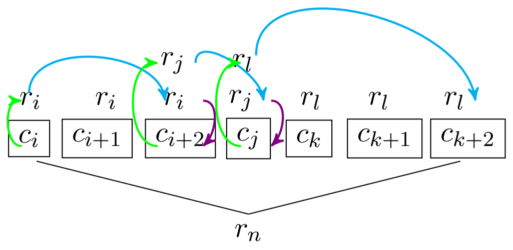
Now we present our algorithm for solving Problem EC-MIN: we look for a cycle that is composed of rows and columns that appear in the parts located between a gap, P_a and P_b, and in r_n also as this is the row creating the Tucker pattern. For the clarity of the presentation, we assume that $a = 1$ and $b = k$, or equivalently that we consider only the submatrix of M that is composed of columns that appear in the parts located between P_a and P_b (included) and of the rows with entries 1 in these columns. Given $\{a, b\}$, such a submatrix can obviously be extracted in $O(e)$ time, so this preprocessing does not impact the overall time complexity.

Definition 3.2.3. We say that a row r *spans* an interval (a, b) of a partition refinement if $r \in R_t$ for $a \leq t \leq b$.

In another preprocessing, we also remove all rows that span $(1,k)$ as such rows can not belong to the cycle we aim at finding without creating a chord. Again, this preprocessing can be done in linear time.

```
Algorithm 3.2.4. Finding Cycles using Partition Refinement
input: $\mathcal{P}, r_n$, such that $\{1,k\}$ is an inside gap and no row spans $(1,k)$.
  1. Let $j = 1$, $i = 1$, and $x_0$ be chosen arbitrarily among the elements of $S_1 \cap r_n$
  2. While $j \neq k$ do
       • Let $r \in R_j$ such that $ri(r)$ is maximum
       • If $ri(r) = j$ then return $\emptyset$ else $j = ri(r)$, $x_i = r$, $i = i + 1$
       • If $j < k$ let $x_i$ be chosen arbitrarily among $S_j$ else let $x_i$ be chosen arbitrarily among
         $S_k \cap r_n$; $i = i + 1$
  3. return $x_0, x_1, \ldots, x_i, r_k$
output: $x_0, x_1, \ldots, x_i, r_k$ or $\emptyset$
```

The general idea of algorithm 3.2.4 can be visualized with the following figure



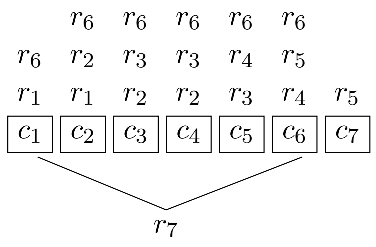
We can see that the algorithm alternates between selecting rows (green arrows) and columns (purple arrows), thus defining a cycle in the bipartite graph G_M. The algorithm jumps from a part on the left to a part on the right (blue arrows) to avoid picking a vertex that could create a chord.

Example 3.2.5. Let M be the binary matrix

$$\begin{array}{c|ccccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \\ \hline
r_1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
r_2 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
r_3 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
r_4 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
r_5 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
r_6 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
r_7 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline
\end{array}$$

M has two Tucker patterns of type I: $\{c_1, r_1, c_2, r_2, c_4, r_3, c_5, r_4, c_6, r_7\}$ and $\{c_1, r_1, c_2, r_2, c_3, r_3, c_5, r_4, c_6, r_7\}$.

The failed partition refinement has the structure


r_6 r_6 r_6 r_6 r_6
r_6 r_2 r_3 r_3 r_4 r_5
r_1 r_1 r_2 r_2 r_3 r_4 r_5
c_1 c_2 c_3 c_4 c_5 c_6 c_7
r_7

Note that the two cycles mentioned above are contained in the gap $(a = 1, b = 6)$. Using Algorithm 3.2.4 we can find a cycle between this gap. First, we remove r_6 from $\mathcal{P}$ since it spans (a, b). Then, we start to construct a cycle with vertex $x_0 = c_1$ and follow by choosing r_1 and c_2. Then, we choose r_2 and follow by choosing an element in the last part where r_2 belongs to, so we choose c_4. Continuing this process, the algorithm returns the cycle $\{c_1, r_1, c_2, r_2, c_4, r_3, c_5, r_4, c_6, r_7\}$.

Proposition 3.2.6. *If Algorithm 3.2.4 returns a non-empty sequence of vertices, then this sequence of vertices is a chordless cycle of length at least 6.*

Proof. Since the algorithm gives a sequence of vertices alternating between rows and columns, then the sequence is a cycle of even length in G_M. The cycle is of length at least 6 because if the cycle had length 4, (x_0, x_1, x_2, r_k), then x_1 would be a row that belongs to every R_t, for $1 \leq t \leq k$, which can not happen due to the preprocessing that discarded rows that span the whole gap.

Next we need to prove that this cycle is chordless. This follows from the fact that, for each row that belongs to the cycle, we include only two columns incident to this row, one from the part R_j the While loop starts in, then one from the last (furthest to the right) part that this row spans. Thus every row other than r_n in the cycle is incident to exactly two columns. Regarding r_n, by construction we select exactly two columns incident to it, in parts P_1 and P_k. So the cycle is chordless. □

Proposition 3.2.7. *If Algorithm 3.2.4 returns $\emptyset$, then there is no chordless cycle of length at least 6 in M.*

Proof. First, a chordless cycle can not include a row that span the whole gap, so the initial preprocessing that discarded these rows has no impact.

Now we can look at the case where the algorithm returns $\emptyset$. This happens only when there is no row in the label of the current part P_j that does contain columns located in the parts to the right of P_j, which implies that there are in fact two partitions, connected through r_n. Thus the columns in $S_1 \cap r_n$ and $S_k \cap r_n$ belong to different connected components in the bipartite graph associated to the gap minus the row r_n. There can then be no path between these two sets of elements, and thus no cycle containing the row r_n. □

Theorem 3.2.8. *Algorithm 3.2.4 solves Problem EC-MIN in time $O(e)$.*

Proof. The fact that the algorithm solves Problem EC-MIN follows from Proposition 3.2.6 and Proposition 3.2.7.

Regarding the time complexity, the main task of the Algorithm consists in finding, within the labels of the current part P_j, the row whose span goes the furthest to the right. Computing $ri(r_i)$ for all rows r_i can be done with a linear preprocessing time. Then, looking for the desired rows in the labels of the current parts can be done by a simple scan of the whole set of labels, with an amortized $O(e)$ time complexity.

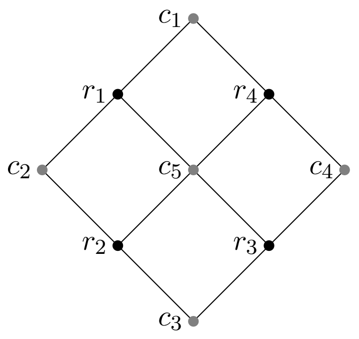
As discussed previously, the other preprocessing (removing rows spanning the gap) can be implemented easily in linear time. □

We now have an algorithm that is able to extract, if it exists, a cycle that is spanned by a given minimal gap. In order to solve Problem EC-GENERAL, one could first ask if, given a matrix with cycles, there is always one spanning a minimal gap. The answer to the question is no, as we can see in the following example.

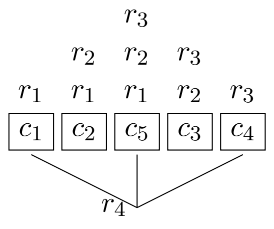
Example 3.2.9. Let M be the following matrix

$$\begin{array}{c|ccccc|}
 & c_1 & c_2 & c_3 & c_4 & c_5 \\
\hline
r_1 & 1 & 1 & 0 & 0 & 1 \\
r_2 & 0 & 1 & 1 & 0 & 1 \\
r_3 & 0 & 0 & 1 & 1 & 1 \\
r_4 & 1 & 0 & 0 & 1 & 1 \\
\hline
\end{array}$$

with corresponding bipartite graph



Then, partition refinement has the following structure



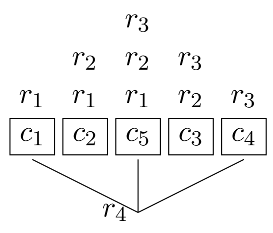
We can see that $(r_4, c_1, r_1, c_2, r_2, c_3, r_3, c_4)$ is a cycle and there is no minimal gap that defines it. Note that there is only one chordless cycle in M due to the position of c_5: since c_5 is adjacent to every row, all other cycles of length at least 6 in M are not chordless.

