ON CERTIFICATES THAT A MATRIX DOES NOT HAVE THE CONSECUTIVE ONES PROPERTY

by

Mehrnoush Malekesmaeili
BSc of Applied Mathematics, Tarbiat Moallem University of Iran, 2007

a 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
Fall 2011

APPROVAL

Name: Mehrnoush Malekesmaeili

Degree: Master of Science

Title of Thesis: On certificates that a matrix does not have the consecutive ones property

Examining Committee: Dr. Abraham Punnen
Chair

Dr. Tamon Stephen, Assistant Professor
Senior Supervisor

Dr. Cedric Chauve, Associate Professor
Supervisor

Dr. Ladislav Stacho, Associate Professor
SFU Examiner

Date: November 24, 2011

Abstract

A binary matrix has the *consecutive ones property* (C1P) if there exists a permutation of its columns which makes the 1s consecutive in every row. The C1P has many applications which range from computational biology to optimization. We give an overview of the C1P and its connections to other related problems.

The main contribution of this thesis is about certificates of non-C1Pness. The notion of incompatibility graph of a binary matrix was introduced in [McConnell, SODA 2004] where it is shown that odd cycles of this graph provide a certificate for a non-C1P matrix. A bound of $k+2$ was claimed for the smallest odd cycle of a non-C1P matrix with k columns. We show that this result can be obtained directly via Tucker patterns, and that the correct bound is $k+2$ when k is even, but $k+3$ when k is odd.

Furthermore we empirically study the *minimal conflicting set* certificate on synthetic data.

To my parents!

"Every new body of discovery is mathematical in form,

because there is no other guidance we can have!"

— *Mathematical Maxims and Minims*, Rome Press Inc., *1988*

Acknowledgments

First and foremost, I want to express my sincere gratitude to my advisor Dr. Tamon Stephen, my senior supervisor for his incredible support and constructive advice from the very early stage of this research, for his patience, motivation, and immense knowledge. During the difficult times when writing this thesis, he gave me the moral support. I am glad that I am his graduate student and indebted to him more than he knows.

Besides my advisor, I would like to thank Dr. Cedric Chauve, my committee member whose encouragement, guidance and hard questions enabled me to develop an understanding of the subject.

I was delighted to interact with Dr. Ross McConnell for his helpful discussion, inspiration and willingness to share his bright ideas with me.

I owe my most sincere gratitude to Dr. Luis Goddyn, and Dr. Matt DeVos, who gave me the opportunity to work with them in the Department of Mathematics, and gave me untiring help.

My sincere thanks also goes to Dr. Randall Pyke, the TA coordinator of Surrey campus, a great advisor and a good friend.

I would also like to acknowledge the help of Mrs. Diane Pogue, the graduate secretary of Mathematics department for her assistance during my graduate studies in Simon Fraser University.

Last but not the least, million thanks to my parents, Azam and Mohammadhossein, without whom I would never have been able to achieve so much. Thank you for your endless support and love, no matter how technical this is for you. I have no suitable word

that can fully describe your love for me.

Many thanks to my brother, Mani Malekesmaeili, for always believing in me. I know I have some one to turn to and I am not all alone. Without whom I could not have made it here.

Finally I would like to thank Alborz for being the best of friends. For his personal support, great patience, and companionship at all times. I could not finish my thesis so soon without his encouragements. Thank you so much.

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

1.1 Consecutive Ones Property

A binary matrix (0-1 matrix) has the *consecutive ones property*(C1P) if its columns can be ordered in such a way that all the 1 entries become consecutive in all rows of the matrix. This property has many applications in computational biology such as ancestral genome reconstruction [12] and physical mapping [2]. It also plays an important role in graph theory [22, 6, 27] and Interval Routing in topology [20]. See Section 1.3 for additional applications and Section 1.4 for some related problems.

Checking whether a matrix has the C1P can be done quickly via polynomial time algorithms that give an ordering of columns of the matrix that places 1s consecutive in each row. We discuss later that it is not as obvious how to find a certificate that a matrix does not have the C1P. In this thesis we consider problems related to such certificates.

Providing a certificate when the given binary matrix does not have the C1P is an important problem. Tucker in [48] proposed such a certificate as a class of forbidden submatrices known as *Tucker patterns*, however, he did not propose an algorithm to find these submatrices. The notion of *Incompatibility graph* was introduced by McConnell in [38]. He showed that this graph is bipartite if its corresponding binary matrix has the C1P and that the existence of an odd cycle in the graph is a certificate that the matrix does not have the C1P.

McConnell claimed an upper bound of $k+2$ for the size of the smallest odd cycle in the incompatibility graph of a non-C1P matrix with k columns. However we prove in Chapter 3 that the correct bound is $k+2$ when k is odd and $k+3$ when k is even, by using Tucker's characterization[1].

A C1P matrix can be represented by a data structure called a *PQ-tree* [6] which was then extended in [43, 38] to a general binary matrix. We discuss the PQ-tree and its generalizations in Chapter 2. We also describe the modular decomposition and related algebraic notions associated to the set representation of a binary matrix. We then discuss Tucker patterns [48] as a certificate of non-C1Pness. In Chapter 3 we discuss certificates for a matrix that does not have the C1P, incompatibility graphs [38]. We will use both certificates in order to obtain a tight bound on the size of the smallest odd cycle in the incompatibility graph.

In Chapter 4 we discuss another concept, the *minimal conflicting set* that is a subset of rows of a matrix that does not have the C1P. These are row-minimal obstructions that contain certificates of non-C1Pness, i.e Tucker patterns. These matrices can be represented by an undirected graph known as the *row-column graph* whose vertices (edges) correspond to columns (rows) of the matrix. We study the class of matrices with exactly two 1s per row and enumerate the total number of minimal conflicting sets. We then consider two examples of matrices that do not have the C1P. Our goal is to find extreme cases, where the number of minimal conflicting sets can be exponential.

Another concept that we discuss in Chapter 4 is the *maximal C1P*, which is dual to the minimal conflicting set. We consider some simulated data sets that have many maximal C1P but few minimal conflicting sets. We study the row-column graph of each data set and find the common subgraphs that appear in most of these data sets, combine them and use them to compute the total number of MC1P for these data.

[1] some of these results are contained in a paper submitted to Information Processing Letters, September 2011. See [36] for details

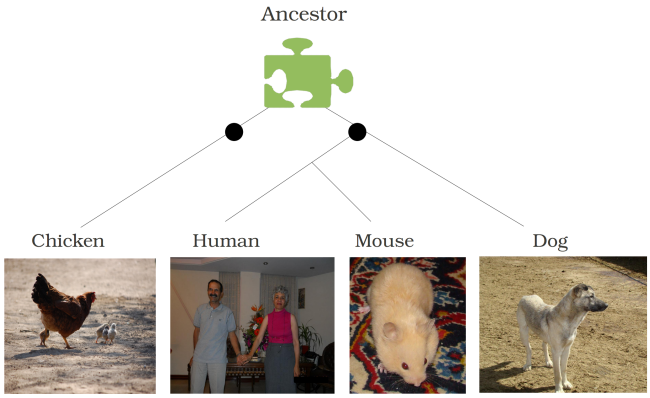


Figure 1.1: Mammalian chromosome evolution

1.2 Biological motivation

It is certainly of interest to find the origin of the human genome. Ancestral genome reconstruction tries to discover the genome rearrangements that happened during the evolution. The first reconstruction of genomes was provided in [50] by a method in which DNA probes are labeled with different colored fluorescent tags to visualize one or more specific regions of the genome.

We focus on genome reconstruction using *markers* on chromosomes (See Figure 1.1), which was first introduced by Adam in [1]. Markers are long and distinctive sequences of DNA which are presumed to exist in ancestral species and appear exactly once in each of the current species that are being studied. There might be mutations over time that result in a slight difference between sequences of a marker appearing in some species, but they are not taken into account. Therefore these sequences are considered identical and thus labeled with a same marker.

The ancestral genome G can be represented by a binary matrix M_G [12]. The columns of M_G represent markers assumed to have been present in G. Each row is a subset of columns that represents *ancestral synteny*, a set of markers that are believed to be consecutive in G [10]. Therefore $M_G(i, j) = 1$ if the ancestral synteny r_i contains the marker c_j. In fact we focus on the ordering of markers that are present in the ancestral genome.

We know which markers are consecutive in the present-day genomes. Also each current species provides evidence that markers may have been in a certain order, but we do not know which order is correct. Therefore our goal in ancestral genome reconstruction is to find an order of present markers in G that is consistent with most of the syntenies. For instance in a consistent order, if two markers c_i and c_j where $i < j$ are present on an ancestral synteny r, but another marker c_k does not appear on r, then c_k should appear before marker c_i or after marker c_j. This order makes the 1s consecutive in the row corresponding to that ancestral synteny which results in a correct order of markers in G. Therefore in order to find a consistent order of markers in G, M_G needs to have the consecutive ones property.

In practice it is often impossible to find the correct order of markers and consequently order the columns in the matrix so that it has the consecutive ones property. In this case the goal is to detect syntenies whose removal results in a correct order of markers in the remaining syntenies. Hence removing the rows of M_G corresponding to these syntenies leads to a C1P matrix. These removed syntenies are known as *false positives* or (errors) and we discuss them later in Chapter 4 in more detail.

1.3 Some other applications

The consecutive ones property has application in graph theory where the goal is to detect whether a graph is an interval graph [6, 27], i.e. interval graph recognition. This property was first used by Hoffman in [25] and Fulkerson et al. in [22]. It was then developed by Booth and Lueker in [6] based on building a data structure called the *PQ-tree*. Novick in [46], Meidanis et al. in [43] and McConnell in [38] gave a generalization of PQ-trees to an arbitrary binary matrix.

Another application of C1P is in physical mapping [2] of a chromosome or a genome which shows the physical locations of genes and other DNA sequences needed to be studied.

In linear programming (LP) in optimization, ($\{\min cx \mid Ax = b, x \geq 0\}$ or $\{\min cx \mid Ax \geq b\}$), if the elements of the right-hand side vectors are integers, and the coefficient matrix A has the C1P, then the LP has an integral optima. Another application of C1P

is in approximation of a combinatorial problem, a restricted Set Cover problem known as *rectangle stabbing* problem [17].

1.4 Some related problems

The *Consecutive ones Submatrix* (COS) problem is a generalization of the C1P when the given binary matrix does not have the C1P. The COS problem is to find a maximum submatrix with the consecutive ones property in the given binary matrix. This problem is NP-complete [24]. It has been shown that the Hamiltonian path problem can be reduced to COS problem (see [28] for details).

The *Gapped C1P* is another extension of C1P that detects if there exists a permutation of columns of the matrix such that each row contains at most k sequences of 1's and any two consecutive sequences of 1's are not separated by a gap of more than δ 0s. In [11] it has been shown that when $k = 2$ and $\delta \geq 2$, this problem is already NP-complete.

Another problem related to C1P is to partition the set of rows of the given binary matrix into two sets that have the C1P. This problem is also proved to be NP-complete. See [33] for detail. In fact most of the generalizations of the C1P are NP-complete.

1.5 Basic definitions

Let M be a binary matrix with m rows and n columns. We consider $C_M = \{c_1, c_2, \ldots, c_n\}$ as a set of columns of M, and $R_M = \{r_1, r_2, \ldots, r_m\}$ as the set of its rows. We can treat R_M as a subset of columns, i.e. $r_i = \{c_j | M_{ij} = 1\}$ for $i = 1, \ldots, m$. Let $s = \sum_{i=1}^{m} |r_i|$ be the total number of 1s in M.

Definition 1.5.1. *A binary matrix has the* consecutive ones property *if its columns can be ordered in such a way that all 1s on each row are consecutive.*

Definition 1.5.2. *The* degree *of a matrix M is the maximum number of entries 1 found in a single row of M.*

Example 1.5.3. *Let M be the binary matrix below that has degree 4. It is easy to check that M is a C1P matrix, since interchanging its third and fourth columns results in consecutive 1s in each row.*

$$M = \begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \end{array} \begin{array}{c} \begin{array}{cccccc} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

Remark 1.5.4. *A binary matrix with $|C_M| \leq 2$ is always a C1P matrix. We can ignore those rows of M that have degree one, since their 1s are consecutive regardless of their position. We can also ignore rows (columns) of 0s and rows (columns) of 1s.*

The set-definition of the C1P is as follows:

Definition 1.5.5. *Given a domain set V and a family F of subsets of the domain set, F has the* consecutive ones property*, if there exists an ordering of elements of V that all the elements of F appear in a consecutive order.*

It can be observed that binary matrices are in one-to-one correspondence with families of subsets of a set. Given M, let $C_1, \ldots, C_n$ be the domain set, then each row represents the subset of columns corresponding to its non-zero entries. Therefore a set family has the C1P if and only if its representing matrix does.

Definition 1.5.6. *A permutation of columns of a matrix M is called a* valid permutation*, if it rearranges the elements of C_M such that it places 1 entries consecutive in each row. Such a permutation is said to give the* consecutive ones ordering *of a binary matrix M, that is an ordering $\mathcal{R}$ of elements of C_M that makes M a C1P matrix.*

The consecutive ones ordering can be seen as a valid permutation of columns of the binary matrix.

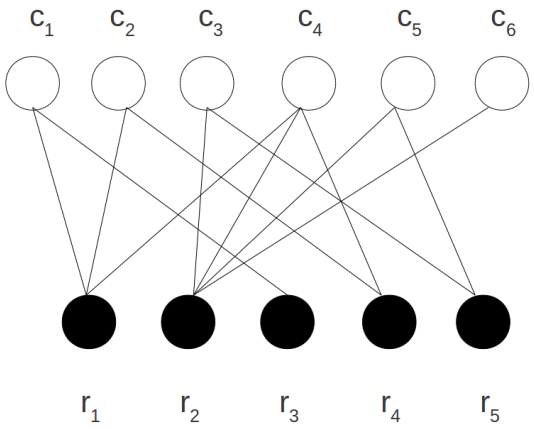


Figure 1.2: The bipartite graph b_M

Example 1.5.7. *For the binary matrix in example 1.5.3,* $\mathcal{R} = \{c_1, c_2, c_4, c_3, c_5, c_6\}$ *is a consecutive ones ordering.*

Definition 1.5.8. *The* bipartite graph associated *to a binary matrix* M *denoted by* $b_M = (C_M \cup R_M, E)$ *is a symmetric bipartite graph whose vertex set is the union of the set of columns and rows of* M*, and edge set is the set* $\{(c_j, r_i) | M_{ij} = 1, i = 1, \ldots m, j = 1, \ldots n\}$*. Vertices corresponding to columns and rows are illustrated with white and black circles respectively.*

Example 1.5.9. *A bipartite graph of matrix* M *in example 1.5.3 is shown in Figure 1.2.*

Let F be a set family on domain set V.

Definition 1.5.10. *Two sets* X *and* Y*,* $X, Y \in F$ overlap *or have nontrivial intersection if they intersect but neither is a subset of the other, that is, they overlap if* $X - Y$*,* $X \cap Y$*, and* $Y - X$ *are all nonempty.*

So two sets have trivial intersection if one is the subset of the other.

Definition 1.5.11. *An* overlap graph $O_M = (V, E)$ *of a binary matrix* M *is an undirected graph whose vertices are members of* R_M *and two vertices are adjacent if they overlap.*

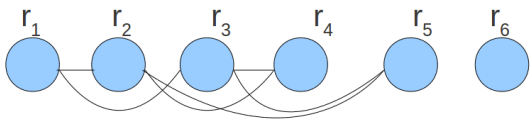


Figure 1.3: The overlap graph O_M

Example 1.5.12. *Let M be the binary matrix below. The overlap graph of M is shown in Figure 1.3*

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

Definition 1.5.13. *A* connected component *of an undirected graph is a subgraph in which any two vertices are connected to each other by paths.*

Example 1.5.14. *The overlap graph in Figure 1.3 has two connected components $C_1 = \{r_1, r_2, r_3, r_4, r_5\}$ and $C_2 = \{r_6\}$.*

The overlap graph is a tool to represent those sets that have non-trivial intersection. We use this graph along with its connected component mainly in Section 2.5.4.1. Finding the connected component can be done in linear time using a breadth-first search. See [30] for details.

1.6 Matrices that do not have the consecutive ones property

Let M be a binary matrix that does not have the C1P, thus there is no valid permutation of columns that makes M a C1P matrix. Tucker in [48] found five forbidden patterns known as *Tucker patterns* that obstruct the matrix from having C1P. Hence showing a binary matrix M has at least one of the Tucker forbidden patterns as minor can be used as a certificate of non-C1Pness, but it is not clear how to find one. These are discussed in details in Chapter 2. In here we briefly discuss two concepts, *Minimal Conflicting Sets* and *Maximal C1P* of a non-C1P matrix.

Definition 1.6.1. *A* Minimal Conflicting Set (MCS) *of a binary matrix M is a set R of rows that does not have the C1P but such that any proper subset of R has the C1P.*

So MCS is a minimal set of rows contained in a non-C1P matrix

Definition 1.6.2. *A* Maximal C1P Set (MC1P) *of a binary matrix M is the set of rows that has the C1P, but adding any row of the matrix to it results in a set of rows that does not have the C1P.*

It can be seen that MC1P is a dual concept of MCS.

