Exploring Different Methods to Find Neumaier Graphs

Arendsen, Fleur J.

Award date:
2023

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science

Exploring Different Methods to Find Neumaier Graphs

by

Fleur Arendsen
1638645

Supervisors: Dr. Aida Abiad
Dr. Christopher Hojny

June 2023

Abstract

A Neumaier graph is a non-complete graph that is edge-regular and has a regular clique. Moreover, if a Neumaier graph is not strongly regular, it is called a strictly Neumaier graph. Determining whether a strictly Neumaier graph exists is a challenging problem. In this thesis, we propose new methods for finding these strictly Neumaier graphs. In particular, we formulate the problem as an integer linear program and as a boolean satisfiability problem. Although these methods work, they lose efficiency for graphs with a lot of vertices. Our findings serve as an initial exploration that paves the way for further development of this approach.

Contents

Chapter 1

Introduction

Neumaier graphs form a parameterized family of graphs that have certain properties after the seminal work by Neumaier in the 1980s who studied edge-regular graphs with a regular clique. The study of these graphs started after he posed the question 'Is every edge-regular graph with a regular clique strongly regular?' [18]. We provide a detailed definition of a Neumaier graph in Chapter 2. For some specifications of parameters, infinite families of Neumaier graphs are known to exist, but in general the problem is still wide open. In this thesis, we explore new methods to decide whether a Neumaier graph with certain specifications exists. In particular we wish to find new classes of Neumaier graphs. The two approaches we look into are formulating the problem of finding a Neumaier graph as an integer linear program and as a boolean satisfiability problem. To the best of our knowledge, these methods have not been previously applied to the study of Neumaier graphs. We analyze these methods and provide computational results on various test cases.

Note that integer programming is a well-known NP-complete problem [10]. However, there are several algorithms that have proven successful in solving integer programs, such as the branch-and-bound algorithm. This method decomposes the problem into smaller sub-problems and uses a bounding function to remove sub-problems that cannot contain the optimal solution [4]. Likewise, boolean satisfiability (SAT) is a well-known NP-complete problem [5]. However there are a number of algorithms that have been developed to solve SAT problems efficiently. One such algorithm is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm that uses the unit-class-rule. This algorithm iteratively chooses a literal and checks whether the SAT problem is satisfiable both when this literal is true and when it is false. The algorithm takes advantage of unit clauses, which are clauses with only one literal and uses that these literals can only take one value [2].

We are aiming to find graph structures with certain properties by transforming the problem into an integer linear program and a SAT problem. For other problem this has been done before. For example in [17] linear programming is used to find flows and matchings in graphs. Another example is [7] where linear programming is used to find regular graphs with a shortest cycle of 7. Moreover, in [15] the problem of finding a maximum independent set is reduced to a SAT problem.

The remainder of this thesis is organized as follows. In Chapter 2, we provide background and related work on Neumaier graphs and their properties. In Chapter 3, we describe our integer linear program that finds Neumaier graphs for given parameters and we go over the results. In Chapter 4, we describe how we formulated the problem as a boolean satisfiability problem and we go over the results. In Chapter 5, we define what it means for a graph to be close to a Neumaier graph and look for such graphs. Finally, in Chapter 6 we conclude our research.