Next one could then ask in the case there is no cycle spanning a minimal gap, if applying our algorithm onto a non-minimal gap could be sufficient to solve Problem EC-GENERAL. The answer to this question is again no as shown by the next example, that reuses the matrix introduced in the previous example.

Example 3.2.10. Let M be the binary matrix defined in example 3.2.9

$$\begin{array}{c|ccccc|} & c_1 & c_2 & c_3 & c_4 & c_5 \\ \hline r_1 & 1 & 1 & 0 & 0 & 1 \\ r_2 & 0 & 1 & 1 & 0 & 1 \\ r_3 & 0 & 0 & 1 & 1 & 1 \\ r_4 & 1 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

with failed partition refinement



Recall that $(r_4, c_1, r_1, c_2, r_2, c_3, r_3, c_4)$ is the only chordless cycle of M of length at least 6.

Applying Algorithm 3.2.4 to all gaps of the failed partition refinement will give us:

- $a = 1, b = 3$: $\emptyset$, since there are no cycles contained in this gap
- $a = 3, b = 5$: $\emptyset$, since there are no cycles contained in this gap
- $a = 1, b = 5$: $c_1, r_1, c_5, r_3, c_4, r_4$, which contains a chord, due to the column c_5.

We can however observe that the problem in the previous example comes from the fact that, while trying to reach the right extremity of the larger gap, we "stop" onto a part that contains columns (here c_5) belonging to the row that caused the partition refinement to fail (r_n in the terminology of the algorithm), which creates a chord. So, when dealing with a non-minimal gap, we can in fact apply our algorithm, provided that we have removed these columns from the considered matrix. This leads to the following algorithm.

Algorithm 3.2.11. Finding Cycles using Partition Refinement

```
input: $\mathcal{P}, r_n$
  1. For each inside gap $\{a, b\}$ of $\mathcal{P}, r_n$ do
       • Remove from $\mathcal{P}$ all columns that belong to $r_n$ and are spanned by $\{a, b\}$
       • Remove from $\mathcal{P}$ all rows that span $\{a, b\}$
       • Let $R$ be the result of applying Algorithm 3.2.4 on gap $\{a, b\}$ of the resulting failed
         partition refinement.
       • If $R \neq \emptyset$ then return $R$
  2. return $\emptyset$
output: a chordless cycle of length at least 6 or $\emptyset$
```

Theorem 3.2.12. *Algorithm 3.2.11 solves Problem EC-GENERAL in time $O(ge)$, where g is the number of gaps of the failed partition refinement defined by $\mathcal{P}, r_n$.*

Proof. The correctness follows from two facts. First, if Algorithm 3.2.11 outputs a cycle, then it is a chordless cycle of length at least 6, for the same reasons as in the proof of correctness of Algorithm 3.2.4. Next, assume there is a chordless cycle of length at least 6. If it is in a minimal inside gap, then, from the correctness of Algorithm 3.2.4 we can assume that it will be found by Algorithm 3.2.11 when this gap is processed. So we can assume there is no cycle that spans a minimal inside gap. From the structure of the failed partition refinement for M_I, we know that every cycle spans an inside gap. So we may assume that our algorithm processes such a non-minimal inside gap $\{a, b\}$ that spans a cycle. No column belonging to r_n and spanned by $\{a, b\}$ can belong to the cycle, otherwise it creates a chord. So we can remove all these columns without destroying the property that the parts spanned by $\{a, b\}$ in the resulting failed partition refinement span a cycle, and thus Algorithm 3.2.4 will find this cycle.

The time complexity follows trivially from the time complexity of Algorithm 3.2.4 and the fact that we apply it onto each gap created by r_n. □

Note that, for a given r_n, the parameter g can belong to $O(n^2)$, so the time complexity of Algorithm 3.2.11 can also be stated as $O(n^2 e)$, in the sparse representation of the matrix, while the best known algorithms for computing a chordless cycle have time complexity $O((n+m)^2)$ and $O(n+m+e\log(n+m)))$.

3.3 Conclusion

To conclude this chapter, we will discuss two points: the result of the previous section and an outline of an alternative strategy we explored without conclusion.

First, by considering the problem of extracting a pattern M_I, we followed an approach that looks for ad-hoc algorithms for each of the five Tucker patterns, in special matrices that are precisely connected matrices that are C1P once their last row is discarded; this implies that our algorithm does decide if a general bipartite graph is chordal bipartite. However, considering such matrices does not reduce the generality of this approach when looking for Tucker patterns, as partition refinement on a non-C1P matrix always lead to such matrices, in linear $O(e)$ time. Focusing first on pattern M_I is interesting because it is the only pattern that is spanned, in the corresponding failed partition refinement, by a minimal gap, and also because algorithms already exist to solve this problem, albeit in general binary matrices. While the complexity analysis of our algorithm leads to an $O(ge)$ time complexity, it is an open question to see if it can be implemented more efficiently by avoiding repeating steps that are common while dealing with two different but overlapping non-minimal gaps. Compared to the best known algorithms for finding chordless cycles in general bipartite graphs, our algorithm has a worse time complexity, when expressed in terms of the parameters n, m, e, but it is interesting to have introduced the parameter g in the complexity statement. It however remains open to see if applying the known algorithms to the special graphs we consider leads to an improved time complexity.

More generally, the approach that looks for ad-hoc algorithms for each family of Tucker patterns seem difficult to extend. The main difficulty is that only few other patterns are spanned by a gap, and in fact several contain only a small gap. It remains open to see if the general approach that considers all possible gaps of r_n and tries to see if a given gap can be used as a seed to extract a given Tucker pattern (or more precisely a given failed partition refinement as described in Figures 3.1 to 3.5) can lead to algorithms to find a Tucker pattern in $O(ge)$.

Next, we outline a possible alternative approach that would avoid the drawback of relying on the precise structure of the failed partition refinement of Tucker patterns, and relies on the more generic notion of Minimal Conflicting Set. Minimal Conflicting Sets for the C1P were introduced in [4] as minimal sets of rows that were obstructing the C1P. It has since proved to be useful in several problems of computational biology involving the C1P [11, 41].

Definition 3.3.1. A subset $R = \{r_{i_1}, \ldots, r_{i_p}\}$ (resp. $C = \{c_{i_1}, \ldots, c_{i_q}\}$) is a *Minimal Conflicting Set for Rows* (MCS-R) (resp. Minimal Conflicting Set for Columns (MCS-C)) for M if it is not C1P but every proper subset of R (resp. C) is C1P.

The minimality of Tucker patterns immediately implies the following result, that was introduced in [13].

Lemma 3.3.2. *A submatrix M' of a non-C1P matrix M, defined by rows $R = \{r_{i_1}, \ldots, r_{i_p}\}$ and columns $C = \{c_{i_1}, \ldots, c_{i_q}\}$, is a Tucker pattern if and only if R is an MCS-R and C is an MCS-C.*

Following Lemma 3.3.2, we extract a Tucker pattern from M in two stages: we first extract a subset R of rows of M that is an MCS-R, then a subset C of the columns of R that is an MCS-C.

Algorithm 3.3.3. Finding Tucker Patterns using Minimum Conflicting Sets

```
input: $M = \{r_1, \ldots, r_n\}$, such that $M$ is non-C1P but $M - r_n$ is C1P.
  1. Let $T = \{r_1, \ldots, r_n\}$
  2. For $i = 1$ to $p$, if $T - \{r_i\}$ is non-C1P, then $T = T - \{r_i\}$.
  3. Let $C = \{c_1, \ldots, c_q\}$ be the support of $T$.
  4. For $i = 1$ to $q$, if $T - \{c_i\}$ is non- C1P, then $T = T - \{c_i\}$.
  5. Return $T$.
output: $T$, a binary matrix.
```

Proposition 3.3.4. *If M is a non-C1P binary matrix, the binary matrix T returned by Algorithm 3.3.3 is a Tucker pattern.*

Proof. If M is a non-C1P binary matrix, then the binary matrix T returned by Algorithm 3.3.3 is an MCS-R and MCS-C. Then, the result follows from Lemma 3.3.2. The algorithmic principle used in the loops in lines 2 and 4 to extract a minimal conflicting set was introduced in [11]. □

Proposition 3.3.5. *Algorithm 3.3.3 computes a Tucker pattern in time $O((n + m)e)$.*

Proof. If M is a non-C1P binary matrix, then Algorithm 3.3.3 reduces to $(n+m)$ C1P tests, that can each be done in $O(e)$ time. $\square$

The main interest of the general algorithmic scheme that underlies Algorithm 3.3.3 is that it is generic and does not rely on ad-hoc structural properties of the different families of Tucker patterns. The drawback is that a naive implementation has a quadratic worst-case time complexity.

