ABSTRACT

Algorithms to Find the Girth and Cogirth of a Linear Matroid

by

John David Arellano

In this thesis, I present algorithms to find the cogirth and girth, the cardinality of the smallest cocircuit and circuit respectively, of a linear matroid. A set covering problem (SCP) formulation of the problems is presented. The solution to the linear matroid cogirth problem provides the degree of redundancy of the corresponding sensor network, and allows for the evaluation of the quality of the network. Hence, addressing the linear matroid cogirth problem can lead to significantly enhancing the design process of sensor networks. The linear matroid girth problem is related to reconstructing a signal in compressive sensing. I provide an introduction to matroids and their relation to the degree of redundancy problem as well as compressive sensing. I also provide an overview of the methods used to address linear matroid cogirth/girth problems, the SCP, and reconstructing a signal in compressive sensing. Computational results are provided to validate a branch-and-cut algorithm that addresses the SCP formulation as well as an algorithm which uses branch decompositions and dynamic programming to find the girth of a linear matroid.
Acknowledgments

I would like to thank my parents and family for the love and support throughout my graduate career. I would like to thank my advisor, Dr. Illya Hicks, and the rest of my committee for their guidance and support. I would also like to thank Dr. Maria Cristina Villalobos for her guidance and support during my undergraduate career. A thanks also goes to Nabor Reyna, Dr. Russell Carden and the rest of the AGEP family for their support.
## Contents

Abstract ii
Acknowledgments iii
List of Illustrations vi

1 Introduction 1
   1.1 Problem Formulation .............................................. 2
   1.1.1 Degree of Redundancy and Linear Matroids ............... 3
   1.1.2 Compressive Sensing ........................................ 13

2 Literature Review 16
   2.1 The Set Covering Problem ....................................... 16
   2.2 The Linear Matroid Cogirth Problem ......................... 23
   2.3 The Hitting Set Problem ...................................... 29
   2.4 Compressive Sensing .......................................... 31

3 Linear Matroid Cogirth Algorithms 36
   3.1 An Algorithm for the Set Covering Problem .................. 36
   3.1.1 Generating Cutting Planes ................................. 41
   3.1.2 Branching .................................................. 44
   3.1.3 Finding Feasible Solutions ................................ 45
   3.2 Rearranging Matrices into Bordered Block Diagonal Form .... 46
   3.3 Cogirth Heuristics ............................................ 47
   3.3.1 Advantages of the Bordered Block Diagonal Form ....... 48
   3.4 Hitting Set Algorithm ....................................... 50
4 Linear Matroid Girth Algorithms
   4.1 From Cogirth to Girth ........................................... 61
   4.2 Linear Matroid Girth Algorithm ................................. 63
   4.3 Linear Matroid Circuit Algorithm Variation .................... 80
   4.4 Genetic Algorithm .................................................. 85
      4.4.1 Generating an Initial Population ........................... 86
      4.4.2 Producing Children ............................................ 88
      4.4.3 Mutation ....................................................... 88
      4.4.4 Children Feasibility .......................................... 90
      4.4.5 Computational Results ....................................... 91

5 Conclusions and Future Work ........................................... 93

Bibliography ........................................................................ 99
Illustrations

1.1 Sensor network example 1 ........................................... 5
1.2 Sensor network example 2 ........................................... 5
1.3 Matroid Example ......................................................... 7
1.4 Hypergraph example for a matroid ............................... 12

2.1 Bordered Block Diagonal Form ..................................... 25

3.1 Flow chart of basic Branch-and-Cut Algorithm .................. 39
Chapter 1

Introduction

Combinatorial and discrete optimization problems arise in a wide variety of areas that include operations research, computer science, and various engineering disciplines. Among the most famous problems are those described by Karp [46]. These problems include but are not limited to set cover, vertex cover, hitting sets, knapsack, cliques, partitioning, coloring, and Hamiltonian cycles. In particular, this work will concentrate on the girth and cogirth of linear matroids, and various models and algorithms to find them. In this thesis, I present an integer programming based branch-and-cut algorithm in which the implicitly defined constraints must be found and inserted into the model. I adapt a generic algorithm for hitting sets proposed in [22] which in itself falls under the scope of cutting plane algorithms. I also present other heuristics based on other models and concepts during the course of this work. Furthermore, I present computational results as validation for the proposed algorithm and heuristics. I also propose an algorithm which uses the branch-decomposition of a matrix and dynamic programming and examine the performance of a genetic algorithm to find the girth of a linear matroid, and discuss the investigation of theoretical results concerning linear matroids, and the corresponding matrices. It will also be seen, that any algorithm to find the girth of a linear matroid can be used to find the
cogirth of a linear matroid and vice versa. Much of what is contained within the first three chapters can be found in [3].

1.1 Problem Formulation

It is not uncommon to construct linear and integer programs for graph theory problems. This section gives a brief discussion of the set covering formulation. Let $N = \{1, ..., n\}$ and $K = \{1, ..., k\}$ for $n, k \in \mathbb{Z}^+$. Let $N_1, N_2, ..., N_k$ be a given collection of subsets of $N$. Each $N_i$ is given a weight $c_i$. $D$ which is a subset of $K$ is called a cover of $N$ if $\bigcup_{i \in D} N_i = N$. The weight of a cover $D$ is $\sum_{i \in D} c_i$. The set covering problem (SCP) is $\min \{c^T x \mid Ax \geq 1, \ x \text{ binary}\}$ where $A$ is an incidence matrix of $\{N_i \mid i \in K\}$. The entries of $A$, $a_{ij}$, are 1 if $j \in N_i$ and 0 otherwise for all $j \in N$. The rows of the $A$ are characteristic vectors of the collection of subsets $N_1, N_2, ..., N_k$. In many cases, the subsets are explicitly known, but there can be some instances where constraints are not explicit. This type of problem will be referred to as an implicit set covering problem (ISCP). Further introduction of the SCP can be found in [13], [59] and [75]. Karp [46] showed that solving a hitting set problem is equivalent to solving a set covering problem. For the hitting set problem, a family $\{U_i\}$ of subset of $\{s_j : j = 1, ..., r\}$ is given. The objective is to find a hitting set $W$ such that $W \cap U_i \neq \emptyset$ for each $i$. The rest of this chapter will focus on introducing the girth and cogirth of linear matroids, sensor network reliability, compressive sensing signal reconstruction and how they relate to linear matroids.
1.1.1 Degree of Redundancy and Linear Matroids

Before I show the relation between linear matroids and set covering, I will discuss an application in which linear matroids can be used. Sensor networks play an important role in industry such as monitoring chemical plants [71]. Therefore, the ability to design a reliable sensor network is important. When designing a sensor network, the degree of redundancy of the network is a way to measure the reliability of the network. Measurements acquired from the network are redundant if they cannot only be obtained directly, but can also be obtained from another set of measurements. It is common practice to represent the relationship between sensor measurements $y$ and the system states $u$ via a linear model:

$$y = Hu + \varepsilon,$$

(1.1)

where $y$ is an $n \times 1$ vector, $u$ is a $p \times 1$ vector, $H$ is an $n \times p$ ($n > p$) matrix with rank $p$, and $\varepsilon$ is an $n \times 1$ vector representing noise accumulated by linearizing the system [48, 67]. The rank of the matrix refers to the dimension of the column space, the set of all possible linear combinations of its column vectors. In the remainder of this thesis including the computational results, $\varepsilon$ is considered to be the zero vector (the system is noise free). The degree of redundancy (DoR) of the network refers to the number of sensors of the network that can fail while maintaining the ability to obtain all the measurements [51]. Therefore, a sensor network with a high DoR would
be more reliable than a sensor network with a low DoR. By obtaining the DoR, it is possible to evaluate which sensor networks are more reliable than others.

Consider the example networks in Figures 1.1 and 1.2. In Figure 1.1, \( S_1, \ldots, S_5 \) represent sensors and \( M_1, \ldots, M_5 \) represent streams of information or measurements in the network measured by sensors to the right of them. Here, we assume that a measurement can be obtained by a sensor, or if a measurement to the right of it is known. In other words, if \( M_5 \) is known, then \( M_4 \) can be obtained. The same goes for \( M_1, M_2 \) and \( M_3 \). If all the measurements are known, then this network has a degree of redundancy of 4 since each measurement can be obtained from the measurement directly to the right of it. In Figure 1.2, we now have six sensors, and six streams of information. In this example, a measurement can only be obtained from other measurements if all the measurements directly to the right of it are known. For instance, both \( M_4 \) and \( M_5 \) are needed to obtain \( M_2 \) without using sensor \( S_2 \). If we assume that all the measurements are known except for \( M_4 \), then the network has a DoR of 2 since \( M_3 \) can be obtained from \( M_6 \), and \( M_1 \) can be obtained from \( M_2 \) and \( M_3 \). If all the measurements are known except for \( M_6 \), then we still have a DoR of 2 since \( M_2 \) can be obtained from \( M_4 \) and \( M_5 \) and \( M_1 \) is obtained as before. However, if \( M_4 \) and \( M_6 \) are not known, then the DoR is 1 since only \( M_1 \) can be obtained from other known measurements. Different sensor network configurations may provide different matrices in the linear system described above. That is, the matrices may or may not have structure. In particular a bordered block diagonal form (BBDF)
Figure 1.1: $S_1,\ldots,S_5$ are sensors in a sensor network. $M_1,\ldots,M_5$ represent streams of information in the network. This example is adapted from an example given by Bagajewicz and Sanchez [5].

Figure 1.2: $S_1,\ldots,S_6$ are sensors in a sensor network. $M_1,\ldots,M_6$ represent streams of information in the network. This example is adapted from an example given by Bagajewicz and Sanchez [5].

which is discussed in Chapter 2. However, the cases in which this structure does arise are still of great importance. It will be seen in Chapter 3 that this structure can be exploited to enhance computational results. In order to find the DoR of a network, we can describe a sensor failure using the matrix $H$.

Removing the $k$th row from $H$ simulates a failure of the $k$th sensor in the network. Considering this, the degree of redundancy is defined as:

$$d^* = \min\{d - 1 : \text{there exists } H_{(-d)} \text{ such that } r(H_{(-d)}) < p\},$$

where $H_{(-d)}$ represents a submatrix of $H$ obtained by deleting $d$ of the rows of $H$, $r(H_{(-d)})$ is the rank of the resulting matrix and $p$ is the rank of $H$. It is assumed that
$H$ is a matrix with full column rank. If $H$ were rank deficient, the only difference would be the initial upper bound for the optimal solution since the upper bound on the girth and cogirth is based on the rank of $H$. It will be seen that the girth of a linear matroid $M[Z]$ is bounded above by $\text{rank}(Z) + 1$, and the cogirth is bounded above by $n - \text{rank}(Z) + 1$, where $Z$ is the $m \times n$ matrix corresponding to $M$. The DoR of the network can be found by finding the cogirth of the matroid obtained from the matrix $H$. The relationship between the DoR of a sensor network and the matroid cogirth problem will be discussed in the following section. In Chapter 2, I will discuss existing methods that attempt to solve the cogirth problem accurately and efficiently.

Before I proceed any further, there is some terminology that needs to be introduced. The following definitions and properties are common in the literature. For a more comprehensive review of matroids, the reader is referred to Oxley [61]. Matroids were first introduced by Whitney [74] as a generalization of linear independence and linear dependence. Therefore, many of the terms and concepts may appear familiar.

A matroid, $M$, consists of an ordered pair $(S, \mathcal{I})$. $S$ is a finite set, and $\mathcal{I}$ is a collection of subsets of $S$ satisfying the following three properties.

(I1) $\emptyset \in \mathcal{I}$.

(I2) If $I \in \mathcal{I}$ and $J \subseteq I$, then $J \in \mathcal{I}$.

(I3) If $I_1, I_2 \in \mathcal{I}$ and $|I_1| < |I_2|$, then $\exists e \in I_2 - I_1$ s.t. $I_1 \cup e \in \mathcal{I}$.

$S$ and the elements of $\mathcal{I}$ are referred to as the ground set and independent sets of $M$. 
Consider the real-valued matrix given in Figure 1.3. In this example, the ground set, \( \{1, 2, 3, 4, 5\} \), corresponds to the rows of \( Z \), and the independent sets correspond to sets of linearly independent rows of the matrix.

The matroid obtained from \( Z \) is called a linear matroid and is denoted by \( M[Z] \). Note that a matroid can also be obtained by letting the ground set correspond to columns of a matrix and the independent sets correspond to sets of linearly independent columns. However, the matroid obtained by using the columns of the matrix is different than the matroid obtained by using the rows of the same matrix. For \( M[Z] \), we have the following:

\[
S = \{1, 2, 3, 4, 5\}
\]

\[
\mathcal{I} = \{\emptyset, \{1\}, \{2\}, \{4\}, \{5\}, \{1, 2\}, \{1, 5\}, \{2, 4\}, \{2, 5\}, \{4, 5\}\}
\]

\[
S - \mathcal{I} = \{\{3\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{3, 4\}, \{3, 5\}\} \cup \{X \subseteq S : |X| \geq 3\}
\]

A maximally independent set of a matroid \( M \) is a set \( J \in \mathcal{I} \) such that \( J \cup x \) is dependent for all \( x \in S - J \), and is referred to as a basis \( B \) of \( M \). The collection of all bases of \( M \) is denoted as \( \mathcal{B}(M) \). Consider rows 1 and 2 from \( Z \). If you

\[
Z = \begin{pmatrix}
1 & 1 & 0 \\
2 & 0 & 1 \\
3 & 0 & 0 \\
4 & 1 & 0 \\
5 & 1 & 1
\end{pmatrix}
\]
include any other row of $Z$ with rows 1 and 2, then the set of rows becomes linearly
dependent. Therefore, $\{1, 2\}$ is a basis of $M[Z]$. The collection of bases of $M[Z]$ is
$\{\{1, 2\}, \{1, 5\}, \{2, 4\}, \{2, 5\}, \{4, 5\}\}$ and satisfy the following two properties. Observe
that all the bases have the same cardinality.

(B1) $\mathcal{B}(M) \neq \emptyset$.

(B2) If $B_1, B_2 \in \mathcal{B}(M)$ and $x \in B_1 - B_2$, then $\exists y \in B_2 - B_1$ s.t. $(B_1 - x) \cup y \in \mathcal{B}(M)$.

A \textit{minimally dependent set} of a matroid $M$ is called a \textit{circuit} $C$ of $M$. A circuit
$C$ is a subset of $S$ such that $C - x \notin \mathcal{I}$ for all $x \in C$. The collection of all circuits
of $M$ is denoted as $\mathcal{C}(M)$. The collection of circuits of $M$ satisfy the following three
properties.

(C1) $\emptyset \notin \mathcal{C}(M)$.

(C2) If $C_1, C_2 \in \mathcal{C}(M)$, and $C_1 \subseteq C_2$, then $C_1 = C_2$.

(C3) If $C_1, C_2$ are distinct members of $\mathcal{C}(M)$ and $e \in C_1 \cap C_2$, then $\exists C_3 \in \mathcal{C}(M)$ s.t.
$C_3 \subseteq (C_1 \cup C_2) - e$.

The cardinality of the smallest circuit is called the \textit{girth} of the matroid. Referring
back to the example, the collection of circuits of $M[Z]$ is $\{\{3\}, \{1, 4\}, \{1, 2, 5\}, \{2, 4, 5\}\}$
and the girth is 1. Given a matroid $M = (S, \mathcal{I})$, we consider another matroid
whose ground set is also $S$. Given $S$, define the following collection of subsets
of $S$, \(\{S - B : B \in \mathcal{B}(M)\}\). This set, which will be denoted by \(\mathcal{B}^*(M)\), is the set of bases of another matroid. This matroid denoted by \(M^*\) is called the dual matroid of \(M\). Thus, \(\mathcal{B}(M^*) = \mathcal{B}^*(M)\) and \((M^*)^* = M\). If \(B \in \mathcal{B}^*(M)\), it is called a \textit{cobasis} of \(M\). Referring back to the example matrix \(Z\), recall that \(\mathcal{B}(M[Z]) = \{\{1,2\}, \{1,5\}, \{2,4\}, \{2,5\}, \{4,5\}\}\). Therefore, \(\mathcal{B}^*(M[Z]) = \{\{3,4,5\}, \{2,3,4\}, \{1,3,5\}, \{1,3,4\}, \{1,2,3\}\}\).

Similarly, \(\mathcal{C}(M^*) = \mathcal{C}^*(M)\), and if \(C \in \mathcal{C}^*(M)\), it is called a \textit{cocircuit} of \(M\). The cardinality of the smallest cocircuit is called the cogirth of the matroid. Referring back to the example, \(\mathcal{C}^*(M[Z]) = \{\{1,2,4\}, \{1,2,5\}, \{2,4,5\}, \{1,4,5\}\}\) and the cogirth is 3. It should be noted that the bases and circuits of \(M^*\) are the cobases and cocircuits of \(M\) respectively and vice versa.

Recall the degree of redundancy (DoR) of a sensor network provides a measure for the reliability of a given network. The ability to design reliable networks that are cost efficient is important. Therefore, it is important to find the DoR of a sensor network. Below, I describe the relationship of the degree of redundancy (DoR) of a sensor network and the cogirth of a matroid.

Although the mathematical definition of the DoR of a sensor network provides a simple algorithm to find the DoR, the algorithm itself is by no means practical. Therefore, it is necessary to take a different approach towards the problem. Recall the linear system, \(y = Hu + \varepsilon\), which is a standard description of the sensor network where \(H\) is an \(n \times p\) matrix, \(n > p\). Define \(R = \{1, \ldots, n\}\) to be the set of row indices,
SR to be the subset of rows of $H$ removed, and $d$ to be the cardinality of $SR$. The cardinality of the smallest subset, $\hat{SR}$, of rows such that the rank of the resulting matrix is one less than the rank of $H$ is what needs to be computed. The DoR is $\hat{d} - 1$, where $\hat{d} = |\hat{SR}|$, the cardinality of $\hat{SR}$. Note that one could consider $H^T$ and remove sets of columns. Define $RowS = \{r_1, r_2, \ldots, r_l\}$ where $l$ is the number of distinct bases of the row space of $H$ and $r_j$ is a distinct basis for $j \in \{1, \ldots, l\}$. Therefore, $|\hat{SR}| = \min\{|J| : J \subseteq R, J \cap r_j \neq \emptyset \forall r_j \in RowS\}$. In other words, the rows of the resulting matrix $H_{(-d)}$ do not span the row space of $H$ because the removed rows intersect every basis of the row space of $H$. Therefore, the rank of $H_{(-d)}$ must be less than the rank of $H$. Recall the example matrix from Figure 1.3. The row space of $Z$ is $\mathbb{R}^2$. By removing rows $\{1, 2, 4\}$, which is a cocircuit of $M[Z]$, the resulting matrix does not span $\mathbb{R}^2$. Switching back to the matrix $H$, we can obtain a linear matroid $M[H]$ from $H$, where the distinct bases of $M[H]$ correspond to distinct bases of the row space of $H$. Based on the observation that $\{1, 2, 4\}$ is a cocircuit of $M[Z]$, one might think that removing rows from $H$ that correspond to a cocircuit will reduce the rank of the resulting matrix. Upon this belief, in order to find the DoR, one can consider finding the cogirth of the linear matroid $M[H]$. The following paragraphs build upon this observation, and describe why the cogirth of a linear matroid provides the DoR of the corresponding sensor network.