Example 1.6.3. *The matrix below does not have the C1P, since its columns cannot be ordered in a way that makes M a C1P matrix.*

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

It can be seen that deleting any row of this matrix results in a C1P matrix. Therefore $\{r_1, r_2, r_3, r_4, r_5\}$ is MCS, and any combination of four rows of M is MC1P.

The minimal conflicting set was first introduced by Bergeron et al. in [4] and expanded by Chauve et al. in [10]. They also showed that when all rows of the matrix have degree 2,

it is possible to find the MCS and MC1P using a graph corresponding to the matrix known as the *row-column graph*. We discuss this in Chapter 4.

Chapter 2

Background on the consecutive ones property

In this chapter we discuss known data structures and algorithms related to the C1P. We first discuss the five classes of matrices introduced by Tucker in [48] that can be used as a certificate when a matrix does not have the C1P. We then discuss a tree-based data structure for a C1P matrix known as the *PQ-tree* and its generalizations. The PQ-tree represents all the valid permutations of the columns of a C1P matrix. It was first introduced to recognize interval graphs, which are linked to the consecutive ones property. It is shown in [34] that it can be used for detecting gene clusters in multiple genomes. In [8] it has been shown that it can be used for implementing a well-known generalization of the dynamic programming algorithm for the TSP. We do not discuss the original algorithm to construct the PQ-tree, since it is rather complicated.

We discuss gPQ-trees and PQR-trees which are generalizations of PQ-trees, that can be constructed for any binary matrix and are simple to implement. These trees convey more information when the given matrix does not have the C1P, since they indicate the elements that obstruct the matrix from having the C1P. The generalized PQ-tree as another extension of PQ-trees were introduced by McConnell in [38]. It uses the modular decompositions and a partitioning algorithm. This data structure, like other PQ-tree generalizations, can be

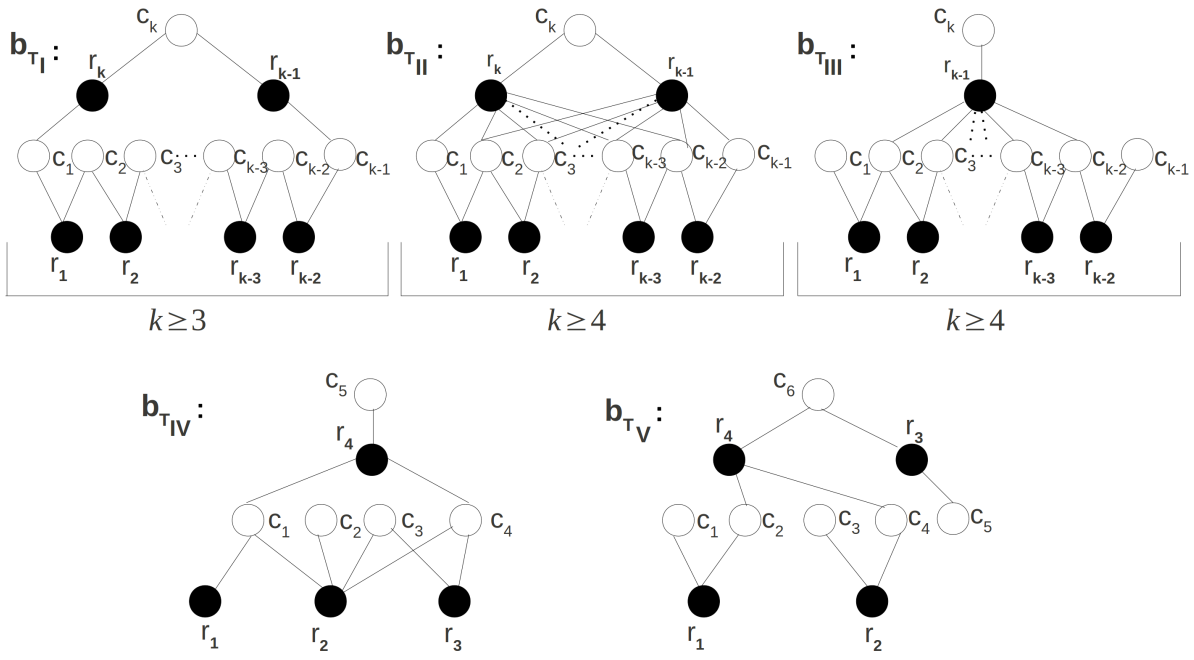


Figure 2.1: Bipartite graphs corresponding to Tucker patterns

applied to an arbitrary binary matrix. It has been shown that all these generalizations are equivalent to the PQ-tree when the matrix has the C1P. We include in particular a detailed presentation of McConnell's partitioning algorithm which uses these generalized PQ-tree, and on which his proof of an upper bound for the size of the certificate is based. In Section 3.2 we show that this bound is indeed incorrect.

2.1 Tucker patterns

The first structural result on non-C1P matrices is due to Tucker, who proved in [48] that a binary matrix does not have the C1P if and only if it contains a submatrix from one of five families of binary matrices that define certificates for non-C1P matrices.

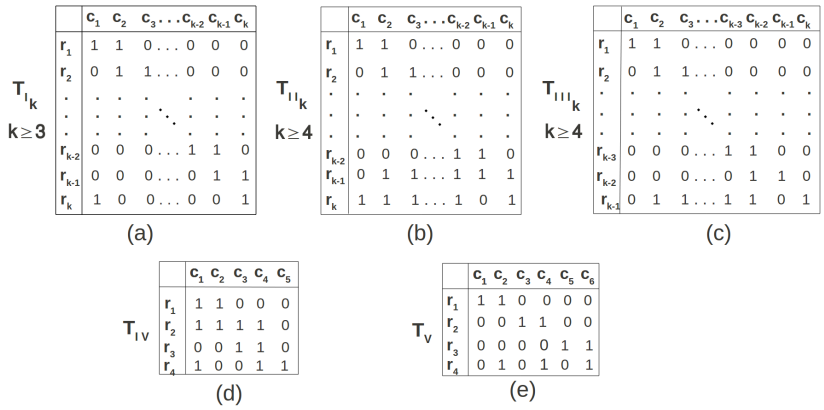


Figure 2.2: The five Tucker patterns

2.1.1 Asteroidal triples and Tucker patterns

In Chapter 1 we defined a bipartite graph b_M associated to a binary matrix M.

Definition 2.1.1. *Let $G = (V, E)$ be a simple undirected graph and v_i, v_j, v_k be three distinct vertices. Then v_i, v_j, v_k form an* asteroidal triple (AT)*, if there exists a path P between two of these vertices and no vertex in P is adjacent to the third vertex.*

Tucker proved that the binary matrix M has the consecutive ones property if b_M does not contain an asteroidal triple between any three vertices of C_M (or R_M).

Theorem 2.1.2. *[48] In the bipartite graph $b_M = (C_M \cup R_M, E)$ the vertex set C_M contains no asteroidal triple if and only if b_M contains none of the graphs $b_{T_I}, b_{T_{II}}, b_{T_{III}}, b_{T_{IV}}, b_{T_V}$ shown in Figure 2.1.*

Lemma 2.1.3. *[48] A binary matrix has the consecutive ones property if and only if it contains none of the five Tucker patterns as a submatrix.*

Tucker characterized C1P matrices via excluded minors known as "Tucker patterns" illustrated in Figure 2.2 and showed that these patterns are due to an AT contained in b_M. These patterns are the minimal structures that obstruct the matrix M from having

the C1P, i.e. removing a row (or a column) from each of these structures results in a C1P-matrix. These patterns can be detected in polynomial time [16], however the algorithm is complicated to implement since it uses many reduction rules and search trees.

In our analysis in Chapter 3, for all of the five Tucker matrices we consider the order of columns given in Figure 2.2.

2.2 PQ-tree

Booth and Lueker in [6] introduced the PQ-tree of a binary matrix M. It is a rooted, labeled tree in which each element of C_M is represented by one of the leaf nodes, i.e. nodes that do not have children. The non-leaf nodes of this tree are of two types,

- P nodes that have at least two children,
- Q nodes that have at least three children.

Therefore each internal nodes of the tree is either a P-node or a Q-node. P nodes are conventionally drawn as circles and Q nodes as rectangles. PQ-trees have applications where the goal is to find a consecutive ordering of the given objects. Thus they can be used for detecting interval graphs or planar graphs [32].

Definition 2.2.1. *The* universal PQ-tree *of a matrix M is a tree that has a single root node labeled as a P-node whose children are the elements of C_M. The* null tree *is used to represent an empty set.*

The strategy for building a PQ-tree is to first build the universal PQ-tree and then restrict the tree by the members of R_M in each step such that the ordering of children is consistent with all members of R_M that are processed. Booth and Lueker proposed the PQ-tree algorithm for a C1P matrix and then showed that the algorithm can be done in linear time in m, n and s, where s is the total number of 1s in M.

Suppose that M has the C1P, its PQ-tree encodes all the valid permutations of the columns of the binary matrix by considering two types of operations:

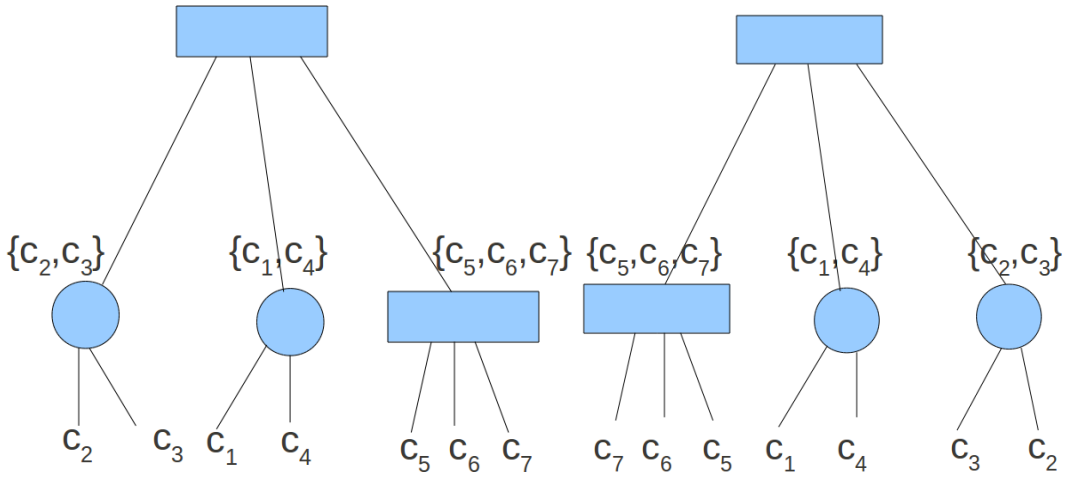


Figure 2.3: Two equivalent PQ-trees

- For each P-node, any permutation of its children is accepted, so they can be rearranged in any order.
- For each Q-node, the order of the children can be reversed, but its children may not otherwise be reordered.

Example 2.2.2. *Let M be the binary matrix below.*

$$M = \begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{ccccccc} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 & c_7 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}.$$

The tree T in Figure 2.3(a) is a PQ-tree representing M. So $\mathcal{R} = \{c_2, c_3, c_1, c_4, c_5, c_6, c_7\}$ is a consecutive ones ordering. $\mathcal{R}' = \{c_7, c_6, c_5, c_1, c_4, c_3, c_2\}$ is also a consecutive ones ordering. T' in Figure 2.3(b) is another PQ-tree that represents $\mathcal{R}'$ and is equivalent to T since it can be obtained by reversing the order of the children of Q nodes $\{c_1, c_2, \ldots, c_7\}$, $\{c_5, c_6, c_7\}$, and then reversing the order of children of the P node $\{c_2, c_3\}$.

It is trivial that the reverse of any consecutive ones ordering is always another consecutive ordering and this is reflected in the PQ-tree representation. In Sections 2.3-2.5 we discuss generalizations of PQ-trees. These trees are same but developed independently.

2.3 gPQ-tree

Since implementing the original PQ-tree algorithm is complicated, different algorithms have been proposed as well as generalizations of PQ-trees that can be built for any binary matrix. The gPQ-tree is a generalization of the PQ-tree that was introduced by Novick in [46] to solve the trivial intersection problem. This concept can be also used for recognizing interval and parity graphs.

Let M be a binary matrix. Each gPQ-tree is constructed such that it represents those subsets of C_M that have trivial intersection with every member of R_M. We denote the set of these subsets by $N_M(R)$ and we will discuss it in Section 2.5.

Definition 2.3.1. *For a* gPQ-tree T, T' *is its* reduction with respect to X, *if it represents those subsets of* C_M *that have trivial intersection with* X.

For solving the trivial intersection problem, the algorithm is based on first building the universal gPQ-tree which is same as the universal PQ-tree and then reducing the tree by members of R_M in each step. So the result would be a gPQ-tree which represents all members of $N_M(R)$, i.e. all subsets of C_M that have trivial intersection with members of R_M. Novick showed that these reductions can be done in $\mathcal{O}(nm)$ and guarantees that the following have trivial intersection with members of R_M:

- The set of leaves that are descendant of some Q-node,
- For any subset of the children of a P-node, the set of leaves that are descendants of this subset,
- $\{c_i\}$ for $i = 1, \ldots, n$, and C_M itself,

Theorem 2.3.2. *[46] If T, the result of the PQ-tree after reducing the universal PQ-tree by members of R_M, is a non-null tree, then ignoring the left-to-right order of children in T results in a* gPQ-tree *that is equivalent with the result of reducing the universal* gPQ-tree *by members of R_M.*

It is trivial to see that a family F of subsets of V can be represented by a gPQ-tree on a domain set V.

2.4 PQR-tree

The *PQR-tree* was introduced by Meidanis and Munuera in [43] for an arbitrary binary matrix. This tree also generalizes the gPQ-tree introduced by Novick. It is a rooted tree over C_M with four types of nodes: P, Q, R and leaves. The leaves are in one to one correspondence with the elements of C_M, and every The R node has at least three children and its children can be ordered arbitrarily. As we discussed before this tree is an extension of the PQ-tree which gives more information when the given binary matrix does not have the C1P.

R nodes represent subsets of columns that obstruct the matrix from having the C1P, thus removing one column from each R node results in a C1P matrix. The PQR-tree of a C1P matrix does not have any R nodes and therefore it is equivalent to its PQ-tree. Similar to the PQ-tree notations, two PQR-trees are equivalent when one can be obtained from the other by reversing the order of the children of Q nodes and arbitrary permutation of the children of P, or R nodes. Meidanis et al. [47] proposed a recursive algorithm, that build the PQR-tree in quadratic time on m, n and s and is simpler than the PQ-tree algorithm of Booth and Lueker.

Example 2.4.1. *Let $C_M = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$ and the set family $R_M = \{\{c_1, c_2, c_3\}, \{c_3, c_4, c_5, c_6, c_7\}, \{c_4, c_5\}, \{c_5, c_6\}, \{c_5, c_7\}\}$. The PQR-tree on domain set C_M is shown in Figure 2.4. The node with two circles is a R-node which shows that $\{c_4, c_5, c_6, c_7\}$ obstructs the matrix M from having C1P.*

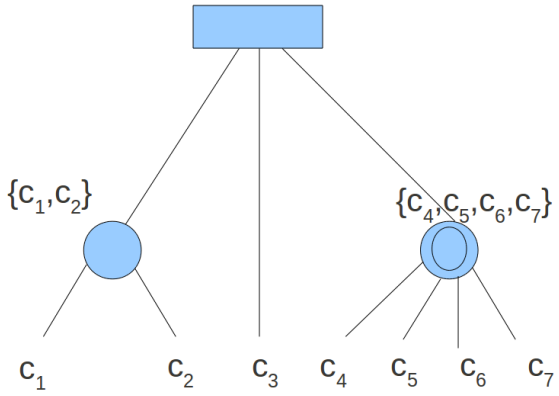


Figure 2.4: A PQR-tree

The generalized PQ-tree is another generalization of PQ-tree that was introduced by McConnell in [38]. We describe the algorithm of finding the generalized PQ-tree in the next section. All generalizations of PQ-tree use similar concepts. They all are based on finding the overlapping members of R_M, however the details are different. For example the orthogonal sets defined in [43] are the same as sets with non-trivial intersection in [46] and the non-overlapping sets in [38]. Essentially a gPQ-tree is a PQR-tree whose R nodes are transformed into Q nodes, and the order of children is ignored [43].

As we discussed in Chapter 1, a binary matrix M can be represented by domain set V whose members are columns of the matrix, and a set family F whose members represents rows of the matrix. In Section 2.5 we represent the binary matrix by a domain set V and its set family F. We do this to follow McConnell's notation.

2.5 Constructing the generalized PQ tree

The generalized PQ-tree is very similar to a PQR-tree, but was developed independently by McConnell in [38]. Like the PQR-tree, the generalized PQ-tree have three types of internal nodes. *Prime* nodes correspond to the P nodes, *linear* nodes to Q nodes and *degenerate* nodes to R node [39].The difference between these algorithms is in the running time of

finding the union of connected components of the overlap graph.