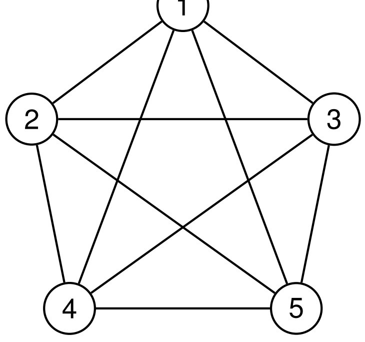
Chapter 2

Neumaier graphs

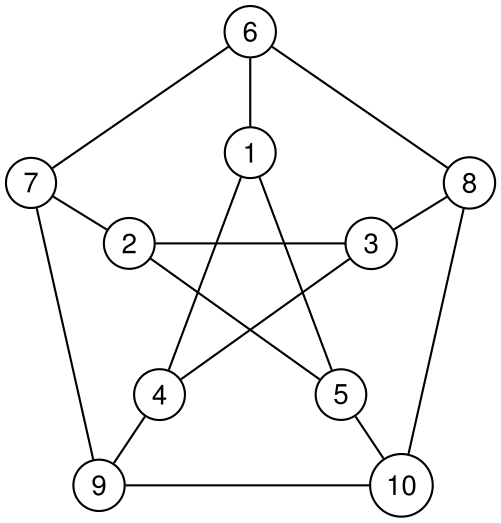
As previously stated, Neumaier graphs are a graph class named after the work of A. Neumaier, who studied edge-regular graphs with a regular clique [18]. We start by giving the definition of a Neumaier graph. Later in this chapter we provide an overview of the known results on Neumaier graphs.

2.1 Definitions

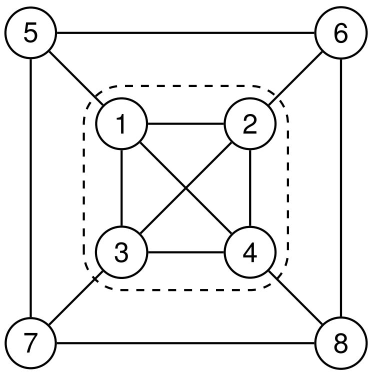
To explain what it means for a graph to be a Neumaier graph, we show some examples. We only consider undirected graphs with a finite number of vertices without loops or multiple edges. We say a vertex is a neighbor of another vertex if they are connected by an edge. A graph is *k-regular* if each vertex has exactly k neighbors. We say a graph is regular if it is k-regular for some integer k. A graph is *λ-edge-regular* if it is a non-empty regular graph in which any two adjacent vertices have exactly λ common neighbors. A graph is said to be non-empty if it has at least one edge. Moreover, two vertices are said to be adjacent if they are connected by an edge. We say a graph is edge-regular if it is λ-edge-regular for some integer λ. An example of a 3-edge-regular graph can be seen below, this means that the graph is non-empty, it is regular and every two adjacent vertices have 3 common neighbors. For example, vertices 1 and 2 share common neighbors 3, 4, and 5. Note that the graph is 4-regular as all vertices have exactly 4 neighbors. For example, vertex 1 has 4 neighbors which are vertices 2, 3, 4, and 5. We can see that this graph is actually complete, this means that any two vertices are connected by an edge.



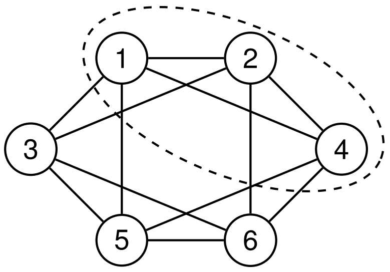
A graph is *μ-co-edge-regular* if it is a non-complete regular graph in which any two non-adjacent vertices have exactly μ common neighbors. A graph is said to be non-complete if it has at least one missing edge. Moreover, two vertices are said to be non-adjacent if they are not connected by an edge. We say a graph is co-edge-regular if it is μ-co-edge-regular for some integer μ. An example of a co-edge-regular graph is the Petersen graph which is illustrated below. This graph is 1-co-edge-regular, which means that the graph is non-complete, it is regular and any two non-adjacent vertices share 1 common neighbor. For example, vertices 1 and 2 share common neighbor 5. Note that the graph is 3-regular as all vertices have exactly 3 neighbors. Furthermore, a graph is *strongly regular* if it is both edge-regular and co-edge-regular. Note that actually the Petersen graph is strongly regular because it is moreover 0-edge-regular, as it is non-empty, regular, and any two adjacent vertices share 0 common neighbors.