We have tried to improve the time complexity using the structure of partition refinement. More precisely, for the problem of finding an MCS-R (questions and problems regarding MCS-C are similar), we have assumed, as in the previous sections, that we have a failed partition refinement $\mathcal{P}$, $\mathcal{Q}$ then looked at the following question, that addresses precisely the task done in the loop of line 2 of Algorithm 3.3.3.

Question 3.3.6. *Given an arbitrary row r, can we implement lines 2 and 4 of Algorithm 3.3.3 in amortized time $O(e)$?*

We were not able to provide a positive answer to this question. In fact we found examples that showed that removing a row from a failed partition refinement can remove all gaps, but at the price of a global modification of this structure, which implies that removing a row r from the current partition refinement can require more than $O(|r|)$ time. However, this does not prevent the existence of a specific order for considering rows that would lead to a linear time amortized complexity therefore conclude this chapter with this question, that, were it to be answered positively, would lead to a linear time algorithm to find a Tucker pattern in a non-C1P binary matrix.

Chapter 4

Enumerating all Tucker Patterns

We propose an output-sensitive algorithm for finding all Tucker patterns on a C1P matrix and use it to analyze some real data. The motivation is to identify all rows and columns that participate in a conflict. This information can be used to transform a non-C1P matrix into a C1P matrix or to detect traces of convergent evolution.

4.1 Theoretical Results

Below we describe an output-sensitive algorithm to find all Tucker patterns in a binary matrix. For Tucker patterns G_{M_I} the algorithm modifies the Read and Tarjan algorithm for finding cycles in a graph [37], so that it only finds chordless cycles. For Tucker patterns $G_{M_{II}}$ and $G_{M_{III}}$, the algorithm enumerates all possible triples of column vertices that could possibly be C_M-AT and then enumerates all ways to extend them into a Tucker pattern. Tucker patterns $G_{M_{IV}}$ and G_{M_V} are a modification of an exhaustive search algorithm.

Theorem 4.1.1. *Let M be a non-C1P binary matrix with m rows, n columns, e entries 1 and containing k Tucker patterns. All Tucker patterns of M can be enumerated in time $O((m^4+n^3m^2)(n+m)+e(k+1))$.*

The remainder of this section presents a proof of Theorem 4.1.1.

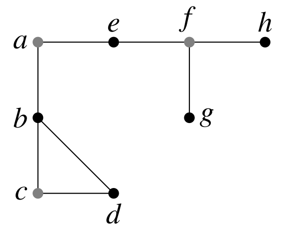
We denote by k_i the number of Tucker patterns of type G_{M_i}, for $i = I, II, III, IV, V$, and we describe, for each pattern, how to enumerate all its occurrences. Roughly speaking, pattern G_{M_I} can be enumerated using a simple variation on the classical cycle enumeration algorithm of [37], while patterns $G_{M_{II}}$ and $G_{M_{III}}$ can be enumerated by checking configurations of vertices for the vertices (or some subset) x, y, z, a, b, c, d (see Figure 2.3) and listing all paths between y and z. Patterns $G_{M_{IV}}$ and G_{M_V}, which are of bounded size, can be enumerated by a brute-force approach that requires time polynomial in the size of M.

Pattern G_{M_I}. To describe the algorithm for Tucker pattern G_{M_I} we begin by describing the algorithm by Read and Tarjan to enumerate cycles on a graph. We follow with a definition of chord and then we show how the Read and Tarjan algorithm is modified to find chordless cycles.

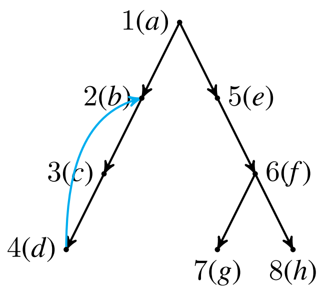
In [37], Read and Tarjan describe an algorithm that can enumerate all C cycles of G_M in time $O(n + m + e + eC)$.

The algorithm starts by using a depth first search to enumerate all vertices on the graph and label all edges of the graph. The vertices are labeled from 1 to $n + m$ in the order of visit of the depth first search. The edges are divided in two classes: a set of edges forming a directed rooted tree and a set of cycle edges, which point from a descendant to an ancestor in the tree. Note that tree edges are directed edges from a smaller to a larger numbered vertex, while cycle edges are directed edges from a larger to a smaller numbered vertex. We illustrate this process with the following example.

Example 4.1.2. Let G be the graph



Then, using depth first search we get the directed graph



where tree edges are black edges and cycle edges are blue edges. Using depth first search we have labeled the vertices of G in the order that they were visited.

Then, a set of cycle starting vertices is constructed, which will be used to start and extend a path that will form a cycle. The set of cycle starting vertices is the set of vertices with entering cycle edges. For each cycle starting vertex s, a recursive procedure extends the path that will form a cycle. The path P is extended from the last vertex of the path v to a vertex w if there is a path from w to s that avoids P. After the path is extended, the recursive procedure is applied: first, it checks if a cycle was completed and if not, it looks for a new vertex of extension.

Their algorithm can be described as follows:

Algorithm 4.1.3. Enumerate cycles (G) [37]

input: Graph G

```
1. DFS$(G)$
2. For all connected components of $G$
        For all starting vertices $s$ of $G$
            $P = \{s\}$
            Enumerate cycles recursive $(G, P)$
```

output: All cycles of G

Algorithm 4.1.4. Enumerate cycles recursive (G, P) [37]

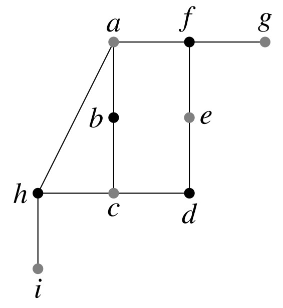
input: Graph G, path P in G

```
1. Let $v$ be the last vertex in $P$ and $s$ the first vertex in $P$
2. If $\{v, s\}$ is an edge of $G$ then output $P$
3. Else
        For each outgoing arc $w$ of $v$
            If there exists a path from $w$ to $s$ that avoids $P$ then
                Add $w$ to $P$
                Enumerate cycles recursive$(G, P)$
```

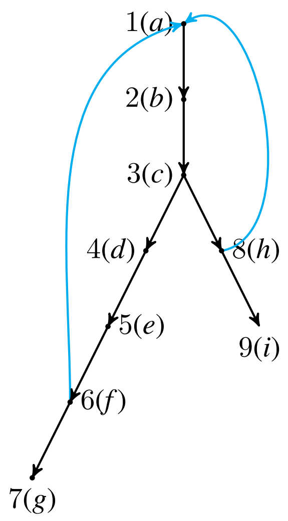
In the above algorithms, DFS is a depth first search algorithm for computing a spanning tree of G that labels the vertices according to the order they are visited, identifies starting vertices and partitions edges into tree edges and cycle edges.

We will illustrate Read and Tarjan algorithm for finding cycles on a graph with an example.