In this section we follow McConnell's notation and describe the generalized PQ-tree with respect to partitive families and modular decomposition. McConnell in [38, 39] proposed a linear time algorithm that finds the generalized PQ-tree for any binary matrix M. He then showed that the generalized PQ-tree of a C1P matrix is equivalent to its PQ-tree.

2.5.1 Partitive families

In Section 1.5 we observed that a binary matrix M can be represented by a family F of subsets of a domain set V, where V indexes C_M and the rows of M are indicator functions of sets in F. In McConnell's terminology $V = \{v_1, \ldots, v_n\}$ is called the domain set and $F = \{f_1, \ldots, f_m\}$, the set family of V.

Definition 2.5.1. *A set family F of a domain set V is a* tree-like *family, if the followings hold:*

- $\emptyset \notin F$,$V \in F$, $\forall\ v \in V, \{v\} \in F$,
- $\forall\ i, j = 1, \ldots, m,\ f_i$ *and* f_j do not overlap.

In Section 1.5 we discussed that two overlapping sets have non-trivial intersection. We denote $N(F)$ as the family of non empty subsets of V that do not overlap with any member of F.

Each tree-like family corresponds to an *inclusion tree* whose nodes are members of the family and there is an edge between two nodes if one is the subset of the other.

Example 2.5.2. *Let $V = \{a, b, c, d, e, f\}$. By the definition above $F = \{\{a, b, c\}, \{c, d, e, f\}, \{a\}, \{b\}, \{d, e\}, \{e, f\}\}$ is not a tree-like family, since $\{a, b, c\}$ and $\{c, d, e, f\}$ overlap.*
Also $N(F) = \{\{a, b, c, d, e, f\}, \{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}, \{a, b\}\}$.

Definition 2.5.3. *F is a* weakly partitive *family if*

- $\emptyset \notin F$,$V \in F$, $\forall\ v \in V, \{v\} \in F$,

- $\forall\ i, j = 1, \ldots, m$, *if* f_i *and* f_j *overlap, then* $f_i \cup f_j,\ f_i \cap f_j,\ f_i - f_j,\ f_j - f_i$ *are in* F.

A member of a weakly partitive family is *strong* if it does not overlap any other member of F. It can be seen that the set of strong members of a weakly partitive family is a tree-like family. The inclusion tree of the set of strong members of F is denoted by $T(F)$.

Definition 2.5.4. *A set family* F *is* strongly partitive *family or decomposable set family if,*

- $\emptyset \notin F$,$V \in F$, $\forall\ v \in V, \{v\} \in F$,
- $f_i \cup f_j,\ f_i \cap f_j,\ f_i \Delta f_j \in F$, *if* f_i *and* f_j *overlap.*

It can be seen that a strongly partitive family is a weakly partitive family.

Example 2.5.5. *let* $F = \{\{a, b, c\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}\}$. *It can be observed that* F *is a weakly partitive family and since for any* $X,\ Y \in F,\ X \Delta Y \in F$ *so it is also a strongly partitive family.*

Theorem 2.5.6. *[13, 45] Each internal node* X *of T(F) is one of the following types:*

- Degenerate*, if the union of two or more of its children is in* F.
- Prime*, if other than* X *itself, the union of two or more of its children is not in* F.
- Linear*, if the union of two or more of its children is a member of* F *if and only if they are consecutive in the ordering.*

Definition 2.5.7. *The* decomposition tree *of a set family* F *is an inclusion tree* $T(F)$*, whose internal nodes are labeled as in Theorem 2.5.6.*

Theorem 2.5.8. *[32] If* F *has the C1P, its PQ-tree is the decomposition tree of* $N(F)$ *whose prime nodes are* Q *nodes and degenerate nodes are* P *nodes.*

When F has the C1P, as we discussed in Section 2.2 the reverse ordering of children of Q nodes and any ordering of P nodes in the PQ-tree of F gives the consecutive ones ordering of members of V that is consistent with F. Therefore the decomposition tree of $N(F)$ does not give the ordering of children of Q nodes and there is a need for a better characterization.

Definition 2.5.9. *Let $W(F)$ denote the* smallest weakly partitive family *of a set family F, and $S(F)$* the smallest strongly partitive family *of F.*

The *generalized PQ-tree* is the decomposition tree of $W(F)$ and it has no degenerate nodes when F has the C1P.

Observation 2.5.10. *If F is a strongly partitive family, then $S(F) = F$, since $S(F)$ is the smallest strongly partitive family that contains F as a subfamily. Also the non-overlapping family of $N(F)$ is same as F, i.e. $N(N(F)) = F$.*

Theorem 2.5.11. *[38] The decomposition tree of $S(F)$ can be obtained from the decomposition tree of $N(F)$ by changing prime nodes to degenerate nodes and vice versa, or it can be obtained from $W(F)$ by relabeling all linear nodes as degenerate nodes.*

When F has the C1P, as a result of Theorem 2.5.11, its PQ-tree is the decomposition tree of $S(F)$ where prime nodes are P nodes and degenerate nodes are Q nodes. Also the decomposition tree of $W(F)$ can be obtained from the decomposition tree of $S(F)$, by relabeling degenerate nodes (Q nodes) as linear nodes. Hence the generalized PQ-tree of F is equivalent to its PQ-tree.

2.5.2 Substitution decomposition

Definition 2.5.12. *A* module *of a graph $G = (V, E)$ is a set X of nodes such that for any x in $V(G) - X$, either x is adjacent to every element of X or x is adjacent to no element of X. A domain set V, and its singleton elements $\{v\}$ are called* trivial modules.

A module is a generalization of a connected component of a graph. The modular decomposition represents all the modules of G. It was first described by Gallai as a tree in [23] whose internal nodes are labeled as *prime* or *degenerate*. McConnell et al. in [42] proposed a linear time algorithm for finding the modular decomposition tree of a graph. They also proved that modular decompositions form a *strongly partitive family* and therefore have *decomposition tree* which we discuss further.

Let $\mathcal{P}$ be a partition of a domain set V then every partition class is a module, since the partition classes are disjoint, their adjacencies constitute a new graph called a *quotient graph.*

Definition 2.5.13. The quotient $G/\mathcal{P}$ *is a graph whose nodes are the members of* $\mathcal{P}$*, and there is an edge between two nodes of* $G/\mathcal{P}$ *if their corresponding modules are adjacent.*

For trivial partitions (empty set, or the domain set V), it is not hard to see that when $\mathcal{P} = V$, its quotient $G/\mathcal{P}$ is the one-node graph, and when $\mathcal{P} = \{v|\ v \in V\}$ then $G/\mathcal{P} = G$ [23].

2.5.3 Modular decomposition for a set family

What is important for us is the relation between the substitution decomposition and the generalized PQ tree of a set family F. Since modules are strongly partitive family, McConnell in [38] discussed that the only module-like structures of a set family F are members of $N(F)$.

Let $\mathcal{P}$ be a partition class where every class is a module, therefore every member of $\mathcal{P}$ is a member of $N(F)$, and the quotient $F/\mathcal{P}$ is $\mathcal{Q} = \{X|X$ is a nonempty subfamily of $\mathcal{P}$ *and* $\cup X \in F\}$. Therefore each internal node of the decomposition tree of F is a quotient.

Example 2.5.14. *Let* $F = \{\{c_1, c_2, c_3, c_4\}, \{c_4, c_5, c_6, c_7, c_8\}, \{c_1, c_2\}, \{c_2, c_3\}, \{c_5, c_6, c_7\}, \{c_6, c_7, c_8\}\}$*. It is not hard to see that* $\mathcal{P} = \{\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}, \{c_1\}, \{c_2\}, \{c_3\}, \{c_4\}, \{c_4\}, \{c_5\}, \{c_6\}, \{c_7\}, \{c_8\}, \{c_1, c_2, c_3\}, \{c_5, c_6, c_7, c_8\}\{c_6, c_7\}\}$ *and* $\mathcal{Q} = \{\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}, \{c_1, c_2, c_3\}, \{c_5, c_6, c_7, c_8\}, \{c_6, c_7\}\}$*. The generalized PQ-tree of* F *is shown in Figure 2.5. it is equivalent to its PQ-tree, since* F *has the C1P. The internal nodes are members of the quotient. Also using the Theorem 2.5.6 the internal node* $\{c_6, c_7\}$ *is a prime node which can be labeled as a* P *node, and* $\{c_1, c_2, c_3\}$, $\{c_5, c_6, c_7, c_8\}$ *are linear nodes that are labeled as* Q *nodes in Figure 2.5.*

For an example of what happens when M does not have the C1P see Section 2.5.5.

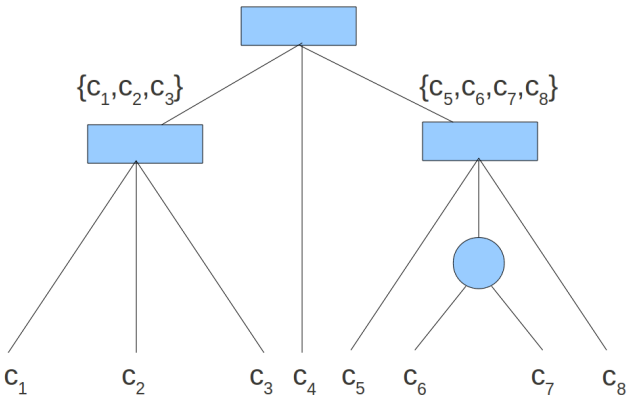


Figure 2.5: The generalized PQ-tree

2.5.4 Constructing the generalized PQ-tree

Now let us discuss the linear time algorithm for finding the generalized PQ-tree proposed by McConnell in [38]. We first find the nodes of the decomposition tree of $N(F)$. Then the decomposition tree of $S(F)$ can be obtained. The only thing that remains is to find those degenerate nodes in decomposition tree of $S(F)$ that are linear in the decomposition tree of $W(F)$ which is equivalent to the generalized PQ-tree of F.

2.5.4.1 Finding nodes of the generalized PQ-tree

Consider a connected component $\mathcal{C}$ of the overlap graph of F, O_F. McConnell defined a notion of blocks which correspond to internal nodes of the generalized PQ-tree.

Definition 2.5.15. *The* equivalence relation $B_{\mathcal{C}}$ on $\cup\mathcal{C}$ *between any two* $c, c^{'} \in \cup\mathcal{C}$ *is that,* $cB_{\mathcal{C}}c^{'}$ *if and only if the family of members of* $\mathcal{C}$ *that contain* c *is the same as the family of members of* $\mathcal{C}$ *that contains* $c^{'}$.

Definition 2.5.16. Blocks of the connected component $\mathcal{C}$ *are the equivalence classes of the equivalence relation* B_c *on* $\cup\mathcal{C}$.

Example 2.5.17. *Let* F *be the same as the one in example 2.5.14. The overlap graph is shown in Figure 2.6. Using definition 2.5.16, the blocks are:* $\{c_1\}$,$\{c_2\}$,$\{c_3\}$,$\{c_4\}$,$\{c_5\}$, $\{c_6, c_7\}$,$\{c_8\}$.

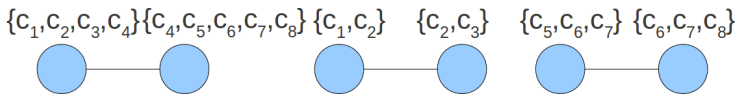


Figure 2.6: $O_(F)$

Theorem 2.5.18. *[38] $X \subseteq V$ is a node of the decomposition tree of $N(F)$ if and only if it is one of the following :*

1. *V or a one-element subset of V,*
2. *$\cup\mathcal{C}$ for some connected component $\mathcal{C}$ of F's overlap graph,*
3. *A block of a connected component of F's overlap graph.*

The next step is to find the spanning tree of the connected components. Dahlhaus showed in [15] that the overlap components can be found in linear time without computing the overlap graph.

2.5.4.2 Finding the spanning tree of an overlap component

Definition 2.5.19. *Let L, be the sorted list of members of F in ascending order of cardinality. $Max(X)$ denotes the rightmost member $Y \in L$ that overlaps X.*

In order to find the spanning tree of each component, for each node X we compute $Max(X)$. It is trivial to see that X and $Max(X)$ are adjacent in the overlap graph.

Lemma 2.5.20. *[38] If $X, Y \in L, X \cap Y \neq 0$; and Y lies between X and $Max(X)$ in L, then Y overlaps either X or $Max(X)$ and hence they belong to same connected component.*

Dahlhaus then introduced a graph $G_c(F)$ whose connected components are the same as the connected components of the overlap graph of F and edges are based on Lemma 2.5.20, i.e. if X, $Y \in F$ where Y is before X in L and $x \in X, Y$, then there is an edge between

these two members of F if there exists $Z \in F$ such that $x \in Z$, $Max(Z) = Y$ or $Max(Z)$ appears after Y in L. To find the spanning tree of a component $\mathcal{C}$ it suffices to traverse the component of $G_c(F)$ using a graph traversal algorithm. McConnell modified the Dahlhaus' algorithm and proved that finding the spanning tree of an overlap graph of F can be done in linear time.

2.5.4.3 Finding the ordering of children of Q nodes using partition refinement

By having the decomposition tree of $N(F)$ we can obtain the decomposition tree of $S(F)$, then we can obtain the generalized PQ-tree by checking which degenerate nodes should be linear in the decomposition tree of $W(F)$.

Definition 2.5.21. *A set family F is* simple *if its generalized PQ-tree has one internal node, and depending on type of the node it is either degenerate, prime or linear.*

Suppose that a set family is simple and prime, consider the internal node X of its decomposition tree along with its children $X_1, X_2, \ldots, X_k$. Since the node is prime, using Theorem 2.5.6, the union of two children of X or more is not a member of F, therefore members of F are either the domain set V or its singleton elements. Conversely it can be proved that when any member of F is either V or $\{v\}$ for $v \in V$, then F is simple and prime. This can be recognized in a linear time. McConnell proposed a partitioning algorithm similar to the one in [27]. This algorithm clarifies whether a simple set family is linear or degenerate in linear time.

McConnell showed that the partitioning algorithm maintains the following four criteria:

1. The processed members of the connected component induce a subtree of the spanning tree T of the overlap component.

2. The members of $\mathcal{P}$ are the blocks of the processed members of F.

3. The ordering of members of $\mathcal{P}$ is consistent with a consecutive-ones ordering on the processed members of connected component i.e. if x is a member of an earlier partition class of $\mathcal{P}$ than y, then x is earlier than y in the consecutive ones ordering.

4. The ordering of members of $\mathcal{P}$ and its reverse are the only possible orderings of $\mathcal{P}$ that are consistent with a consecutive ones ordering on the processed members of the connected component.

Suppose that X is a degenerate node in the decomposition tree of $S(F)$. In Section 2.5.3 we discussed that every member of $\mathcal{P}$ is a member of $N(F)$. Also the quotient of X in the decomposition tree of $S(F)$ is simple and it is either degenerate or linear. The algorithm takes a simple family as input. Every node of the generalized PQ-tree has a quotient and every quotient is a simple family. Since the quotients comprise F, to detect whether F has the C1P, the algorithm must be successively run on every quotient. The quotient of a prime node is trivial, so it suffices to consider the quotients of non-prime nodes. Every non-prime quotient is either degenerate or linear. Since quotients represent disjoint subfamilies, it suffices to give an algorithm that determines whether a simple family is degenerate or linear. The partitioning algorithm chooses an unprocessed set $R \in \mathcal{Q}$ and refines the partition $\mathcal{P}$ in each iteration.

The first iteration is an arbitrary $Y \in \mathcal{Q}$, and partition $\mathcal{P} = (Y)$, with $S = V - Y$. Let $P = (X_1, \ldots, X_k)$ be our current partition and let $S_X = V - X - \bigcup P$, we check for the following:

- S_R is empty and $X_{i+1}, \ldots, X_{j-1}$ are contained in R, or
- S_R is nonempty, $i = 1$, and $X_1, \ldots, X_{j-1}$ are contained in R, or
- S_R is nonempty, $j = k$, and $X_{i+1}, \ldots X_j$ are contained in R.

If none holds, then the quotient $\mathcal{Q}$ must be degenerate. Otherwise, $\mathcal{P}$ is refined by R.

Now suppose that $\mathcal{Q}$ is a linear family, if the first case happened, then $\mathcal{P}$ after refining by R would be $\mathcal{P} = (X_1, X_2, \ldots, X_{i-1}, X_i - R, X_i \cap R, X_{i+1}, \ldots, X_{j-1}, X_j \cap R, X_j - R, \ldots, X_k)$. If the second case happened, then $\mathcal{P}$ is modified to $\mathcal{P} = (S_R, X_1, X_2, ..., X_{j-1}, X_j \cap R,$ $X_j - R, \ldots, X_k)$. Finally if the third case happened $\mathcal{P}$ is modified to $\mathcal{P} = (X_1, X_2, ..., X_{i-1},$ $X_i - R, X_i \cap R, X_{i+1} \ldots, X_k, S_R)$. By criteria 4, the linear ordering on members of $\mathcal{P}$ gives

the linear ordering of children of Q nodes. McConnell in [38] proved that this algorithm can be done in linear time for all quotients.