Recall that for a matroid $M = (S, \mathcal{I})$, the cobases are just the complement of the bases with respect to $S$. Since a basis, $B$, is maximally independent, $B \cup x$ contains
a circuit which must contain $x$ for all $x \in S - B$. Also, since $x \in S - B$, $x$ is a
member of a cobasis $B \in B(M)$. This might lead one to wonder if a circuit has
a nonempty intersection with every basis in $B^*(M)$. Oxley [61] shows that this is
in fact the case. A hypergraph, $G_{hyp} = (V, E)$, is a generalization of a graph where
members of $E$ can connect two or more members of $V$. Define $G_{hyp,H} = (V, E)$, to
be a hypergraph where $V$ corresponds to the set of row indices of $H$ and a hyperedge
$f \in E$ corresponds to a basis of $M[H]$. A transversal of $G_{hyp}$ is a subset $U$ of $V$
such that every $f \in E$ is adjacent to at least one member of $U$. Observe that since $E$
and $U$ correspond to the set of bases and a subset of row indices respectively and $U$
touches every hyperedge, $U$ corresponds to a cocircuit of $M[H]$. Therefore, if we want
to find the cogirth of $M[H]$, we need to find the smallest transversal of the $G_{hyp,H}$.
Similarly, if we wanted to find the girth of $M[H]$, we would consider a hypergraph
with members of $E$ corresponding to cobases of $M[H]$.

To provide a more illustrative example, refer back to the matrix $Z$. We can
find the girth of $M[Z]$ by considering the above mentioned hypergraph with mem-
bers of $E$ corresponding to cobases of $M[Z]$. Recall the collection of cobases of
$M[Z]$, $B^*(M[Z]) = \{\{3, 4, 5\}, \{2, 3, 4\}, \{1, 3, 5\}, \{1, 3, 4\}, \{1, 2, 3\}\}$. The correspond-
ing hypergraph can be seen in Figure 1.4. Notice that every member of $C(M) = \{\{3\}, \{1, 4\}, \{1, 2, 5\}, \{2, 4, 5\}\}$ is a transversal of the hypergraph and the smallest transversal is $\{3\}$. Therefore, the girth of the $M[Z]$ is 1. In order to find the mini-
mum transversal, we can turn to discrete optimization and model the problem as an
integer program. A description of the set covering problem (SCP), and how it relates to the matroid cogirth problem is provided in Section 1.1.

Figure 1.4: This figure shows the hypergraph $G_{hyp} = (V, E)$ where $V$ corresponds to row indices of the matrix $Z$ and members of $E$ correspond to cobases of $M[Z]$.

Recall the cogirth problem can be solved by addressing a minimum transversal problem. The problem of finding the minimum transversal can be seen as a SCP. In order to find the cogirth of $M[H]$, let $N = \{1, ..., n\}$, where $n$ is the number of rows of $H$ and let $K = \{1, ..., k\}$ where $k$ is the number of distinct bases of $M[H]$. Let $N_i$ correspond to a distinct basis of $M[H]$ with $c_i = 1$ for each $i \in K$. For the incidence matrix $A$, $a_{ij}$ will be 1 if row $j$ is a member of basis $N_i$ and 0 otherwise. Therefore, the problem now becomes $\min \{1^T x \mid Ax \geq 1, \ x \text{ binary}\}$. The constraints demand that any feasible solution intersect every basis of $M[H]$, which is exactly what is required.

In order to find the girth, let $k$ be the number of distinct cobases of $M[H]$, and let $N_i$ correspond to a distinct cobasis of $M[H]$. Therefore, a solution to the SCP provides a solution to the cogirth problem for linear matroids, which in turn provides the DoR of the corresponding sensor network. However, since one does not normally have
access to all the bases of a linear matroid, they must be found. Therefore, we arrive at a set covering problem with implicitly defined constraints. Since both problems are NP-hard, note that in general, it is possible to take a set covering problem, and model it as a cogirth problem for a linear matroid so that a solution to the cogirth problem provides a solution to the set covering problem. However, describing a clear path to do so is not discussed here. To view the linear matroid cogirth problem as a hitting set, simply consider the bases of the matroid as the family of subsets \( \{ U_i \} \), and a hitting set \( W \) as a cocircuit of the matroid. Further discussion of the SCP, and the hitting set problem as well as methods devised to solve these problems will be given in Chapter 2.

1.1.2 Compressive Sensing

The two main problems of compressive sensing [15], [9], [21] are designing a suitable measurement matrix and reconstructing a signal. Given an \( m \times 1 \) measurement vector \( y \) that represents an encoded signal, one wishes to recover the original signal represented by an \( n \times 1 \) vector using a \( M \times N \) measurement matrix \( \Phi \), i.e., solve \( y = \Phi x \). Of particular interest is when \( x \) has a sparse representation. In other words, when there exists a basis represented by a matrix \( \Psi \) and a sparse vector \( s \) with \( K \) nonzero elements such that \( x = \Psi s \). In that case, \( x \) is called \( K \)-sparse. The problem then becomes \( y = \Theta s \), with \( \Theta = \Phi \Psi \). Therefore, the two main problems are:

1. Design a suitable matrix \( \Phi \).
2. Solve $y = \Theta s$.

I will focus on how the latter of the two relates to the girth of a linear matroid, and subsequently on how to find the girth. In solving the second problem, one wishes to find a vector $s$ with the smallest number of non-zeros. This amounts to solving

$$\min ||s||_0 \text{ s.t. } y = \Theta s,$$

where $|| \cdot ||_0$ refers to the number of non-zeros of the vector. In other words, find the smallest number of columns of $\Theta$ needed such that $y = \Theta s$. If we consider the matrix $[\Theta, y]$, then solving 2.3 is equivalent to finding the smallest set of linearly dependent columns of $[\Theta, y]$. Recall that given a matrix, and its resulting linear matroid, dependent sets of a matroid are known as circuits of the matroid, and the cardinality of the smallest circuit is called the girth. Therefore, solving 2.3 is equivalent to finding girth of the matroid $M[\Theta, y]$ which can be modeled as a set covering problem with implicit constraints. The difference is that the characteristic vectors of the constraints will represent cobases of $M[\Theta, y]$ instead of bases.

The remainder of this thesis focuses on developing algorithms to find the girth and cogirth of a linear matroid. A review of the necessary background material for the discussed problems is presented in Chapter 2. In Chapter 3, I present algorithms to find the cogirth of a linear matroid and present computational results. In Chapter 4, I present algorithms to find the girth of a linear matroid. In Chapter 5, I offer
some concluding remarks and future work.
Chapter 2

Literature Review

This Chapter will focus on discussing the set covering problem, the linear matroid cogirth problem, the hitting set problem, and the compressive sensing signal reconstruction problem. Despite the vast contributions to the growing fields, it is my goal to discuss the relevant background.

2.1 The Set Covering Problem

The set covering problem has applications to problems such as facility location, graph coloring, the linear matroid cogirth problem, and compressive sensing. In particular, the linear matroid cogirth problem, and compressive sensing problems involve implicitly defined constraints. The goal of this and the following sections is to provide the necessary background for the set covering problem, the linear matroid cogirth problem, compressive sensing, and the hitting set problem, which is equivalent to the set covering problem.

Recall the set covering problem. Let \( N = \{1, \ldots, n\} \) and \( K = \{1, \ldots, k\} \) for \( n, k \in \mathbb{Z}^+ \). Let \( N_1, N_2, \ldots, N_k \) be a given collection of subsets of \( N \). Each \( N_i \) is given a weight \( c_i \). \( D \) which is a subset of \( K \) is called a cover of \( N \) if \( \bigcup_{i \in D} N_i = N \). The weight of a cover \( D \) is \( \sum_{i \in D} c_i \). The set covering problem (SCP) is \( \min \{ c^T x \mid Ax \geq \} \).
1, \( x \) binary} where \( A \) is an incidence matrix of \( \{N_i \mid i \in K\} \). The entries of \( A, a_{ij}, \) are 1 if \( j \in N_i \) and 0 otherwise for all \( j \in N \). The decision version of the SCP was proved to be NP-complete [46]. Posed as an optimization problem, the SCP is NP-hard. The SCP is a well-studied problem. There are a variety of known methods that have been developed to address the set covering problem. These methods include greedy heuristics, genetic algorithms, and other methods that implement cutting plane methods and Lagrangian heuristics.

Greedy heuristics are discussed by Chvátal [28] and Johnson [45]. Greedy heuristics work by trying to gain as much as possible at each given step of the procedure. Given the system \( Ax \geq 1 \), with \( A \) an \( m \times n \) matrix, the greedy algorithm would consist of the following steps.

Algorithm 2.1 Greedy Algorithm

1: Set \( Cov = \emptyset, Uncov = \{1, \ldots, m\}, Sol = \emptyset \)
2: \textbf{while} \( Uncov \neq \emptyset \) \textbf{do}
3: \hspace{1em} Choose \( j \)th column of \( A \) and \( Sol = Sol \cup j \) based on weights given to the columns of \( A \)
4: \hspace{1em} \textbf{for} \( i = 1 \rightarrow m \) \textbf{do}
5: \hspace{2em} \textbf{if} \( a_{ij} = 1 \) \textbf{then}
6: \hspace{3em} \( Cov = Cov \cup i, Uncov = Uncov - i \)
7: \hspace{2em} \textbf{end if}
8: \hspace{1em} \textbf{end for}
9: \hspace{1em} \textbf{end while}
10: \( Sol \) is a solution to the SCP

The weights given to the columns of \( A \) can be obtained in many ways. For instance, let \( CR_j \) be the subset of rows of \( A \) such that \( a_{ij} \) is 1 for the \( j \)th column of \( A \). Then we can let the weight of the \( j \)th column, \( w_j \), be \( |CR_j|/c_j \), where \( c_j \) is the cost of the \( j \)th
column of $A$. In the SCP discussed in Chapter 2, $w_j = |CR_j|$ for all columns of $A$ since $c = 1$. There have also been attempts to improve upon this initial greedy algorithm. Marchiori and Steenbeek [53] offer what they call an iterated approximation algorithm (ITEG). Their algorithm basically starts by finding a cover using the procedure listed above with the weight of the $j$th column $w_j = |CR_j|$. A function they call **Enhanced Greedy** tries to improve the cover by adding or removing columns from the original cover using criteria which is discussed in more detail in [53]. The following procedure is performed for a specified number of iterations. Given a best solution $S_{best}$, a subset of $S_{best}$ is chosen, and **Enhanced Greedy** is used to extend the partial cover into a cover $S$. If $S$ is better than $S_{best}$, then $S_{best} \leftarrow S$. The algorithm performs well on the reported instances. Another improvement on the greedy algorithm was introduced by Musliu [56]. As with ITEG, an initial cover is produced using the above procedure. A neighborhood of the current solution, $S$, is created. A neighbor of this initial cover is a cover that can be found by adding and/or removing columns of $A$ from the initial cover. A column index $j$ is put into and saved in a “tabu list” for a specified number of iterations if the $j$th column of $A$ has been added or removed from a cover in one of the recent iterations. An upper bound on the size of the neighborhood is also specified to limit the search time. Once a neighborhood is found, the best neighbor is chosen using a fitness function. For a cover $D$, define $Fitness_D = \text{number of uncovered elements} + |D|$. The fitness function is among criteria that is used to choose a new cover from the neighbors. The tabu list and
upper bound are updated, and the procedure is repeated until stopping criteria is met. According to the computational results reported, the heuristic is competitive with ITEG.

A genetic algorithm developed by Beasley and Chu [11], uses the idea of natural selection. The general idea of the genetic algorithm is to produce a population of solutions until a sufficient solution is found. Begin by creating an initial population of solutions, and evaluate the fitness of the solutions. For a solution $D$, $Fitness_D = |D|$ since $D$ is a binary vector. Solutions, referred to as parents, are chosen and mated with each other. The children are evaluated, some or all of the population is replaced by the children. The process of generating children and replacing solutions is repeated as needed. The actual procedure involves a little more work. What follows is a basic description of the algorithm. Once an initial population is generated and $t$ is initialized to 0, two solutions $P_1$ and $P_2$ and mated and a child $C$ is produced. $C$ is “mutated”, a random number of chosen elements of $C$ are switched from 0 to 1 and vice versa, to randomize the search of solutions. If $C$ is not a solution, a heuristic is used to extend it to a solution. If $C$ is identical to a solution in the population, the process of producing a child is repeated, else $t = t + 1$ and a randomly chosen fit solution is replaced by $C$. This process is repeated until $t = M$ for some specified number $M$. The fittest solution is chosen as the optimal solution.

Algorithms and heuristics that focus on the polyhedral structure of the SCP have also been developed. In fact, Beasley [10, 12] has conducted further research into
solving the SCP using subgradient optimization techniques, problem reduction, and tree search procedures. Balas and Ho [7] also discuss a class of algorithms which incorporate heuristics, cutting planes generated from conditional bounds and subgradient optimization. The heuristics are used to find feasible solutions to the SCP and feasible solutions to the dual program of the linear relaxation of the SCP. The generation of conditional bounds from solutions to the dual program of the linear relaxation of the SCP is discussed by Balas [6]. Balas and Ho ran their algorithms on several randomly generated test instances. According to the reported instances, the algorithm works efficiently in most cases, and provides good approximations when a solution cannot be found. Further computational experiments are presented by Grossman and Wool [41]. They discuss several approximation algorithms including several versions of greedy algorithms. They provide a basic description of each algorithm and provide results for random generated test instances as well as instances that arise from combinatorial questions. It was reported that the neural network algorithm performed the best overall, and that the greedy algorithms performed competitively in many instances. The best algorithm to use would depend on the test instance.

Another approach has been to use swarm intelligence methods such as ant colony optimization [65], [66]. The idea is to use several ”ants” or agents to find the best cover. Initially, an ant will choose columns of the incidence matrix $A$ based on some given probability. This is done until a cover has been found. Once a cover has been found, the ant will leave a ”pheromone trail” on the columns chosen to increase
the likelihood that they are chosen again since they a part of a solution. Once the pheromones for the columns have been updated, another ant is sent out to look for a cover. This is process is repeated for a specified number of ants. Although this provides another alternative to solve the set covering problem, like the other methods discussed, this one requires that all the constraints be known. For the ISCP, this type of approach could only be used to solve explicit set covering problems where the constraints only represent some subset of the entire implicit constraint set.

Ashley et al. [4] introduce an implicit set covering problem that arises in biology. They provide some results on the complexity of the set cover instance as well as some results on the when it is possible to provide an approximation to the problem. They also present a greedy algorithm which is similar to the algorithm presented in this proposal in the sense that they both find uncovered sets. However, the greedy algorithm discussed by Ashley et al. stops once all the sets have been covered. The algorithm discussed in this proposal uses this idea of adding uncovered sets as a subroutine, not as the complete algorithm. However, this type of greedy algorithm could be used for the linear matroid cogirth, and compressive sensing problems discussed in Chapter 1.

The methods discussed above are representative of the variety of different approaches that can be taken when faced with solving a SCP. Although many of these traditional methods work well and are viable options to solve many set covering problems, it is not possible to use them directly to solve the SCP that arises from the
matroid cogirth problem with the exception of the greedy algorithm discussed in [4]. This stems from the fact that in order to use these methods directly, the entire original system of constraints, $Ax \geq 1$, must be known. In fact, at the initialization of the problem of interest, no constraints are known. In the cases of Balas and Ho [7], and Beasley [12], primal and dual heuristics cannot be incorporated into an algorithm. Similarly, finding cutting planes from conditional bounds will not be possible without the dual formulation to the linear relaxation of the SCP. The biggest challenge involving the greedy heuristics and genetic algorithms is the ability to find a good initial solution. Although it might be possible to use genetic algorithms to generate children from an initial population of solutions, generating a good initial population would probably be difficult without any weights to decide which columns of $A$ are better in the greedy sense. In fact, a modification of a genetic algorithm to find the girth of a linear matroid is presented in Chapter 4. The same can be said for generating a good initial solution for the greedy heuristics discussed at the beginning of this chapter. However, it may be possible to incorporate the underlying concepts into a branch-and-cut algorithm. The greedy heuristics may provide some insight into obtaining feasible solutions to the SCP while only having knowledge of a subset of $Ax \geq 1$. It may also be possible to generate other cutting planes without knowing $Ax \geq 1$ in its entirety. The developed branch-and-cut algorithm and computational results will be presented in Chapter 3.
2.2 The Linear Matroid Cogirth Problem

Finding the cogirth of a general linear matroid is equivalent to finding the minimum transversal of a hypergraph, which can be solved as a set covering problem. As an optimization problem, the set covering problem is NP-hard [46]. Therefore a polynomial time algorithm to find the cogirth of a general linear matroid does not likely exist. However, there have been several attempts to develop an algorithm or heuristic that solves the matroid cogirth problem accurately and/or efficiently. The first algorithm, which is fairly obvious and simple in nature, comes from the definition of the DoR. Recall that for a sensor network with model matrix $H$, the DoR is defined as the cardinality of the smallest set of rows whose removal from $H$ reduces the rank of the resulting matrix $H_{(-d)}$ where $d$ is the number of sets removed. Exhaustive rank testing is a brute force technique. As the definition of the DoR suggests, the idea is to iteratively remove sets of rows from $H$ until the rank of $H_{(-d)}$ is one less than the rank of $H$. Although the algorithm is easy to understand, the number of possible combinations of sets of rows that are to be removed is $n$ choose $d$ where $n$ is the number of rows of $H$. As $d$ increases, so does the number of possible combinations. The singular value decomposition (SVD) is typically used to find the rank of matrices. However, since the computational time needed to compute the SVD of a matrix is dependent on the number of rows and columns of the matrix, the SVD can be computationally expensive on large submatrices. For practical system matrices, as $d$ increases, the number of submatrices considered increases. In turn, the algorithm
becomes computationally expensive as the number of submatrices increases. Therefore, it would not be advantageous nor practical to do this exhaustive rank testing as $d$ and the size of the matrices increase.

Boros et al. [47] presented an algorithm to enumerate all the circuits of a matroid. The algorithm is based on property (C3) from Chapter 1. Let $M = (S, \mathcal{I})$ be a matroid with ground set $S$. Given such a matroid, a basis, $B \subset S$, of $M$, and an element $x \in S\setminus B$, $B \cup x$ contains a circuit $C$ of $M$. Using the same $B$ and all $x \in S\setminus B$, a collection of initial circuits is obtained. Using (C3), it is concluded that all the circuits of the matroid have been enumerated or a new circuit is found, and added to the collection. These steps are repeated until all the circuits of the matroid have been enumerated. In the case of linear matroids, finding a basis, finding circuits and using (C3) as a check is not difficult. Although using (C3) as a check is not difficult using a brute force methodology, it is essentially trying to find a new circuit of another matroid whose groundset is a subset of $S$. In other words, it is the problem that is trying to be solved. This new circuit is of course a circuit of $M$. As the algorithm is guaranteed to enumerate all the circuits of the matroid, it proves to be quite accurate. However, as the number of circuits increases, the number of pairwise comparisons using (C3) increases. The number of circuits may not be known, and the algorithm also requires the enumeration of all the circuits of the matroid before it terminates. This enumeration is unnecessary since only the ones of smallest cardinality are needed. Having to generate all the circuits can become quite taxing, especially if there are a
large number of them. It would be advantageous to avoid this enumeration.