Example 4.1.5. Let G be the graph



Then, using depth first search we get the directed graph



The list of starting vertices is $\{1(a)\}$. Then, we start our path as $P = \{1(a)\}$ with $s = 1(a)$. We extend the path with a vertex w if there is a path from w to s that avoids P. Then, we add vertices $2(b)$ and $3(c)$ to P since there is a path from each of them to s. So far, $P = \{1(a), 2(b), 3(c)\}$. Now we look for the next vertex that we can use to extend the path. In this case, there are two possible extensions, namely $4(d)$ and $8(h)$. First, we consider adding $4(d)$ to P. We continue to extend the path by adding $5(e)$ and $6(f)$. Since $\{6(f), 1(a)\}$ is an edge in G, we have completed a cycle, and the algorithm outputs the cycle $C_1 = \{1(a), 2(b), 3(c), 4(d), 5(e), 6(f), 1(a)\}$. Now we look at the other possibility, which was adding vertex $8(h)$ instead of $4(d)$. In this case, we continue extending the path by adding $1(a)$. Since $\{8(h), 1(a)\}$ is an edge in G, then we have completed another cycle and enumerate it, $C_2 = \{1(a), 2(b), 3(c), 8(h), 1(a)\}$.

Note that the algorithm also looks at edges $7(g)$ and $9(i)$, but since there are no further neighbours of $7g$ and $9(i)$, respectively, then the algorithm returns without output.

Now, we discuss the complexity of Read and Tarjan algorithm for finding cycles in a graph $G = (V, E)$, where e is the number of edges, n is the number of vertices and C is the number of cycles. The time required to do depth first search and label the edges and vertices is $O(n+e)$. As we assume the graph is connected, we have that $e \geq n-1$. Within the recursive function, if one leaves aside the recursive calls, the time complexity is dominated by the search for a path from the neighbours of v to the starting vertex s of the current path and that avoids this path. This can be done in $O(e)$ time by a classical DFS for example, where (1) vertices visited when processing a given neighbour

w of v are marked as belonging or not to such path, avoiding thus that an edge will be visited several times, and (2) vertices belonging to the current path are initially marked as not belonging to a path ending in s. Then, the total number of recursive calls is at most $C + e$, where C corresponds to the number of enumerated cycles and e corresponds to the number of explored dead ends .

Therefore, the complexity of Algorithm 4.1.3 is $O(n + e + e(C + |E|))$.

Note that if we apply Algorithm 4.1.3 to a binary matrix, then the corresponding complexity is $O(n + m + e + e(c + e))$, where m is the number of rows, n is the number of columns, e is the number of entries equal to 1, and c is the number of chordless cycles of length at least 6.

We will now show how to modify this algorithm for enumerating chordless cycles.

Definition 4.1.6. A *chord* is a single edge linking two non-consecutive vertices of the cycle. A *chordless cycle* is a graph of length at least 4 with no chords.

Example 4.1.7. Figures 4.1 and 4.2 illustrate two 8-cycles. The cycle on the left has a chord from vertex 4 to vertex 7. The cycle on the right is chordless since it does not have any chords.

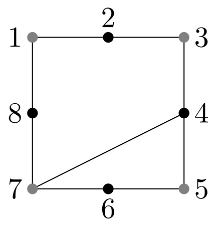


Figure 4.1: A cycle with a chord from vertex 4 to vertex 7.

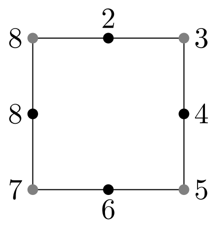


Figure 4.2: A chordless cycle.

Occurrences of the pattern G_{M_I} are exactly all chordless cycles of length at least 6 of G_M. The Read and Tarjan algorithm for finding cycles on a graph can easily be modified to avoid computing the cycles with chords. The modified algorithm has another restriction before extending the current path, that can be described as follows: let v be the last vertex on the current path P. The restriction is that when extending P from v to a new vertex w we first remove all vertices t that are adjacent to a vertex u of P and then look for w such that there is a path from w to s that avoids P. This restriction ensures that w is not adjacent to any vertex of P and therefore that there are no chords on the cycle. Below we describe the algorithm for finding all chordless cycles in a graph.

Algorithm 4.1.8. Enumerate chordless cycles (G)

input: Graph G

```
1. DFS($G$)
2. For all connected components of $G$
        For all starting vertices $s$ of $G$
            $P = \{s\}$
            Enumerate chordless cycles recursive $(G, P)$
```

output: All cycles of G

Algorithm 4.1.9. Enumerate chordless cycles recursive (G, P)

input: Graph G, path P in G

```
1. Let $v$ be the last vertex in $P$ and $s$ the first vertex in $P$
2. If $\{v, s\}$ is an edge of $G$ then output $P$
3. Else
            For each outgoing arc $w$ of $v$
                Remove all vertices $t$ that are adjacent to a vertex $u$ of $P$
                If there exists a path from $w$ to $s$ that avoids $P$ then
                    Add $w$ to $P$
                    Enumerate cycles recursive$(G, P)$
```

Figures 4.3 and 4.4 illustrate the difference between the modified algorithm to find chordless cycles and the original algorithm of Read and Tarjan. Figure 4.3 represents a cycle that can be returned by Read and Tarjan's algorithm, where w can be adjacent to a vertex w_2 while on the second figure (4.4) w can not be adjacent to any vertex in P and the returned cycle is chordless.

Remark 4.1.10. Algorithm 4.1.8 can be used to find a path between a and b by finding a cycle with starting vertex a (instead of s) and looking if there is path from w to b (instead of s) to extend the path.

Note that removing vertices that are adjacent to a vertex of P can be done in $O(e)$ time as it reduces to looking at all edges of the graph, and thus does not increase the complexity of checking if the current path can be extended. We can then enumerate all chordless cycles of length at least 6 in time

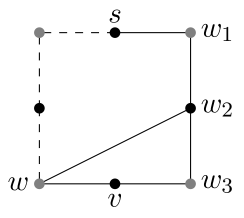


Figure 4.3: A cycle with a chord returned by Read and Tarjan's algorithm.

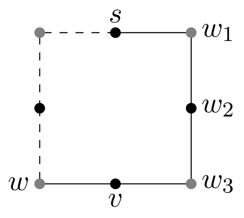


Figure 4.4: A chordless cycle returned by the modified version of Read and Tarjan's alforithm.

$O(n + m + e(1 + c + e))$.

Proposition 4.1.11. *Let M be a non-C1P binary matrix with m rows, n columns, e entries 1 and containing k_I Tucker patterns of type I, G_{M_I}. Then, all Tucker patterns G_{M_I} can be enumerated with Algorithm 4.1.8 in time $O(n + m + e(k_I + e + 1))$.*

Proof. Read and Tarjan proved that the complexity of enumerating all C cycles in a graph is $O(n + m + e(C + e + 1))$ [37]. Then, we only need to show that the complexity remains the same with the new restriction added to find only chordless cycles in a graph. The complexity of removing all vertices t that are adjacent to a vertex u of P is $O(e)$, since it reduces to the complexity of looking at all edges of the graph. Then, the time needed to enumerate all chordless cycles in a graph is $O(n + m + e(k_I + e + 1))$, where k_I is the number of chordless cycles in G_{M_I}. □

Pattern $G_{M_{II}}$. Figure 4.6 illustrates the bipartite graph of Tucker pattern $G_{M_{II}}$, and Figure 4.5 illustrates the steps involved in enumerating Tucker pattern $G_{M_{II}}$. To enumerate all occurrences of pattern $G_{M_{II}}$, we begin by enumerating all possible sets of two row vertices $\{a, b\}$, in time $O(n^2)$ (orange vertices in Figure 4.5). Then, we compute the set $V_{a,b}$ of column vertices that are adjacent to both a and b (green vertices on Figures 4.5 and 4.6), in time $O(m)$ by simple comparison of both sets associated to a and b to select common elements. We also pick x from $V_{a,b}$ arbitrarily (orange vertex in Figure 4.5). For a given (a, b, x), we pick y as an element in the neighborhood of a that is not in $V_{a,b}$ and similarly we pick z as an element in the neighborhood of b that is not in $V_{a,b}$. We can pick y and z in time $O(m^2)$ (orange vertices in Figure 4.5). Then, we compute the set $V_{y,z}$ of row vertices that are adjacent to either y (resp. z) and a vertex of $V_{a,b}$, or at least two vertices of $V_{a,b}$ (blue vertices on Figures 4.5 and 4.6), but not x, in time $O(n)$. Then, to enumerate all patterns $G_{M_{II}}$, we only need to enumerate all chordless paths between y and z using only vertices from $V_{a,b} \cup V_{y,z}$ (dashed red path in Figure 4.5). This can be achieved in time $O(n+m+e(k_{II}+1))$ using Read and Tarjan's algorithm as modified above. The total complexity is then $O(n^2m^3(n+m)+e(k_{II}+1))$.