2.5.5 Generalized PQ-tree of Tucker patterns

In Section 2.5 we discussed how to find the generalized PQ-tree of a binary matrix. In here we find the generalized PQ-tree of Tucker patterns discussed in Section 2.1. The columns of each pattern are members of the domain set V and the rows are members of the set family F. It can be seen that the non-overlapping family of F, $N(F)$ is the set of all singleton elements of V, and V itself, i.e. $N(F) = \{\{c_1\}, \ldots, \{c_{k-1}\}, \{c_k\}, \{c_1, \ldots, c_{k-1}, c_k\}\}$. Also the overlap graph associated to each patterns has only one component, hence the blocks are same as members of $N(F)$. The overlap graph of each pattern is shown in Figure 2.7. Using Theorem 2.5.18 we can find the decomposition tree of $N(F)$ of each pattern and then the decomposition tree of $S(F)$ which is shown in Figure 2.8. This tree is same for all the five patterns and only k changes. The members of the partition class $\mathcal{P}$ are the singleton elements of V, $c_1, \ldots, c_k$. Using the partitioning algorithm, the type of the Q node of the decomposition tree of each pattern is degenerate, since the conditions do not hold. For each pattern there exist X_i, X_j, X_k that X_i, X_k belong to the same member of F that does not contain X_j. For example for the first pattern $X_1 = c_1$ and $X_k = c_k$ are present in $r_k \in F$, but $X_j = c_l$ $(l = 2, \ldots, k-1) \notin F$.

If a binary matrix M contains at least one of these patterns, then its generalized PQ-tree has degenerate nodes and probably no linear nodes, and therefore does not have the C1P. In Chapter 3 we discuss certificates for a matrix that does not have the C1P which are based on Tucker patterns.

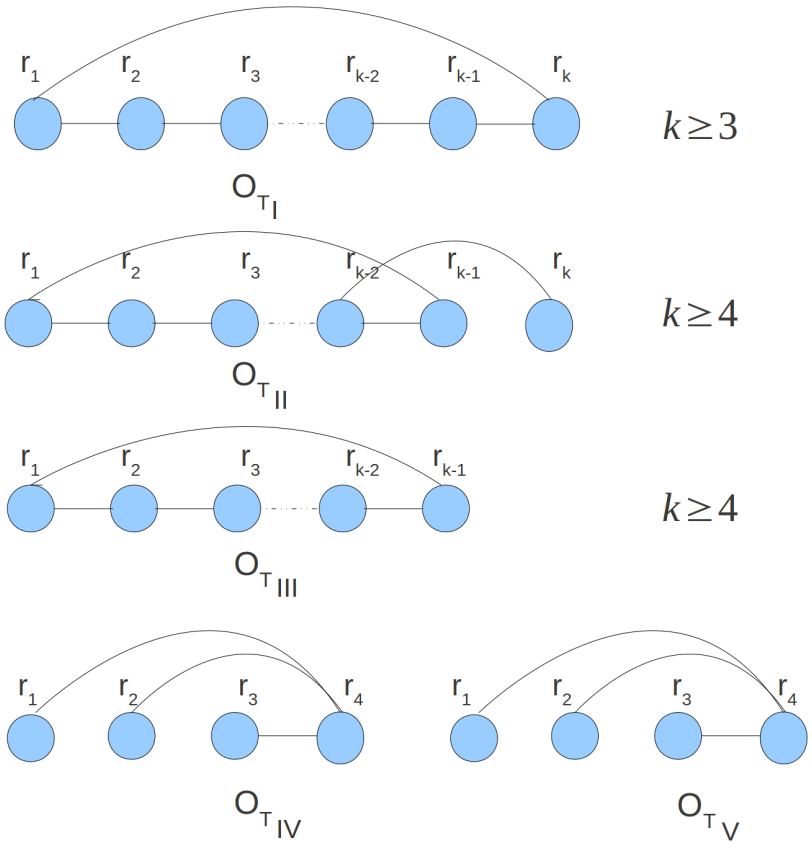


Figure 2.7: The overlap graphs of Tucker patterns

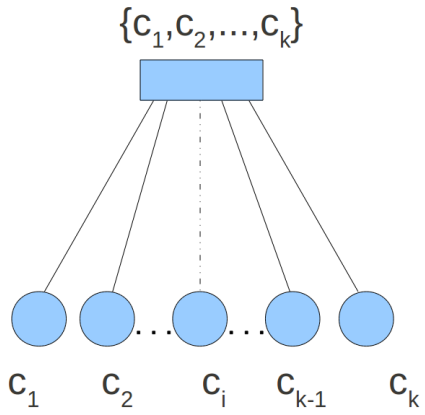


Figure 2.8: The decomposition tree of Tucker patterns

Chapter 3

Incompatibility graph certificates

We discussed in Chapter 1 that the problem of deciding whether a given binary matrix has the C1P can be solved efficiently. Checking the claim that a matrix has the C1P is easy with providing a valid permutation of the columns. However it is not as obvious how to verify that a matrix is not C1P. A *certifying algorithm* is an algorithm which produces a proof with an output that can be verified by a polynomial-time algorithm. For example a certifying algorithm for checking whether a graph G is bipartite outputs an odd cycle as a certificate when G is a non-bipartite graph. In this chapter we describe McConnell's certifying algorithm in [McConnell, SODA 2004 768-777] for a matrix that does not have the consecutive ones property. This uses the incompatibility graph of a binary matrix. He showed that odd cycles of this graph proves the non-C1Pness of the given binary matrix. McConnell claimed a bound of $k+2$ for the smallest odd cycle contained in the incompatibility graph of a non-C1P matrix, where k is the number of columns. We show that the small certificate always exists using Tucker patterns and correct the bound to $k+3$ when k is odd.

3.1 Incompatibility graph

McConnell, in [38] defined an elegant certificate for non-C1P matrices. He introduced the notion of the *incompatibility graph* of a binary matrix, and proved that a matrix is C1P if and only if this graph is bipartite. Hence, an odd cycle in this graph is a non-C1P certificate.

Definition 3.1.1. *Let M be an $m \times n$ binary matrix with rows $R_M = \{r_1, r_2, \ldots, r_m\}$ and columns $C_M = \{c_1, c_2, \ldots, c_n\}$. The* incompatibility graph *of M is an undirected graph $G_M = (V, E)$, whose vertices are pairs (c_i, c_j) (for $i, j = 1, \ldots, n, i \neq j$). Two vertices (c_i, c_j) and (c_j, c_k) are adjacent, if 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 $M_{lj} = 0$.*

It can be seen that there are two vertices for each unordered pair $\{c_i, c_j\}$. Edges of the incompatibility graph represent incompatible pairs of orderings, i.e. each edge corresponds to two relative orderings of the columns that cannot appear simultaneously in a consecutive ones ordering of the matrix. So for instance when (c_i, c_j), (c_j, c_k) are adjacent, there is no consecutive ordering that places c_j between c_i, c_k. McConnell noted that the incompatibility graph is bipartite if and only if the matrix is C1P. Thus odd cycles in the incompatibility graph certify that a matrix is not C1P.

The *forcing graph* $F_M = (V, E')$ is an undirected graph whose vertex set is same as that of G_M and whose edge set is a set of all pairs $((c_i, c_j), (c_k, c_j))$ where $((c_i, c_j), (c_j, c_k))$ is an edge of G_M. It is not hard to see that the incompatibility graph and the forcing graph both have $n(n-1)$ vertices and are symmetric. As with the incompatibility graph, the forcing graph can be used to certify that a matrix is not C1P: a path in this graph from (c_i, c_j) to (c_j, c_i), represents a chain of implications ("forcings") leading to a contradiction.

In fact, McConnell observed that these certificates are almost the same: such a path in the forcing graph can be transformed to a cycle in the incompatibility graph and vice versa [38]. The following statement is from McConnell, we provide an alternate proof.

Lemma 3.1.2. *If there exists a path with m vertices in F_M between (c_i, c_j) and (c_j, c_i) there is an odd cycle of length $m-1$ (if m is even) or m (if m is odd) in G_M containing the vertex (c_i, c_j). Conversely, if there is an odd cycle of length m in G_M containing (c_i, c_j) there is a path with at most $m+1$ vertices in F_M from (c_i, c_j) to (c_j, c_i).*

Proof. Without loss of generality the path in F_M is:

$$P: \quad (v_1 = (c_i, c_j), v_2, v_3, v_4, \ldots, v_m = (c_j, c_i)).$$

Let $v_k^{'} = (c', c)$ when $v_k = (c, c')$. Then in G_M we can build the walk:

$$P': \quad (v_1, v_2', v_3, v_4', \ldots, v_m^{(')})$$

When m is even, the final vertex in this walk is $v_m' = v_1$, and we have a cycle with $m-1$ vertices. When m is odd, the final term is v_m, and we can complete the cycle using the type 1 edge $(v_m, v_m' = v_1)$. In this case the odd cycle has length m.

Similarly, an odd cycle of length m in G_M can be transformed into a walk of length $m+1$ in F_M by performing the reverse operation on the walk with even length $m+1$ by taking the cycle vertices starting an ending in v_1. Note that if this path contains some type 1 edge (v_k, v_k') in G_M, this becomes a trivial edge (v_k, v_k) or (v_k', v_k') in F_M and should be contracted, reducing the length of the found path. □

Given G_M (F_M), we define G_M^1 (F_M^1) and G_M^2 (F_M^2) to be the subgraphs induced by the vertex sets $V_1 = \{(c_i, c_j) \mid i < j\}$ and $V_2 = \{(c_i, c_j) \mid i > j\}$ respectively. We observe that the two pairs of subgraphs are isomorphic.

Suppose now that we build G_M and F_M graph for a given Tucker pattern M from Figure 2.2 by first generating the type 1 edges of G_M and then adding the edges generated by each row in turn, beginning at the top. Edges $e = ((c_i, c_j), (c_l, c_j))$ and $e^{'} = ((c_j, c_i), (c_j, c_l))$ in F_M are generated by triples (i, j, l) from a given row r exactly when $M_{ri}, M_{rl} = 1$ but $M_{rj} = 0$. The edges corresponding to the rows from the top of the matrix then come in pairs e, $e^{'}$, where one is contained in F_M^1 and the other in F_M^2. As we descend the rows, the ones in the rows are consecutive until we reach the final row, r_t, that has gaps between its ones entries.

The edges of F_M generated by the gaps in r_t, i.e. triples (i, j, l) of columns where $i < j < l$ are the only edges which go between V_1 and V_2. We call these edges *critical*.

3.2 Finding odd cycles using Tucker configurations

We now give a tight bound on the smallest odd cycle in the incompatibility graph of a non-C1P matrix using Tucker matrices.

Theorem 3.2.1. *The length of the smallest odd cycle in the incompatibility graph of a binary matrix with $k \geq 4$ columns is at most $k+2$ if k is odd or $k+3$ if k is even, and this bound is tight.*

We begin by remarking that since any non-C1P matrix M contains a Tucker pattern as a submatrix, we can restrict our attention to Tucker patterns when looking for short odd-cycles in the incompatibility graph. This is because if we look at the subgraph of the incompatibility graph induced by considering only the columns (vertices) and rows (edges) of M (G_M) containing the Tucker pattern, we get exactly the incompatibility graph of the submatrix, which has at most as many columns as M. So the upper bound for Tucker patterns holds for all M, and the worst case for a given number of columns will occur at a Tucker pattern.

We remark that for $k \leq 2$ all binary matrices have the C1P, and for $k = 3$ if a matrix is not C1P it must contain the Tucker pattern T_{I_3} as a submatrix, and thus have an odd cycle of length 3 in its incompatibility graph, see Section 3.2.1. For $k \geq 4$, the tight bound of $k+2$ or $k+3$ is attained by T_{III_k}, see Section 3.2.3. We proceed to analyze each Tucker pattern separately.

3.2.1 First Tucker pattern

T_{I_k} is shown in Figure 2.2 (a); it is a square matrix of size k where $k \geq 3$.

$$M = T_{I_k} = \begin{array}{c} \\ r_1 \\ r_2 \\ . \\ . \\ . \\ r_{k-1} \\ r_k \end{array} \begin{array}{c} \begin{array}{cccccccc} c_1 & c_2 & c_3 & . & . & . & c_{k-1} & c_k \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & . & . & . & 0 & 0 \\ 0 & 1 & 1 & . & . & . & 0 & 0 \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & . & . & . & 1 & 1 \\ 1 & 0 & 0 & . & . & . & 0 & 1 \end{pmatrix} \end{array}.$$

Lemma 3.2.2. *For $k \geq 3$, the length of the smallest odd cycle in the incompatibility graph of T_{I_k} is k when k is odd and $k+1$ when k is even.*

Proof. We find a path in F_M in T_{I_k} between (c_1, c_{k-1}) and (c_{k-1}, c_1). Since $M_{11}, M_{12} = 1$ but $M_{1\ k-1} = 0$, we have that $((c_1, c_{k-1}), (c_2, c_{k-1}))$ is an edge of F_M.
Similarly $((c_i, c_{k-1}), (c_{i+1}, c_{k-1}))$ is an edge of F_M for $i = 2, \ldots, k-3$ using row i of M. Using row $k-1$, we get that (c_{k-2}, c_{k-1}) forces (c_{k-2}, c_k) and using row $k-2$ that (c_{k-2}, c_k) forces (c_{k-1}, c_k). Observe that $e = ((c_1, c_{k-1}), (c_k, c_{k-1}))$ is a critical edge of F_M. Therefore $(c_1, c_{k-1}), (c_2, c_{k-1}), (c_3, c_{k-1}), \ldots, (c_{k-2}, c_{k-1}), (c_{k-2}, c_k), (c_{k-1}, c_k), (c_{k-1}, c_1)$ is a path with $k+1$ vertices in F_M. By Lemma 3.1.2, this gives the required cycle.

Finally, we note that if there is any odd cycle in the incompatibility graph of length less than k, we would derive a contradiction to the C1P using fewer than k columns, contradicting the minimality of the Tucker pattern. Thus the length of this odd cycle is in fact minimal.

□

3.2.2 Second Tucker pattern

For $k \geq 4$, T_{II_k} shown in Figure 2.2 (b) is a square matrix of size k. We use a strategy to find a cycle in the incompatibility graph of T_{II_k} that is similar to that of T_{I_k}.

$$M = T_{II_k} = \begin{array}{c} \\ r_1 \\ r_2 \\ . \\ . \\ . \\ r_{k-2} \\ r_{k-1} \\ r_k \end{array} \begin{array}{c} \begin{array}{ccccccccc} c_1 & c_2 & c_3 & . & . & . & c_{k-2} & c_{k-1} & c_k \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & . & . & . & 0 & 0 & 0 \\ 0 & 1 & 1 & . & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & . & . & . & 1 & 1 & 0 \\ 1 & 1 & 1 & . & . & . & 1 & 0 & 1 \\ 0 & 1 & 1 & . & . & . & 1 & 1 & 1 \end{pmatrix} \end{array}.$$

Lemma 3.2.3. *The smallest odd cycle in the incompatibility graph of T_{II_k} has length k when k is odd and $k+1$ when k is even.*

Proof. From row i of the matrix for $i = 1, \ldots, k-2$, we get that (c_i, c_k) forces (c_{i+1}, c_k). From row k, (c_1, c_k) forces (c_1, c_{k-1}). Finally, $((c_1, c_{k-1}), (c_k, c_{k-1}))$ is a critical edge. Then $(c_1, c_k), (c_2, c_k), \ldots, (c_{k-1}, c_k), (c_{k-1}, c_1), (c_k, c_1)$ is a path of $k+1$ vertices in the forcing graph of T_{II_k}. Using Lemma 3.1.2 we can find an odd cycle of length either k or $k+1$ containing all rows of the pattern. Again the minimality of the Tucker pattern ensures that we cannot have an odd cycle of length less than k. □

3.2.3 Third Tucker pattern

Now we consider the third Tucker pattern in Figure 2.2 (c) that has $(k-1)$ rows and k columns where $k \geq 4$.

$$M = T_{III_k} = \begin{array}{c} \\ r_1 \\ r_2 \\ . \\ . \\ . \\ r_{k-2} \\ r_{k-1} \end{array} \begin{array}{c} \begin{array}{ccccccccc} c_1 & c_2 & c_3 & . & . & . & c_{k-2} & c_{k-1} & c_k \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & . & . & . & 0 & 0 & 0 \\ 0 & 1 & 1 & . & . & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & . & . & . & 1 & 1 & 0 \\ 0 & 1 & 1 & . & . & . & 1 & 0 & 1 \end{pmatrix} \end{array}.$$

Lemma 3.2.4. *The smallest odd cycle in the incompatibility graph of the third Tucker pattern has length $k+2$ if k is odd and $k+3$ if k is even.*

Proof. In this case, because we need to prove a non-trivial lower bound, we will describe the full structure of F_M. The graph F_M for $k = 6$ is illustrated in Figure 3.1 and captures the features we are interested in. Consider first the portion of the graph generated by excluding the last row and column on T_{III_k}. In this case each row has a unique pair of 1 entries; these can be combined with any zero entry to get a forcing triple. The result is a triangular grid on V_1 (and symmetrically, V_2), where vertex (c_i, c_j) is connected to all of $(c_{i\pm1}, c_j)$ and $(c_i, c_{j\pm1})$ that are also vertices of V_1 with coordinates between 1 and $k-1$ and in increasing order. Now, returning our attention the last column, we see it generates a path $(c_1, c_k), (c_2, c_k), \ldots (c_{k-1}, c_k)$ by considering the pair of ones in each row in turn. These two components, and their symmetric copies in V_2 are the entirety of the graph if we exclude the final row.