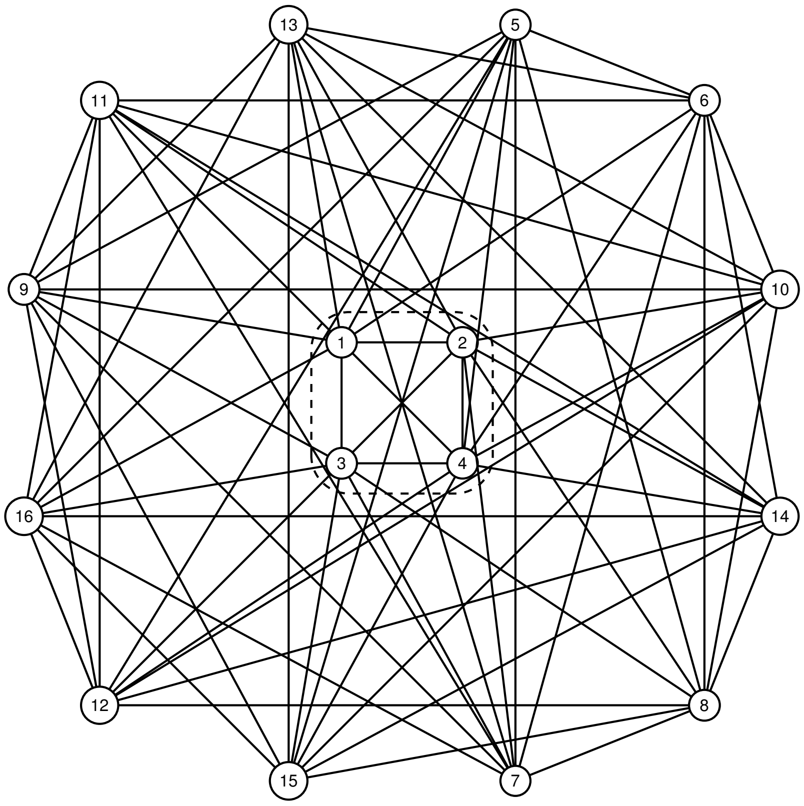
A graph has an *s-clique* if there is a subset of its vertices of size s such that every two distinct vertices in the subset are adjacent. We say a graph has a clique if it has an s-clique for some integer s. A graph has an *e-regular clique* if it has a clique such that any vertex outside the clique has e neighbors in the clique. We say a graph has a regular clique if it has an e-regular clique for some integer e. An example of a graph with a 1-regular clique can be seen below. Namely the vertices 1, 2, 3, and 4 are all pairwise adjacent and so they form a clique of size 4 and all vertices outside the clique are connected to 1 vertex in the clique. For example vertex 5 is connected to vertices 1, 6, and 7 of which only vertex 1 is in the clique.



Now we can define what it means for a graph to be a Neumaier graph. A *Neumaier graph* is a non-complete graph that is edge-regular and has a regular clique. We let $NG(v,k,\lambda,e,s)$ denote the collection of Neumaier graphs with v vertices, that are k-regular, λ-edge-regular and contain an e-regular s-clique. An example of a graph in $NG(6,4,2,2,3)$ can be seen below. This graph has 6 vertices. It is 4-regular as all vertices have exactly 4 neighbors. It is 2-edge-regular as it is a non-empty regular graph in which any two adjacent vertices have 2 common neighbors. Moreover, the vertices 1, 2, and 4 form a 2-regular clique of size 3, since they are all connected and all vertices outside the clique are connected to 2 vertices in the clique.



Note that also the above graph is 4-co-edge-regular, namely any two non-adjacent vertices have exactly 4 common neighbors. So it is strongly regular as it is co-edge-regular and edge-regular. A *strictly Neumaier graph* is a Neumaier graph that is not strongly regular. So the above graph is not a strictly Neumaier graph. Actually, the smallest strictly Neumaier graph has 16 vertices and can be seen below. This graph is an example of a graph in $NG(16, 9, 4, 2, 4)$. This graph was mentioned in [9]. One can check that this graph is not strongly regular, it is 9-regular and 4-edge-regular, and the vertices 1, 2, 3, and 4 form a 2-regular clique of size 4.