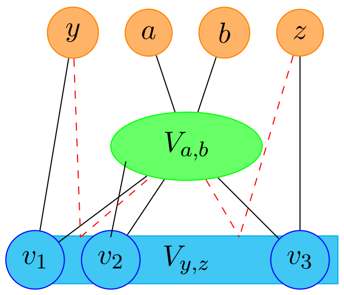


Figure 4.5: Enumerating Tucker pattern $G_{M_{II}}$: Enumerate the vertices a, b (in orange), then find the green set $V_{a,b}$ of vertices adjacent to both a and b. Later, pick x, y and z (orange vertices). Then, find the blue set $V_{y,z}$ of vertices adjacent to either two vertices in $V_{a,b}$, or adjacent to y and a vertex in $V_{a,b}$ or adjacent to z and a vertex in $V_{a,b}$. Finally, find a path between y and z.

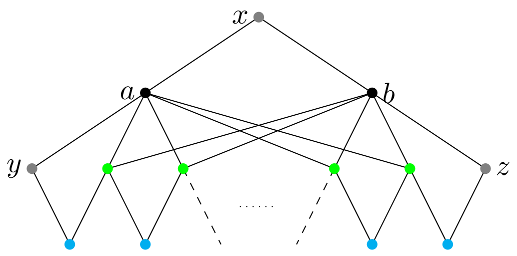


Figure 4.6: Tucker pattern $G_{M_{II}}$

Pattern $G_{M_{III}}$. Figure 4.8 illustrates the bipartite graph of Tucker pattern $G_{M_{III}}$, and Figure 4.7 illustrates the steps involved in enumerating Tucker pattern $G_{M_{III}}$. Occurrences of pattern $G_{M_{III}}$ can be enumerated in a manner similar to pattern $G_{M_{II}}$. The main difference is that the initial step requires to enumerate all possible row vertex $\{a\}$, instead of two row vertices $\{a, b\}$, and can then be done in time $O(n)$ (orange vertex in Figure 4.7).

Then, we compute the set V_a of column vertices that are adjacent to a (green vertices on Figures 4.7 and 4.8), in time $O(m)$ by simple comparison of the set associated to a. We also pick x from V_a

arbitrary (orange vertex in Figure 4.7). For a given (a, x), we pick y and z as elements that are not in V_a, in time $O(m^2)$ (orange vertices in Figure 4.7). Then, we compute the set $V_{y,z}$ of row vertices that are adjacent to either y (resp. z) and a vertex of V_a, or at least two vertices of V_a (blue vertices on Figure 4.8), but not x. As this set of vertices is defined by bounded size subgraphs of G_M, they it can be computed in time $O(m)$. Then, to enumerate all patterns $G_{M_{III}}$, we only need to enumerate all chordless paths between y and z using only vertices from $V_a \cup V_{y,z}$ (dashed red path in Figure 4.7). This can be achieved in time $O(n + m + e(k_{III} + 1))$ using Read and Tarjan's algorithm as modified above. The total complexity is then $O(nm^3(n + m) + e(k_{III} + 1))$.

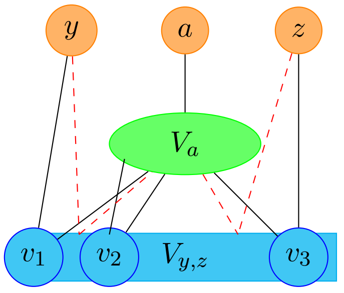


Figure 4.7: Enumerating Tucker pattern $G_{M_{III}}$: Enumerate the vertex a (in orange), then find the green set V_a of vertices adjacent to a. Later, we pick x from V_a arbitrarily and y and z not in V_a. Then, find the blue set $V_{y,z}$ of vertices adjacent to either two vertices in $V_{a,b}$, y and a vertex in $V_{a,b}$ or z and a vertex in $V_{a,b}$. Finally, find a path between y and z.

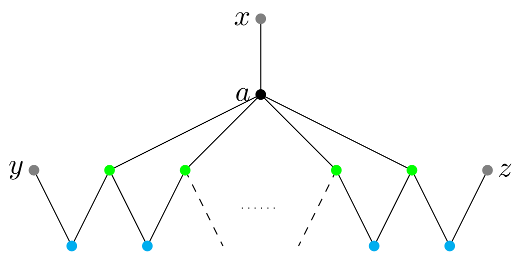


Figure 4.8: Tucker pattern $G_{M_{III}}$

Pattern $G_{M_{IV}}$. To enumerate all occurrences of pattern $G_{M_{IV}}$ more efficiently than checking all configurations of 4 row vertices and 6 column vertices, we start by enumerating only the row vertices $\{a, b, c, d\}$, in time $O(m^4)$. Then, all potential sets of column vertices compatible with this quadruplet (denoted by V_x for possible vertices x, V_y, V_z, V_u, V_v and V_w being defined similarity) are computed, in time $O(n)$ as they are pairwise disjoint. It is then sufficient to list all $(x, y, z, u, v, w) \in V_x \times V_y \cdots \times V_w$, as each defines an occurrence of a Tucker pattern $G_{M_{IV}}$. This can then be done in time $O(m^4 n + k_{IV})$.

Pattern G_{M_V}. Pattern G_{M_V} can be enumerated in a similar way, starting from all possible quadruples $\{a, b, c, d\}$ of row vertices, in time $O(m^4 n + k_{IV})$.

Summing the complexities of these five steps, we then obtain the complexity stated in Theorem 4.1.1.

4.2 Analyzing Real Data

In this section we apply the main theorem for enumerating Tucker patterns, described in section 4.1 to a binary matrix that resulted from reconstructing the ancestral genome of a set of several mammalian amniotes species (mammalians, marsupials and avian species).

Recall that we analyzed the data from the set of genomes of the phylogenetic tree, consisting of the following species: Homo sapiens (humans), Pan troglodytes (chimpanzee), Pongo pygmaeus (orangutan), Macaca mulatta (monkey), Mus musculus (mouse), Rattus norvegicus (rat), Equus caballus (horse), Canis familiaris (dog), Bos taurus (cows), Monodelphis domestica (opossum), Gallus gallus (chicken), and Taeniopygia guttata (zebra finch). The authors wanted to reconstruct the genome of the common ancestor of amniote genomes species, that is mammals, marsupials and avian genomes species [35]. Figure 4.9 represents the phylogenetic tree of this set of species. Each node is labeled according to the species that it corresponds to. The species that is being reconstructed is labeled as "ancestor".

From this set of data the authors constructed a binary matrix M, of size 1861×1546, where each column represents a marker and every row represents an ancestral contiguous set (ACS), that is, a set of ancestral markers that are believed to be contiguous in the ancestor species. As we apply the principles described in the introduction to find ancestral syntenies, every row is a group of markers conserved in both a mammalian/marsupial genome and a bird genome. This is a property of the matrix. What is interesting is that the bird genomes are well known to be quite stable [17]: it can be seen in Accessory figure 3 of [17] that there are no inter-chromosomal rearrangements (that is, ex-

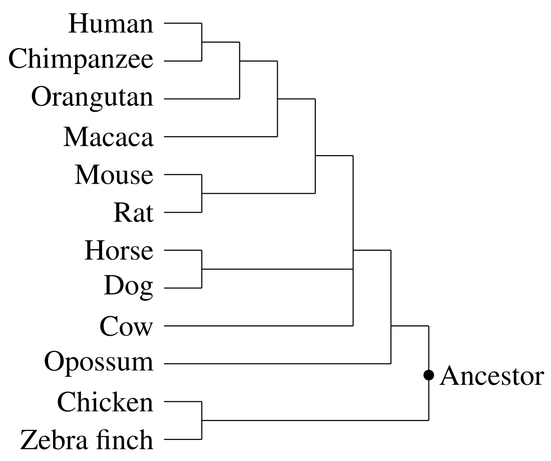


Figure 4.9: [12] Phylogenetic tree of mammals, marsupials and avian genomes. The common ancestor of amniote genomes is being illustrated and reconstructed.

change of genetic material between chromosomes since the divergence of the last common ancestor of chicken and zebra finch), but a few intra-chromosomal rearrangements (that is, rearrangements within a chromosome). Hence one can expect a matrix with low conflict [35] and we can have a look at the precise causes of the few observed conflicts.