The first zero from the final row combines with the many pairs of ones to connect the two components of V_1 by fusing $(c_1, c_2), (c_1, c_3), \ldots (c_1, c_{k-2})$ and (c_1, c_k) (but not (c_1, c_{k-1})) into a clique. Finally the second zero (in row $k-1$) produces the critical edges $(c_i, c_{k-1}), (c_k, c_{k-1})$ for $i = 2, \ldots, k-2$ and fuses all these vertices into a clique.

We can then see that a path with $(k+3)$ vertices in F_M from (c_1, c_k) to (c_k, c_1) is given by: $(c_1, c_k), (c_2, c_k), \ldots, (c_{k-1}, c_k), (c_{k-1}, c_2), (c_{k-1}, c_1), (c_{k-2}, c_1), (c_k, c_1)$. By Lemma 3.1.2,

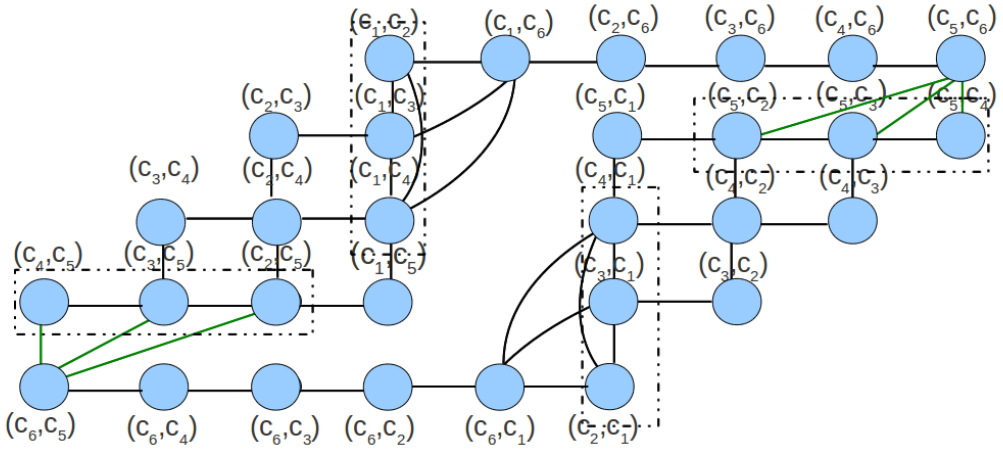
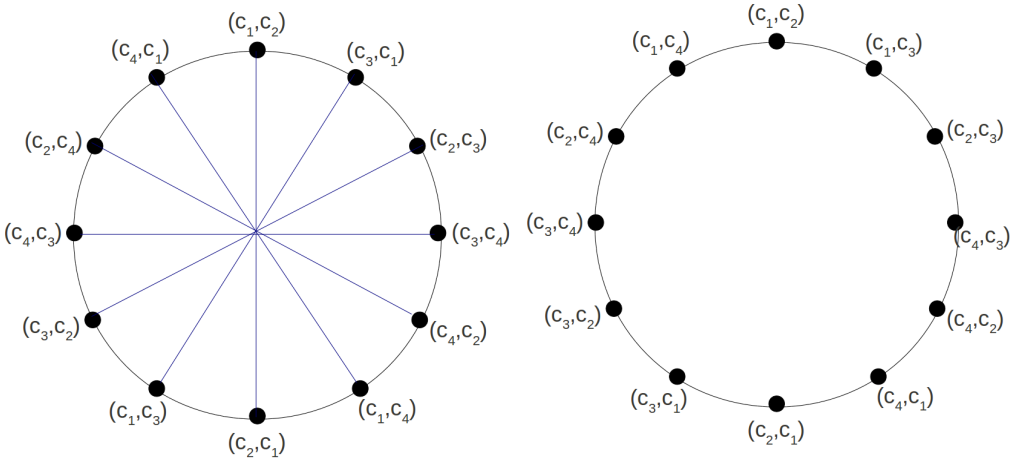


Figure 3.1: $F_{T_{III_6}}$

this gives the required cycle in the incompatibility graph of length $(k+3)$ if k is even, and $(k+2)$ if k is odd.

To prove that this is shortest, it suffices to show that this is the shortest path between (c_i, c_j) and (c_j, c_i) for some i, j in F_M, since if there is a shorter odd cycle in the incompatibility graph of length $(k+1)$ with k even, or k with k odd, by Lemma 3.1.2 there would be a path of length at most $k+2$ between some (c_i, c_j) and (c_j, c_i).

We can see that there is no shorter path by contracting the groups of vertices illustrated in Figure 3.1, i.e. (c_1, c_j) for $j = 2, 3, \ldots k-2$; (c_i, c_{k-1}) for $i = 2, 3, \ldots k-2$; and the symmetric groups on F_{V_2}. This will not increase the distance between any pair of vertices. We can see that what remains is $2k-2$ cycle with vertices (c_i, c_j) opposite (c_j, c_i), an additional part of F_{V_1} attached to the two contracted vertices, and symmetrically in F_{V_2}. The shortest path between opposite vertices on the cycle has $(k+3)$ vertices, as does the shortest path between any of additional vertices in F_{V_1} and those of F_{V_2}, though for some choices of (i, j) the shortest path from (c_i, c_j) to (c_j, c_i) may be longer. □



(a) G_M (b) F_M

Figure 3.2: Counter example to McConnell's Theorem 6.1

Taking $k \geq 4$ even, this gives a family of counter examples to Theorem 6.1 of McConnell in [38]. For example, taking $k = 4$, we have $M = \left(\begin{smallmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{smallmatrix}\right)$. Then F_M is a 12-cycle and G_M is a 12-cycle with 6 chords added between opposite vertices of the cycle. It is clear that the smallest odd cycle is of length 7. This graph is shown in Figure 3.2 (a).

3.2.4 Fourth Tucker pattern

The fourth Tucker pattern is of size 4 by 5, (see Figure 2.2 (d))

$$M = T_{IV} = \begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{ccccc} c_1 & c_2 & c_3 & c_4 & c_5 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array}.$$

With $((c_1, c_3), (c_5, c_3))$ as a critical edge of F_M. Here (c_1, c_3) forces (c_5, c_3), (c_5, c_3) forces (c_5, c_2), (c_5, c_2) forces (c_4, c_2), (c_4, c_2) forces (c_3, c_2) and (c_3, c_2) forces (c_3, c_1), which gives a path with 6 vertices in F_M and an odd cycle of length 5 in the incompatibility graph of

T_{IV}.

3.2.5 Fifth Tucker pattern

The last Tucker pattern has 6 columns and 4 rows (see Figure 2.2 (e)).

$$M = T_V = \begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ r_4 \end{array} \begin{array}{c} \begin{array}{cccccc} c_1 & c_2 & c_3 & c_4 & c_5 & c_6 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix} \end{array}.$$

It can be observed that $((c_2, c_3), (c_6, c_3))$ is a critical edge of F_M. Now (c_2, c_3) forces (c_6, c_3) which forces (c_6, c_4); (c_6, c_4) forces (c_5, c_4), (c_5, c_4) forces (c_5, c_2), (c_5, c_2) forces (c_5, c_1) which forces (c_6, c_1). (c_6, c_1) forces (c_4, c_1) which forces (c_4, c_2); (c_4, c_2) forces (c_3, c_2). This gives a path with 10 vertices in F_M and an odd cycle of length 9 in T_V. In this case the length of the smallest odd cycle also attains the bound of Theorem 3.2.1.

Combining these five case allows us to conclude Theorem 3.2.1.

3.3 Discussion of McConnell's proof

Let us consider the set representation of a binary matrix. McConnell in [38] modified the partitioning algorithm in which it returns a certificate when the set family F does not have the C1P. He showed that this algorithm returns an odd cycle of length at most $k + 2$. We showed in Section 3.2 that this bound is $k + 2$ when k is odd, $k + 3$ when k is even and proved that this bound is tight. McConnell did not specify which critical edges the modified partitioning algorithm chooses when the conditions fail. Depending on the choice, i.e. if the algorithm chooses $((c_k, c_{k-1}), (c_2, c_{k-1}))$, it will return an odd cycle of length up to $2k - 1$.

In the main theorem of [38], McConnell showed that when F does not have the C1P, the required conditions of the partitioning algorithm fails as R is processed. Therefore there exist X_i, X_j, X_k where $i < j < k$ (or $k < j < i$) such that $a, c \in R$, $b \notin R$ $a \in X_i$, $b \in X_j$ and $c \in X_k$. Therefore $((a, b), (c, b))$ is a critical edge of the forcing graph. In here we

briefly discuss how the modified partitioning algorithm finds the odd cycle as the certificate of non-C1Pness [37].

Definition 3.3.1. *Partition P of a set V is a* refinement *of a partition P' of V if each partition class in P is a subset of a set in P'.*

Example 3.3.2. *For example $P' = \{\{a,b\},\{c\},\{d\},\{e,f\}\}$ is a refinement of $P = \{\{a,b,c\}, \{d\},\{e,f\}\}$. On the other hand $P''\{\{a,b,d\},\{c\},\{e,f\}\}$ is not, because $\{a,b,d\}$ isn't a subset of any of the partition classes in P.*

By producing a sequence of refinements, we can use the inclusion tree. Each set appearing anywhere in these partitions is a node. Now the partitioning algorithm finds the *lowest common ancestor* of (a,b) and (c,b) in the inclusion tree which gives the shortest path between these nodes in the forcing graph that can be transformed into the smallest odd cycle in the incompatibility graph. This can be done in linear time [29]. McConnell used the similar approach as we did in Lemma 3.1.2.

Remark 3.3.3. *Running the partitioning algorithm of [38] on T_{III_k} may generate an odd cycle of length as much as $2k-1$ for the certificate, depending on which critical edge is processed from the last row.*

As we discussed in 2.5.5, the partitioning algorithm fails for Tucker patterns, and therefore it gives three columns c_i, c_j, and c_k, where $i \leq k \leq j$ (or $j \leq k \leq i$) such that there there exists a row which has entries 1 in c_i, c_k and 0 in c_k. So $((c_i,c_j),(c_k,c_j))$ would be a critical edge in forcing graph of the pattern which yields to an odd cycle in the incompatibility graph. But this cycle is not necessarily the smallest odd cycle.

For example, for the third Tucker pattern, considering $((c_{k-2},c_{k-1}),(c_k,c_{k-1}))$ as a critical edge yields to an odd cycle of length $2k-1$. The F_M^1 of the third pattern is illustrated in Figure 3.3.

In McConnell's paper, the claim in Corollary 6.1 of a path of length $k-1$ from (c_i,c_j) to (c_k,c_l) where $i<j, k<l$ in the forcing graph of length $k-1$ is false. For example in Figure 3.2 (b), it can be seen that there does not exist a path of length $k-1$, i.e. 3 from

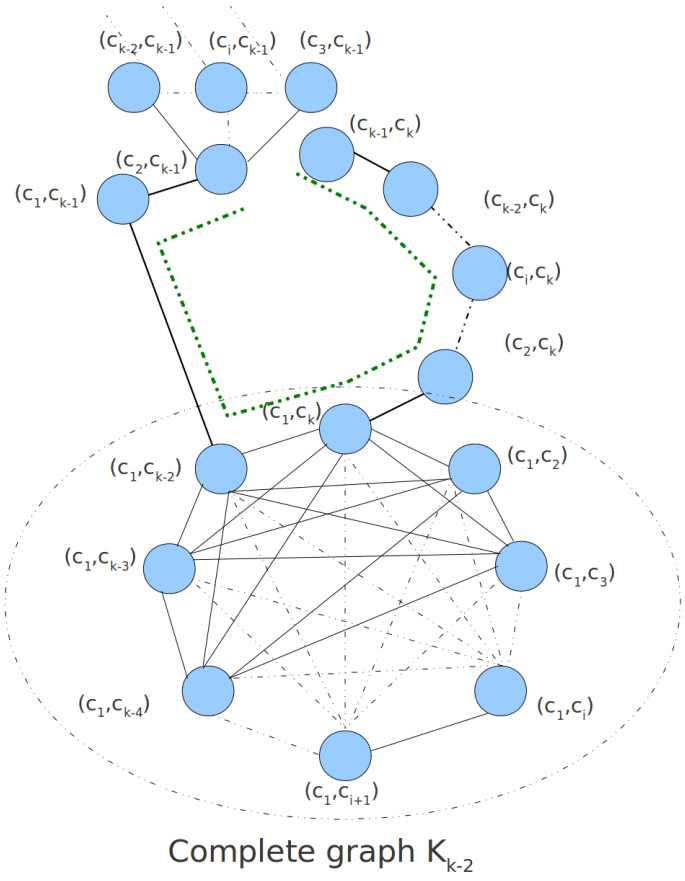


Figure 3.3: F_M^1

(c_2, c_3) to (c_2, c_4). One expects that this type of argument will produce some small linear, bound on the path length, McConnell [37] believes that $3k$ will work and is incorporating this into a revised version of [38].

Chapter 4

Minimal conflicting sets

4.1 Introduction

In Chapters 2 and 3 we discussed some algorithms for deciding whether a matrix has the consecutive ones property and that when a matrix does not have the C1P, there are some structures that obstruct the matrix from having the C1P. For purposes of ancestral genome reconstruction, we would like to identify rows known as *false positives* that contribute to such structures. A *minimal conflicting set (MCS)* is a subset of rows where any proper subset of these rows has the C1P. Thus a MCS contains a certificate of non-C1Pness. The reason is that the MCS is a minimal obstruction to the C1P, so it contains a Tucker pattern. The *maximal C1P (MC1P)* is another concept that is dual to MCS, it represents a subset of rows that have the C1P but adding any row to it results in a non-C1P matrix.

In this chapter we consider matrices with two 1s per row that do not have the C1P. This was studied in [10] using the joint generation algorithm [26, 21] which simultaneously generates all MCS/MC1P's. We discuss simple ways of constructing small C1P problems that have many MCS/MC1P's. We first consider two examples that have many MCS's and we enumerate MCS's for each case for the purpose of computing statistics.

The running time of the joint generation algorithm depends on the quantity of $|MC1P|+|MCS|$, and we expect that performance for generating MCS's will depend on the number of

MC1P's. So when the number of MC1P's is much greater than the number of MCS's, then the generation is slow for MCS's. In the general joint generation problem, the representation of the two dual functions may be quite large. We show in Section 4.4 that this is also the case for the MCS/MC1P dual functions arising from the C1P problem.

In the case where we have many MC1P's but few MCS's, we would like to see if there is a shorter representation of the MCS's. We see in these examples that is often possible to divide the problem of enumerating MCS's into subproblems where each MCS can be understood as different path through a fixed set of obstacles. This is an initial step in understanding their structures, and by finding each structure we can enumerate MC1P's. We are motivated in part by the synthetic data experiments of [10] where we see many MC1P's per MCS and much repetitions in the list of MC1P's.

4.2 Basic definitions

When a binary matrix M has exactly two 1s per row, then each row r can be labeled by the pair of columns (c_i, c_j), where $M_{r,i}, M_{r,j} = 1$. This class of binary matrices can be represented by an undirected graph.

Definition 4.2.1. *The* row-column graph C_M *corresponding to a binary matrix M is the undirected graph whose vertices are columns of M and two vertices c_i and c_j are adjacent if and only if there exists a row labeled by (c_i, c_j).*

Example 4.2.2. *Let M be a binary matrix below.*

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

Rows of M can be labeled by $(c_1, c_2), (c_2, c_3), (c_3, c_4), (c_1, c_4)$. The row-column Graph of matrix M is shown in Figure 4.1

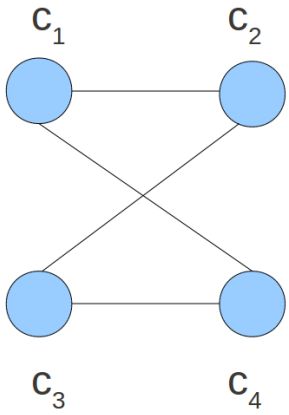


Figure 4.1: $C_{T_{I_4}}$

In chapter 3 we discussed Tucker patterns as certificates of non-C1Pness. These patterns are illustrated in Section 2.1. It can be observed that for a matrix M with two 1s per row, the only Tucker patterns that can be contained in M, are the first Tucker pattern T_{I_k} for all $k \geq 3$ and the third Tucker pattern T_{III_4}. The row-column graph of the first Tucker pattern T_{I_k} corresponds to a cycle of size k. Figure 4.1 shows the row-column graph of T_{I_4}. The row-column graph C_M of T_{III_4} corresponds to a *claw*, a star graph with three edges, three leaves, and one central vertex. Hence finding all Tucker patterns of M is equivalent to finding cycles and claws in its corresponding row-column graph. We are particularly interested in enumerating minimal conflicting sets of a binary matrix generated by permutation of two 1s in some of its columns.