As an alternative, Cho et al. [26] presented a branch-and-decompose algorithm. The theory behind this method stems from the connectivity of the matroid constructed from the given matrix. It is assumed that the matroid maintains a degree of disconnectedness. For a more detailed discussion of matroid connectivity and the properties used to develop this algorithm, one is referred to Cho et al. [26]. The most important observation is that for a matrix $T$ with a particular structure, the cogirth of $M[T]$ can be found by looking at submatrices of $T$. The matrix $T$ is transformed into bordered block diagonal form (BBDF), which can be seen in Figure 2.1, with a border, $P$, of rows. The BBDF matrix is a rearrangement of the columns and rows of $T$. While $d$ is small, the exhaustive rank test is used on the entire matrix. That is, while $d < (\frac{n_b}{n_b-1})|P| - 1$ where $n_b$ is the number of blocks and $|P|$ is the number of rows in the border, the exhaustive rank testing is used. Once $d$ reaches this bound, the algorithm intelligently chooses rows and columns of the matrix to create submatrices, and implements the exhaustive rank test on each submatrix to compute the cogirth of the original matrix. This method depends on the structure of the matroid.
and inherently on the matrix. Although it has the ability to provide nice results as presented by Cho et al. [26], it is not guaranteed to do so for all matrices with the desired structure. In particular, as the number of rows in $P$ increases, so does the number of exhaustive rank tests in the initial stage of the algorithm. This algorithm, although very clever, runs the risk of being very inefficient if the matrix and in turn the matroid does not have the structure illustrated by the figure above with $|P|$ small. Other methods have made the attempt to solve the problem for general linear matroids.

In contrast to the methods discussed above, Kianfar et al. [48] approached the problem from an optimization standpoint. They presented a 0-1 mixed integer program (MIP) to solve the cogirth problem. Given an $n \times p$ matrix $H$, with $n > p$, the formulation attempts to find a nonzero vector $x \in \text{null}(H)$, the null space of $H$, that minimizes the number of nonzero inner products $h_i x$ for $i = 1, \ldots, n$, where $h_i$ is a row of $H$. The assumptions are that $||h_i||_1 = 1$ for each row of $H$ and that $H$ has full column rank since $n > p$. The following is the 0-1 MIP formulation:

$$\begin{align*}
\min & \sum_{i=1}^{n} q_i \\
\text{s.t.} & -q_i \leq \sum_{j=1}^{p} h_{ij} x_j \leq q_i, \quad i = 1, \ldots, n \\
& -1 + 2z_j \leq x_j \leq 1, \quad j = 1, \ldots, p \\
& \sum_{i=1}^{p} z_i = 1
\end{align*}$$

(2.1)

where $x_j$ is a real variable for all $j$ and $q_i, z_j$ are binary variables for all $i$ and $j$. The
first set of constraints correspond to \( x \) being in the null space of \( H \). The second and third set of constraints ensure that the trivial solution \( x = 0 \) is not chosen. If the matrix is not full rank then another set of constraints must be added to ensure that \( x \) is in the row space of \( H \). The formulation is then solved using the MIP solver CPLEX [1]. Computational comparisons to the exhaustive rank test and Cho’s branch-and-decompose algorithm are presented in Table 3.4 in [48]. In some cases presented, the algorithm proposed by Cho et al. performs better, but the MIP formulation is shown to do better in most instances, especially as the size of the matrix increases. Upper and lower bounds for the degree of redundancy can still be acquired when optimality is not achieved [48]. The 0-1 MIP formulation reported is an initial approach, and the investigation is ongoing. The formulation sounds promising, especially if strong valid inequalities can be added to strengthen the formulation.

In order to find the cogirth of a matroid, Govindaraj [39] concentrates on the circuits of the dual matroid, \( M^* \), which are the cocircuits of \( M \). Rather than enumerate all cocircuits, an \( \ell^0 \)-norm minimization formulation is used to find the smallest cocircuits containing each element in the ground set \( S \). For a given matrix \( A \) and a vector \( b \), the following formulation will provide the minimum number of columns of \( A \) needed to obtain \( b \).

\[
\min \|x\|_0 \quad \text{s.t.} \quad Ax = b
\]  

(2.2)

It should be noted that the \( \ell^0 \)-norm is not actually a norm. It refers to the number of non-zeros of a vector. Rather than solve the \( \ell^0 \)-norm minimization problem, an
\(\ell^1\)-norm minimization approximation is used. In other words, replace \(\|x\|_0\) with \(\|x\|_1\). Here, \(\|x\|_1\) refers to the 1-norm of a real-valued vector. Recall the matrix \(H\) from the linear system. This heuristic would be performed on \(H^T\). The full algorithm consists of computing the matrix representation of the dual matroid, which is discussed in Oxley [61], and then solving a sequence of \(\ell^1\)-norm minimization problems. For each column, \(H_j, j \in \{1,\ldots,n\}\), of the \(H^T\), let \(b = H_j\) and let \(A\) be the matrix \(H^T\) excluding \(H_j\) and then solve the \(\ell^1\) approximation. Computational results reported by Govindaraj [39] indicate that the algorithm is efficient, but because the solutions are approximated, the algorithm is not always guaranteed to provide exact solutions. That being said, the inaccurate solutions still provide a good approximation to the true solution and the cogirth. The algorithm provides a viable option to give an approximation to the cogirth problem. The algorithm provided by Govindaraj is based on a technique to find an approximation to a problem that arises in compressive sensing. This problem is equivalent to finding the girth of a linear matroid. Along those lines one could also consider using other algorithms that address the compressive sensing problem such as the Basis Pursuit algorithm [25] and the Orthogonal Matching Algorithm (OMP) [62],[70] as possible approximation algorithms to find the cogirth of a linear matroid. In turn, the algorithms discussed above as well as the one presented in this paper could be considered to address the problem that arises in compressive sensing. Another possible option, which will be discussed in more detail in the following section, is given in [22].
2.3 The Hitting Set Problem

Recall the hitting set problem, a family \( \{U_i\} \) of subset of \( \{s_j = 1, ..., r\} \) is given. The objective is to find a hitting set \( W \) such that \( W \cap U_i \neq \emptyset \) for each \( i \). As mentioned in Chapter 1, Karp [46] showed that the set covering problem is equivalent to the hitting set problem. Therefore, it is relevant to discuss methods that have been presented to solve hitting set problems. This section presents such methods, including a generic algorithm introduced in [22] to solve implicitly constrained algorithms.

One method to find a minimum hitting set for a given instance would be to enumerate all the hitting sets and choose the one with smallest cardinality. However, in many instances, such an approach would prove unnecessary and computationally expensive. Reiter [64] introduces an approach based on logic to enumerate only minimal hitting sets in his work on theory of diagnosis. First, the family of subsets \( \{U_i\} \) must be computed, and then using what is called a pruned HS-tree (hitting set tree) the minimal hitting sets are enumerated. Reiter provides the details for the construction of the HS-tree. Greiner et al. [40] later showed that Reiter’s pruning techniques did not find all minimal hitting sets under certain conditions. They provide a method that creates an acyclic directed graph that, according to them, better adheres to Reiter’s description than his algorithm involving the HS-tree. Wotawa [76] discusses a variant of Reiter’s algorithm which does create HS-trees to compute the minimal hitting sets. Nyberg [60] presents a more generalized version of an algorithm described in [32] which iteratively updates a set of minimal hitting sets. As Reiter
states, the family $\{U_i\}$ must be known before implementing the hitting set algorithm. For implicitly defined problems, these algorithms could be incorporated in a heuristic to find minimum hitting sets for subsets of the family of sets, but cannot provide optimal solutions in every case.

Lin and Jiang [49], [50] also propose algorithms for the hitting set problem. The former is a genetic algorithm that follows the same principles as the algorithm discussed 2.1. The algorithm begins with a population of solutions that is constructed. Then through operations that mimic concepts such as mating and mutation, better solutions are introduced. The latter presents an approach similar to the HS-tree approach known as a binary HS-tree and another that treats components as boolean variables, and uses properties of boolean formulas to compute the hitting sets. These two approaches only enumerate minimal hitting sets. However, using these techniques requires the knowledge of the entire family of subsets that they are trying to hit. Therefore, using these techniques directly for an implicit problem is not plausible. However, they could be used to solve explicit subproblems. An analysis of a greedy algorithm on average case hitting sets by considering its performance on several random instances is provided in [73]. In each iteration, an element is chosen at random from a set of elements which intersect the largest number of sets which have not been hit yet.

A generic algorithm presented by Chandrasekaran et al. [22] to solve an implicit hitting set problem (IHSP) is in some ways similar to the proposed branch-and-cut
algorithm for the set covering problem. Both algorithms begin without any constraints, and iteratively add constraints and search for feasible solutions. There are a few of main differences between the algorithms. One is that the generic algorithm sometimes solves an explicit hitting set problem (EHSP) given the current subset of constraints while the branch-and-cut algorithm only solves the linear relaxations. Another is that the generic algorithm finds feasible solutions for the current subset of constraints, while the branch-and-cut algorithm searches for solutions that are feasible for the implicit problem. Finally, the branch-and-cut algorithm chooses variables to bound at upper and lower integral bounds to help reduce the search space by pruning solutions that will not provide better solutions. Overall, the generic algorithm provides a viable approach to these types of implicitly constrained problems. I will present the algorithm in more detail in Chapter 3, discuss some modifications and discuss how it can be used for the linear matroid cogirth problem. With the exception of the last algorithm discussed, the hitting set algorithms discussed do not provide a direct means to solve the implicit set covering problem.

2.4 Compressive Sensing

Recall the reconstruction problem described in Chapter 1,

$$\min \|s\|_0 \text{ s.t. } y = \Theta s \quad (2.3)$$
The idea is to reconstruct \( s \) using a minimal number of measurements. As is common in the literature, 2.3 is often transformed into the following form, and is referred to as *basis pursuit*.

\[
\min \| s \|_1 \text{ s.t. } y = \Theta s
\]

(2.4)

Santosa and Symes [68] were among the first to note that the \( \ell^1 \)-norm is sparse promoting. Holtz [42] offers a good survey of many theoretical results and algorithms developed to solve 2.4. Although I will not provide an exhaustive account of the work that has been done in the area of compressive sensing, I will highlight relevant work many of which can be found referenced in [42].

Although Santosa and Symes [68] were among the first to realize the ability of the \( \ell^1 \)-norm to promote sparse solutions under certain conditions, as Holtz [42] notes, Donoho [36] made great strides in providing applications and theoretical results for compressive sensing. Candès, Romberg and Tao [20],[16],[17] showed the validity of sparse signal recovery, and provided theoretical results that laid the foundation for many other results concerning the exact recovery of sparse signals. Among those is the *restricted isometry property*. Candès, Romberg and Tao [19], [18] discuss this property and a result for determining when a matrix with this property can be used to recover a sparse signal exactly; however, I will give the description provided in [42].

**Definition 2.1** A matrix \( \Phi \) is said to satisfy the *restricted isometry property* (RIP)
of order $k$ with constant $\delta := \delta_k \in (0, 1)$ if

\[(1 - \delta_k)\|x\|_2^2 \leq \|\Phi x\|_2^2 \leq (1 + \delta_k)\|x\|_2^2 \tag{2.5}\]

for any $x$ such that $\|x\|_0 \leq k$.

**Theorem 2.1** If the $n \times N$ matrix $\Phi$ satisfies the RIP of order $3k$ for some $\delta \in (0, 1)$, then for any vector $x \in \mathbb{R}^N$, the $\ell_1$ minimization problem 2.4 has a solution $x^*$ such that

\[\|x - x^*\| \leq \frac{C}{\sqrt{k}}\|x - x_k\|_1, \tag{2.6}\]

where $x_k$ denotes the best $k$-sparse approximation to $x$ and $C$ denotes a constant.

The theorem states that the larger $k$ is, the better approximation the $\ell_1$-norm provides. In fact, optimal results are obtained when $k = \Theta(n/\log(N/n))$ [34]. When actually performing this sparse signal recovery, there are various models and algorithms which have been proposed. I will highlight the most notable ones. As I mentioned above, 2.4 and variants of this problem have become the default models to consider. However, solving 2.4 is equivalent to solving the following linear program

\[
\min 1^T[u; v] \text{ s.t. } y = \Theta(u - v), \; u, v \geq 0 \tag{2.7}
\]
Another problem introduced by Tibshirani [69] is the following model

$$\min ||y - \Theta s||_2 \text{ s.t. } ||s||_1 < \tau$$

(2.8)

for some parameter $\tau$. Small $\tau$ promotes sparsity in $s$. This model can then be solved using any number of solvers.

There are in fact methods used to solve 2.4. One such solver, whose details are described in [77], [78], [33] is YALL1. Other methods involve a greedy approach. Such methods include matching pursuit [52] and many variants OMP [62],[70], stage-wise OMP (StOMP) [37], regularized OMP [58], and CoSaMP [57]. These methods work to minimize the objective function (residual) in 2.8 up to a certain tolerance by iteratively adding the one or a set of columns of the $\Phi$ that will provide the best solution based on the current residual. The approaches are greedy in the sense that they try to decrease the residual as much as possible at each iteration.

Other types of methods include that of Cormode and Muthukrishnan [30], [29]. They describe a method to construct the measurement matrix $\Phi$ and then build a set of coefficients to accompany the measurement matrix. Iwen [44] has also done some work in constructing matrices with the RIP. Recall that 2.3 is equivalent to finding the girth of linear matroid. Many of the algorithms used for compressive sensing rely on the matrix having the RIP. Since 2.3 is equivalent to finding the smallest linearly dependent set of columns that contains $y$ of the augmented matrix $[\Theta \ y]$, this leads to the following question. Is there a way to find the smallest set of linearly dependent
columns without using brute force to test every possible subset of columns? I address this question in Chapter 4. I propose an algorithm which uses a branch-decomposition and dynamic programming to address the problem, and a genetic algorithm to address this question.
Chapter 3

Linear Matroid Cogirth Algorithms

3.1 An Algorithm for the Set Covering Problem

As discussed in Chapter 2, there are several exact methods and heuristics to solve the following set covering problem (SCP),

\[ \min 1^T x \]
\[ \text{s.t. } Ax \geq 1. \]

However, these methods require knowledge of the entire system \( Ax \geq 1 \). For the SCP formulated from the cogirth problem, every constraint corresponds to a basis of \( \text{Row}(H) \) and each variable which has a cost of 1 corresponds to a row of \( H \). Therefore, in order to have the entire system \( Ax \geq 1 \), all the bases of the \( \text{Row}(H) \) must be known. Since this is generally not the case, the previously discussed methods cannot explicitly be used in this case. Instead, I implement a branch-and-cut algorithm, which is a branch-and-bound algorithm that incorporates cutting plane methods, to solve the SCP. I will first briefly discuss the branch-and-bound algorithm and why cutting planes are used.
A branch-and-bound algorithm is designed to solve integer programs to optimality. It begins by considering the linear relaxation of an integer program or mixed-integer program. In this case, the integer program is the set covering problem. At the root node, before the branching begins, the integer constraints are relaxed and the linear relaxation is solved. At this point, the solution, \( \bar{x} \), may be a mixture of fractional and integral variables. If it is integral and satisfies all explicit and implicit constraints, then the solution is optimal. If not, then the fractional variables are selected and the branching begins. Various branching rules are discussed by Achterberg et al. [2]. At each branch node, a variable is chosen to bound at its upper and lower integral bounds, and a subproblem is created for each bound. The same procedure is used on all the subproblems. Subproblems are pruned (eliminated) if they are infeasible, or if the optimal value of the subproblem is not better than the current best integral solution. Since there are an exponential number of branches, it would be advantageous to find better optimal values early in the procedure. Branching rules can have an effect on the number of branches considered and how quickly optimal solutions are found. Cutting plane methods are used for linear, integer, and mixed integer programs in order to improve efficiency of the algorithms used to solve the problems. The idea is to find valid inequalities or cutting planes for the program that will reduce the feasibility region, the solution set of the problem. By reducing the feasibility region, an optimal solution may be found more quickly. A discussion of cutting plane methods can be found in both Nemhauser and Wolsey [59] or Wolsey
However, the general cutting plane methods discussed may not be viable options in this case. In many of the cutting plane generation procedures discussed, the entire constraint set is needed.

Referring back to the SCP that we formulated to solve the matroid cogirth problem, since no bases of $\text{Row}(H)$ are initially known, there are not any constraints. Therefore, we must use a branch-and-cut algorithm to solve the SCP. The branch-and-cut algorithm is guaranteed to solve the SCP to optimality. A more general discussion of the branch-and-cut algorithm can be found in Wolsey [75]. Figure 3.1 shows a flow chart of the basic algorithm. The implementation of the branch-and-cut algorithm is discussed below. The following subsections describe the intricacies of the algorithm. For the flow chart, $\min \{ c^T x \mid x \in X \}$ is the initial formulation with $X$ the set of feasible solutions, $P$ is the original formulation of the problem, $z$ is the optimal value for the original formulation, $x^*$ is the optimal solution and $\bar{z}$ is the bound used to prune solutions. $P^i$ and $X^i$ correspond to the formulation and solution set for the $i$th subproblem picked from the node list. $P^{i,k}$ is the $k$th formulation for the $i$th subproblem after a cutting plane has been added to it, $x^{i,k}$ is the solution to the formulation and $z^{i,k}$ is the optimal value. $P^i_t$ and $X^i_t$ are subproblems created after a fractional variable has been chosen.