2.2 Previous work

After Neumaier posed the question regarding the existence of edge-regular graphs with a regular clique in [18], the study of Neumaier graphs began. In [12], Greaves and Koolen present the first infinite families of strictly Neumaier graphs, providing an answer to Neumaier's question. In [8], Evans, Goryainov, and Panasenko present significant results on parameters of strictly Neumaier graphs. Using these results, they found the smallest strictly Neumaier graph. In [1], Abiad, De Boeck, Castryck, Koolen, and Zeijlemaker present infinitely many new strictly Neumaier graphs. Furthermore, Table 1 of [1] provides an overview of parameters of strictly Neumaier graphs up to 64 vertices and whether it is known if a strictly Neumaier graph exists. We repeat this table below, it is shown in Table 2.1. For graphs for which it is known whether they exist, a reference is provided indicating where the proof of the result can be found. Note that for many of the parameters, it is not known whether a strictly Neumaier graph exists or not. Hence, our goal is to find new strictly Neumaier graphs with these parameters.

v	k	λ	e	s	Exists?
16	9	4	2	4	Yes, [8]
21	14	9	4	7	No, [1]
22	12	5	2	4	No, [8]
24	8	2	1	4	Yes, [8, 11, 13]
25	12	5	2	5	
	16	9	3	5	
26	15	8	3	6	
27	18	12	5	9	No, [1]
28	9	2	1	4	Yes, [8, 12]
	15	6	2	4	
		8	3	7	
	18	11	4	7	
33	22	15	6	11	No, [1]
	24	17	6	9	
34	18	7	2	4	
35	10	3	1	5	
	16	6	2	5	
	18	9	3	7	
	22	12	3	5	
36	11	2	1	4	
	15	6	2	6	
	20	10	3	6	
	21	12	4	8	
	25	16	4	6	
39	26	18	7	13	No, [1]
	30	23	9	13	No, [1]
40	12	2	1	4	Yes, [8]
	21	8	2	4	
		12	4	10	
	27	18	6	10	
	30	22	7	10	
42	11	4	1	6	
	21	10	3	7	
	26	15	4	7	
44	28	18	6	11	
45	12	3	1	5	
	20	7	2	5	
		10	3	9	
	24	13	4	9	
	28	15	3	5	
		17	5	9	
	30	21	8	15	No, [1]
	32	22	6	9	
46	24	9	2	4	
	25	12	3	6	
	27	16	5	10	
48	12	4	1	6	
	14	2	1	4	
	35	26	10	16	No, [1]
49	18	7	2	7	
	24	11	3	7	
	30	17	4	7	
	36	25	5	7	
50	28	15	4	8	
51	20	7	2	6	
	34	24	9	17	No, [1]
52	15	2	1	4	Yes, [12]
	27	10	2	4	
		16	5	13	
	36	25	8	13	
54	13	4	1	6	
55	14	3	1	5	
	24	8	2	5	
	30	17	5	11	
		18	3	5	
	34	21	6	11	
	36	23	6	10	
56	27	12	3	7	
	30	14	3	6	
	33	20	6	12	
	45	36	12	16	No, [1]
57	24	11	3	9	
	38	27	10	19	No, [1]
	40	27	6	9	
	42	31	10	15	
58	30	11	2	4	
60	14	4	1	6	Yes, [13]
	17	2	1	4	
	35	22	7	15	
	38	25	8	15	
63	14	5	1	7	
	30	13	3	7	
	32	16	4	9	
	38	21	4	7	
		22	5	9	
	42	30	11	21	No, [1]
	50	40	15	21	No, [1]
	52	43	16	21	No, [1]
64	18	2	1	4	
	21	8	2	8	
	28	12	3	8	
	33	12	2	4	
		20	6	16	
	35	18	4	8	Yes, [8]
	36	20	5	10	
	42	26	5	8	
	45	32	10	16	
	48	36	11	16	
	49	36	6	8	

Table 2.1: Feasible parameters for strictly Neumaier graphs up to 64 vertices.

Chapter 3

Integer linear programming approach

As explained, we want to use linear programming to find new Neumaier graphs. Feasible parameters for strictly Neumaier graphs up to 64 vertices can be found in Table 2.1. We consider only these parameters. We want to find graphs with v vertices, that are k-regular, λ-edge-regular and contain an e-regular s-clique and are not strongly regular for given parameters v, k, λ, e and s that can be found in this table.