4.3 Examples of binary matrices with many MCS's

In this section we show that it is possible to have very large number of MCS/MC1P's on a small matrix. We consider two examples of binary matrices that have many MCS's and therefore many Tucker patterns as a submatrix. These matrices are extreme cases, i.e. they have the maximum number of MCS's among all matrices of the same dimensions.

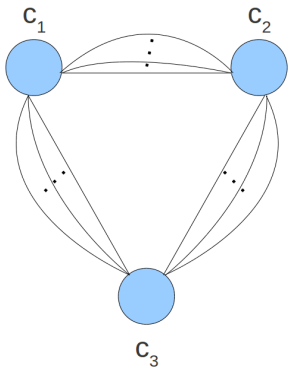


Figure 4.2: $C_{M_{I_k}}$

4.3.1 Example 1: Effect of repeated rows

We consider a binary matrix M_{I_k} with three columns and three distinct rows, where each row is repeated k-times. These rows are permutation of 1s in three columns.

$$M_{I_k} = \begin{array}{c} \\ r_1 \\ . \\ r_{k+1} \\ . \\ r_{2k+1} \\ . \\ r_{3k} \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ . & . & . \\ 0 & 1 & 1 \\ . & . & . \\ 1 & 0 & 1 \\ . & . & . \\ 1 & 0 & 1 \end{pmatrix} \end{array}$$

When $k = 1$, this matrix is equivalent to T_{I_3}, and thus has 1 MCS= $\{r_1, r_2, r_3\}$. When $k = 2$, it contains 8 Tucker patterns of same size, T_{I_3}, and therefore 8 MCS's (see Example 4.3.2).

The corresponding row-column graph of M_{I_k} is a multigraph with three vertices, and k edges between any two vertices, see Figure 4.2. This graph has many cycles of size three that corresponds to the first Tucker pattern of size 3. It is trivial to see that $C_{M_{I_k}}$ does not have any claw, and thus M_{I_k} does not contain any T_{III_4}.

Theorem 4.3.1. *Let M_{I_k} be a binary matrix with 3 columns and $3k$ rows. M_{I_k} has k^3*

MCS's.

Proof. There are exactly k^3 cycles formed by picking one of the k edges between each pair. This is equal to the number of first Tucker pattern T_{I_3} contained in M_{I_k}. Since the set of rows of each T_{I_3} is a MCS, thus the total number of MCS's are equal to k^3. □

For matrices with 3 columns, there are only three possible rows, if we have k_1, k_2, k_3 of each, there are $t = k_1 \cdot k_2 \cdot k_3$ such MCS's. For $k_1 + k_2 + k_3 = k$, t is maximized when $k_1 = k_2 = k_3 = k$. Therefore M_{I_k} has the maximum number of MCS's between matrices of size $3k$ by 3. It is obvious that M_{I_k} has only 3 MC1P's, each of which has $2k$ rows.

Example 4.3.2. *Let $k = 2$, then the matrix M_{I_2} is:*

$$M_{I_2} = \begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} \end{array}.$$

Minimal conflicting sets are: $\{r_1, r_3, r_5\}, \{r_1, r_3, r_6\}, \{r_1, r_4, r_5\}, \{r_1, r_4, r_6\}, \{r_2, r_3, r_5\}, \{r_2, r_3, r_6\}, \{r_2, r_4, r_5\}, \{r_2, r_4, r_6\}$. *Each MCS represents edges of the cycle of size 3 which corresponds to the first Tucker pattern of size 3.*

4.3.2 Example 2: Exponential behaviour of MCS's

Now we consider a binary matrix M_{II_k} that has k columns and $\binom{k}{2}$ rows.

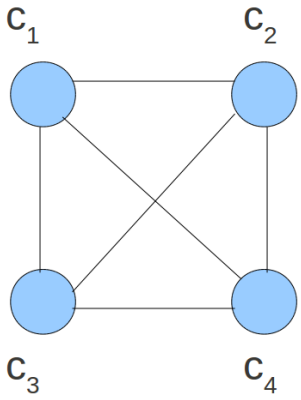


Figure 4.3: $C_{M_{II_4}}$

$$M_{II_k} = \begin{array}{c} \\ r_1 \\ r_2 \\ r_3 \\ . \\ . \\ . \\ r_{\frac{k(k-1)}{2}} \end{array} \begin{array}{c} \begin{array}{ccccccccc} c_1 & c_2 & c_3 & . & . & . & c_{k-2} & c_{k-1} & c_k \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & . & . & . & 0 & 0 & 0 \\ 1 & 0 & 1 & . & . & . & 0 & 0 & 0 \\ 1 & 0 & 0 & . & . & . & 0 & 0 & 0 \\ 0 & 0 & 0 & . & . & . & 0 & 0 & 0 \\ 0 & 0 & 0 & . & . & . & 1 & 1 & 0 \\ 0 & 0 & 0 & . & . & . & 1 & 0 & 1 \\ 0 & 0 & 0 & . & . & . & 0 & 1 & 1 \end{pmatrix} \end{array}$$

In order to find the total number of MCS's we need to find the Tucker patterns, and therefore we need to enumerate all cycles of size l for $l = 3, 4 \ldots, k$ and the claws.

Example 4.3.3. *As a base case let $k = 4$. The row-column graph of M_{II_4} is shown in Figure 4.3. This graph has 4 cycles of size 3, and 3 cycles of size 4. Minimal conflicting sets for corresponding to these cycles are $\{r_1, r_2, r_4\}, \{r_1, r_3, r_5\}, \{r_2, r_3, r_6\}, \{r_4, r_5, r_6\}, \{r_1, r_3, r_4, r_6\}$, $\{, r_1 r_2, r_5, r_6\}, \{r_2, r_3, r_4, r_5\}, \{r_2, r_4, r_6\}$. Also $\{r_1, r_2, r_3\}, \{r_1, r_4, r_5\}, \{r_2, r_4, r_6\}, \{r_3, r_5, r_6\}$ correspond to claws of $C_{M_{II_4}}$, where each vertex of $C_{M_{II}}$ is a central vertex of a claw. Therefore M_{II_4} has total of 11 MCS's.*

It can be observed that the row-column graph of M_{II_k}, $C_{M_{II_k}}$ is a complete graph (see

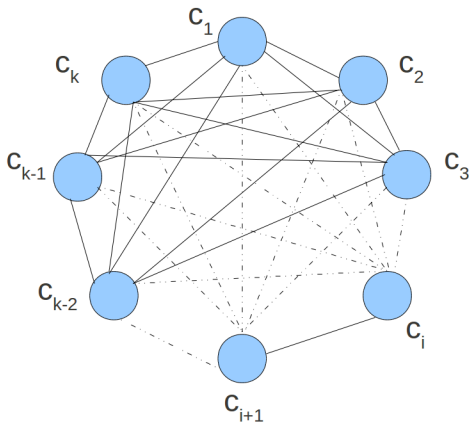


Figure 4.4: $C_{M_{II_k}}$

Figure 4.4).

Theorem 4.3.4. *Let M_{II_k} be a binary matrix with k columns and $\frac{k(k-1)}{2}$ rows. M_{II_k} has $\sum_{l=3}^{k} \frac{1}{2} \cdot \binom{k}{l} \cdot (l-1)! + k \cdot \binom{k-1}{3} = \frac{k}{4} \left[2\ {}_3F_1\ (1,1,1-k;2;-1) - k - 1\right] + \frac{(k-1)(k-2)(k-3)}{6}$ MCS's, where ${}_pF_q(a_1,...,a_p;b_1,...,b_q;x)$ is the generalized hypergeometric function.*

Proof. Finding all cycles of a complete graph K_n can be found by computing $\sum_{k=3}^{n} \frac{1}{2} \cdot \binom{n}{k} \cdot (k-1)!$ [9] and can be written as a generalized hyper geometric function [18]. Also each vertex in K_n is connected to $n-1$ vertices, hence the total number of claws centered in a vertex v_i of K_n $(i = 1, 2 \ldots, n)$ is equal to $\binom{n-1}{3}$, number of ways of picking 3 outcomes from $n-1$ possibilities. □

For enumerating MC1P's of M_{II_k} it suffices to enumerate all subgraphs that do not contain claws and cycles and have maximal number of edges. This is equivalent with enumerating Hamiltonian cycles with one edge removed. The total number of Hamiltonian cycles in $C_{M_{II_k}}$ is $\frac{(k-1)!}{2}$ Thus the total number of MC1P's is equal to $\frac{k!}{2}$.

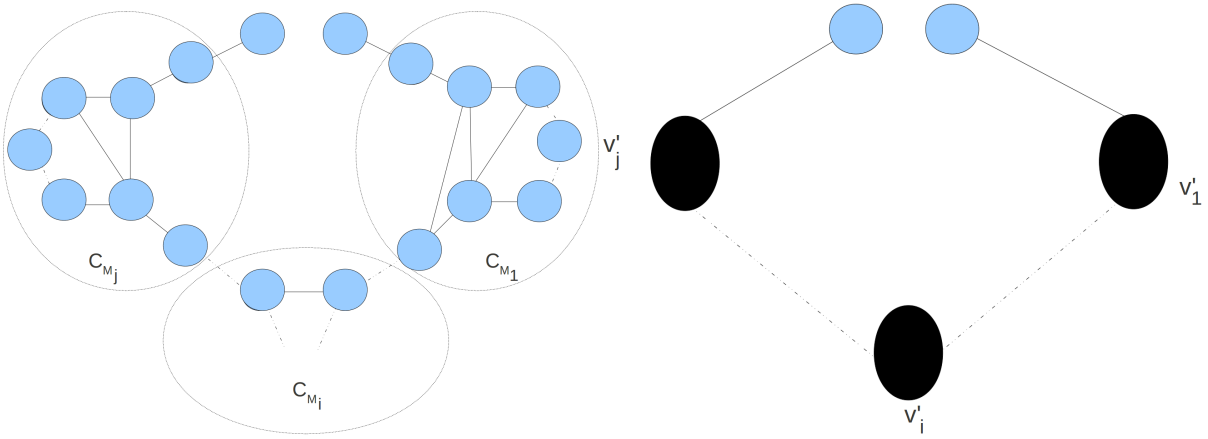
4.4 Enumerating the Maximal C1P

A related interesting problem for a non-C1P matrix M is enumerating MC1P's when there is few MCS's. From the definition of MCS and MC1P, one can be obtained from the other, however it is not easy. Chauve et al. described that the problem of generating MCS/MC1P's can be reduced to the problem of generating *Minimal True Clauses* (MTC) and *Maximal False Clauses (MFC)* of monotone boolean functions. This can be done by a quasi-polynomial time algorithm of Fredman and Khachian [21]. Chauve et al. in [10] performed experiments where in real data, these turned out to be many MCS's where on the other hand in simulated data there were few MCS's but many MC1P's. In this section we show that in simulated data it is possible to break the problem of enumerating MC1P's to enumerating MC1P's in the subgraphs of the row-column graph. Thus the total number of MC1P's equals to the product of total number of MC1P's found in each subgraphs.

For this problem by studying the row-column graph, C_M, we can obtain the MC1P by detecting subgraphs that do not contain cycles and claws, and adding any edge to these subgraphs results in a cycle or a claw. This can be also done by first testing the graph for acyclicity in linear time using a depth-first search, and then by checking whether it is a claw-free graph in polynomial time [19].

4.4.1 Decompositions of the row-column graph

The problem of enumerating MC1P's of a binary matrix M can sometime be divided into separate subproblems where each subproblem is a subgraph C_{M_j} of C_M. We are interested in the case where the graph C_M has cut-edges, as this is often the case for our synthetic data experiments. For each C_{M_j}, we then find subgraphs that are acyclic and claw-free, i.e. those subgraphs of C_{M_j} that are forest and do not contain claws. Counting these subgraphs gives us the total number of MC1P's contained in a submatrix M_j of M. We then put vertices of each C_{M_j} into a group and treat them as a pseudonode $v_j^{'}$ of C_M. Replacing each C_{M_j} with its claw-free acyclic subgraphs and expanding the corresponding pseudonode $v_j^{'}$, yields a row-column graph of the C1P matrix. Therefore the total number of MC1P's of M is



(a) C_M (b) C_M

Figure 4.5: C_M. Dotted circles shows subgraphs that contains claws or cycles

equal to the product of the total number of MC1P's for each M_j (See Figure 4.5) .

4.4.2 Finding MC1P's of the simulated data

We consider 10 simulated data used in [10] with 40 markers and 44-45 ancestral syntenies. Each data set represented by a binary matrix M with 40 columns, and 44-45 rows. Some rows of M do not have the C1P, i.e. are false positives. Each matrix has exactly two 1s per row and there are 39 rows in each matrix that have consecutive 1s and 5-6 false positives, i.e rows that break the C1P. It is trivial to see that the rows of each matrix can be reordered such that the i-th row has only 1s in i and $i+1$ columns, i.e. $r_i = \{c_i, c_{i+1}\}$ for $i = 1, \ldots, 39$. Each row that is a false positive can be represented by a pair (c_i, c_j) where $2 \leq j - i \leq 3$. They are generated uniformy at random. Because of this we expect (and observe) that most of the MCS's and MC1P's represent local obstacles, which allows us to decompose the full MC1P into several local configurations. Table A.1 shows these pairs for each data set.

For each data set we first considered its corresponding row-column graph C_M, and found subgraphs $C_{M_1}, \ldots, C_{M_j}$ as *configurations* of the graph. We found total of 12 configurations

Table 4.1: False positives of the simulated data

Data	False positives					
Data 1	(c_{11}, c_{13})	(c_{13}, c_{16})	(c_{16}, c_{19})	(c_{23}, c_{26})	(c_{26}, c_{29})	
Data 2	(c_2, c_4)	(c_{11}, c_{13})	(c_{33}, c_{35})	(c_{35}, c_{38})	(c_{37}, c_{39})	
Data 3	(c_1, c_3)	(c_9, c_{11})	(c_{10}, c_{12})	(c_{25}, c_{28})	(c_{27}, c_{29})	(c_{36}, c_{38})
Data 4	(c_2, c_5)	(c_6, c_8)	(c_7, c_{10})	(c_{22}, c_{24})	(c_{23}, c_{25})	(c_{31}, c_{34})
Data 5	(c_{11}, c_{13})	(c_{12}, c_{15})	(c_{15}, c_{17})	(c_{16}, c_{18})	(c_{33}, c_{36})	(c_{36}, c_{39})
Data 6	(c_2, c_4)	(c_8, c_{11})	(c_{25}, c_{28})	(c_{26}, c_{28})	(c_{27}, c_{30})	(c_{30}, c_{32})
Data 7	(c_{10}, c_{13})	(c_{27}, c_{30})	(c_{29}, c_{31})	(c_{31}, c_{33})	(c_{33}, c_{35})	(c_{35}, c_{38})
Data 8	(c_2, c_5)	(c_3, c_5)	(c_9, c_{12})	(c_{30}, c_{32})	(c_{31}, c_{34})	(c_{34}, c_{37})
Data 9	(c_1, c_4)	(c_9, c_{12})	(c_{10}, c_{12})	(c_{13}, c_{15})	(c_{26}, c_{28})	
Data 10	(c_4, c_7)	(c_7, c_9)	(c_{13}, c_{16})	(c_{15}, c_{17})	(c_{22}, c_{25})	(c_{34}, c_{36})

for all data sets. These configurations are listed in Appendix A and the total number of claw-free acyclic subgraph for each configuration is shown.

Table 4.2: Simulated data results

Data	false positives	cycles	claws	MCS's	Configuration	MC1P's
Data 1	5	5	17	22	2,6	468
Data 2	5	6	12	18	3,8,8	560
Data 3	6	8	11	19	8,8*,11,12	1248
Data 4	6	8	12	20	9,11,(8,12)	2240
Data 5	6	8	16	24	2,4	962
Data 6	6	9	16	25	5,8,9	1500
Data 7	6	7	18	25	9,(7,9)	1615
Data 8	6	8	16	24	(2,8),9,10	1755
Data 9	5	6	11	17	8,9*,(8,10)	464
Data 10	6	7	14	21	1,8,9,12	3120

Table 4.2 shows the results. In this table * shows a configuration that contains either the first or the last column of the matrix. In this case the number of claw-free and acyclic subgraphs of the configurations is reduced by 1. In some cases, we found combination of two configurations that are shown in pairs in table 4.2. These configurations either share a node or are connected by an edge.

Chapter 5

Conclusion

5.1 Conclusion and results

We studied the consecutive ones property of binary matrices in detail. We discussed the PQ-tree to be representing a C1P matrix. We then consider other data structures that are extension of PQ-trees and are more useful since they can be applied to any binary matrix and have extra information when the matrix does not have the C1P. We discussed that all of these structures uses similar concepts.