Since no basis is initially known, the algorithm begins at the root node with no constraints; the constraint set, which will be referred to as $Q$, is empty. The initial solution to the relaxation, the zero vector, is not a feasible solution to the SCP. At this
INITIALIZATION
\[ z = \min \{ c^T x | x \in X \} \ \bar{z} = \infty, \ x^* \text{ is empty. Process initial problem and put on Nodelist} \]

NODE
If Nodelist is empty, go to EXIT. Else choose and remove node \( i \) from Nodelist and go to RESTORE

RESTORE
The formulation \( P^i \) of the set \( X^i \). Set \( k = 1 \), and \( P^{i,1} = P^i \)

LP RELAXATION
Iteration \( k \). Solve \( z^{i,k} = \min \{ c^T x | x \in P^{i,k} \} \). If infeasible, prune and go to NODE. Else solution \( x^{i,k} \) and go to CUT

CUT
Iteration \( k \). Try to cut off \( x^{i,k} \). If no cuts found, go to PRUNE. Else add cuts to \( P^{i,k} \) giving \( P^{i,k+1} \). Increase \( k \) by 1, and go to LP RELAXATION

PRUNE
If \( z^{i,k} \leq \bar{z} \), go to NODE. If \( x^{i,k} \notin X \), set \( \bar{z} = x^{i,k} \), update \( x^* \leftarrow x^{i,k} \) and go to NODE. Else go to BRANCHING

BRANCHING
Create two or more new problems \( X^j \) with formulations \( P^j \). Add them to the Nodelist

EXIT
Incumbent \( x^* \). Optimal Value \( z \)

Figure 3.1: Flow chart of basic Branch-and-Cut Algorithm.
point a basis of $\text{Row}(H)$ is found and a constraint is added to $Q$. A new solution, $x$, to the relaxation is found. The values of the individual variables are used as weights for the rows of $H$ since each variable corresponds to a row. In other words, rows are compiled together one by one by minimum weight until a basis is found. A constraint corresponding to a minimum weight basis $B$ is added to $Q$ if $x$ violates the constraint. That is, if the inner product between the solution $x$ and the rows corresponding to the newly found basis $B$ is less than 1, the constraint is added. The constraint set $Q$ is a subset of the system $Ax \geq 1$. Because $x$ is used as weights to find a new basis, it is guaranteed that a newly added constraint was not previously in $Q$. If it was, then $x$ would have already satisfied the corresponding constraint. The idea behind adding constraints this way is that not all the constraints of $Ax \geq 1$ are needed. Only significant bases are used to add constraints to the system. It would not be advantageous to find every basis of $H^T$ since there may be a large number of them in general. With this in mind, it is often not the case that the optimal solution to the SCP will be found at the root node since a relaxation of a subset of constraints of the SCP is being solved in the algorithm and the polyhedron for the SCP may not be totally unimodular. Therefore, after adding as many constraints as possible at the root node, the actual branching part of the algorithm is begun. Along with branching on fractional elements of $x$, more cutting planes are added at each node of the branch tree. Since the system $Ax \geq 1$ is not known in its entirety, constraints from the system are added the same way they were at the root node. Other cutting plane
methods are also incorporated to find other valid inequalities. The rest of this section will concentrate on cutting plane generation subroutines that were incorporated in the algorithm, the branching rule that was used, and a heuristic that was used to find feasible solutions to the SCP.

3.1.1 Generating Cutting Planes

As discussed above, constraints from $Ax \geq 1$ are added iteratively by finding bases of $\text{Row}(H)$. The algorithm attempts to add these constraints at the root node and at each branch node. In order to find bases, a greedy algorithm is used. Once a solution, $x = (x_1, \ldots, x_n)^T$, is found, the elements are sorted. The sorted solution is $x_{\sigma(1)} \leq x_{\sigma(2)} \leq \ldots \leq x_{\sigma(n)}$ where $\sigma$ is a permutation of the indices. Using the weights, rows of $H$ are added to a matrix, say $H_s$, as long as they increase the rank of $H_s$. The rows are added in order of minimum weight until the rank of $H_s$ is equal to the rank of $H$. Once a basis is found, the corresponding constraint is added to constraint set $Q$ if it is violated by the current solution $x$. An initial set of bases is found by first finding a feasible solution to the SCP. The initial set of bases is obtained in such a way that they are as disjoint as possible. The method used to find a feasible solution is discussed in Subsection 3.1.3. Once a feasible solution, $x$, to the SCP is found, a greedy algorithm is used to find a set of bases. For each $j$ such that $x_j = 1$, a basis of $\text{Row}(H)$ containing row $j$ is found. The weight of each row of $H$ is initially set to zero. Every time a row is added to a basis, its weight is increased by one. Rows
with minimum weight are added to a basis first. Bases were found this way to make them as disjoint as possible in an attempt to ensure diversity of the constraints. After this initial set of bases that are as disjoint as possible, bases are found based on the the solution $x$ to the linear relaxation. All the constraints obtained from bases are global cuts. The algorithm also attempts to add valid inequalities generated from these basis constraints.

Balas and Ng [8] consider the set covering polytope, $P_I(A) := \text{conv}\{x \in \mathbb{R}^n \mid Ax \geq 1, x \text{ binary}\}$. They discuss the following class of inequalities. For a subset of rows $S$ of the matrix $A$ from $Ax \geq 1$, the inequality $\alpha^S x \geq 2$ associated with $S$ is defined by

$$\alpha^S_j = \begin{cases} 
0 & \text{if } a_{ij} = 0 \text{ for all } i \in S, \\
2 & \text{if } a_{ij} = 1 \text{ for all } i \in S, \\
1 & \text{otherwise.}
\end{cases}$$

According to Balas and Ng [8], this class of inequalities, which will be referred to as $CP$ and is valid for $P_I(A)$, can be generated using the following procedure D:

\begin{algorithm}
1: (i) add the inequalities $a^i x \geq 1, i \in S$
2: (ii) divide the resulting inequality by $|S| - \varepsilon, 0.5 < \varepsilon < 1$
3: (iii) round up all coefficients to the nearest integer
\end{algorithm}

It would be advantageous to incorporate inequalities from this class. Fortunately, Balas and Ng [8] proved the following theorem. Let $\bar{x}$ be a fractional solution to the $Ax \geq 1, 0 \leq x \leq 1, R$ be the row indices, $N$ be the column indices, $I := \{j \in N|\bar{x}_j = \}$
1}, and \( R(I) := \{ i \in R \mid a_{ij} = 0, \forall j \in I \} \).

**Theorem 3.1** Let \( a^Sx \geq 2 \) be an inequality in the class \( CP \) that cuts off \( \bar{x} \). Then \( a_j^S = 0 \) for all \( j \in I \); i.e., \( S \subseteq R(I) \).

The theorem states that the search for the set \( S \) of rows associated with the \( a^Sx \geq 2 \) that cuts of \( \bar{x} \) can be restricted to \( R(I) \). Procedure D is incorporated into the branch-and-cut algorithm at each branch node. After as many inequalities from \( Ax \geq 1 \) have been added to \( Q \), the theorem proved by Balas and Ng [8] is used to try to find constraints using procedure D. Since only a subset of the original constraints is known, inequalities from the class \( CP \) may not always be found using this procedure. Note that this procedure can be used at branch nodes incrementally. Further discussion of these cuts and their use in the branch-and-cut algorithm is discussed later in this chapter.

Beasley et al. [12] also discuss feasible solution exclusion constraints for the SCP. Suppose \( T_c \) is a set of column indices that correspond to the best feasible solution for the SCP. It is assumed without loss of generality that \( T_c - j \) is not a feasible solution for all \( j \in T_c \). Then according to Beasley et al. [12], the following two constraints can be added to the program:

\[
\sum_{j \in T_c} x_j \leq |T_c| - 1
\]

\[
\sum_{j \notin T_c} x_j \geq 1
\]
The idea is that if a better solution exists, it can be obtained by replacing at least one column in the current solution. These constraints are incorporated into the branch-and-cut algorithm. These constraints are only considered after branching has begun. After a feasible solution is found using the heuristic discussed below, a check is performed to see if it is better than the current best feasible solution. If so, then two feasible solution exclusion constraints are added. This procedure is done every time the best feasible solution is replaced in a branch node. As I mentioned earlier, Achterberg [2] discusses many possible branching techniques. The next section discusses which rule was used in the proposed branch-and-cut algorithm.

3.1.2 Branching

Various branching rules can be incorporated into a branch-and-cut algorithm. The branching rule used for the proposed branch-and-cut algorithm is that the most infeasible variable is chosen to branch on [2]. Since \( 0 \leq x \leq 1 \), the closer a variable is to 0.5, the more infeasible it is. The branch-and-cut algorithm implements depth-first search branching, i.e. once a fractional variable, say \( x_k \) is chosen, \( x_k \) is set to zero and the resulting subproblem is explored before \( x_k \) is set to one. The depth-first branching is incorporated in a recursive manner. In order to enhance the branch-and-cut algorithm, a method to find better feasible solutions to the SCP is also included. The method is discussed in the following subsection.
3.1.3 Finding Feasible Solutions

Recall that a feasible solution to the SCP is actually a cocircuit of the matroid \( M = (S, \mathcal{I}) \) where the ground set \( S \) corresponds to the indices of the rows of \( H \) and \( \mathcal{I} \) is the collection subsets corresponding to linearly independent sets of rows of \( H \). Therefore, it would be beneficial to find feasible solutions as often as possible without enumerating all possible solutions. In order to find feasible solutions, a greedy algorithm is implemented. The algorithm is similar to that used to find bases. As before, the solution, \( x \), to the linear program is used as weights. However, in this case, rows with max weight are removed from the matrix \( H \) one at a time until the rank of the resulting matrix is reduced by one. The rows removed represent a feasible solution to the SCP. The intuition behind removing rows in this manner is that rows with larger weights are more significant to the current relaxed subproblem and may be more likely to be contained in a cocircuit of \( M[H] \). The only way to ensure that a feasible solution to the SCP found this way represents the smallest cocircuit is to enumerate all possible feasible solutions. However, it is not practical for this method to be used by itself. Instead, this method is used to find feasible solutions of smaller cardinality if possible. In doing so, better upper bounds for the branch-and-cut algorithm can be found and more branch nodes can be pruned earlier in the branch-and-cut algorithm. This method is applied after a new solution \( x \) to the relaxation is found. The next section describes a procedure to arrange a matrix into BBDF. The BBDF is incorporated into a modification of the branch-and-cut
algorithm. Computational results of the modified algorithm are presented in later in this chapter.

3.2 Rearranging Matrices into Bordered Block Diagonal Form

Recall that part of the method described by Cho et al. [26] involved using a matrix that had been rearranged into bordered block diagonal form (BBDF), Figure 2.1. In particular, Cho et al. [26] were considering sparse matrices when discussing the BBDF. The idea is that if a matrix $T$ can be arranged into BBDF, then the cogirth of $M[T]$ can be found by considering submatrices if the cogirth is greater than $\frac{n_b}{n_b-1})|P| - 1$ where $n_b$ is the number of blocks, or the blocks of the rearranged matrix $BT$ along with elements of the border, e.g the submatrices $[A_1, P_1], [A_2, P_2], ... , [A_2, P_2]$ if the cogirth is greater than $2|P| - 1$. To put the matrix in this form, a set of border rows must be found. To find these border rows we use a method that is discussed by Cho [27]. Create a graph $\hat{G} = (\hat{V}, \hat{E})$ such that the members of $\hat{V}$ represent rows of the matrix $T$ and an edge $(u, v) \in \hat{E}$ if $T_{uk}$ and $T_{vk}$ are both nonzero for some column $k$ of the matrix $T$. That is, an edge exists if rows $u$ and $v$ share a common column $k$. Once $\hat{G}$ is created, a minimum separating set of vertices $S_v$ of $\hat{G}$ is found. A separating set of vertices is a set $S \subset \hat{V}$ whose removal disconnects the remaining subgraph.

Menger’s Theorem, which follows, is significant to finding the smallest separating set.

**Theorem 3.2** Let $G = (V, E)$ be a graph and $A, B \subseteq V$. Then the minimum number of vertices separating $A$ from $B$ in $G$ is equal to the maximum number of disjoint $A-B$
paths in $G$.

In order to find the smallest separating set, we find the maximum number of $u - v$ paths for all $u, v \in \hat{G}$ with $u \neq v$. We can do this by solving a series of max-flow min-cut problems from another graph obtained from $\hat{G}$. The details can be found in [27]. After the separating set is found the blocks of $BT$ can be found. Let $\hat{G} = (V, E)$ be a bipartite graph where members of $V$ represent rows and columns of $T$. Let $(u, k) \in E$ if $T_{uk}$ is nonzero for row $u$ and column $k$. Remove the separating set $S_v$ from $\hat{G}$ and any edges adjacent to it. The resulting connected components represent the blocks of the matrix $BT$. In section 3.5, the branch-and-cut algorithm is run on several test instances, and the computational results are presented. A discussion of possible avenues for further research is discussed in Chapter 5.

### 3.3 Cogirth Heuristics

Up to this point, I have mainly talked about solving the set covering algorithm to optimality and thus finding the cogirth of a linear matroid. However, suppose there is not enough time to solve the problem to optimality, or given the application of sensor network reliability, it is enough to have an upper bound. We know that finding cocircuits of linear matroids is an NP hard problem. However, is it possible to take advantage of the aforementioned bordered block diagonal form (BBDF) of a matrix? The short answer is yes. The long answer is discussed below.
3.3.1 Advantages of the Bordered Block Diagonal Form

Recall that the BBDF, Figure 2.1, consists of a block diagonal matrix with a set of border rows at the end with more rows than columns. If the border, $P$, of the matrix is removed, and the rank of the rest of the matrix is less than the original matrix, then the border contains a cocircuit. Therefore, the cogirth of the matrix is bounded above by $|P|$ if removing $P$ reduces the rank of the matrix. Also, the search for the cogirth can be restricted to submatrices that have less than $|P|$ columns. Even if the rank of the matrix is not reduced, the same thing can be done with the rows corresponding to the blocks of the matrix. For each, block $A_i$ of the matrix, remove the rows corresponding to $A_i$. If the rank is reduced, then $A_i$ contains a cocircuit, and $|A_i|$ can act as an upper bound. Observe that if the BBDF matrix is full rank and removing $P$ reduces the rank, then the remaining matrix is rank deficient. This means that at least one of the blocks, say $A_j$, is rank deficient. Therefore, $|A_j| - \text{rank}(A_j) + 1$ is an upper bound on the cogirth of the matrix since any $|A_j| - \text{rank}(A_j) + 1$ rows of $A_j$ are a cocircuit. If $|P| > |A_j| - \text{rank}(A_j) + 1$, then a better upper bound has been found indirectly. Furthermore, according to Cho [26], if the cogirth is larger than $2|P| - 1$, the search for the cogirth can be restricted to $[A_i P_i]$ for each block of the matrix.

In the case that the search can be restricted to each of the blocks, $[A_i P_i]$, the submatrix provides an upper bound on the cogirth of the entire matrix. That is, the cogirth of $[A_i P_i]$ is bounded above by $|A_i \cup P_i| - \text{rank}([A_i P_i]) + 1$. A cobasis plus
another row is a cocircuit, and is therefore an upper bound on the cogirth. A cobasis of \([A_i \ P_i]\) can be found by first finding a basis. The complement of the basis with respect to \([A_i \ P_i]\) is a cobasis. Therefore, a cobasis along with a row from the basis that was found is a cocircuit. Outlines for the two heuristics to find an upper bound are provided below.

**Algorithm 3.2 Cogirth Heuristic 1**

1: Given \(\tilde{A}\) with border \(P\)
2: if Removing \(P\) reduces the rank then
3: \(|P|\) is an upper bound \(U\) on the cogirth
4: end if
5: for \(A_i \in A\) do
6: Remove \(A_i\)
7: if rank\((\tilde{A}_{-A_i})\) < rank\((\tilde{A})\) then
8: \(|A_i|\) is an upper bound on the cogirth
9: end if
10: end for

**Algorithm 3.3 Cogirth Heuristic 2**

1: Given \(\tilde{A}\) with border \(P\)
2: for \([A_i P_i] \in A\) do
3: \(|A_i \cup P_i| - \text{rank}\([A_i P_i]\) + 1\) is an upper bound \(U\)
4: Find basis \(B\) of \([A_i P_i]\), \(\bar{B}\) is complement of \(B\)
5: \(B \cup \{b\}\) for \(b \in \bar{B}\) is a cocircuit with cardinality \(U\)
6: end for

I also considered using the formulation presented by Kianfar et al. [48] as a heuristic. Instead of solving the 0-1 MIP as a MIP, for each \(j \in \{1, ..., p\}\), set \(z_j = 1\) and relax the binary constraints on \(q_i\) for all \(i \in \{1, ..., n\}\); i.e., \(0 \leq q_i \leq 1\) for all \(i \in \{1, ..., n\}\). Then, solve the resulting sequence of problems for each \(z_j\). Let
\{z_{\sigma(1)}, \ldots, z_{\sigma(k)}\} be the \(k\) variables from \(\{z_1, \ldots, z_p\}\) that provide the best \(k\) optimal values when each of the variables was set to 1. Note \(k\) is specified by the user. Since \(\sum_{j=1}^{p} z_j = 1\) and \(z_j\) is binary for each \(j \in \{1, \ldots, p\}\), only one variable is set to 1 in each instant. Given these \(k\) variables, \(k\) 0-1 MIP are solved with the original restriction that \(q_i\) is binary for all \(i \in \{1, \ldots, n\}\) and the new restriction \(z_{\sigma(j)} = 1\) for each \(j \in \{1, \ldots, k\}\). In other words, for each \(j \in \{1, \ldots, k\}\), solve the following problem.

\[
\begin{align*}
\min & \quad \sum_{i=1}^{n} q_i \\
\text{s.t.} & \quad -q_i \leq \sum_{i=1}^{p} h_{ij} x_j \leq q_i, \quad i = 1, \ldots, n \\
& \quad -1 + 2z_j \leq x_j \leq 1, \quad j = 1, \ldots, p \quad (3.1) \\
& \quad \sum_{i=1}^{p} z_i = 1 \\
& \quad z_{\sigma(j)} = 1
\end{align*}
\]

### 3.4 Hitting Set Algorithm

As I mentioned in Chapter 2, there is a generic implicit hitting set algorithm proposed in [22] that could also be used for the implicit set covering problem. Given a universe of elements \(U\), and a collection \(\Upsilon\) of subsets \(B_1, \ldots, B_m\) of \(U\), we want to find some subset \(H \subseteq U\) which has a nonempty intersection with each subset in the collection. The set \(H\) is feasible if it has a nonempty intersection and optimal if it is of minimum cardinality as well. The subset of \(\Upsilon\) for the hitting set algorithm is denoted by \(\Gamma\). The pseudocode for the algorithm is the following:
Algorithm 3.4 Implicit Hitting Set Algorithm

1: Initialize $\Gamma \leftarrow \emptyset$
2: while true do
3:     $H \leftarrow U$
4:     while there exists a $\Gamma$-feasible set $H' = (H \cup X) - Y$ such that $X, Y \subseteq U$, $|X| < |Y|$ do
5:         if $H'$ is feasible then
6:             $H \leftarrow H'$;
7:         else
8:             $\Gamma \leftarrow \Gamma \cup T$ for some $T \in \Upsilon$ such that $H' \cap T = \emptyset$
9:         end if
10:     end while
11:     Construct a $\Gamma$-optimal set $K$
12:     if $|H| = |K|$ then
13:         return $H$ and halt ($H$ is optimal)
14:     end if
15:     if $K$ is feasible then
16:         return $K$ and halt ($K$ is optimal)
17:     else
18:         $\Gamma \leftarrow \Gamma \cup T$ for some $T \in \Upsilon$ such that $K \cap T = \emptyset$
19:     end if
20: end while

Similar to the branch-and-cut algorithm, the idea is to iteratively find bases and add them to the growing list $\Gamma$. When a new basis is found, an element of the basis is chosen to become part of the $\Gamma$-feasible set $H'$. In order to find a new basis, a greedy algorithm can be used to find a minimum weight basis as in the branch-and-cut algorithm where the weights are obtained solving a linear relaxation of the integer program. In order to choose a good partial hitting set $H'$, the same weights can be used. Instead, element with highest weight can be chosen, since it is more likely to be in a feasible solution. Constructing a $\Gamma$-optimal set $K$ can be done by solving an integer program with any integer programming software such as CPLEX [1] or Gurobi.
Although solving the integer program is NP-Hard, the overall performance will depend on the choice of \( \Gamma \), as noted in [22]. Despite the fact that the algorithm is guaranteed to find an optimal solution, finding one in a practical amount of time is of great interest. However, a near optimal upper bound may be sufficient if it can be obtained quickly, as is the case in compressive sensing when the \( \ell_1 \)-norm cannot be used.