We constructed the overlap graph G_M of M, that consists of roughly 160 connected components. There were 5 connected components that contained conflicts. We analyzed those five connected components, each representing a submatrix M_i, for $i = 1, \ldots, 5$, respectively. In this way, we divided M into five smaller matrices, M_1, M_2, M_3, M_4 and M_5, that together form M.

The implementation of the algorithm was written with the help of Brad Jones (funded by an SFU VPR USRA, in the summer 2011).

We analyzed each of these five matrices and found for each of the matrices the number of Tucker patterns, the possible rows and columns that could be removed so that the matrix is C1P, and the possible rows that could be result of convergent evolution.

From an evolutionary point of view, the notion of convergent evolution is as follows. Let A be an ancestor, with two branches leaving towards descendants, that are then split into two groups D_1 and D_2 (the nodes of the two subtrees whose roots are the children of A). A character results from convergent evolution if it was not ancestral but appeared independently along two branches, one within D_1 and one within D_2. A mathematical definition/algorithm is to detect groups of genes that appear along two branches of the tree. It can be misleading, since technically, convergent evolution can also include a character that was not ancestral but appeared along the two branches leaving A for D_1 and D_2 (and thus is conserved in all species), that is, very high in the tree; this is a situation that is very unlikely but could happen in the true evolution. Below we present the results obtained.

	Size	No. G_{M_I}	No. $G_{M_{II}}$	No. $G_{M_{III}}$	No. $G_{M_{IV}}$	No. G_{M_V}
M_1	21×17	40	16	0	0	40
M_2	25×19	29	1	42	68	21
M_3	28×22	1113	12	169	0	0
M_4	6×6	0	0	1	0	0
M_5	11×9	3	0	0	0	0

Table 4.1: Number of Tucker patterns of each type in each of the five matrices.

Matrix M_1. The following tables summarize the number of Tucker patterns each row belongs to and the number of rows each Tucker pattern contains for the first matrix M_1.

	No. G_{M_I}	No. $G_{M_{II}}$	No. $G_{M_{III}}$	No. $G_{M_{IV}}$	No. G_{M_V}
r_0	0	16	0	0	40
r_1	0	16	0	0	0
r_2	40	16	0	0	40
r_3	8	0	0	0	8
r_5	32	0	0	0	32
r_9	40	0	0	0	0
r_{10}	40	16	0	0	40

Table 4.2: Number of Tucker patterns each row belongs to in M_1. Only rows that belong to at least one Tucker pattern are shown.

No. rows	No. G_{M_I}	No. $G_{M_{II}}$	No. $G_{M_{III}}$	No. $G_{M_{IV}}$	No. G_{M_V}
4	40	16	0	0	40

Table 4.3: Number of Tucker patterns in M_1 with number of rows as specified on the first column of the table. That is, there are 40 G_{M_I} with 4 rows as well as 16 $G_{M_{II}}$ with 4 rows.

From Table 4.3 we can see that all Tucker patterns in M_1 contain 4 rows.

Below is a list of rows and their corresponding markers and a list of marker orders in each species.

Row	Markers
0	1 2 3 4 5 6 7 8
1	7 8 9
2	6 7 8 9
3	5 6 or -6 -5
4	6
5	5 6
6	4 5 or -5 -4
7	5
8	4
9	1 2 3 4 5
10	2 3 8
11	1 2 or -2 -1
12	2
13	2 3 or -3 -2
14	3
15	1
16	1 2 3
17	7 8 or -8 -7
18	8
19	7
20	4 5

Table 4.4: List of rows and their corresponding markers in M_1.

Bos taurus	chr2: $< 31 > -9\$$ chr16: $< 10 > -7 - 6 - 5 - 4 - 3 - 2 - 1 - 8 < 6 > \$$
Canis familiaris	chr2: $< 23 > -9 - 8 - 7 - 6\$$ chr5: $< 20 >$ 1 2 3 4 5 $< 15 > \$$
Equus caballus	chr2: $< 12 > -9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 < 23 > \$$
Gallus gallus	chr21:$-3 - 2 - 1$ 4 5 6 $- 8 - 7 - 9\$$
Homo sapiens	chr1: 1 2 3 4 5 6 7 8 9 $< 48 > \$$
Macaca mulatta	chr1: $< 1 >$ 1 2 3 4 5 6 7 8 9 $< 48 > \$$
Monodelphis domestica	chr2: $< 19 > 1 < 97 > \$$ chr4: $< 76 > -9 < 2 > -7 < 1 > -6 - 5$ $2 - 8$ $3 < 6 > -4\$$
Mus musculus	chr4: $< 40 > -9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1\$$
Pan troglodytes	chr1: 1 2 3 4 5 6 7 8 9 $< 48 > \$$
Pongo pygmaeus	chr1:$< 48 > -9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1\$$
Rattus norvegicus	chr5: $< 44 > -9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1\$$
Taeniopygia guttata	chr21: 9 7 8 $- 3 - 2 - 1 - 6 - 5 - 4\$$

Table 4.5: List of marker orders in each species.

We found that there are two rows, row 2 and row 10, that are present in all Tucker patterns. Then, removing any of these two rows will eliminate the conflict, making the matrix C1P. We look now at the question of understanding the conflict we observe: which of these two rows could be considered as resulting from convergent evolution ?

Table 4.4 gives the markers that each row corresponds to. Row 2 corresponds to markers 6 7 8 9 and row 10 to 2 3 8. Table 4.5 gives the markers in each species. Combining these two tables we can see which rows are present in each species. At first it might look like row 10, that is present in only two species (the zebra finch and the opossum) is the most obvious candidate for being an example of convergent evolution. However, this is not so obvious. Indeed if we consider it was an ancestral character, then it has been lost along two branches, the one leading to the chicken and the one leading to the mammalian ancestor; conversely, if we consider it was not ancestral, then it has been gained independently along two branches (zebra finch and opossum). The same can be said about row 2: it could have been lost in the zebra finch and opossum branches or gained in the chicken and mammalian branch. In the present case however, the most likely convergent evolution is row 10 because of two arguments: it is shorter than row 2 and thus more likely to result from independent rearrangements along two branches and the opossum genome is known to have evolved in a very different way than other eutherian genomes (C. Chauve, personal communication).

Hence, we can see that the almost constant nature of both rows in the mammalian genomes leads us to consider in fact a species tree with only four leaves (chicken, zebra finch, opossum, mammalian ancestor), where detecting convergent evolution is not easy. The situation is rendered harder by an unbalanced taxonomic sampling that creates two very long branches (to the birds and the opossum).

Matrices M_2 and M_3. The following tables summarize the results for the second matrix M_2.

	No. $G_{M_{II}}$	No. $G_{M_{III}}$	No. $G_{M_{IV}}$	No. G_{M_V}
r_0	12	72	0	0
r_1	0	169	0	0
r_2	0	3	0	0
r_3	0	36	0	0
r_4	12	169	0	0
r_5	0	1	0	0
r_8	0	3	0	0
r_9	60	30	0	0
r_{10}	0	12	0	0
r_{11}	0	12	0	0
r_{13}	6	30	0	0
r_{14}	0	24	0	0
r_{15}	6	42	0	0
r_{16}	0	27	0	0
r_{21}	6	63	0	0
r_{22}	0	3	0	0
r_{23}	0	6	0	0
r_{27}	0	12	0	0

Table 4.6: The table illustrates the number of Tucker patterns each row belongs to in M_2.

No. rows	No. $G_{M_{II}}$	No. $G_{M_{III}}$	No. $G_{M_{IV}}$	No. G_{M_V}
3	0	4	0	0
4	12	132	0	0
5	0	24	0	0
6	0	9	0	0

Table 4.7: The table illustrates the number of Tucker patterns in M_2 with number of rows as specified on the first column of the table.

Table 4.7 says that all Tucker patterns in M_2 are small, since they only contain 3, 4, 5 or 6 rows.