To find such graphs we develop an integer linear program. We focus on linearizing all constraints in our program. According to [14], there is no efficient all-purpose algorithm for solving nonlinear programming problems, and some problems cannot be satisfactorily solved by any method. We can use well-known linear programming techniques by linearizing every constraint in our program. Furthermore, all constraints in our problem are straightforward to linearize, allowing us to avoid the computational complexity of nonlinear constraints. Note that we are only interested in whether strictly Neumaier graphs with given parameters exist and we are not interested in optimizing an objective function. This is a feasibility problem. As objective we can use the zero function that has to be minimized.

Similar as in [7], [15], and [17] we can define variables for every possible edge in the graph since we only consider graphs with a finite number of vertices in which there are no multiple edges. We label the vertices of the Neumaier graph from 1 up to v. We define binary variables $x_{i,j} \in \{0, 1\}$ for $i, j \in \{1, \dots, v\}$ to represent whether there is an edge in the resulting graph between vertices i and j. If there is such an edge, $x_{i,j} = 1$, otherwise $x_{i,j} = 0$. We then define a graph $G = (V, E)$ with vertices $V = \{1, \dots, v\}$ and edges $E = \{\{i, j\} \mid i, j \in \{1, \dots, v\}$ and $x_{i,j} = 1\}$. We now introduce constraints on these variables x to make sure the resulting graph is a strictly Neumaier graph.

3.1 Constraints

3.1.1 Preliminary constraints

We first introduce constraints on these variables to make sure they represent edges in a graph. We do not allow self loops in the graph. So there should be no edges between vertices that are the same. For any vertex i, $x_{i,i}$ represents whether the vertex is connected to itself, so this should equal

0. We get the following constraints

$$x_{i,i} = 0 \text{ for } i \in \{1, \ldots, v\}. \tag{3.1}$$

Edges are symmetric, meaning there is an edge between vertices i and j for two vertices $i, j \in \{1, \ldots, v\}$ if and only if there is also an edge between j and i. So $x_{i,j}$ should equal $x_{j,i}$ for any two vertices $i, j \in \{1, \ldots, v\}$. We get the following constraints

$$x_{i,j} = x_{j,i} \text{ for } i, j \in \{1, \ldots, v\}. \tag{3.2}$$

With these constraints we have made sure that the binary variables indeed represent edges in an undirected graph with no self loops. We now introduce constraints on these variables to make sure the resulting graph is a strictly Neumaier graph.

3.1.2 s-clique

The graph needs to have a clique of size s. Without loss of generality, we let the first s vertices, which are vertices 1 up to s, form such a clique. This assumption is valid because the vertices are indistinguishable and relabeling them does not change the underlying graph structure. This means there is an edge between every two distinct vertices in the first s vertices. So $x_{i,j}$ should equal 1 for any two distinct vertices $i, j \in \{1, \ldots, s\}$. We get the following constraints

$$x_{i,j} = 1 \text{ for } i, j \in \{1, \ldots, s\} \text{ s.t. } i \neq j. \tag{3.3}$$

Note that assuming the first s vertices form this clique makes the constraint relatively easy, this will also be convenient when adding the constraints to ensure this is an e-regular clique.

3.1.3 e-regular clique

There should be an e-regular s-clique in the graph, meaning that every vertex outside the clique is adjacent to e vertices in the clique. Note that for this constraint we can use that the first s vertices, 1 up to s, form a clique of size s. We will ensure this is an e-regular clique. We can use that the number of neighbors a vertex $i \in \{s+1, \ldots, v\}$ outside the clique has in the clique is equal to the number of incident edges to vertices in the clique. This is the sum of all $x_{i,j}$ over j ranging from 1 to s. This sum has to equal e. We get the following constraints

$$\sum_{j=1}^{s} x_{i,j} = e \text{ for } i \in \{s+1, \ldots, v\}. \tag{3.4}$$

We used the assumption that the first s vertices form a clique. By satisfying these constraints, we guarantee that the resulting clique becomes e-regular.