We then discussed the incompatibility graph of a binary matrix, and reduced the C1P problem to a problem of testing graph bipartiteness which can be detected by graph algorithms that use two-colorability. Then we used the fact that the Tucker patterns are the minimal obstructions contained in a non-C1P matrix to find a tight bound on the smallest odd cycle of the incompatibility graph that can be applied to any non-C1P matrix.

We considered a class of binary matrices that have two 1s per row and study those matrices that have many minimal conflicting sets. We discussed that these matrices can be represented by a row-column graph whose vertices are labeled by columns of the matrix. It is proved that the cycles and claws of this graph correspond to the first and the third Tucker patterns respectively. We discussed strategies for enumerating the total number of MCS and MC1P using this property.

We also worked on some simulated data with many MC1P that can be represented by a binary matrix with two 1s per row. We discussed that one way to attack the problem of enumerating the total number of MC1P is to find all claw-free acyclic subgraphs of the row-column graph. We found configurations that are common in some of them, and we used graph theoretic methods to enumerate the total number of MC1P for each simulated data. These configurations can be seen as a certificate of non-C1Pness.

5.2 Future works

McConnell claimed the the partitioning algorithm can be modified, such that it outputs an odd cycle as a certificate of non-C1Pness. The advantage of this algorithm is that it can be done in linear time, but the disadvantage is that it does not necessarily output the smallest odd cycle. It would be challenging to develop a fast algorithm that always finds the shortest odd cycle of the incompatibility graph. Implementing an efficient algorithm for finding all odd cycles of the incompatibility graph leads to an algorithm of finding all Tucker patterns in a non-C1P matrix.

Appendix A

Configurations

Figure A.1 shows the 12 configurations (local obstructions) found in the synthetic data experiments of [10]. Since the edges and vertices of the row-column graph are labeled, all of these configurations have labeled edges.

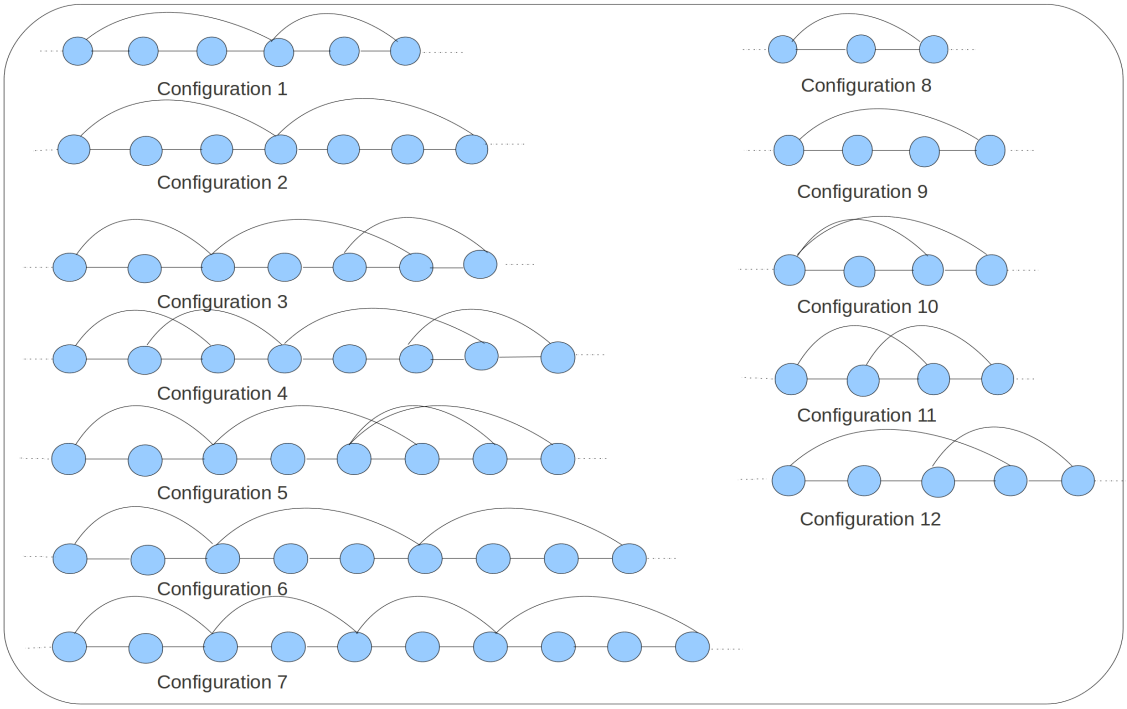


Figure A.1: Configurations found in simulated data

Table A.1: Number of claw-free and acyclic subgraphs of configurations

Configurations	Claw-free and Acyclic subgraphs
Configuration 1	12
Configuration 2	13
Configuration 3	35
Configuration 4	74
Configuration 5	75
Configuration 6	36
Configuration 7	99
Configuration 8	4
Configuration 9	5
Configuration 10	9
Configuration 11	8
Configuration 12	13

Enumerating the claw-free and acyclic subgraph of an undirected graph can be done in output linear time using a depth-first search.

Appendix B

MATLAB codes

B.1 Incompatibility graph

This appendix contains MATLAB code for computing the incompatibility graph. It reads a binary matrix M along with its name from a file and then plots the incompatibility graph of M using the code and stores its adjacency matrix which is the input to another program (B.2) which computes its cycles. Code B.3 finds all cycles of the incompatibility graph but it only keeps its odd cycles. It creates a folder for odd cycles of same length and stores them.

Program B.1: Incompatibility Graph

```
%The IncompatibilityGraphVersion2() reads the binary matrix M from the folder
%and draw its incompatibility graph
%INPUT: A binary matrix with two 1s per row along with its name
%OUTPUT: Draw the incompatibility graph with all edges labeled and Show the
%odd cycle if there is any.
%PURPOSE: Check the incompatibility of a given binary matrix for odd cycles
%which results in non-C1Pness of the matrix.
```

```
8  % AUTHOR:   Mehrnoush,Malekesmaeili, May 2011
9  % EMAIL:    mmalekes@sfu.ca
10 function []= IncompatibilityGraphVersion2(M,Name)
11 global A;
12 global EdgeSet;
13 global B;
14 global dirname;
15 count=0;
16 ColumnSize=size(M,2);
17 VertexSize=ColumnSize*(ColumnSize-1);
18 l=1;
19 A=zeros(VertexSize);
20 % B is an 2 by ColumnSize*(ColumnSize-1) matrix that keeps the vertices (a,b)
21 %for any two columns a,b
22 B=zeros(2,VertexSize);
23 D=zeros(size(M,2)*2*ColumnSize-2,2);
24 VertexSet=zeros(1,VertexSize);
25 C=zeros(2,size(M,1));
26 % The for loop Save adjacent vertices of type ((a,b) (b,a))
27 %in adjaceny matrix A
28 for j1=1:ColumnSize
29    j2=j1+1;
30    while(j2<=ColumnSize)
31     count=count+1;
32     B(:,count)=[j1 j2];
33     B(:,count+(VertexSize/2))=[j2 j1];
34
35     A(count,count+(VertexSize/2))=1;
36     A(count+(VertexSize/2),count)=1;
37     D(l,:)=[count count+(VertexSize/2)];
38     l=l+1;
39     j2=j2+1;
40    end
41 end
42 %This loop stores name of each vertex in an array
```

```
for j3=1:size(B,2)
    VertexSet(j3)=str2num((strcat(num2str(B(1,j3)),num2str(B(2,j3)))));
end
for k=1:size(M,1)
    C(:,k)=find(M(k,:));
    %Finds the member of the set family.(index of two columns of M having
    %entry 1 in a same row) and stores them
    EdgeSet(:,k)=C(:,k);
    % this loop finds all incompatible pairs of type ((a,b),(b,c))
    %and construct the adjacency matrix of the graph
    for i=1:ColumnSize
          if (i~=C(1,k) && i~=C(2,k))
        r1=find(all(repmat([C(1,k);i],1,size(B,2))==B));
        r2=find(all(repmat([i;C(2,k)],1,size(B,2))==B));
        c1=find(all(repmat([C(2,k);i],1,size(B,2))==B));
        c2=find(all(repmat([i;C(1,k)],1,size(B,2))==B));
        A(r1,r2)=1;
        A(r2,r1)=1;
        D(l,:)=[r1 r2];
        l=l+1;
        A(c1,c2)=1;
        A(c2,c1)=1;
        D(l,:)=[c1 c2];
        l=l+1;
         end
    end
end
% stores the edge list in a text file
save A;
StoreEdgeList(D);
mkdir(strcat('Result/',dirname,num2str(date)),Name)
 tStart=tic;
    PlotGraph(VertexSize,VertexSet,Name);
    RunLoopsVersion2(Name);
    tElapsed=toc(tStart);
```

```
78      save(strcat('Result/',dirname,num2str(date),'/',...
79      Name,'/tElapsed'),'tElapsed');
80  end
81  ————————————————————————————————
```

Program B.2: Store Edge list

```
1  %The StoreEdgeList(E) Stores the adjacency graph in a txt format.
2  %INPUT: The adjacency matrix E
3  %OUTPUT: AdjMatrix.txt
4  function []= StoreEdgeList(E)
5  fid = fopen('AdjMatrix.txt','w');
6  fprintf(fid,' %2d %3d\n ',E');
7  end
```