### 3.5 Computational Results

In this section, some computational results are presented to validate the proposed branch-and-cut algorithm, and compare its performance to that of Cho et al. [26], and Kianfar et al. [48], two of the methods discussed in Chapter 2. The branch-and-cut algorithm is validated using the entire matrix as input. A version that exploits the BBDF structure of the matrix is also presented. However, since the branch-and-cut algorithm was not able to verify an optimal solution in the allotted time for many of the test instances, only the version which exploits the BBDF structure compared to the other algorithms. The results for branch-and-cut algorithm with the entire matrix as input are still shown in Table 3.2. The algorithms are compared using computational time which is recorded in seconds. A note on the branch-and-cut algorithm is that the class of inequalities developed by Balas and Ng [8] were difficult to find. In the instances tested, cuts from this class were not found and added to the set of constraints.
Structural information about the test instances is contained in Table 3.1 and Table 3.4 contains the comparison of computational results. The test instances can be found at http://ise.tamu.edu/people/faculty/kianfar/personal/DRinstances.zip. For test instance categories 1, 2, and 5-8, there are five different instances. For categories 3 and 4, there is only one instance. Table 3.1 shows the size of the matrix \((n \times p)\), the number of rows in the border, \(|P|\), which was computed using the method discussed in [27], and the average DoR, \(\eta\), for the test instances. In the first column, the first number indicates the category, and the number in parentheses is the number of instances for that category. For the test instances in which “\(> 18000\)” is seen in the computational times, the algorithms were unable to solve the problem of interest to optimality for the given time limit. The algorithms were stopped manually. Note that Cho’s algorithm may consider 1 or more blocks to combine when searching for the cogirth, whereas the BBDF branch-and-cut looks at individual blocks along with the corresponding border rows to find the cogirth. Also, note that since \(\eta > 2|P| - 1\) for the test instances [26], the BBDF branch-and-cut makes sense. All three algorithms were tested on the same machine, a Dell Precision T3500 Tower Workstation with an Intel® Xeon® Dual Core W3505 2.53GHz processor. The algorithms were coded in C++, and Gurobi [43] was used to solve all the linear, and mixed integer program formulations created during the experiments.

In Table 3.2, two versions of the branch-and-cut algorithm are reported. One version found an initial feasible solution using brute force to find an upper bound.
The other version did not. The initial feasible solution was found by removing rows one at a time until the rank of the matrix was reduced. An initial set of constraints based on this feasible solution were then found. Although both versions find optimal or near optimal upper bounds quickly, the algorithm is unable to verify the optimality of or find better solutions. This is due to the size of the tree created during the branch-and-cut procedure. The size of the tree increases exponentially with respect to the number of columns of the matrix. However, the algorithm works fairly well as a heuristic to find an upper bound on the set of test instances. An implementation of the hitting set algorithm for the cogirth of linear matroids is also presented in Table 3.3. The algorithm is able to a lower bound that is equal to the optimal value but is only able to verify optimality for two of the categories. It is unable to find good lower bounds for the other categories. This is due in large part to the number of constraints needed as well as the fact the hitting set algorithm solves explicit set covering problems.

Although not reported here, the BBDF branch-and-cut algorithm finds the best feasible solutions within seconds. These solutions turned out to be optimal or near optimal. The rest of the time was spent verifying the optimality of the current solution or trying to reduce the upper bound. All the algorithms perform well on the first two sets of instances with the BBDF branch-and-cut doing slightly better. For categories 3 and 4, the 0-1 MIP performed better. For the instances in category 5, we see that Cho’s algorithm performs the best overall, with the BBDF branch-and-cut following.
For the 0-1 MIP, only one of the five instances was solved to optimality. The rest reached the time limit. Therefore, the average only includes the one instance. This is noted in Table 3.4. For categories 6-8, none of the algorithms find an optimal solution, but the BBDF branch-and-cut and 0-1 MIP find feasible solutions that are much closer to the actual optimal value than Cho’s algorithm. Overall, it appears that the BBDF branch-and-cut and the 0-1 MIP perform better when the cogirth is larger than 4 and the matrix contains several blocks with the exception of category 5. This may indicate that Cho’s algorithm and the 0-1 MIP are better to use when \( \eta \) is small, and Algorithm 2 is better to use when \( \eta \) is large and the given matrix has the suggested BBDF structure. The 0-1 MIP also provides good bounds for higher cogirth values. Unfortunately, a range for \( \eta \) must be known beforehand in order to decide which algorithm to use. This is highly unlikely. Something else to consider is the number of blocks and size of the blocks that have to be considered by the algorithm. In all, the BBDF branch-and-cut algorithm and 0-1 MIP are competitive. The key components to the efficiency of the BBDF branch-and-cut algorithm appear to be the development of the constraint system, and the size and number of blocks that need to be solved. Although it is not clear in the tables, it should be noted that some of the categories contained outliers when the average run times were being computed.

Although the methods discussed in this article provide good upper bounds for the matroid cogirth problem, when considering a minimization problem, a lower bound
on the optimal solution is often more useful than an upper bound. However, a good lower bound is often harder to obtain. Similarly, when considering how many sensor failures are needed to lose the integrity of a sensor network, a lower bound is also more useful. With this in mind, we note a relationship between the spark of a matrix and the girth of a linear matroid. Donoho and Elad [35] define the spark of a matrix as the smallest number of linearly dependent columns of a matrix. Consider a matrix $F$ and $M[F]$ with the indices of the columns as the ground set. The circuits of $M[F]$ are all the sets of minimally dependent columns. Therefore, the girth, cardinality of the smallest circuit, of $M[F]$ is equal to the spark of $F$. Also, the cogirth of $M[F]$ is equal to the spark of the matrix representation of $M^*[F]$, the dual matroid of $M[F]$. Donoho and Elad [35] provide a bound on the spark of the matrix. So, one could use this lower bound as a lower bound on the cogirth of a linear matroid. Since we are looking for the smallest cocircuit, finding the smallest set of pairwise disjoint bases can also be used as a lower bound. We note that the optimal value of the linear relaxation found during the branch-and-cut procedure also provides a lower bound.

The lower bound can be computed from the linear relaxation during the branch-and-cut procedure. Let $Z$ of size $m \times n$, $m > n$, with rank $r$. The branch-and-cut procedure finds bases, feasible solutions and solves linear relaxations. To find a basis, we consider $Z^T$. Every column of $Z^T$ might need to be considered, and the rank a submatrix of $Z$ is computed to check if it provides a basis using the singular value decomposition (SVD). The complexity of the SVD is $O(mr^2 + r^3)$ [38] since the size of
the submatrices considered is $O(r)$. Therefore, computing a basis is $O(n(mr^2 + r^3))$. A feasible solution is found in a similar manner, and thus can also be found in $O(n(mr^2 + r^3))$. The subroutines to find bases and feasible solutions are called in a while loop at each node of the branching procedure, but the number of iterations of the while loop can be limited. There are an exponential number of branching nodes, but in practice the branch-and-cut procedure does not consider all the nodes. A linear relaxation of the current SCP is also solved after a new basis is added to the model using Gurobi.

Table 3.5 contains computational results for two cogirth heuristics presented in 3.3 which correspond to algorithms 3.2 and 3.3 respectively. For all the instances, both heuristics perform about the same in terms of efficiency and precision with the exception of categories 3 and 4. In fact, the heuristics find optimal solutions for all the instances in category 2. The heuristics find relatively near optimal solutions which provide good upper bounds on the cogirth. Observe that all the upper bounds on average are found in less than 15 seconds with the exception of categories 7 and 8. However, the cogirth for the instances in categories 7 and 8 are fairly efficient with average times hovering around 75 and 30 seconds respectively. Table 3.6 contains computational results for a heuristic which uses linear relaxations of the 0-1 MIP. Observe that the heuristic finds optimal solutions for every instance with the exception of the instances in category 8. The heuristic performs extremely well. Overall, all three heuristics perform well. The BBDF cogirth heuristics can be used to generate
upper bounds when the matrix has the BBDF. The heuristic which solves linear relaxations of the 0-1 MIP is useful whether or not the matrices have the BBDF. In Chapter 5, I present some concluding remarks and further research.

| Cat.(No.) | $n \times p$ | $|P|$ | Num. of Blocks | bound# | Avg. $\eta$ |
|-----------|------------|-------|----------------|--------|-------------|
| 1(5)      | 26 $\times$ 12 | 2     | 4              | 1.67   | 4           |
| 2(5)      | 66 $\times$ 27 | 3     | 9              | 2.37   | 7           |
| 3(1)      | 154 $\times$ 72 | 2     | 2              | 3      | 4           |
| 4(1)      | 318 $\times$ 144 | 2     | 2              | 3      | 4           |
| 5(5)      | 222 $\times$ 55 | 2     | 11             | 1.2    | 14          |
| 6(5)      | 1009 $\times$ 252 | 1     | 42             | 0.025  | 17          |
| 7(5)      | 2018 $\times$ 504 | 2     | 84             | 1.024  | 18          |
| 8(5)      | 501 $\times$ 384 | 9     | 4              | 11     | $\geq 1, \leq 34$ |

Table 3.1: Test instance information. #: $\text{bound} = \frac{\text{Num.of Blocks}}{\text{Num.of Blocks} - 1} |P| - 1.$

<table>
<thead>
<tr>
<th>Cat.</th>
<th>$\eta$</th>
<th>w/ IFS</th>
<th>Time</th>
<th>w/o IFS</th>
<th>Time</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>$\approx 0$</td>
<td>4</td>
<td>$\approx 0$</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>7</td>
<td>$\approx 0$</td>
<td>7</td>
<td>1.8</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>$\approx 0$</td>
<td>4</td>
<td>147</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>139</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>14</td>
<td>$\approx 0$</td>
<td>17</td>
<td>2</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>17</td>
<td>14</td>
<td>19</td>
<td>26</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>18</td>
<td>179</td>
<td>20</td>
<td>386</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>8</td>
<td>[1, 34]</td>
<td>36</td>
<td>20</td>
<td>36</td>
<td>47</td>
<td>$&gt; 18000$</td>
</tr>
</tbody>
</table>

Table 3.2: Computational times in seconds times for the Branch-and-Cut algorithm. IFS denotes an initial feasible solution.
<table>
<thead>
<tr>
<th>Cat.</th>
<th>$\eta$</th>
<th>Best Lower Bound</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>7</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5401</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>7</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>2</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>1</td>
<td>$&gt; 18000$</td>
</tr>
<tr>
<td>8</td>
<td>$[1,34]$</td>
<td>-</td>
<td>$&gt; 18000$</td>
</tr>
</tbody>
</table>

Table 3.3: Computational times in seconds times. For the last category, a lower bound could not be found because the explicit set covering problem could not be solved.

<table>
<thead>
<tr>
<th>Matrix H</th>
<th>BBDF Branch-and-Cut</th>
<th>Cho</th>
<th>0-1 MIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat. $\eta$ $\eta^+$ Time $\eta^+$ Time $\eta^+$ Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 4 4 0.2 4 $\approx 0$ 4 0.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 7 7 0.8 7 10.8 7 23.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 4 4 272 4 2214 4 24.54</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 4 918 4 5408 4 54.42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 14 14 1862.4 14 640.2 14 12169*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 17 $&lt; 19$ $&gt; 18000$ $&gt; 2$ $&gt; 18000$ $[5,17]$ $&gt; 18000$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 18 $&lt; 20$ $&gt; 18000$ $&gt; 2$ $&gt; 18000$ $[4,18]$ $&gt; 18000$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 $[1,34]$ $&lt; 36$ $&gt; 18000$ $&gt; 2$ $&gt; 18000$ $[1,34]$ $&gt; 18000$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.4: $\eta^+$ is the computed DoR for the algorithms. *: For category 5, the 0-1 MIP reached optimality for only 1 of the 5 instances.

<table>
<thead>
<tr>
<th>Matrix H</th>
<th>Heur. 1</th>
<th>Heur. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cat. $\eta$ $\eta^+$ Time $\eta^+$ Time $\eta^+$ Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 4 5 $\approx 0$ 5 $\approx 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 7 7 $\approx 0$ 7 0.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 4 33 $\approx 0$ 33 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 4 53 1 53 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 14 19 $\approx 0$ 17 $\approx 0$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 17 23 13.6 19 7.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 18 20 75 20 75.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 $[1,34]$ 36 23.6 36 26.8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5: Computational times in seconds times for the BBDF cogirth heuristics. $\eta^+$ is the computed DoR for the algorithms.
Matrix H

<table>
<thead>
<tr>
<th>Cat.</th>
<th>$\eta$</th>
<th>$\eta^+$</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>$\approx 0$</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>7</td>
<td>1.6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>14</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>17</td>
<td>757.6</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>18</td>
<td>5747</td>
</tr>
<tr>
<td>8</td>
<td>[1,34]</td>
<td>[1,34]</td>
<td>$&gt; 18000$</td>
</tr>
</tbody>
</table>

Table 3.6: Computational times in seconds for the heuristic which uses linear relaxations of the 0-1 MIP. $\eta^+$ is the optimal value computed by the heuristic.
Chapter 4

Linear Matroid Girth Algorithms

4.1 From Cogirth to Girth

Recall that a linear matroid is derived from a matrix where the ground set corresponds either to the set of columns or rows. In this case, the ground set corresponds to the set of columns. The girth of a linear matroid is the cardinality of the smallest circuit of the linear matroid. A circuit of a linear matroid is a minimally linearly dependent set of columns. A set $X$ of columns is minimally dependent if $X - \{x\}$ is linearly independent for every $x \in X$. I previously proposed a few algorithms and heuristics to find the cogirth or an upper bound on the cogirth of a linear matroid. Recall that circuits of a matroid, $M$, are cocircuits of the matroid’s dual, $M^*$. Therefore, one way to find the girth of a linear matroid is to find the matrix representation of the dual matroid [61] and use one of the previously proposed algorithms. A branch-and-cut algorithm for a set covering problem was previously proposed in Chapter 3 to find the cogirth of a linear matroid. However, the algorithm performed well for a specific class of matrices, those with bordered block diagonal form (BBDF). For general matrices, the algorithm is not a viable option. A similar set covering model
could be formulated to find the girth.

\[
\min 1^T x \\
\text{s.t. } Ax \geq 1
\]

The objective function remains the same, but the constraints require a simple change. In the constraint matrix \( A \) for the cogirth problem, \( A_{ij} = 1 \) if column \( j \) is in basis \( i \) and 0 otherwise. For the girth problem, \( A_{ij} = 1 \) if column \( j \) is in cobasis \( i \) and 0 otherwise. Since a cobasis of a matrix \( Z \) is simply the complement of a basis with respect to \( Z \), generating constraints for \( A \) is a simple matter. However, as can be seen in Table 3.2 in Chapter 3, the implicit set covering model can be quite troublesome to solve even when the structure of the matrix can be exploited. Something to point out is that the bordered block diagonal form (BBDF) would be advantageous as well when considering the girth of a linear matroid. If a matrix \( Z \) had the bordered block diagonal form (BBDF), Figure 2.1, an upper bound on the girth of the corresponding linear matroid could be found using the blocks of the matrix. A basis of a matrix plus another column not in the basis is a circuit. Therefore, given a block \( Z_i \) of \( Z \), a basis of \( Z_{Z_i} \) plus another column of \( Z_i \) is a circuit \( D \) of \( Z_{Z_i} \) which is also a circuit of \( Z \).

Recall that in order to find the girth of a linear matroid, we can consider the matrix from which the matroid was obtained. The circuits of the linear matroid correspond to minimal sets of linearly dependent columns. In order to find the circuits, one
could use brute force and test all possible sets of columns one at a time until the smallest circuit is found. However, this is not efficient since the number of possible sets is $n!$. Another approach is the circuit enumeration proposed by Boros et al. [14]. Recall that this algorithm was discussed in Chapter 2. This approach required enumeration of all the circuits in order to find the girth, and it involved finding a circuit of a matroid with a smaller ground set. The latter is the original problem on a smaller scale. Admittedly, it should be easier to find another distinct circuit using a brute force approach, but a brute force approach is exactly what is trying to be avoided. The other methods mentioned in Chapter 2 can also be used to find circuits since circuits of the a linear matroid are cocircuits of its dual matroid. However, the same obstacles as before arise. I propose a new approach to find the girth of a linear matroid.

4.2 Linear Matroid Girth Algorithm

The proposed method is based on a the Cunningham-Geelen Method proposed in [31] and implemented in [54]. The method is conceptually similar to the Cunningham-Geelen method in that it considers smaller subsystems to find partial solutions, and from those solutions, build full solutions.

The Cunningham-Geelen Method utilizes the branch decomposition of a given matrix and dynamic programming to find a solution. As described in [54], a branch-decomposition of a real $m \times n$ matrix $A$ is a pair $(T, \nu)$ where $T$ is a cubic tree and $\nu$
is a map from the columns of \( A \) denoted by \( E \) to the leaves of \( T \). The edges of \( T \) are weighted via a connectivity function \( \lambda_A \). Given an edge \( e \in E(T) \), \( T - e \) disconnects the tree into two connected components. Since the leaves of the tree correspond to column indices, disconnecting the tree \( T \) is equivalent to partitioning the matrix into two sets of columns, \( X \) and \( E - X \). Let \( A|X \) denote the submatrix of \( A \) containing only columns of \( X \), the connectivity function is defined as

\[
\lambda_A = \text{rank}(A|X) + \text{rank}(A|(E - X)) - \text{rank}(A) + 1.
\]

Note the similarity to

\[
dim(M \cap N) = dim(M) + dim(N) - dim(M + N)
\]

for some finite-dimensional subspaces \( M, N \) of \( \mathbb{R}^n \).

The connectivity function is symmetric, \( \lambda_A(X) = \lambda_A(E - X) \) and submodular, \( \lambda_A(X_1) + \lambda_A(X_2) \geq \lambda_A(X_1 \cap X_2) + \lambda_A(X_1 \cup X_2) \) for all \( X_1, X_2 \subseteq E \). The width of \( (T, \nu) \) is the maximum over all edge weights. Let \( BD(A) \) denote the set of all possible branch-decompositions of \( A \).

\[
\text{branch-width of } A = \min_{(T, \nu) \in BD(A)} \left\{ \max_{e \in E(T)} \{ \text{weight}(e) \} \right\}
\]

Take the following matrix \( Z \) as an example along with two distinct branch-decompositions.
The circuits of the matrix are \{\{1, 2, 3, 4\}, \{1, 2, 5\}, \{3, 4, 5\}\}

\[
Z = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0
\end{pmatrix}
\]

The branch-decompositions are optimal and near optimal with widths 2 and 3. Note that in each branch-decomposition, at least one of the smallest circuits of the linear matroid are neighbors in the tree. Also, the branch-width of \( Z \) is 2. A question arises from this example. Does an optimal branch-decomposition of the matrix provide an upper or lower bound on the girth of the corresponding linear matroid, and can the branch-decomposition itself be used in a heuristic to find a near optimal solution to the girth of a linear matroid? Note, it is possible for an edge \( e \in E(T) \) to have a weight that is less than the rank of the matrix \( A \). In finding optimal branch-decompositions, algorithms try to minimize the maximum \( \lambda_A \). In doing so, they produce submatrices \( A|X \) and \( A|(E - X) \) that have low ranks to try to find small \( \lambda_A \). The linear matroid circuit heuristic is a simple answer to the second question.
Algorithm 4.1 Linear Matroid Circuit Heuristic

1: upbd $= |A|$
2: for all $e \in E(T)$ do
3:     Find partition $X, (E - X)$ in $T - \{e\}$
4:     Compute $\text{rank}(A|X), \text{rank}(A|(E - X))$
5:     if $A|X$ is linearly dependent and $|X| < upbd$ then
6:         upbd $= |X|$
7:     end if
8:     if $A|(E - X)$ is linearly dependent and $|E - X| < upbd$ then
9:         upbd $= |E - X|$
10:    end if
11: end for

The algorithm steps through the tree an edge at a time, and looks at the ranks of the two submatrices created by partitioning the matrix. If either one of the matrices are rank deficient, then we know that it provides an upper bound if not optimal solution to the girth of the linear matroid since the submatrix is linearly dependent.