3.1.4 k-regular

The graph should be k-regular, meaning each vertex in the graph is adjacent to k other vertices. We can use that the number of neighbors a vertex $i \in \{1, \ldots, v\}$ has is equal to the number of incident edges. The number of incident edges is equal to the sum of all $x_{i,j}$ over j ranging from 1 to v. This sum has to equal k. We get the following constraints

$$\sum_{j=1}^{v} x_{i,j} = k \text{ for } i \in \{1, \dots, v\}. \tag{3.5}$$

These constraints ensure that the sum of incident edges from each vertex in the graph is equal to k, thereby guaranteeing the graph is k-regular.

3.1.5 λ-edge-regular

The graph should moreover be λ-edge-regular, meaning the graph is regular and all adjacent vertices have λ common neighbors. We already ensured the graph is k-regular with constraints (3.5). So we only need that any two adjacent vertices $i, j \in \{1, \dots, v\}$ have λ common neighbors. For vertices i and j to share a neighbor $m \in \{1, ..., v\}$, both $x_{i,m}$ and $x_{j,m}$ must be equal to 1. The product $x_{i,m}x_{j,m}$ equals 1 if and only if both $x_{i,m}$ and $x_{j,m}$ are equal to 1, indicating the presence of a common neighbor. The number of common neighbors between vertices i and j is the sum of the products $x_{i,m}x_{j,m}$ over all m ranging from 1 to v. This sum should equal λ for any two adjacent vertices $i, j \in \{1, \dots, v\}$, so for which $x_{i,j} = 1$. We get the following constraints

$$\sum_{m=1}^{v} x_{i,m}x_{j,m} = \lambda \text{ for } i, j \in \{1, \dots, v\} \text{ if } x_{i,j} = 1.$$

Note that these constraints are not linear constraints. Instead we introduce new binary variables y to represent the products in the summation.

$$y_{i,j,m} \in \{0, 1\} \text{ for } i, j, m \in \{1, \dots, v\} \tag{3.6}$$

So we should have $y_{i,j,m} = x_{i,m}x_{j,m}$. In order to ensure that these y variables accurately represent the products within the summation, we can introduce additional linear constraints that ensure this equivalence for all vertices $i, j, m \in \{1, \dots, v\}$. We use the same methods as in [4] to linearize products of binary variables. We get the following constraints

$$y_{i,j,m} \geq x_{i,m} + x_{j,m} - 1 \text{ for } i, j, m \in \{1, \dots, v\} \tag{3.7}$$

$$y_{i,j,m} \leq x_{i,m} \text{ for } i, j, m \in \{1, \dots, v\} \tag{3.8}$$

$$y_{i,j,m} \leq x_{j,m} \text{ for } i, j, m \in \{1, \dots, v\}. \tag{3.9}$$

Now, $y_{i,j,m}$ represents whether m is a common neighbor of i and j for vertices $i, j, m \in \{1, \dots, v\}$. So the number of common neighbors between two vertices $i, j \in \{1, \dots, v\}$ is exactly the sum of all $y_{i,j,m}$ over m ranging from 1 to v. We introduce the integer M that is large enough to indicate the constraint only has to be satisfied when i and j are adjacent, i.e. for $x_{i,j} = 1$. We get the following constraints that replace the non-linear constraint we mentioned above

$$\sum_{m=1}^{v} y_{i,j,m} \leq \lambda + M(1 - x_{i,j}) \text{ for } i, j \in \{1, \dots, v\} \tag{3.10}$$

$$\sum_{m=1}^{v} y_{i,j,m} \geq \lambda - M(1 - x_{i,j}) \text{ for } i, j \in \{1, \dots, v\}. \tag{3.11}$$

We can for example use $M = v$ such that these constraints are always satisfied for two vertices $i, j \in \{1, \ldots, v\}$ if $x_{i,j} = 0$. Note that if $x_{i,j} = 1$ for two vertices $i, j \in \{1, \ldots, v\}$, constraint (3.10) says that i and j cannot have more than λ common neighbors and constraint (3.11) says that i and j cannot have less than λ common neighbors. Together this implies that vertices i and j have exactly λ common neighbors.