We analyzed the rows that could be removed from M_2 to eliminate the conflict and therefore allow M_2 to be C1P. We found that we need to remove two rows from M_2 so that it is C1P. We can remove rows 0 and 1, or rows 0 and 4.

When looking more carefully at this matrix, it appears clearly (see Appendix) that the conflict is created by the mixing of two sets of markers $(425 \dots 428; 164 \dots 177)$. Again, deciding whether they were mixed in the ancestor and were separated in two well defined groups along the mammalian branch (then separated again along the rodents branch, which is not significant as rodent genomes are known to have an increased rate of genome rearrangements), or were in two separate groups in the amniote ancestor and then got mixed along the birds, rodents and opossum lineages is difficult, and it is difficult to detect clear convergent evolution signals.

The following tables summarize the results for the third matrix M_3.

	No. G_{M_I}	No. $G_{M_{II}}$	No. $G_{M_{III}}$	No. $G_{M_{IV}}$	No. G_{M_V}
r_0	3	0	0	0	0
r_1	3	0	0	0	0
r_7	1	0	0	0	0
r_8	1	0	0	0	0
r_9	2	0	0	0	0

Table 4.8: The table illustrates the number of Tucker patterns each row belongs to in M_3.

No. rows	No. G_{M_I}	No. $G_{M_{II}}$	No. $G_{M_{III}}$	No. $G_{M_{IV}}$	No. G_{M_V}
3	2	0	0	0	0
4	1	0	0	0	0

Table 4.9: The table illustrates the number of Tucker patterns in M_3 with number of rows as specified on the first column of the table.

We analyzed the rows and columns that could be removed from M_3 to eliminate the conflict and therefore allow M_3 to be C1P. We found that there are two rows that are present in all Tucker patters:

row 0 and row 1. Therefore, by removing row 0 or row 1 M_3 is C1P. Row 0 is only present in two species (Canis familliaris and Taeniopygia guttata) while row 1 is only present in Gallus gallus and Monodelphis domestica. Thus it is more likely that row 0 results from convergent evolution, and is the most obvious choice for being discarded.

4.3 Conclusion

In this chapter we presented an algorithm that enumerates all Tucker patterns on a binary matrix in polynomial time on the size of the output. Also, we used the algorithm to analyze some real data. We applied the algorithm to a binary matrix that resulted from reconstructing the ancestral genome of a set of mammalian amniotes. Our results show that, at least on the considered dataset, enumerating all Tucker patterns is a good strategy to suggest possible rows to remove to make a matrix C1P, as in most cases these rows belong to a lot of Tucker patterns. This is consistent with previous results that were considering Minimum Conflicting Sets [11], [41], although our approach has the advantage of being output-sensitive. It was also interesting to notice that in most cases, Tucker patterns are small, which might suggest that, as an alternative to the exhaustive generation, generating all small Tucker patterns is an approach that is worthwhile to be explored, in particular the problem of output-sensitive generation, as an alternative to the brute-force approach that would enumerate all subsets of rows of bounded cardinality. Finally, the main points of the analysis of the amniote dataset are twofold: first the taxonomic structure of the considered species makes it difficult to detect traces of convergent evolution, and second, the conflicts appear with relatively small groups of markers that are subject to some pressure to be conserved contiguous in several species but not to a strong pressure to be maintained in the same order, which results in numerous small-scale rearrangements that are the cause of convergent evolution. This raises the interesting question, that is far beyond the scope of this document, to develop computational approaches to understand the evolution of such genome segments.

Appendix A

Appendix

We present here the corresponding markers and a list of marker orders for each species for each matrix.

Matrix M_2. Below is a list of rows and their corresponding markers and a list of marker orders in each species.

Row	Markers
0	165 164 166 167 168 169 171 172 173 174 425 426 427 428 170
1	426 176 or -176 -426
2	426
3	166 167 165 173 174 175
4	175 176 or -176 -175
5	176
6	176 -177 or 177 -176
7	177
8	176 177
9	176 177 175
10	174 175 or -175 -174
11	175
12	174
13	176 177 164 425 426
14	175 174 (unordered)
15	176 177 174 175
16	166 174 165 173
17	166 167 or -167 166
18	167
19	166
20	166 167
21	176 177 426
22	425 426 or -426 -425
23	425 426
24	164 425 or -425 -164
25	425
26	164
27	164 425 426

Table A.1: List of rows and their corresponding markers in M_2.

Bos taurus	chr17: < 12 > −177 − 176 − 175 − 174 − 173 165 166 − 170 − 169 − 168 − 167 171 172 − 164 425 426 427 428\$
Canis familiaris	chr26: −177 − 176 − 175 − 174 − 173 165 166 167 168 169 170 171 172 − 164 425 426 427 428 < 1 > \$
Equus caballus	chr8: −428 − 427 − 426 − 425 − 164 − 172 − 171 − 170 − 169 − 168 − 167 − 166 − 165 173 174 175 176 177 < 17 > \$
Gallus gallus	chr15: 177 − 176 − 175 − 174 − 165 173 166 167 164 425 426 − 428 − 172 − 171 − 170 − 427 − 169 − 168\$
Homo sapiens	chr12: < 33 > 164 165 166 167 168 169 170 171 172 173 174 175 176 177\$ chr22: < 1 > 425 426 427 428 < 10 > \$
Macaca mulatta	chr10: < 26 > 425 426 427 428 < 10 > \$ chr11: < 38 > 164 165 166 167 168 169 170 171 172 173 174 175 176 177\$
Monodelphis domestica	chr3: < 64 > 177 − 176 − 175 < 1 > −166 − 165 173 174 − 164 − 172 − 168 − 167 425 169 170 171 428 426 427\$
Mus musculus	chr5: < 31 > −177 − 176 − 426 − 425 164 − 172 − 171 − 170 − 169 − 168 − 167 − 166 − 165 173 174 175 < 13 > \$ chr11: −428 − 427 < 56 > \$
Pan troglodytes	chr12: < 33 > 164 165 166 167 168 169 170 171 172 173 174 175 176 177\$ chr22: < 1 > 425 426 427 428 < 10 > \$
Pongo pygmaeus	chr12: < 33 > 164 165 166 167 168 169 170 171 172 173 174 175 176 177\$ chr22: < 1 > 425 426 427 428 < 10 > \$
Rattus norvegicus	chr12: < 13 > −175 − 174 − 173 165 166 167 168 169 170 171 172 − 164 426 176 177\$ chr14: < 21 > −428 − 427 < 11 > \$ chr19: < 4 > −425 < 22 > \$
Taeniopygia guttata	chr15: −175 − 174 − 165 173 166 167 164 425 426 176 − 177 − 168 170 171 172 428 169 427\$

Table A.2: List of marker orders in each species.

Matrix M_3. Below is a list of rows and their corresponding markers and a list of marker orders in each species.

Row	Markers
0	445 454 482 2574
1	455 482 or -455 -482
2	482
3	445 482 or -482 -455
4	445 2574 or -2574 -445
5	445
6	445 482 2574
7	454 455 or -455 -454
8	455
9	454
10	454 455

Table A.3: List of rows and their corresponding markers in M_3.

List of marker orders in each species:

Bos taurus	chr1: < 37 > −482 < 4 > 445 446 447$
	chr22: 449 < 4 > 454 < 10 > −455 < 8 > $
	chr27: < 8 > 448$
Canis familiaris	chr20: < 18 > −455 < 3 > $
	chr23: < 4 > 454 − 449 − 448 − 447 − 446 − 445 482 < 6 > $
Equus caballus	chr16: < 18 > −455 − 454 < 4 > −449 − 448 − 447 − 446 − 445 482 < 5 > $
Gallus gallus	chr2: < 23 > 446 447 448 449 − 445 482 − 455 454 < 62 > $
Homo sapiens	chr3: < 6 > 445 446 447 448 449 < 4 > 454 455 < 26 > 482 < 16 > $
Macaca mulatta	chr2: < 27 > −455 − 454 < 17 > −482 445 446 447 448 449 < 3 > $
Monodelphis domestica	chr8: < 80 > −454 455 − 482 445 446 447 448 449 < 6 > $
Mus musculus	chr9: < 36 > −482 < 6 > −449 < 2 > 454 455$
	chr14: < 1 > −448 < 6 > 445 446 < 21 > $
	chr17: < 19 > 447 < 16 > $
Pan troglodytes	chr3: < 6 > 445 446 447 448 449 < 4 > 454 455 < 26 > 482 < 16 > $
Pongo pygmaeus	chr3: < 32 > −455 − 454 < 4 > −449 − 448 − 447 − 446 − 445 482 < 16 > $
Rattus norvegicus	chr8: < 36 > −482 < 6 > −449 < 2 > 454 455$
	chr9: < 1 > −447 < 34 > $
	chr15: < 2 > 448 < 20 > $
	chr16: < 4 > 445 446 < 18 > $
Taeniopygia guttata	chr2: < 33 > 446 447 448 449 − 445 482 − 454 455 < 52 > $

Table A.4: List of marker orders in each species.