A variant of the first heuristic tries to perform a more thorough search of the branch decomposition recursively. As with the first heuristic, the second removes the edges one at a time looking for a tighter upper bound. If a tighter upper bound is found, then the heuristic is called to perform on the submatrix which provided the tighter upper bound. Table 4.1 compares the performance of the heuristic on matrices of various sizes against a few other well known algorithms briefly described below.
Algorithm 4.2 Linear Matroid Circuit Heuristic 2

1: \( \text{upbd} = |A| \)
2: \textbf{for all} \( e \in E(T) \) \textbf{do}
3: \hspace{1em} Find partition \( X, (E - X) \) in \( T - \{e\} \)
4: \hspace{1em} Compute \( \text{rank}(A|X), \text{rank}(A|(E - X)) \)
5: \hspace{1em} \textbf{if} \( A|X \) is linearly dependent and \( |X| < \text{upbd} \) \textbf{then}
6: \hspace{2em} \( \text{upbd} = |X| \)
7: \hspace{1em} \text{Find branch decomposition} \( (T, \nu)_{(A|X)} \)
8: \hspace{1em} \text{Call Linear matroid Circuit Heuristic 2 on} \ A - X
9: \hspace{1em} \textbf{end if}
10: \hspace{1em} \textbf{if} \( A|(E - X) \) is linearly dependent and \( |E - X| < \text{upbd} \) \textbf{then}
11: \hspace{2em} \( \text{upbd} = |E - X| \)
12: \hspace{1em} \text{Find branch decomposition} \( (T, \nu)_{(A|E-X)} \)
13: \hspace{1em} \text{Call Linear matroid Circuit Heuristic 2 on} \ A - (E-X)
14: \hspace{1em} \textbf{end if}
15: \textbf{end for}

Recall that finding the smallest circuit of matroid obtained from a matrix \( A \) is equivalent to

\[
\min ||x||_0 \\
\text{s.t.} \ Ax = 0 \\
x \neq 0,
\]

which is the optimization problem used to find the spark of a matrix. Since the \( \ell_1 \)-norm can be sparse promoting, \( ||x||_0 \) can be replaced with \( ||x||_1 \). A well known \( \ell_1 \) solver which was used to compare the heuristics is YALL1 [78]. YALL1 [77], [78] is a MATLAB [55] software package designed to solve the basis pursuit problem and its variants. In YALL1, an augmented Lagrangian of the basis pursuit variant is solved using alternating direction methods (ADM) that are derived in [77]. The
augmented Lagrangian is obtained from the original optimization problem in the following manner. Let \( f(x) : \mathbb{R}^m \rightarrow \mathbb{R} \) and \( g(y) : \mathbb{R}^n \rightarrow \mathbb{R} \) be convex functions, \( A \in \mathbb{R}^{p \times m} \), \( B \in \mathbb{R}^{p \times n} \) and \( b \in \mathbb{R}^p \). The structured optimization problem \( \min_{x,y} \{ f(x) + g(y) : Ax + By = b \} \) is considered, where variables \( x \) and \( y \) are separate in the objective, and coupled only in the constraint. The augmented Lagrangian function of this problem is given by

\[
L_A(x, y, \lambda) = f(x) + g(y) - \lambda^T (Ax + By - b) + \frac{\beta}{2} \|Ax + By - b\|^2
\]

The variables \( x \) and \( y \) are only coupled together in the constraint. The ADM takes advantage of this by iteratively solving the augmented Lagrangian in the following manner

\[
\begin{align*}
x^{k+1} & \leftarrow \arg \min_x L_A(x, y^k, \lambda^k), \\
y^{k+1} & \leftarrow \arg \min_y L_A(x^{k+1}, y, \lambda^k), \\
\lambda^{k+1} & \rightarrow \lambda^k - \gamma \beta (Ax_{k+1} + By_{k+1} - b)
\end{align*}
\]

where \( \gamma \in (0, 2) \) guarantees convergence.

A variant of the basis pursuit problem \( \{ \min \|x\|_1 : Ax = b \} \) is the basis pursuit denoising problem \( \{ \min \|x\|_1 : \|Ax - b\|_2 < \sigma, \sigma > 0 \} \). Lasso, \( \{ \min \|Ax - b\|_2 : \|x\|_1 < \tau, \tau > 0 \} \) (LS\( \tau \)), is another formulation that is similar to the basis pursuit problems. SPGL1 is an implementation of an algorithm that solves a sequence of (LS\( \tau \)) problems using a spectral projection-gradient (SPG) algorithm [72]. At each iteration, the algorithm searches the projected gradient path of the current gradient \( g_k \) of the function \( \|Ax_k - b\|_2 \) where the set of projected iterates is \( \{ x : \|x\|_1 < \tau \} \).
Another set of algorithms are the iterative reweighted algorithms presented in [24] which seek to address the 4.1 by first replacing it with a weighted least squares and then iteratively solving the reweighted problem. Results have been provided in [24], [23] to show that the iterative reweighting algorithms can produce sparse solutions under certain conditions for the matrix \( A \). The weighted least squares problem becomes

\[
\min_x \sum_{i=1}^{N} w_i x_i \\
\text{s.t. } Ax = 0 \\
x \neq 0
\]

Given this least squares problem, \( x \) is solved iteratively, \( x^{(k)} = Q_k A^T (AQ_k A^T)^{-1} b \) until the stopping criteria is met. The matrix \( Q_k \) is a diagonal matrix with entries \( \frac{1}{w_i} \) where \( w_i = ((x_i^{(k-1)})^2 + \epsilon)^{\frac{p}{2}-1} \) for some \( \epsilon > 0 \).

The test instances all contain one circuit whose cardinality is equal to the girth stated and several circuits whose cardinality is \( r + 1 \) where \( r \) is the rank of the matrix.

In all the cases, SPGL1 did not perform well in terms of precision. The rest of the algorithms performed similarly in terms of precision. The algorithms returned solutions with the number of nonzeros about \( r + 1 \) where \( r \) is the rank of the matrix. The solutions of YALL1 and IRLS_CS had a higher variance in the solutions, but did provide some solutions with upper bounds better than \( r + 1 \). Both algorithms are similar in run time. The first linear matroid circuit algorithm provided
<table>
<thead>
<tr>
<th>No.</th>
<th>Size</th>
<th>rank</th>
<th>girth</th>
<th>LMCH 1</th>
<th>Time</th>
<th>LMCH 2</th>
<th>Time</th>
<th>YALL1</th>
<th>Time</th>
<th>IRLS_CS</th>
<th>Time</th>
<th>SPGL1</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.1443</td>
<td>11</td>
<td>18.4021</td>
<td>10</td>
<td>2.0781</td>
<td>11</td>
<td>0.3664</td>
<td>30</td>
<td>0.2609</td>
</tr>
<tr>
<td>2</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.1166</td>
<td>11</td>
<td>20.8278</td>
<td>10</td>
<td>0.9367</td>
<td>11</td>
<td>0.0247</td>
<td>28</td>
<td>0.0994</td>
</tr>
<tr>
<td>3</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.1149</td>
<td>6</td>
<td>17.3931</td>
<td>11</td>
<td>1.9434</td>
<td>11</td>
<td>0.0228</td>
<td>19</td>
<td>0.0742</td>
</tr>
<tr>
<td>4</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.1195</td>
<td>6</td>
<td>21.8661</td>
<td>10</td>
<td>1.5618</td>
<td>13</td>
<td>0.0403</td>
<td>30</td>
<td>0.0761</td>
</tr>
<tr>
<td>5</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.1283</td>
<td>11</td>
<td>22.5851</td>
<td>10</td>
<td>1.9435</td>
<td>11</td>
<td>0.0195</td>
<td>30</td>
<td>0.0776</td>
</tr>
<tr>
<td>6</td>
<td>20 × 50</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>0.2483</td>
<td>21</td>
<td>72.8655</td>
<td>20</td>
<td>1.9802</td>
<td>21</td>
<td>0.0643</td>
<td>50</td>
<td>0.1361</td>
</tr>
<tr>
<td>7</td>
<td>20 × 50</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>0.2509</td>
<td>21</td>
<td>81.6949</td>
<td>18</td>
<td>1.5714</td>
<td>21</td>
<td>0.0307</td>
<td>50</td>
<td>0.1270</td>
</tr>
<tr>
<td>8</td>
<td>20 × 50</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>0.2492</td>
<td>21</td>
<td>81.8810</td>
<td>21</td>
<td>1.9888</td>
<td>26</td>
<td>0.0492</td>
<td>50</td>
<td>0.1380</td>
</tr>
<tr>
<td>9</td>
<td>20 × 50</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>0.2460</td>
<td>21</td>
<td>68.9922</td>
<td>28</td>
<td>1.9940</td>
<td>21</td>
<td>0.0300</td>
<td>48</td>
<td>0.1301</td>
</tr>
<tr>
<td>10</td>
<td>20 × 50</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>0.2510</td>
<td>21</td>
<td>76.9029</td>
<td>21</td>
<td>1.9830</td>
<td>21</td>
<td>0.0525</td>
<td>47</td>
<td>0.1287</td>
</tr>
<tr>
<td>11</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>50</td>
<td>0.8724</td>
<td>50</td>
<td>628.3333</td>
<td>48</td>
<td>2.5870</td>
<td>50</td>
<td>0.2352</td>
<td>100</td>
<td>0.4339</td>
</tr>
<tr>
<td>12</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>50</td>
<td>0.8630</td>
<td>50</td>
<td>589.5038</td>
<td>45</td>
<td>2.6161</td>
<td>51</td>
<td>0.2122</td>
<td>100</td>
<td>0.3776</td>
</tr>
<tr>
<td>13</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>50</td>
<td>0.8632</td>
<td>50</td>
<td>572.8569</td>
<td>51</td>
<td>2.6433</td>
<td>48</td>
<td>0.1544</td>
<td>100</td>
<td>0.4156</td>
</tr>
<tr>
<td>14</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>50</td>
<td>0.8678</td>
<td>50</td>
<td>571.0964</td>
<td>49</td>
<td>2.6176</td>
<td>56</td>
<td>0.1363</td>
<td>100</td>
<td>0.3948</td>
</tr>
<tr>
<td>15</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>50</td>
<td>0.8567</td>
<td>50</td>
<td>623.9043</td>
<td>49</td>
<td>2.5225</td>
<td>48</td>
<td>0.1417</td>
<td>100</td>
<td>0.4199</td>
</tr>
<tr>
<td>16</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>100</td>
<td>3.5571</td>
<td>100</td>
<td>4947.9500</td>
<td>113</td>
<td>3.0251</td>
<td>103</td>
<td>0.5857</td>
<td>200</td>
<td>0.8932</td>
</tr>
<tr>
<td>17</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>100</td>
<td>3.5876</td>
<td>100</td>
<td>5299.1410</td>
<td>102</td>
<td>3.9567</td>
<td>99</td>
<td>0.4460</td>
<td>200</td>
<td>0.8294</td>
</tr>
<tr>
<td>18</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>100</td>
<td>3.6433</td>
<td>100</td>
<td>4309.7830</td>
<td>112</td>
<td>2.9587</td>
<td>104</td>
<td>0.5203</td>
<td>200</td>
<td>0.8827</td>
</tr>
<tr>
<td>19</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>100</td>
<td>3.5260</td>
<td>100</td>
<td>4932.8470</td>
<td>101</td>
<td>2.9338</td>
<td>102</td>
<td>0.8210</td>
<td>199</td>
<td>0.7877</td>
</tr>
<tr>
<td>20</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>100</td>
<td>3.5937</td>
<td>100</td>
<td>4989.3630</td>
<td>104</td>
<td>2.9383</td>
<td>107</td>
<td>0.5202</td>
<td>199</td>
<td>0.7908</td>
</tr>
</tbody>
</table>

Table 4.1: Linear Matroid Circuit Heuristic
solutions with optimal value \( r \) and \( r + 1 \) and had run times similar to YALL1 and IRLS_CS. The second linear matroid circuit heuristic performed similarly with the exception of test instances 3 and 4 where it was able to find the optimal solution. However, since it performed a more thorough search of the branch decomposition, it did require longer run times. The linear matroid circuit heuristics provide an insight into the importance of the structure of the branch decomposition. The structure of the branch decomposition has an effect on the efficiency and precision of the linear matroid circuit heuristics. If the smallest circuit is grouped together in the branch decomposition, there is a better chance that the heuristics will return optimal solutions. The optimality of the branch decomposition is not necessarily as important as the structure of the tree. Since YALL1 and IRLS_CS are able to find solutions better than \( r + 1 \) in some test instances, these algorithms should be considered when trying to find the girth of a linear matroid.

Referring back to the first question, can the branch-decomposition provide an upper or lower bound of the girth of a linear matroid? It is clear that the girth is bounded above by \( r + 1 \) where \( r \) is the rank of the matrix. This is because the largest circuits have size \( r + 1 \) since a basis \( B \) plus any other column is a minimal linearly dependent set of columns. From the example matrix \( Z \) we see that the girth is not bounded above by the branch-width since the girth is 3 and the branch-width is 2. One might also wonder if the branch-width + 1 is an upper bound. However, consider a \( 3 \times 4 \) matrix consisting of the identity matrix along with a column of ones.
The branch-width of the matrix is 2, but the girth is 4. What about a lower bound? Consider the following example. Let $Z$ be a $4 \times 8$ real matrix such that any 4 columns are independent. Then the matrix has full row rank and any 5 columns are linearly dependent. Let $y = Z_1 + Z_2$, and $Z = [Z, y]$. Now $Z$ has a circuit of size 3. We now have to consider the different partition combinations to figure out the values of $\lambda_A(X)$.

- If $|X| = 1$, then $\text{rank}(A|X) = 1$, $\text{rank}(A|(E - X)) = 4$ and $\lambda_A(X) = 2$

- If $|X| = 2$, then $\text{rank}(A|X) = 2$, $\text{rank}(A|(E - X)) = 4$ and $\lambda_A(X) = 3$

- If $X = [Z_1, Z_2, y]$, then $\text{rank}(A|X) = 2$, $\text{rank}(A|(E - X)) = 4$ and $\lambda_A(X) = 3$; else if $|X| = 3$, then $\text{rank}(A|X) = 3$, $\text{rank}(A|(E - X)) = 4$ and $\lambda_A(X) = 4$

- If $[Z_1, Z_2, y] \subset X$ and $|X| = 4$, then $\text{rank}(A|X) = 3$, $\text{rank}(A|(E - X)) = 4$ and $\text{rank}(A|(E - X)) = 4$; else $\text{rank}(A|X) = 4$, $\text{rank}(A|(E - X)) = 4$ and $\lambda_A(X) = 5$

- If $|X| = 5$, $\text{rank}(A|X) = 4$, $\text{rank}(A|(E - X)) = 4$ and $\lambda_A(X) = 5$

We see that for this example, $\lambda_A(X) \geq 4$ for $|X| \geq 4$. Therefore, the branch-width is 5 and since the girth is 3, the branch-width does not provide a lower bound on the girth. Although the branch-decomposition cannot be used directly to find the girth of a linear matroid or be used to find a lower bound on the girth, it can still be used to find the girth of a linear matroid.

As stated earlier, in trying to find optimal branch-widths, the branch-decomposition is indirectly trying to partition the matrix into submatrices that are rank deficient,
and thus producing smaller values of $\lambda_A(X)$. The proposed algorithm does not look at the matrix in its entirety. Instead it looks at the these submatrices along with the subsets to see if circuits can be found. As an example, refer back to one of the previous branch-decompositions. The tree is traversed in post depth-first search order, $\{5, 1, 2, 8, 7, 3, 4, 6, 9\}$.

A root node is added to the tree if necessary, and then the nodes are traversed in order. The following outlines how the post depth-first order is used and what it means to traverse a node.

**Algorithm 4.3 Linear Matroid Circuit Algorithm**

1: for all $v \in V(T)$ in post depth first search order do
2: \hspace{1em} if $v$ is a leaf node then
3: \hspace{2em} process leaf node
4: \hspace{1em} end if
5: \hspace{1em} if $v$ is an internal node then
6: \hspace{2em} process internal node
7: \hspace{1em} end if
8: \hspace{1em} if $v$ is the root node then
9: \hspace{2em} process root node
10: \hspace{1em} end if
11: end for

Before I discuss what happens at each node, there are some ideas that need to be introduced.
Lemma 4.1 Let $X$, $Y$ be a partition of the column indices of an $m \times n$ matrix $A$ ($m \leq n$). Let $A_W$ denote a submatrix of $A$ where $W$ is a subset of columns of $A$. Let $\hat{X} \subseteq X$ that is linearly independent. Let $B_y$ be a basis for the column space of $Y$. If $A_{B_y \cup \hat{X}}$ has full column rank, then there is no $\hat{Y} \subseteq Y$ and $\bar{X} \subseteq \hat{X}$ such that $\bar{X} \cup \hat{Y}$ is a circuit of $A$.

Proof Let $V$ be the subspace spanned by $A_{\hat{X}}$ with dimension $|\hat{X}|$ since $\hat{X}$ is not a circuit. Let $W$ be the subspace spanned by $B_y$ with dimension $|B_y|$ since $B_y$ is a basis of $A_Y$. If $A_{B_y \cup \hat{X}}$ has full column rank, then the dimension of the span of $V + W$ is $|\hat{X}| + |B_y|$. Since $V$ and $W$ are finite dimensional vector spaces, $\dim(V \cap W) = \dim(V) + \dim(W) - \dim(V + W)$. Therefore, $\dim(V \cap W) = |\hat{X}| + |B_y| - (|\hat{X}| + |B_y|) = 0$. This means that that $\text{span}(V) \cap \text{span}(W) = \{0\}$.

Suppose there is some $\hat{Y} \subseteq Y$ such that $\hat{X} \cup \hat{Y}$ is a circuit. Then $\rank(A_{\hat{Y} \cup \hat{X}}) < |\hat{X}| + |\hat{Y}|$. Let $Z$ be the subspace spanned by $A_{\hat{Y}}$. Then $\dim(V + Z) = t < |\hat{X}| + |\hat{Y}|$. Therefore, $\dim(V \cap Z) = |\hat{X}| + |\hat{Y}| - t \geq 1$. This means that $\text{span}(V) \cap \text{span}(Z)$ contains some non-zero vectors. However, $\text{span}(Z) \subseteq \text{span}(W)$ which means $\text{span}(V) \cap \text{span}(W)$ contains some non-zero vectors (contradiction) $\square$.