3.1.6 Strictly Neumaier graph

We now have all constraints to ensure the resulting graph of the integer linear program is a Neumaier graph. We are however looking for strictly Neumaier graphs. For the graph to be a strictly Neumaier graph, it should not be co-edge-regular. This means that not all pairs of non-adjacent vertices have the same number of common neighbors. So there should at least be two pairs of non-adjacent vertices in the graph such that they have a different number of common neighbors. Note that we introduced new binary variables y such that $y_{i,j,m} = 1$ if and only if vertex m is a common neighbor of vertices i and j. We again use these variables to ensure the resulting graph is not co-edge-regular. We get the following constraint

$$\sum_{m=1}^{v} y_{a,b,m} \neq \sum_{m=1}^{v} y_{c,d,m} \text{ for some } a, b, c, d \in \{1, \ldots, v\} \text{ such that } a \neq b, c \neq d, x_{a,b} = 0 \text{ and } x_{c,d} = 0.$$

This constraint says that there should exist two pairs of non-adjacent vertices a, b and c, d such that they have a different number of common neighbors. Note that this is again not a linear constraint. Instead, without loss of generality, we can make assumptions on which vertices a, b, c, d satisfy this constraint. So far we only assumed which vertices form the s-clique. All vertices in the clique are indistinguishable and relabeling them does not change the underlying graph structure. Also, all vertices outside the clique are indistinguishable and relabeling them does not change the underlying graph structure. Since $a \neq b$ and $x_{a,b} = 0$, not both a and b can be in the clique. Namely for two distinct vertices in the clique, they always have an edge between them. Similarly not both c and d can be in the clique. So for both a, b and c, d, either one of the vertices is in the clique and the other is outside the clique or both vertices are outside the clique. Note that we may assume they satisfy the constraint as $\sum_{m=1}^{v} y_{a,b,m} < \sum_{m=1}^{v} y_{c,d,m}$, which is equivalent to $\sum_{m=1}^{v} y_{a,b,m} \leq \sum_{m=1}^{v} y_{c,d,m} - 1$, i.e. the number of common neighbors of the vertices a and b is smaller than the number of common neighbors of the vertices c and d. Namely, the constraint $\sum_{m=1}^{v} y_{a,b,m} > \sum_{m=1}^{v} y_{c,d,m}$ is the same for a, b and c, d swapped and therefore redundant. We now explain the different cases.

Both a, b and c, d have one vertex in the clique and one vertex outside the clique:
We assume without loss of generality $a = 1, b = s + 1$, i.e. a is the first vertex of the clique and b is the first vertex outside the clique. Then, either c and d are two different vertices from a and b, or one of them is the same. We may assume without loss of generality $c = 2, d = s + 2$ or $c = 1, d = s + 2$ or $c = 2, d = s + 1$. So we find case 1 with values $a_1 = 1, b_1 = s + 1, c_1 = 2, d_1 = s + 2$ and case 2 with values $a_2 = 1, b_2 = s + 1, c_2 = 1, d_2 = s + 2$ and case 3 with values $a_3 = 1, b_3 = s + 1, c_3 = 2, d_3 = s + 1$.

One vertex of a, b is in the clique and the other is outside the clique and c, d are both outside the clique:

We again assume without loss of generality $a = 1, b = s + 1$. Then, either c and d have one vertex the same as a and b outside the clique or they are two different vertices. We may assume without loss of generality $c = s + 2, d = s + 3$ or $c = s + 1, d = s + 2$. So we find case 4 with values $a_4 = 1, b_4 = s+1, c_4 = s+2, d_4 = s+3$ and case 5 with values $a_5 = 1, b_5 = s+1, c_5 = s+1, d_5 = s+2$.