Program B.3: Find loops

```
1  function RunLoopsVersion2(Name)
2  %RunLoops21 Counts the number of loops in a network
3  % This code counts the number of loops (cycles) in a network (graph) that
4  % is composed of nodes and edges. It employs an iterative algorithm that
5  % transforms the network into a tree (the ILCA — Iterative Loop Counting
6  % Algorithm). This is a "brute force" technique as there are no known (to
7  % my knowledge anyway) algorithms for providing a good estimation.
8  %It  also checks the graph for odd cycles
9  %
10 % AUTHOR:   Joseph Kirk,2/2007
11 % EMAIL:    <jdkirk630@gmail.com>
```

```
% USAGE:    >> run_loops;
% Revised by Mehrnoush Malekesmaeili
%Slight changes on functions
%New functions for finding odd cycles added
%March 2011
%mmalekes@sfu.ca
        fid = fopen('AdjMatrix.txt','rt');
        edge_list = fscanf(fid,'%10i',[2,inf]);
         fclose(fid);
         edge_list = edge_list';
        if isempty(edge_list)
            return
        end
        usnet = edge_list2net(edge_list);
% format the edgelist for the loop counting process
        net = sort_net(usnet);
num_nodes = length(net);
num_edges = calc_num_edges(net);


n = get_starting_node(net);
  % give the path a nearly optimal starting node
path = net(n).node;
 % initialize the path
current_edge = net(n).edges(1);
 % initialize the first edge
loop_list = [];
  % initialize the loop list
iterations = 0;
   % initialize the number of algorithm steps
while (length(path)>1 || ~isempty(current_edge))
    [net,path,current_edge,loop_list] = ...
iterate_tree(net,path,current_edge,loop_list);
    iterations = iterations+1;
end
```

```
47 num_loops = length(loop_list);
48 Oddloops(loop_list,Name);
49
50 %-------------------------------------------------------------------------
51 %-------- SUBFUNCTIONS ---------------------------------------------------
52 %-------------------------------------------------------------------------
53 function net = edge_list2net(edge_list)
54 % PURPOSE:  Transform an edge list into a network structure
55 % INPUTS:   edge_list: each row represents an edge connection
56 % OUTPUTS:  net  - network structure containing two fields:
57 % 1.'node' is the ID of the current node
58 % 2.'edges' is a vector that lists all the nodes connected to 'node'
59
60 net = [];
61 if isempty(edge_list)
62     return
63 end
64 edge_list = abs(round(real(edge_list)));
65 ne = size(edge_list);
66 net(1).node = edge_list(1,1); net(1).edges = edge_list(1,2);
67 net(2).node = edge_list(1,2); net(2).edges = edge_list(1,1);
68 for idx = 2:ne(1)
69     node_exists = 0;
70     % if the node is already part of the net, update the list of edges
71     for k = 1:length(net)
72         if (edge_list(idx,1) == net(k).node)
73             % do not update the edge list if the edge already exists
74             if isempty(find([net(k).edges net(k).node] == edge_list(idx,2),1))
75                 net(k).edges = [net(k).edges edge_list(idx,2)];
76             end
77             node_exists = 1;
78             break
79         end
80     end
81     % if the node is new, add it to the end of the net along with the edge
```

```
82      if ~node_exists
83          net(k+1).node = edge_list(idx,1);
84          net(k+1).edges = edge_list(idx,2);
85      end
86      node_exists = 0;
87      % if the node is already part of the net, update the list of edges
88      for k = 1:length(net)
89          if (edge_list(idx,2) == net(k).node)
90              % do not update the edge list if the edge already exists
91              if isempty(find([net(k).edges net(k).node] == edge_list(idx,1),1))
92                  net(k).edges = [net(k).edges edge_list(idx,1)];
93              end
94              node_exists = 1;
95              break
96          end
97      end
98      % if the node is new, add it to the end of the net along with the edge
99      if ~node_exists
100         net(k+1).node = edge_list(idx,2);
101         net(k+1).edges = edge_list(idx,1);
102     end
103 end
104
105 %-------------------------------------------------------------------------
106 function net = sort_net(net)
107 % PURPOSE:  Puts all of the nodes in order from least to greatest
108 % INPUTS:   net  - network structure containing two fields:
109 %1.'node' is the ID of the current node
110 % 2.'edges' is a vector that lists all the nodes connected to 'node'
111 % OUTPUTS:  net  - sorted network structure containing two fields:
112 % 1.'node' is the ID of the current node
113 % 2.'edges' is a vector that lists all the nodes connected to 'node'
114
115 tmp = [];
116 nodes_list = zeros(1, length(net));
```

```
117 for k = 1:length(net)
118     nodes_list(k) = net(k).node;
119 end
120 [sorted, order] = sort(nodes_list);
121 for k = 1:length(net)
122     tmp(k).node = net(order(k)).node;
123     tmp(k).edges = sort(net(order(k)).edges);
124 end
125 net = tmp;
126
127 %--------------------------------------------------------------------------
128 function num_edges = calc_num_edges(net)
129 % PURPOSE:  Calculates the number of edges in an undirected network
130 % INPUTS:   net  - network structure containing two fields: 'node' and 'edges'
131 % 1.'node' is the ID of the current node
132 % 2.'edges' is a vector that lists all the nodes connected to 'node'
133 % OUTPUTS:  num_edges  - number of edges in the network
134
135 num_edges = 0;
136 for k = 1:length(net)
137     num_edges = num_edges + length(net(k).edges);
138 end
139 num_edges = num_edges/2;
140
141 %--------------------------------------------------------------------------
142
143 function n = get_starting_node(net)
144 % PURPOSE:  Pick the (nearly) optimal starting node
145 % USAGE:    >> n = get_starting_node(net);
146 % INPUTS:   net  - network structure containing two fields: 'node' and 'edges'
147 %1.'node' is the ID of the current node
148 % 2.'edges' is a vector that lists all the nodes connected to 'node'
149 % OUTPUTS:  n  - index to the optimal network starting node
150 n = 1;
151 for k = 2:length(net)
```

```
152     if (length(net(k).edges) > length(net(n).edges))
153         n = k;
154     end
155 end
156 
157 %-------------------------------------------------------------------------
158 function [net,path,current_edge,loop_list] =...
159  iterate_tree(net,path,current_edge,loop_list)
160 % PURPOSE:  Execute the current iterative step in the loop counting algorithm
161 % INPUTS:   net  - network structure containing two fields: 'node' and 'edges'
162 % 1.'node' is the ID of the current node
163 % 2.'edges' is a vector that lists all the nodes connected to 'node'
164 % path  - an ordered vector of node values that are connected
165 % current_edge  - the node ID of the current edge
166 % loop_list  - a structure with one field named 'loop'
167 % containing a list of all loops found
168 % OUTPUTS:  net  - same as net input
169 % path  - same as path input,potentially modified
170 % current_edge  - the node ID of the next edge to be considered
171 % loop_list  - same as loop_list input,potentially ammended
172 path_size = length(path);
173 if(path_size <=21)
174 % DONE - finished searching tree
175 if (path_size == 1 && isempty(current_edge))
176     return
177 % CURRENT EDGE LIST FINISHED - go up tree
178 elseif (isempty(current_edge))
179     current_edge = get_next_edge(net,path(path_size-1),path(path_size));
180     path(path_size) = [];
181 % CURRENT EDGE IS THE SAME AS PREVIOUS VERTEX - move to next edge
182 elseif (length(path) > 1 && path(path_size-1) == current_edge)
183     current_edge = get_next_edge(net,path(path_size),current_edge);
184 % LOOP FOUND!
185 elseif (check_path4loop(path,current_edge))
186     loop = loop2std_form(path,current_edge);
```

```
187     if ~compare_loop(loop,loop_list)
188         loop_list = append_loop_list(loop_list,loop);
189     end
190     current_edge = get_next_edge(net,path(path_size),current_edge);
191 % NO LOOP FOUND - keep going down tree
192 else
193     path = [path current_edge];
194     current_edge = get_next_edge(net,path(path_size+1),[]);
195 end
196 end
197
198 %-------------------------------------------------------------------------
199 function loop_list = append_loop_list(loop_list,loop)
200 % PURPOSE:  Adds a loop to the end of a loop_list structure
201 % INPUTS:   loop_list  - a structure with one field named 'loop'
202 % containing a list of all previously found loops
203 % loop  - 1xM vector containing a list of nodes that make a loop
204 % OUTPUTS:  loop_list  - the modified loop_list structure
205
206 if isempty(loop_list)
207     loop_list.loop = loop;
208 else
209     num_loops = length(loop_list);
210     loop_list(num_loops+1).loop = loop;
211 end
212
213 %-------------------------------------------------------------------------
214 function status = check_path4loop(path,current_edge)
215 % PURPOSE:  Check to see if the current edge is in the path
216 % INPUTS:   path  - an ordered vector of node values that are connected
217 % urrent_edge  - a node connected to the last node in path
218 % OUTPUTS:  status  - 1 if a loop has been found,0 otherwise
219
220 status = 0;
221 if find(path == current_edge,1)
```

```
222     status = 1;
223 end
224
225 %-------------------------------------------------------------------------
226 function status = compare_loop(loop,loop_list)
227 % PURPOSE:  Check to see if the loop already exists in the loop_list
228 % INPUTS:   loop  - 1xM vector containing nodes that are connected in a loop
229 % loop_list  - a structure with one field named 'loop'
230 %containing a list of all previously found loops
231 % OUTPUTS:  status  - equals 1 if 'loop' already exists,0 otherwise
232
233 status = 0;
234 if isempty(loop_list)
235     return
236 end
237 for k = 1:length(loop_list)
238     m = length(loop_list(k).loop);
239     n = length(loop);
240     % if the two loops have the same length,check if they are identical
241     if (m == n)
242         status = 1;
243         for kk = 1:n
244             if (loop_list(k).loop(kk) ~= loop(kk))
245                 status = 0;  % loops are different,move on to next
246                 break
247             end
248         end
249         % loops are identical
250         if status
251             return
252         end
253     end
254 end
255
256 %-------------------------------------------------------------------------
```

```
function next_edge = get_next_edge(net,current_node,current_edge)
% PURPOSE:  Find the next edge of the current node in the network structure
% INPUTS:   net  - network structure containing two fields: 'node' and 'edges'
%1.'node' is the ID of the current node
% 2.'edges' is a vector that lists all the nodes connected to 'node'
% current_node  - the ID of the current node in the path
% current_edge  - the node ID of the current edge
% OUTPUTS:  next_edge  - the node ID of the next edge in the edges list
%for the current node
next_edge = [];
for k = 1:length(net)
    if (current_node == net(k).node)
        if isempty(current_edge)    % start with the first edge of the node
            next_edge = net(k).edges(1);
        else    % get the next edge in the list,if there is one
            kk = find(net(k).edges == current_edge);
            if kk < length(net(k).edges)
                next_edge = net(k).edges(kk+1);
            end
        end
        return
    end
end

%-------------------------------------------------------------------------
function loop = loop2std_form(path,current_edge)
% PURPOSE:  Take a loop found in the path and return the loop vector
% INPUTS:   path  - an ordered vector of node values that are connected
% current_edge  - the node ID of the current edge
% OUTPUTS:  loop  - 1xM vector of standard form loop,
%where M is the length of the loop
% NOTES:    Standard form is defined as having the smallest node ID at
%the front of the list,and the smaller of the two neighbors listed second

ii = find(path == current_edge);
```

```
% get the loop from the path
loopy = path(ii:end);
n = length(loopy);
jj = find(loopy == min(loopy));
% order the loop with the smallest value first
loop = loopy([(jj:n) (1:jj-1)]);
% order the rest of the loop with the smaller of the two neighbors second
if loop(2) > loop(n)
    loop = [loop(1) fliplr(loop(2:n))];
end
%-------------------------------------------------------------------------
%PURPOSE: Keep Each odd cycle in its corresponding list
function Oddloops(loop_list,Name)
num_loops = length(loop_list);
leng=ones(1,8); % Each element is a counter for sepecific odd cycle
List3=zeros(1,3);List5=zeros(1,5);List7=zeros(1,7);List9=zeros(1,9);
List11=zeros(1,11);List13=[];List15=[];List17=[];
mkdir('Draft');
for iter=3:2:17

save(strcat('Draft/','List',num2str(iter)),strcat('List',num2str(iter)))
end
for k1=1:num_loops
     h1 = length(loop_list(k1).loop);
  if (mod(h1,2)==1)
        switch num2str(h1)
            case num2str(3)
           List3(leng(1,1),:)=loop_list(k1).loop;
           save('Draft/List3','List3')
           leng(1,1)=leng(1,1)+1;
            case num2str(5)
                List5(leng(1,2),:)=loop_list(k1).loop;
                save('Draft/List5','List5')
                leng(1,2)=leng(1,2)+1;
            case num2str(7)
```

```
327                 List7(leng(1,3),:)=loop_list(k1).loop;
328                 save('Draft/List7','List7')
329                 leng(1,3)=leng(3)+1;
330
331             case num2str(9)
332                 List9(leng(1,4),:)=loop_list(k1).loop;
333                 save('Draft/List9','List9')
334                 leng(1,4)=leng(1,4)+1;
335             case num2str(11)
336                 List11(leng(1,5),:)=loop_list(k1).loop;
337                 save('Draft/List11','List11')
338                 leng(1,5)=leng(1,5)+1;
339             case num2str(13)
340                 List13(leng(1,6),:)=loop_list(k1).loop;
341                 save('Draft/List13','List13')
342                 leng(1,6)=leng(1,6)+1;
343             case num2str(15)
344                 List15(leng(1,7),:)=loop_list(k1).loop;
345                 save('Draft/List15','List15')
346                 leng(1,7)=leng(1,7)+1;
347                  case num2str(17)
348                 List17(leng(1,8),:)=loop_list(k1).loop;
349                save('Draft/List17','List17')
350                 leng(1,8)=leng(1,8)+1;
351            otherwise
352         return
353         end
354   end
355 end
356  ListOfOddCycle(Name);
357 %-------------------------------------------------------------------------
358 %PURPOSE: Finds vertices and edges of each loop in the incimpatibility
359 % Graph and concatenate them horizentally and stores in a matrix
360 function[RESULT]= FindRows(Y)
361 global EdgeSet;
```

```
global B;
n1=size(Y,1);
n2=size(Y,2);
Y(:,n2+1)=Y(:,1);
for m1=1:n1
for m2=1:n2
    endpoint1=Y(m1,m2);
    endpoint2=Y(m1,m2+1);
    E(m1,m2)=str2num(strcat(num2str(B(1,endpoint1)),num2str(B(2,endpoint1))));
      if((B(2,endpoint1)==B(1,endpoint2))&&(B(1,endpoint1)==B(2,endpoint2)))
          F(m1,m2)=0;
       elseif((B(2,endpoint1)==B(1,endpoint2)))
             edgename1=find(all(repmat([B(2,endpoint2);B(1,endpoint1)],1,...
             size(EdgeSet,2))==EdgeSet));
             edgename2=find(all(repmat([B(1,endpoint1);B(2,endpoint2)],1,...
             size(EdgeSet,2))==EdgeSet));
             if(isempty(edgename1))
                edgename1=edgename2;
             end
             F(m1,m2)=edgename1;
      elseif ((B(1,endpoint1)==B(2,endpoint2)))
             edgename1=find(all(repmat([B(2,endpoint1);B(1,endpoint2)],1,...
             size(EdgeSet,2))==EdgeSet));
             edgename2=find(all(repmat([B(1,endpoint2);B(2,endpoint1)],1,...
             size(EdgeSet,2))==EdgeSet));
             if(isempty(edgename1))
                 edgename1=edgename2;
             end
      F(m1,m2)=edgename1;
          end

end
end
RESULT=horzcat(E,F);

```

```
397
398 %-------------------------------------------------------------------------
399 %This function saves all odd cycle of size less than or equal to 17
400 %PURPOSE shows list of odd cycles of the specif size that user
401 %determines first.
402
403 function ListOfOddCycle(Name)
404 global dirname;
405 for i=3:2:17
406     STRING=strcat('List',num2str(i)); %#ok<IJCL>
407     MAT=importdata(strcat('Draft/',STRING,'.mat'));
408     if(any(MAT))
409        n=size(MAT,2);
410        dat=FindRows(MAT);
411        Data=int8([1:n 1:n;dat]);
412        dlmwrite(strcat('Result/',dirname,num2str(date),'/',Name,'/',...
413        STRING,'.txt'), Data, 'newline', 'pc','precision', 2);
414
415   end
416
417 end
418 S=strcat('Result/',dirname,num2str(date),'/',Name);
419 copyfile('AdjMatrix.txt',S)
420 rmdir('Draft','s');
421 %-------------------------------------------------------------------------
422 %%END
```

Bibliography

[1] Zaky Adam, Monique Turmel, Claude Lemieux, and David Sankoff. Common intervals and symmetric difference in a model-free phylogenomics, with an application to streptophyte evolution. In *Comparative Genomics'06*, pages 63–74, 2006.

[2] Farid Alizadeh, Richard M. Karp, Deborah K. Weisser, and Geoffrey Zweig. Physical mapping of chromosomes using unique probes. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA'94, pages 489–500, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[3] Seymour Benzer. On the topology of the genetic fine structure. *Proc. Nat. Acad. Sci. - USA*, 45:1607–1620, 1959.

[4] Anne Bergeron, Mathieu Blanchette, Annie Chateau, and Cedric Chauve. Reconstructing ancestral gene orders using conserved intervals. In *Proc. Fourth Intl Workshop Algorithms in Bioinformatics WABI04*, pages 14–25. Springer, 2004.

[5] Guillaume Blin, Romeo Rizzi, and Stéphane Vialette. A faster algorithm for finding minimum tucker submatrices. In *Proceedings of the Programs, proofs, process and 6th international conference on Computability in Europe*, CiE'10, pages 69–77, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and planarity using PQ-tree algorithms. *Journal of Computational Systems Science*, 13:335–379, 1976.

[7] Binh M. Bui-Xuan, Michel Habib, and Michael Rao. Tree-representation of set families and applications to combinatorial decompositions. *European Journal of Combinatorics*, 2011.

[8] Rainer E. Burkard, Vladimir G. Deineko, and Gerhard J. Woeginger. The travelling salesman and the PQ-tree. In *IPCO'96*, pages 490–504, 1996.

[9] J.P. Char. Master circuit matrix. *Electrical Engineers, Proceedings of the Institution of*, 115(6):762 –770, 1968.

[10] Cedric Chauve, Utz-Uwe Haus, Tamon Stephen, and Vivija P. You. Minimal conflicting sets for the consecutive ones property in ancestral genome reconstruction, 912.

[11] Cedric Chauve, Ján Manuch, and Murray Patterson. On the gapped consecutive-ones property. *Electronic Notes in Discrete Mathematics*, 34(0):121 – 125, 2009.

[12] Cedric Chauve and Eric Tannier. A methodological framework for the reconstruction of contiguous regions of ancestral genomes and its application to mammalian genomes. *PLoS Comput Biol*, 4(11):e1000234, 2008.

[13] M. Chein, M. Habib, and M. C. Maurer. Partitive hypergraphs. *Discrete Mathematics*, 37(1):35 – 50, 1981.

[14] Lin Chen and Yaacov Yesha. Parallel recognition of the consecutive ones property with applications. *Journal of Algorithms*, 12(3):375 – 392, 1991.

[15] Elias Dahlhaus. Parallel algorithms for hierarchical clustering and applications to split decomposition and parity graph recognition. *Journal of Algorithms*, 36:2000, 1998.

[16] Michael Dom, Jiong Guo, and Rolf Niedermeier. Approximation and fixed-parameter algorithms for consecutive ones submatrix problems. *J. Comput. Syst. Sci.*, 76:204–221, 2010.

[17] Michael Dom and Somnath Sikdar. The parameterized complexity of the rectangle stabbing problem and its variants. In *In Proc. 2nd FAW, volume 5059 of LNCS*, pages 288–299. Springer, 2008.

[18] Bernard M. Dwork. *Generalized hypergeometric functions*. Oxford mathematical monographs. Clarendon Press, 1990.

[19] Ralph Faudree, Evelyne Flandrin, and Zdeněk Ryjáček. Claw-free graphs−a survey. *Discrete Math.*, 164:87–147, 1997.

[20] Michele Flammini, Giorgio Gambosi, and Sandro Salomone. Boolean routing. In *Proc. of the 7th Int. Workshop on Distributed Algorithms (WDAG'93),LNCS 725*, pages 219–233, 1993.

[21] Michael L. Fredman and Leonid Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *J. Algorithms*, 21:618–628, 1996.

[22] Delbert R. Fulkerson and Oliver A. Gross. Incidence matrices and interval graphs. *Pac. J. Math.*, 15:835–855, 1965.

[23] Tibor Gallai. Transitiv orientierbare graphen. *Acta Mathematica Hungarica*, 18:25–66, 1967.

[24] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, 1979.

[25] P. C. Gilmore and A. J. Hoffman. A characterization of comparability graphs and of interval graphs. *Canad. J. Math.*, 16:539–548, 1964.

[26] Vladimir Gurvich and Leonid Khachiyan. On generating the irredundant conjunctive and disjunctive normal forms of monotone boolean functions. *Discrete Appl. Math.*, 96-97:363–373, 1999.

[27] Michel Habib, Ross M. McConnell, Christophe Paul, and Laurent Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoretical Computer Science*, 234:59–84, 2000.

[28] MohammadTaghi Hajiaghayi, Yashar Ganjali, and Communicated F. Dehne. A note on the consecutive ones submatrix problem, 2002.

[29] Dov Harel and Robert E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[30] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16:372–378, 1973.

[31] Wen-Lian Hsu. A simple test for the consecutive ones property. In *Proceedings of the Third International Symposium on Algorithms and Computation*, ISAAC '92, pages 459–468, London, UK, 1992. Springer-Verlag.

[32] Wen-Lian Hsu and Ross M. McConnell. Pc trees and circular-ones arrangements. *Theor. Comput. Sci.*, 296:99–116, 2003.

[33] Witold Lipski Jr. Generalizations of the consecutive ones property and related np-complete problems. 1978.

[34] Gad M. Landau, Laxmi Parida, and Oren Weimann. Using PQ trees for comparative genomics. In *CPM'05*, pages 128–143, 2005.

[35] Jian Ma, Louxin Zhang, Bernard B. Suh, Richard C. Raney, Brian J. Burhans, W. James Kent, Mathieu Blanchette, David Haussler, and Webb Miller. Reconstructing contiguous regions of an ancestral genome. *Genome Res*, 16(12):1557–1565, 2006.

[36] Mehrnoush Malekesmaeili, Cedric Chauve, and Tamon Stephen. A tight bound on the length of odd cycles in the incompatibility graph of a non-C1P matrix. 2011.

[37] Ross M. McConnell. Personal Communication.

[38] Ross M. McConnell. *A certifying algorithm for the consecutive-ones property.* SODA '04. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004.

[39] Ross M. McConnell and Fabien de Montgolfier. Algebraic operations on PQ trees and modular decomposition trees. In *WG'05*, pages 421–432, 2005.

[40] Ross M. McConnell and Fabien de Montgolfier. Linear-time modular decomposition of directed graphs. *Discrete Appl. Math.*, 145:198–209, 2005.

[41] Ross M. McConnell and Jeremy P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, pages 536–545, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[42] Ross M. McConnell and Jeremy P. Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189 – 241, 1999.

[43] João Meidanis and Erasmo G. Munuera. A theory for the consecutive ones property. In *Third South American Workshop on String Processing*, volume 4, pages 194–202, 1996.

[44] João Meidanis, Oscar Porto, and Guilherme P. Telles. On the consecutive ones property. *Discrete Applied Mathematics*, 88(1-3):325 – 354, 1998. Computational Molecular Biology DAM - CMB Series.

[45] Rolf H. Möhring. Algorithmic aspects of the substitution decomposition in optimization over relations, set systems and boolean functions. *Annals of Operations Research*, 4:195–225, 1985/6.

[46] Mark B. Novick. Generalized PQ-trees. Technical report, Ithaca, NY, USA, 1989.

[47] Guilherme P. Telles and João Meidanis. Building PQR trees in almost-linear time. 2003.

[48] Alan Tucker. A structure theorem for the consecutive 1's property. *Journal of Combinatorial Theory, Series B*, 12(2):153–162, 1972.

[49] Shih Wei-Kuan and Hsu Wen-Lian. A new planarity test. *Theoretical Computer Science*, 223(1-2):179 – 191, 1999.

[50] Johannes Wienberg and Roscoe Stanyon. Chromosome painting in mammals as an approach to comparative genomics. *Current Opinion in Genetics & Development*, 5(6):792 – 797, 1995.

Index