Lemma 4.2 Let $X$, $Y$ be a partition of the columns indices of an $m \times n$ matrix $A$ ($m \leq n$). Let $A_V$ denote a submatrix of $A$ where $V$ is the subset of columns of $A$. Let $\hat{X} \subseteq X$ that is linearly independent. Let $W = \text{span}(A_X) \cap \text{span}(A_Y)$ and $B_W$ be some matrix that spans $W$. If $[B_W A_{\hat{X}}]$ for some $\hat{X} \subseteq X$ has full column rank, then there is no $\hat{Y} \subseteq Y$ and $\bar{X} \subseteq \hat{X}$ such that $\bar{X} \cup \hat{Y}$ is a circuit of $A$. 
Proof Note that since $W = \text{span}(A_X) \cap \text{span}(A_Y)$, there is some $\hat{Y} \subseteq Y$ such that $A_{\hat{Y}}$ is a basis for $W$, that is $\text{span}(B_W) = \text{span}(A_{\hat{Y}})$. Also, we do not need to worry about $\bar{Y} \subseteq Y$ such that $\text{span}(A_{\bar{Y}}) \cap W = \{0\}$.

If $[B_W \ A_{\hat{X}}]$ has full column rank, then $W \cap \text{span}(A_{\hat{X}}) = \{0\}$. This implies $[A_{\hat{Y}} \ A_{\hat{X}}]$ has full column rank since $\text{span}(B_W) = \text{span}(A_{\hat{Y}})$. From Lemma 4.1, $\hat{X} \cup \hat{Y}$ does not contain a circuit. Also, $\hat{X} \cup \hat{Y}$ does not contain a circuit for any $\hat{Y} \subset Y$ such that $\text{span}(A_{\hat{Y}}) = \text{span}(A_{\hat{Y}})$.

Concerning Lemma 4.1, we consider a matrix $A$ and a given partition $[X, Y]$ of $A$. If $C$ is a circuit of $A$, then $C$ is either entirely contained in $X$, contained in $Y$ or is split between the partitions. Lemma 4.1 provides a way to test if a subset $\hat{X}$ of $X$ that is not linearly dependent is part of circuit that is contained in $A$ and split between $X$ and $Y$ by using a basis of $Y$. Lemma 4.2 states that the same test can be performed using a basis of the intersection of $\text{span}(X)$ and $\text{span}(Y)$. Observe that $\lambda_A(X) = \dim(\text{span}(X) \cap \text{span}(Y)) + 1$. The $\dim(\text{span}(X) \cap \text{span}(Y))$ is smaller than $\dim(\text{span}(Y))$ unless $\text{span}(X) \subseteq \text{span}(Y)$. Therefore, Lemma 4.2 states that a smaller matrix can be used to test if an independent set of columns $\hat{X}$ is part of a circuit. All that needs to be done is to find a matrix representation of the intersection of $\text{span}(X)$ and $\text{span}(Y)$. I follow the CG intersection method described in [31]. Now that we can determine if an independent set is part of a circuit, we can see how the nodes are processed.

As earlier stated, the algorithm works by combining partial solutions to create full
solutions. At the internal nodes, all the sets of the left child are combined with sets from the right child to create new sets to be tested for linear dependence. Recall that at any node in the tree, the leaves that are descendents of that node indicate which submatrix is being considered at the moment. Before I describe what happens at internal nodes in further detail, I will talk about the leaf nodes. Recall that each leaf node corresponds to a column of the matrix. Therefore, at the leaf nodes, a check is done to see if the single column is in the span of the rest of the matrix. If the column is in the span, then it is part of a circuit in the rest of the matrix and any subset of $A$ that contains it needs to be considered as a possible linearly dependent set. If not, then any possible combination with it can be discarded. Therefore, the column itself can be discarded since passing it along to an internal node would create combinations that contain it. Doing this for each column will help determine if it is needed, and possibly decrease the overall number of sets considered for dependence. The internal nodes work in a similar fashion.

At the internal nodes, combinations of sets from left and right children are created and tested for dependence. If a set is dependent, then it is an upper bound on the smallest circuit. If not, then the set is independent, and it can be tested to see if it is part of a dependent set. Lemma 4.2 is used to determine if independent sets should be passed up to the parent node. Observe that only sets smaller than the current upper bound, $|C|$, need to be tested for dependence. Also, only independent sets with cardinality less than $|C| - 1$. Any larger set will create a new set larger than
$|C|$ when combined at the parent node. At the root node, every possible combination that can be created from the children is checked if it is less than the current upper bound. Once the root node has been traversed, the smallest dependent set will have been found. This is because at each node in the tree, the only subsets in a submatrix that are discarded are independent sets that are not part of a dependent set.

**Algorithm 4.4** Processing the leaves

1. **for** $i = 1 : n$ **do**
2. Get intersection $I_i = SubspaceIntersection(\hat{A}_i, \hat{A} - \hat{A}_i)$
3. **if** $[\hat{A}_i I_i]$ is linearly dependent **then**
4. $A_i$ is part of some circuit, store $\{A_i, I_i\}$
5. **else**
6. discard
7. **end if**
8. **end for**

**Algorithm 4.5** Processing the internal node $p$:

1. $C$ is a linearly dependent set that has been previously obtained
2. **for** $\ell \in L$ **do**
3. **for** $r \in R$ **do**
4. $s = \ell \cup r$
5. **if** $|s| < |C|$ **then**
6. **if** $A_s$ is linearly dependent **then**
7. $C \leftarrow s$
8. **else**
9. **if** $|s| \geq |C| - 1$ **then**
10. discard it
11. **end if**
12. **if** $\text{span}(A_s) \cap I_p = \{0\}$ **then**
13. discard it
14. **end if**
15. **end if**
16. **else**
17. Discard $s$ since $|s| \geq |C|$
18. **end if**
19. **end for**
20. **end for**
Algorithm 4.6 Processing the root
\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{for} $\ell \in L$ \textbf{do}
\State \textbf{for} $r \in R$ \textbf{do}
\State \hspace{1em} $s = \ell \cup r$
\State \hspace{1em} \textbf{if} $|s| < |C|$ \textbf{then}
\State \hspace{2em} \textbf{if} $A_s$ is a linearly dependent \textbf{then}
\State \hspace{3em} $C \leftarrow s$
\State \hspace{1em} \textbf{end if}
\State \textbf{end if}
\State \textbf{end for}
\State \textbf{end for}
\end{algorithmic}
\end{algorithm}

Although the total number of sets considered can be decreased as lower upper bounds are found, is it possible to discard more sets? Let $X, Y$ be a partition of $A$. Also, let $\hat{X} \subseteq \bar{X} \subseteq X$ be independent sets such that $\hat{X} \cup Y$, and $\bar{X} \cup Y$ contain circuits. Is it possible to keep $\hat{X}$ or $\bar{X}$ and discard the other. It could be said that $\hat{X}$ dominates $\bar{X}$ since they are both part of some circuit and $|\hat{X}| \leq |\bar{X}|$. Suppose $\hat{X} \cup \hat{Y}$ contains a circuit and $\hat{X} \cup \hat{Y}$ is the circuit, then $\hat{X}$ cannot be discarded. Although $|\hat{X}| \leq |\bar{X}|$ it is possible that there are $\hat{Y}, \bar{Y} \subseteq Y$ such that $|\hat{X} \cup \hat{Y}| > |\bar{X} \cup \bar{Y}|$. Therefore, this definition of domination does not work. However, what if the span of an independent set is considered. That is, if $\hat{X}, \bar{X} \subseteq X$ are independent sets and $span(\hat{X}) = span(\bar{X})$, then both sets are not needed. This leads to couple of lemmas.

Lemma 4.3 Let $X_1$ and $X_2$ be distinct sets of independent columns of an $m \times n$ real matrix $A$, and $|X_1| < |X_2|$. Then $span(X_1) \neq span(X_2)$.

Proof Suppose $span(X_1) = span(X_2)$. Then since $X_1$ is an independent set of columns, $\text{dim}(span(X_1)) = \text{rank}(A_{X_1}) = |X_1|$. Since $span(X_1) = span(X_2)$, $\text{rank}(A_{X_2}) = |X_2|$. Since $|X_1| < |X_2|$, we have $\text{rank}(A_{X_1}) < \text{rank}(A_{X_2})$. Therefore, $span(X_1) \neq span(X_2)$. 

\[
dim(\text{span}(X_2)) = \dim(\text{span}(X_1)) = \rank(A_{X_1}) = |X_2|. \text{ Therefore, } \rank(A_{X_2}) = |X_1| < |X_2|. \text{ Therefore, } X_2 \text{ is dependent (contradiction)} \square.
\]

**Lemma 4.4** Let \( X_1 \) and \( X_2 \) be distinct sets of independent columns of an \( m \times n \) real matrix \( A \), and \( |X_1| = |X_2| \). If \( \rank(A_{X_1} \cup X_2) = \rank(A_{X_1}) \), then the \( \text{span}(X_1) = \text{span}(X_2) \).

**Proof** Since \( |X_1| = |X_2| \) and they are both independent, \( \rank(A_{X_1}) = \rank(A_{X_2}) \).

Also

\[
\rank(A_{X_1}) = \dim(\text{span}(X_1))
\]

\[
\rank(A_{X_1} \cup X_2) = \dim(\text{span}(X_1) \cup \text{span}(X_2)) \text{ and }
\]

\[
\dim(\text{span}(X_1) \cup \text{span}(X_2)) + \dim(\text{span}(X_1)) \cap \text{span}(X_2)) = \dim(\text{span}(X_1)) + \dim(\text{span}(X_2)).
\]

Therefore,

\[
\dim(\text{span}(X_1) \cap \text{span}(X_2)) = \dim(\text{span}(X_2)) = \dim(\text{span}(X_1)).
\]

This means that the subspace intersection of \( X_1 \) and \( X_2 \) has the same dimension as \( X_1 \) and \( X_2 \). That means the \( \text{span}(X_1) = \text{span}(X_2) \square \).

Lemma 4.3 means that independent sets only need to be compared to each other if they are the same size. Lemma 4.4 shows a simple way to test if two independent sets have the same span. Therefore, it can be said that a set of columns \( X_1 \) dominates
\( X_2 \) or vice versa if \( \text{span}(X_1) = \text{span}(X_2) \). Even though this sense of domination can reduce the total number of possible solutions, and thus reduce memory storage, quite a bit of storage is still required.

Is there a way to use the branch-decomposition and dynamic programming without using so much memory? The next section discusses this issue.

### 4.3 Linear Matroid Circuit Algorithm Variation

Recall that the branch decomposition is used to decide which submatrix is currently being tested. The order was decided using a post-order depth-first search. The post-order depth-first search for the following example is \( \{5, 1, 2, 8, 7, 3, 4, 6, 9\} \).

![Tree diagram](image)

By looking at the tree on the right with the root node, we see that on either side of the root node, submatrices are created by adding columns one at a time. On the left side of the tree, submatrices can be built as \([1], [1, 2], \) and \([1, 2, 5]\). The submatrices on the right side would be produced in the same manner. These submatrices are built at the internal nodes of the linear matroid circuit algorithm. Possible solutions are created by combining sets that were previously identified as being part of some...
solution. Although some sets are discarded, the majority must be stored. However, it is possible to create the partial solutions while storing the discarded sets instead of the partial solutions. In general, the number of discarded sets will be less than the number of possible solutions. Therefore, amount of storage will be decreased. In the circuit algorithm, possible solutions are created in order of size $k = 1 : w$ where $w$ is the number of columns in the submatrix. The advantage of storing possible solutions is that solutions are not tested more than once. For example, the set $[1, 2]$ is tested in the submatrix $[1, 2]$, but not again in the submatrix $[1, 2, 5]$. This advantage can be recreated in a variation of the circuit algorithm. Observe that single columns do not need to be tested since it is assumed that there are no zero columns. For $[1, 2]$, only $\{1, 2\}$ needs to be tested. To create this subset, start with $[1]$, then add $\{2\}$ to get $[1, 2]$. For $[1, 2, 5]$, $\{1, 5\}$, $\{2, 5\}$, and $\{1, 2, 5\}$ need to be tested. In order to test these sets, the sets $\{1\}, \{2\}, \{1, 2\}$ would be created, and then $\{5\}$ would be added to get $\{1, 5\}$, $\{2, 5\}$, and $\{1, 2, 5\}$. For a submatrix $Z$ with $w$ columns, create the subsets of size $k = 1 : \max((ub - 1), (w - 1))$ in $[Z_1 \cdots Z_{w-1}]$ where $ub$ is an upper bound on the girth, and then add column $Z_w$. The subsets of size $k$ can be generated using the following algorithm found in [63].
Algorithm 4.7 Next k subset

1: Given a subset $S$ of size $k$, $1 \leq k \leq w$
2: if $k == w$ then
3: return
4: end if
5: $q = -1$
6: for $p = (k - 1) : 0$ do
7: if $S(p) < w - (k - 1) + p$ then
8: $q \leftarrow p$
9: break
10: end if
11: end for
12: if $q \neq -1$ then
13: $S(q) \leftarrow S(q) + 1$
14: if $q < (k - 1)$ then
15: for $p = (q + 1) : (k - 1)$ do
16: $S(q) \leftarrow S(q - 1) + 1$
17: end for
18: end if
19: end if
20: return $S$

This variation on subset creation replaces the previous method of creating possible solutions from stored partial solutions. Subsets are still discarded based on the same concepts previously discussed. In fact, the discarded subsets are stored instead of possible solutions. The created subsets are checked against discarded sets to see if they need to be tested. The reason for this will become apparent momentarily.

Before, the submatrix $[1, 2, 5]$ was considered for possible solutions, $\{1\}, \{2\}, \{1, 2\}$, and $\{5\}$ needed to be stored in order to create all the solutions. With the new method of subset creation, only $\{1, 2, 5\}$ needs to be stored. The difference in storage increases as the size of the submatrix increases. However, if $\{1, 2\}$ was discarded in
the original method of creating subsets, then any subsequent subset that would have contained \{1, 2\} would not be created. This can be mimicked in the new method of subset creation. If a newly created subset \( S \) contains \{1, 2\}, then it does not need to be checked, and it can be discarded. In general, if \( d \subseteq S \) for a discarded set \( d \) and subset \( S \), then \( S \) can be discarded. However, \( S \) does not need to be stored as a discarded set. Since \( d \subseteq S \), any subset that would contain \( S \), would also contain \( d \). Therefore, only a minimal number of discarded sets need to be stored.

Below, in Table 4.2, are the results of the linear matroid circuit algorithm using the new method of subset creation. The algorithm was tested on the same set of test instances as the linear matroid circuit heuristic. The limitations of the algorithm appear for test instances as small as \( 20 \times 50 \). Therefore, larger test instances were not included in the table. The algorithm is able to find optimal solutions for many of the instances. However, even for matrices that are not relatively large, the performance of the algorithm begins to waiver. There could be several reasons for this. First note that the value of the girth of the linear matroid, where the circuits are located in the branch-decomposition, and the rank of the matrix have an affect on the algorithm. If a near optimal circuit is found early, or the rank of the matrix is small, then the number of total subsets checked is decreased since the current upper bound on the girth is lower. Recall that the girth is bounded above by \( r + 1 \) where \( r \) is the rank of the matrix. Something else to consider is where the smallest circuit is located on the branch-decomposition. If the branch-decomposition is a caterpillar tree, and
the smallest circuit is split between the left and right children of the root node of the tree, then the girth will not be found until the algorithm reached the root node (looks at the entire matrix). The branch-decompositions of all the test instances were caterpillar trees. In all the test instances, the smallest circuit was not found until the algorithm reached the root node. This is because the smallest circuit was divided between the left and right child of the root node. This along with possible solutions will be discussed further in Chapter 5.

<table>
<thead>
<tr>
<th>No.</th>
<th>Size</th>
<th>rank</th>
<th>girth</th>
<th>comp. girth</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$3 \times 5$</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>≈ 0</td>
</tr>
<tr>
<td>2</td>
<td>$5 \times 10$</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>≈ 0</td>
</tr>
<tr>
<td>3</td>
<td>$5 \times 15$</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>≈ 0</td>
</tr>
<tr>
<td>4</td>
<td>$5 \times 20$</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>≈ 0</td>
</tr>
<tr>
<td>5</td>
<td>$6 \times 20$</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>≈ 0</td>
</tr>
<tr>
<td>6</td>
<td>$7 \times 11$</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>≈ 0</td>
</tr>
<tr>
<td>7</td>
<td>$7 \times 20$</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>$9 \times 15$</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>≈ 0</td>
</tr>
<tr>
<td>9</td>
<td>$9 \times 20$</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>$10 \times 30$</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>$10 \times 30$</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>$10 \times 30$</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>75</td>
</tr>
<tr>
<td>13</td>
<td>$10 \times 30$</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>$10 \times 30$</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>83</td>
</tr>
<tr>
<td>15</td>
<td>$20 \times 50$</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>&gt; 18000</td>
</tr>
<tr>
<td>16</td>
<td>$20 \times 50$</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>&gt; 18000</td>
</tr>
<tr>
<td>17</td>
<td>$20 \times 50$</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>&gt; 18000</td>
</tr>
<tr>
<td>18</td>
<td>$20 \times 50$</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>&gt; 18000</td>
</tr>
<tr>
<td>19</td>
<td>$20 \times 50$</td>
<td>20</td>
<td>11</td>
<td>21</td>
<td>&gt; 18000</td>
</tr>
</tbody>
</table>

Table 4.2 : Linear Matroid Circuit Algorithm
4.4 Genetic Algorithm

Another algorithm that was discussed in Chapter 2 was a genetic algorithm. The genetic algorithm presented below follows the overall format of the one developed by Beasley and Chu [11] to solve the set covering problem. Since finding the girth of a matroid is equivalent to solving a set covering problem, it is easy to see that a genetic algorithm can be considered to find circuits of a matroid without creating the set covering model. This is advantageous since in this case, all the constraints are not initially known. The concept behind the genetic algorithm is survival of the fittest for the solutions. The ideal setting is that the algorithm will run through iterations, the fittest (best) solutions will rise, and an optimal or near optimal solution will be found. Genetic algorithms are not guaranteed to solve problems to optimality since they do not usually look at the entire population of solutions. An initial population of solutions for the problem is generated by some means. Using this initial population, new generations are created by mating parents (solutions) with each other and replacing solutions that are less fit than others. These solutions are created until a specified number of new solutions have been created. A solution is new if it is not in the current set of solutions. Therefore, it is possible to generate the same solution more than once. A general outline of the algorithm and details are described below. Note a difference between the algorithm described below and that described in [11] is that there is not a column redundancy removal step. A solution $C$ that has redundant columns is a linearly dependent set that is not a circuit, i.e. the set is not minimally
dependent. Therefore, finding the redundant columns is similar to finding a circuit that is contained by $C$. In fact, a circuit is a set of columns without any redundant columns. It would not be efficient to find the smallest circuit of a child $C$ for every child created. Therefore, that step is left out of the genetic algorithm. An outline of the genetic algorithm is provided below. Computational results are provided below as well in Table 4.3.