Bibliography

[1] V. A. Albert, *Parsimony, phylogeny, and genomics*, Oxford Press University, 2006.

[2] N. Alon, R. Yuster, and U. Zwick, *Finding and counting given length cycles*, Algorithmica **17** (1997), 209–223.

[3] S. Benzer, *On the topology of the genetic fine structure*, Proc. Nat. Acad. Sci. USA **47** (1961), 403–415.

[4] A. Bergeron, M. Blanchette, A. Chateau, and C. Chauve, *Reconstructing ancestral gene orders using conserved intervals*, Algorithms in bioinformatics (wabi), 2004, pp. 14–25.

[5] M. Blanchette, E. D. Green, W. Miller, and D. Haussler, *Reconstructing large regions of an ancestral mammalian genome in silico*, Genome Res. **14** (2004), 2412–2423.

[6] G. Blin, R. Rizzi, and S. Vialette, *A faster algorithm for finding minimum tucker submatrices*, Computability in europe (cie), 2010.

[7] K. S. Booth and G. S. Luker, *Testing for the consecutive ones property, interval graphs, and planarity using pq-tree algorithms*, J. Comput. Syst. Sci. **13** (1976), 335–379.

[8] G. Bourque, P. Pevzner, and G. Tesler, *Reconstructing the genomic architecture of ancestral mammals: lessons from human, mouse and rat genomes*, Genome Res **14** (2004), 507–516.

[9] T. A. Brown, *Genomes*, Oxford: Wiley-Liss, 2002.

[10] E. Callaway, *Plague genome: The black death decoded*, Nature **478** (2011), 444–446.

[11] C. Chauve, U.-U. Haus, T. Stephen, and V. P. You, *Minimal conflicting sets for the consecutive ones property in ancestral genome reconstruction*, J. Comput. Biol. **17** (2010), 1167–1181.

[12] C Chauve and E. Tannier, *A methodological framework for the reconstruction of contiguous regions of ancestral genomes and its applications to mammalian genomes*, PLoS Comput. Biol. **4** (2008), e1000234.

[13] M. Dom, J. Guo, and R. Niedermeier, *Approximation and fixed-parameter algorithms for consecutive ones submatrix problems*, J. Comput. Syst. Sci. **76** (2010), 204–221.

[14] M. Hoss et al., *Dna damage and dna sequence retrieval from ancient tissues*, Nucleic Acids Res. **24** (1996), no. 7, 1304–1307.

[15] P. Charbit et al., *On the consecutive ones property*, Inf. Process. Letters **108** (2008), no. 4, 186–191.

[16] R. E. Green et al, *A draft sequence of the neandertal genome*, Science **328** (2010), no. 5979, 710–722.

[17] W. C. Warrer et al., *The genome of a songbird*, Nature **464** (2010), 757–762.

[18] W. Miller et al, *Sequencing the nuclear genome of the extinct woolly mammoth*, Nature **456** (2008), 387–390.

[19] T. Faraut, *Addressing chromosome evolution in the whole-genome sequence era*, Chromosome Res. **16** (2008), no. 1, 5–16.

[20] G. Fertin, A. Labarre, I. Rusu, E. Tannier, and S. Vialette, *Combinatorics of genome rearrangements*, The MIT Press, 2009.

[21] L. Froenicke, M. Garcia Caldés, A. Graphodatsky, S. Müller, L. A. Lyons, T. J. Robinson, M. Volleth, F. Yang, and J. Wienberg, *Are molecular cytogenetics and bioinformatics suggesting diverging models of ancestral mammalian genomes?*, Genome Res. **16** (2006), no. 3, 306–310.

[22] D. R. Fulkerson and O. A. Gross, *Incidence matrices and interval graphs*, Pacific J. Math. **18** (1965), 835–855.

[23] M. Habib, C. Paul, and L. Viennot, *Partition refinement techniques: An interesting algorithmic tool kit*, Int. J. Found. Comput. Sci. **10** (1999), 147–170.

[24] Q Ji, Z. Luo, C. Yuan, J. Wible, Zhang J., and Georgi J., *The earliest known eutherian mammal*, Nature **416** (2002), 816–882.

[25] D. G. Kendall, *Incidence matrices, interval graphs and seriation in archaelogy*, Pacific Journal of Mathematics **28** (1969), 565–570.

[26] D. Kratsch, R. M. McConnell, K. Mehlhorn, and J. P. Spinrad, *Certifying algorithms for recognizing interval graphs and permutation graphs*, SIAM J. Comput. **36** (2006), no. 2, 326–353.

[27] D. A. Liberles, *Ancestral sequence reconstruction*, Oxford University Press, 2007.

[28] T. Lust and J. Teghemll, *Multiobjective decomposition of integer matrices: application to radiotherapy*, CoRR **abs/1006.1031** (2010).

[29] M. Malekesmaeili, *On certificates that a matrix does not have the consecutive ones property*, Master's Thesis, 2011.

[30] R. M. McConnell, *A certifying algorithm for the consecutive ones property*, Acm/siam symposium on discrete algorithms (soda), 2004, pp. 119–161.

[31] R. M. McConnell, K. Mehlhorn, S. Naher, and P. Schweitzer, *Certifying algorithms*, Comput. Sci. Rev. **5** (2011), 119–161.

[32] F. R. McMorris, C. Wang, and P. Zhang, *On probe interval graphs*, Discrete Appl. Math. **88** (1998), 315–324.

[33] J. Meidanis, O. Porto, and G. Telles, *A note on computing set overlap classes*, Discrete Appl. Math. **88** (1998), no. 1-3, 325–354.

[34] M Muffato and Crollius H. Roest, *Paleogenomics in vertebrates, or the recovery of lost genomes from the mist of time*, Bioessays **30** (2008), no. 2, 122–134.

[35] A. Ouangraoua, E. Tannier, and C. Chauve, *Reconstructing the architecture of the ancestral amniote genome*, Bioinformatics **27** (2010), no. 19, 2664–2671.

[36] M. Raffinot, *Consecutive ones property testing: Cut or swap*, Computability in europe (cie), 2011, pp. 239–249.

[37] R. C. Read and R. E. Tarjan, *Bounds on backtrack algorithms for listing cycles, paths, and spanning trees*, Networks **5** (1975), 237–252.

[38] F. Richard, M. Lombard, and Dutrullaux B., *Reconstruction of the ancestral karyotype of eutherian mammals*, Chromosome Res. **11** (2003), no. 6, 605–618.

[39] J. Romiguier, V. Ranwez, E. J.P. Douzery, and N. Galtier, *Contrasting gc-content dynamics across 33 mammalian genomes: Relationship with life-history traits and chromosome sizes*, Genome Res. **20** (2010), no. 8, 1001–1009.

[40] F. Rubin, *A search procedure for hamilton paths and circuits*, J. ACM **21** (1974), 576–580.

[41] J. Stoye and R. Wittler, *A unified approach for reconstructing ancient gene clusters*, IEEE/ACM Trans. Comput. Biology Bioinform **6** (2009), 387–400.

[42] N. Tamnura, *Eomaia*, 2010. `http://en.wikipedia.org/wiki/File:Eomaia_NT.jpg`, accessed November 27 2012.

[43] A. C. Tucker, *A structure theorem for the consecutive 1's property*, J. Combinat. Theory (B) **12** (1972), 153–162.

[44] D. B. West, *Introduction to graph theory*, Pearson, 2000.