**Algorithm 4.8 Genetic Algorithm**

1. Generate initial population of solutions, $t \leftarrow 0$
2. Select two solutions $P_1$ and $P_2$ from the population using binary tournament selection
3. Combine $P_1$ and $P_2$ using the fusion crossover operator to form a child $C$
4. Mutate $C$
5. Make $C$ feasible if necessary
6. If $C$ is identical to a solution in the population, produce a new $C$, else $t \leftarrow t + 1$
7. Replace a randomly selected solution with an above-average fitness level with $C$
8. Repeat until $t = M$, where $M$ is specified by the user

### 4.4.1 Generating an Initial Population

A greedy algorithm is used to create the initial population of solutions. The algorithm chooses columns of the matrix based on the weights, $w_{A_i}$, of each of the columns of a matrix $\hat{A}$. This is done to create a diverse population of solutions. Recall that a circuit has a nonempty intersection with every cobasis. Although the set of cobases is not known to find an initial population, it is possible to generate feasible solutions. At the very least, a basis $B$ of a matrix $\hat{A}$ along with an extra column is a circuit of $\hat{A}$. Therefore, for each column in the matrix, a circuit can be found which contains it. To find a solution for the initial population, the algorithm
adds columns to an independent set of columns until the set becomes dependent. In particular, the algorithm searches for a basis, then adds an extra column to create a circuit. If a dependent set is found before a basis is found for a particular column, then this dependent set is kept as a solution. Even though it might not be a circuit, it provides an upper bound on the girth, the cardinality of the smallest circuit. After a solution $C$ is found, $w_{A_i} = w_{A_i} + 1$ for each column $A_i \in C$. Solutions are created based on minimum weights in order to ensure diversity of the population.

**Algorithm 4.9 Initial Population**

1: Given an $m \times n$ matrix $\hat{A}$ and empty set Circuits
2: for each column $A_k$ of $\hat{A}$ do
3: $sc_k = 0$
4: end for
5: for each column $A_k$ of $\hat{A}$ do
6: $C \leftarrow \{k\}$
7: rankbasis, ranktempbasis = 1
8: $sc_k = sc_k + 1$
9: index = sort($sc$) in ascending order
10: for $j = 1 : n$ do
11: if $sc_j \neq k$ then
12: $C \leftarrow C \cup index(j)$
13: rankbasis = rank($A_C$);
14: if rankbasis == ranktempbasis then
15: if rankbasis == rank($A$) then
16: Circuits $\leftarrow$ Circuits $\cup$ $C$
17: break
18: end if
19: else
20: ranktempbasis = rankbasis
21: end if
22: end if
23: end for
24: end for
4.4.2 Producing Children

The fusion operator is a crossover technique used to create child solutions from parents. A few other popular crossover techniques such as the one-point and two-point crossover techniques are discussed in [11]. However, there are many variations of crossovers. For a child $C$, the fusion operator includes a column $k$ in the solution if $k$ is contained in both parents. Otherwise, the column is included using a certain probability if it included in one of the parents.

**Algorithm 4.10 Fusion Operator**

1: $p = \frac{\text{fitness}_{P_2}}{\text{fitness}_{P_1} + \text{fitness}_{P_2}}$
2: $q = 1 - p$
3: for $k = 1 : \text{length}(P_1)$ do
4: if $P_1(k) == P_2(k)$ then
5: $C(k) \leftarrow P_1(k)$
6: else
7: $\text{temp} \leftarrow \text{random number between 0 and 1}$
8: if $\text{temp} < p$ then
9: $C(k) \leftarrow P_1(k)$
10: else
11: $C(k) \leftarrow P_2(k)$
12: end if
13: end if
14: end for

4.4.3 Mutation

After a child solution $C$ is created it is mutated. Algorithmically, the mutation flips the zeros to ones and the ones to zeros for a randomly selected number of columns. As the algorithm proceeds, the population may converge. The mutation is a means to prevent the algorithm from only searching for new solutions locally. Two different
methods of mutation are considered. The first is a bit-flip mutation. In the bit-flip mutation, a uniform random number $f_i$ in $[0, 1]$ is selected for each column in a circuit $C$. If $p \geq f_i$ for some probability $p$, then the value of $C(j)$ is flipped. In this case, $p = \frac{1}{\text{length}(C)}$.

**Algorithm 4.11** Bit-Flip Mutation

1: Given a solution $C$
2: $p \leftarrow 1/\text{length}(C)$
3: for $j=1:\text{length}(C)$ do
4:   $f_i$ is random number chosen uniformly from 0 to 1
5:   if $p \geq f_i$ then
6:      $C(j) \leftarrow 1 - C(j)$
7:   end if
8: end for

The second method chooses $k$ randomly chosen columns. The value $k$ is specified using the formula presented by Beasley and Chu [11],

$$
k = \left\lceil \frac{m_f}{1 + \exp\left(-4m_g(t - m_c)/m_f\right)} \right\rceil,
$$

where $t$ is the number of child solutions that have been generated, $m_f$ is the final stable mutation rate, $m_c$ specifies the number of child solutions generated at which a mutation rate of $m_f/2$ is reached and $m_g$ specifies the gradient at $t = m_c$. The values for $m_f$, $m_c$, and $m_g$ are those found by Beasley to work well, 10, 200, 2.0 respectively.
4.4.4 Children Feasibility

Once a child is produced, it is not guaranteed that the child will be a solution.
It is simple enough to check if the child is linearly dependent. Although it cannot be
guaranteed that the child is a circuit, ideally the child will be a better solution than
either of its parents. If the child is not a solution, then columns must be added to
it in order to make it so. Columns are added to the child in a greedy fashion. The
more solutions a column \( A_i \) is a member of, the more likely it is to be added to the
child. Below are some computational results of the genetic algorithm.

\begin{algorithm}
\caption{Dependence Check}
\begin{algorithmic}[1]
\STATE Given an \( m \times n \) matrix \( \hat{A} \), Circuits and a child solution \( C \)
\IF {\text{\texttt{rank}}(\hat{A}C) > |AC|} \text{\texttt{then}}
\STATE \( C \) is linearly dependent, return \( C \)
\ENDIF
\FOR {\( j = 1 : n \)}
\FOR {\( D \in \text{Circuits} \)}
\IF {column \( j \) is a member of \( D \) \text{\texttt{then}}}
\STATE \( sc_j = sc_j + 1 \)
\ENDIF
\ENDFOR
\ENDFOR
\STATE index = sort(sc) in descending order
\FOR {\( j = 1 : n \)}
\IF {index(\( j \)) is not in \( C \)} \text{\texttt{then}}
\STATE \( C \leftarrow C \cup \text{index}(\( j \)) \)
\ENDIF
\IF {\text{\texttt{rank}}(AC) < |AC|} \text{\texttt{then}}
\STATE return \( C \)
\ENDIF
\ENDFOR
\end{algorithmic}
\end{algorithm}
4.4.5 Computational Results

The algorithm was run on the same computer as before, but was implemented in MATLAB [55]. For each instance the algorithm was run 10 times. The computed girths and times are an average. The first time corresponds to the algorithm run with bit-flip mutation while the second time corresponds to the algorithm run with the mutation presented by Beasley and Chu [11]. The algorithm stopped when either \( t = \max(500, n) \) where \( t \) is the number of solutions and \( n \) is the number of columns of the matrix, or the number of iterations hit 1000 for \( n < 100 \) and 2000 otherwise.

The genetic algorithm presents similar results to the linear matroid circuit heuristic in that the computed girth is the rank of the matrix plus one. The algorithm was never able to generate a circuit with a smaller cardinality than the best circuit from the initial population. Because all the cobases of the matrix are not initially known, it is difficult to generate a good initial population. Concluding remarks and further research are discussed in Chapter 5.
<table>
<thead>
<tr>
<th>No.</th>
<th>Size</th>
<th>rank</th>
<th>girth</th>
<th>comp. girth</th>
<th>Time 1 (secs)</th>
<th>Time 2 (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.3042</td>
<td>0.3028</td>
</tr>
<tr>
<td>2</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.2729</td>
<td>0.2662</td>
</tr>
<tr>
<td>3</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.2719</td>
<td>0.2763</td>
</tr>
<tr>
<td>4</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.2802</td>
<td>0.2739</td>
</tr>
<tr>
<td>5</td>
<td>10 × 30</td>
<td>10</td>
<td>6</td>
<td>11</td>
<td>0.2620</td>
<td>0.2724</td>
</tr>
<tr>
<td>6</td>
<td>20 × 50</td>
<td>21</td>
<td>11</td>
<td>21</td>
<td>0.6184</td>
<td>0.6181</td>
</tr>
<tr>
<td>7</td>
<td>20 × 50</td>
<td>21</td>
<td>11</td>
<td>21</td>
<td>0.6010</td>
<td>0.6141</td>
</tr>
<tr>
<td>8</td>
<td>20 × 50</td>
<td>21</td>
<td>11</td>
<td>21</td>
<td>0.5790</td>
<td>0.6197</td>
</tr>
<tr>
<td>9</td>
<td>20 × 50</td>
<td>21</td>
<td>11</td>
<td>21</td>
<td>0.5905</td>
<td>0.6072</td>
</tr>
<tr>
<td>10</td>
<td>20 × 50</td>
<td>21</td>
<td>11</td>
<td>21</td>
<td>0.5683</td>
<td>0.5876</td>
</tr>
<tr>
<td>11</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>51</td>
<td>3.5553</td>
<td>3.8227</td>
</tr>
<tr>
<td>12</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>51</td>
<td>3.4617</td>
<td>3.5506</td>
</tr>
<tr>
<td>13</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>51</td>
<td>3.8764</td>
<td>3.6818</td>
</tr>
<tr>
<td>14</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>51</td>
<td>3.7441</td>
<td>3.7137</td>
</tr>
<tr>
<td>15</td>
<td>50 × 100</td>
<td>50</td>
<td>26</td>
<td>51</td>
<td>3.7884</td>
<td>3.7541</td>
</tr>
<tr>
<td>16</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>101</td>
<td>40.7021</td>
<td>41.9762</td>
</tr>
<tr>
<td>17</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>101</td>
<td>40.3148</td>
<td>40.6185</td>
</tr>
<tr>
<td>18</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>101</td>
<td>40.9893</td>
<td>40.7602</td>
</tr>
<tr>
<td>19</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>101</td>
<td>41.3223</td>
<td>42.2368</td>
</tr>
<tr>
<td>20</td>
<td>100 × 200</td>
<td>100</td>
<td>51</td>
<td>101</td>
<td>40.6124</td>
<td>41.7873</td>
</tr>
<tr>
<td>21</td>
<td>200 × 500</td>
<td>200</td>
<td>101</td>
<td>201</td>
<td>695.4952</td>
<td>905.5381</td>
</tr>
<tr>
<td>22</td>
<td>200 × 500</td>
<td>200</td>
<td>101</td>
<td>201</td>
<td>654.5522</td>
<td>673.0402</td>
</tr>
<tr>
<td>23</td>
<td>200 × 500</td>
<td>200</td>
<td>101</td>
<td>201</td>
<td>687.5282</td>
<td>893.0445</td>
</tr>
<tr>
<td>24</td>
<td>200 × 500</td>
<td>200</td>
<td>101</td>
<td>201</td>
<td>680.1853</td>
<td>668.3996</td>
</tr>
<tr>
<td>25</td>
<td>200 × 500</td>
<td>200</td>
<td>101</td>
<td>201</td>
<td>673.5710</td>
<td>673.2532</td>
</tr>
</tbody>
</table>

Table 4.3 : Linear Matroid Circuit Genetic Algorithm
Chapter 5

Conclusions and Future Work

In this thesis, I proposed a branch-and-cut algorithm to solve a set covering problem. In turn, the solution to the set covering problem provides a solution to the matroid cogirth problem for a linear matroid. Although the algorithm is designed to find a solution for the matroid cogirth problem, it can be adapted to find a solution for the matroid girth problem. One simply has to flip the values of the binary constraint matrix $A$ for the set covering problem.

The results showed that Algorithm 1 performed well as a heuristic to find an upper bound on the cogirth, but did not perform very efficiently in verifying or finding an optimal solution for most of the test instances. Some possible reasons are the following. First, as the size of the test matrices grew, the size of a basis for the matroid, and the number of bases grew as well. The number of significant bases required to solve the problems increased and so did the time needed to find them. Also, it was difficult to find cuts from the class of inequalities developed by Balas and Ng [8] discussed in Chapter 3 in both Algorithms 1 and 2. Therefore, it was harder to reduce the feasibility region using these cutting planes. However, the feasible solution exclusion constraints described by Beasley et al. [12] were useful in both Algorithms 1 and 2. Algorithm 2 performed better than Algorithm 1, but this
was due to the number and size of matrices in the BBDF of the test matrices. A promising result is that both algorithms were able to find optimal solutions or near optimal quickly even though they took longer to terminate. This was partially due to the number of branch nodes that needed to be pruned in both algorithms. Although good feasible solutions were found quickly, the algorithms needed to consider other branch nodes in order to confirm that better solutions did not exist, or to find those better solutions. Therefore, the algorithms could be manually terminated after a given amount of time, and used as heuristics. Recall that the hitting set algorithm built its constraint set by developing a hitting set $H$, searching for a basis it had not hit, and choosing an element that hit the basis until a basis could not be found. An integer program was then solved to produce a solution $K$. If $|H| = |K|$ or if $K$ is feasible, i.e., dependent, then an optimal solution has been found. The hitting set $H$ was built using a greedy algorithm and the overall solution to the problem was found by raising the lower bound until the lower bound was equal to the optimal value. The Hitting Set algorithm did not perform well overall either. The reasons for this are similar to those for the branch-and-cut algorithm. The number of constraints needed to find an optimal solution increases greatly as the size of the matrix increases. Also, the Hitting Set algorithm sometimes solves an explicit set covering problem to optimality as a subproblem. The branch-and-cut algorithm was able to find good upper bound in an efficient amount of time. It would be interesting to try to hybridize the two methods to search for an optimal solution. That is, it might be possible to run the algorithms
concurrently to decrease the duality gap. As I mentioned in Chapter 3, the 0-1 MIP formulation presented by Kianfar et al. [48] can be used as a heuristic. In fact, the suggested heuristic appears to be accurate for most of the test instances and more efficient than just solving the 0-1 MIP formulation as it was originally presented.

The main goal of this research was to consider a well-known problem from a different perspective. The problem remains difficult to solve in its explicit form and even more so in its implicit form. For Algorithms 2 and 3, the ability to find cuts for the system of constraints faster is the biggest challenge. Even though both algorithms found optimal or near optimal solutions within seconds, exploring as little branch nodes as possible in the both algorithms is also important. Therefore, studying and finding good branching rules is also important. The ability of Algorithms 1 and 2 to find optimal or near optimal solutions quickly points to the possibility of using them as heuristics. Using the 0-1 MIP formulation in a heuristic should also be considered. Solving the matroid cogirth problem for linear matroids using mixed integer programming techniques appears to be a promising avenue. In any case, ongoing efforts to solve this problem using any sort of MIP formulation should focus on finding strong valid inequalities for the problem.

I also presented algorithms to find the girth of a linear matroid. The first algorithm uses a branch-decomposition of the matrix and dynamic programming to look for sets of columns which are linearly dependent. Instead of doing an exhaustive search, the algorithm tries to prune sets of columns which do not contain a part of some circuit,
or whose size is larger than a current best circuit. The lowest upper bound on the
girth of a matrix is $|B| + 1$ where $B$ is a basis of an $m \times n$ matrix with $m < n$. In such a
matrix without any duplicate columns, the algorithm would have to consider at most
all sets of size 3 to $|B|$ to confirm the girth of the matroid. This would only be the case
if the girth of the matrix was $|B| + 1$. One major caveat of original algorithm is that
it must store partial solutions in order to create new possible solutions. It was shown
that this can be avoided by using a different method to generate possible solutions.

Another possibility would be to considered the branch-decompositions generated from
the matrices. The ability to generate good branch-decompositions from the matrices
is something to research in and of itself. The branch-decompositions generated for the
test instances were caterpillar trees. If a branch-decomposition was a caterpillar tree,
it would be advantageous for the smallest circuit to be fully contained on either the
left or right child of the root node. This would make finding good feasible solutions
faster, and decrease the number of solutions to be considered. It is possible to have
branch-decompositions with a different structure. Consider the following example

```
Such a branch-decomposition would lend itself to different heuristics or variations
```
of the algorithms. For example, the linear matroid circuit algorithm could possibly be written in parallel. That is, based on the branch-decomposition, the matrix could be divided into $k$ submatrices that have at least $N$ columns. One suggestion would be based on the level of the tree at which to start. In other words, the level of the tree would dictate the size of the first submatrices tested. Each submatrix could be given to a different cpu. The subsets of columns of each submatrix could be tested to see if they contain a circuit. Using the example, start with matrices of size at least two. Then, the matrices $[1, 2, 3]$, $[4, 5]$, $[6, 7]$, $[8, 9]$ would be tested. Then, the matrices $[1, 2, 3, 4, 5]$ and $[6, 7, 8, 9]$ would be tested. Finally, the entire matrix would be tested. The upper bounds could be updated accordingly. It could be that the smallest circuit would be contained in one of these submatrices. In any case, developing better branch-decompositions would be advantageous to the linear matroid circuit algorithm.

The second algorithm was based on a genetic algorithm developed by Beasley and Chu [11]. The algorithm proved to be very efficient in generating solutions. However, the solutions generated were not better than solutions in the initial population. The ability to generate a better initial population of solutions would greatly improve the genetic algorithm. The branch-and-cut algorithms were able to find good solutions in a small number of iterations. It might be of interest to see if such an algorithm could be used to generate the initial population. Another aspect that could be tested is the production of children. Recall that the fusion operator proposed in [11] was used.
Testing other techniques such as the single-point or double-point operators provide further insight into generating better solutions.

Since finding the girth of a matroid is equivalent to finding the cogirth of the matroid’s dual, all of the algorithms proposed in this thesis can be used to generate solutions to either problem. It is also clear that reconstructing a signal in compressive sensing is equivalent to finding the girth of a matroid. Although many of the algorithms in compressive sensing only guarantee an optimal solution if the matrix contains certain properties, another avenue to consider would be using any one of these algorithms to produce an upper bound on the girth. After finding such a solution $C$, a search of the submatrix could be done to find the girth of the submatrix, thus providing a better upper bound on the girth of the entire matrix or using the solution $C$ as a warm start to the proposed girth algorithms. Another possibility is performing these steps iteratively on submatrices. That is find a submatrix $C_1$ of a matrix $\hat{A}$, then use a compressive sensing algorithm to search for a submatrix $C_2$ of $C_1$. This could be done iteratively restarting with the original matrix several times.

A few of the algorithms proposed in this thesis performed quite well while others did not. These algorithms provide insight into finding the girth and cogirth of linear matroids. They enable us to see what works, what does not work, and what needs to be improved. Knowing what does not work can provide as much information as knowing what does work.
Bibliography


[54] Susan Margulies, Illya V. Hicks Hicks, and Jing Ma. The cunningham-geelen method in practice: Branch decompositions and integer programming.