One vertex of c, d is in the clique and the other is outside the clique and a, b are both outside the clique:
We again assume without loss of generality $c = 1, d = s + 1$. Then, either a and b have one vertex the same as c and d outside the clique or they are two different vertices. We may assume without loss of generality $a = s + 2, b = s + 3$ or $a = s + 1, b = s + 2$. Note that these cases are similar to the previous cases where a, b and c, d are switched. We need this case as we assumed the inequality is satisfied as $\sum_{m=1}^{v} y_{a,b,m} < \sum_{m=1}^{v} y_{c,d,m}$ where the number of common neighbors of the vertices a and b is smaller than the number of common neighbors of the vertices c and d. So we find case 6 with values $a_6 = s + 2, b_6 = s + 3, c_6 = 1, d_6 = s + 1$ and case 7 with values $a_7 = s + 1, b_7 = s + 2, c_7 = 1, d_7 = s + 1$.

All vertices a, b, c, d are outside the clique:
We assume without loss of generality $a = s + 1, b = s + 2$, i.e. a is the first vertex outside the clique and b is the second vertex outside the clique. Then, either c and d have one vertex the same as a and b outside the clique or they are two different vertices. We may assume without loss of generality $c = s+3, d = s+4$ or $c = s+2, d = s+3$. So we find case 8 with values $a_8 = s+1, b_8 = s+2, c_8 = s + 3, d_8 = s + 4$ and case 9 with values $a_9 = s + 1, b_9 = s + 2, c_9 = s + 2, d_9 = s + 3$.

We find a total of 9 different cases of which at least one needs to hold. We can linearize this by introducing binary variables that represent whether a case holds. We also introduce a new constraint that says that the sum of these binary variables is at least 1, so at least one of these cases should hold. We introduce new binary variables z.

$$z_n \in \{0, 1\} \text{ for } n \in \{1, \dots, 9\} \tag{3.12}$$

We again make use of the integer M that is large enough to indicate the corresponding constraints only have to be satisfied when the case holds. We now add the constraints to make sure these z variables represent whether a case holds. We add the following constraints for all cases mentioned above where n is the case number

$$x_{a_n,b_n} \leq 1 - z_n \tag{3.13}$$

$$x_{c_n,d_n} \leq 1 - z_n \tag{3.14}$$

$$\sum_{m=1}^{v} y_{a_n,b_n,m} \leq \sum_{m=1}^{v} y_{c_n,d_n,m} + M(1 - z_n) - 1. \tag{3.15}$$

Again, $M = v$ is large enough such that these constraints are always satisfied if $z_n = 0$. If $z_n = 1$, constraint (3.13) says that $x_{a_n,b_n} \leq 0$ and constraint (3.14) says that $x_{c_n,d_n} \leq 0$, implying $x_{a_n,b_n} = 0$ and $x_{c_n,d_n} = 0$ since these variables are binary. Constraint (3.15) says that the number of common neighbors between vertices a_n and b_n is at least one less than the number of common neighbors between vertices c_n and d_n. We moreover add the following constraint to make sure at least one of the cases is satisfied

$$\sum_{n=1}^{9} z_n \geq 1. \tag{3.16}$$

We have successfully established all the necessary constraints to ensure that the resulting graph from our integer linear program $G = (V, E)$ is a strictly Neumaier graph with parameters v, k, λ, e, and s. Here $V = \{1, \ldots, v\}$ and $E = \{\{i, j\} \mid i, j \in \{1, \ldots, v\}$ and $x_{i,j} = 1\}$. This graph consists of v vertices. It is non-complete as this follows from the parameters given in Table 2.1. It is edge regular as it is non-empty, k-regular, and any two adjacent vertices have λ common neighbors. Additionally, there exists a clique of size s, namely $\{1, \ldots, s\}$, that is e-regular. Moreover, the resulting graph is not strongly regular.

We now know that a strictly Neumaier graph exists for the given parameters v, k, λ, e, s if and only if the corresponding integer linear program is feasible because we have not imposed any additional assumptions on these graphs, aside from the usual 'without loss of generality'.

3.2 Implementation

We implement the integer linear program in Python (version 3.7.6) using the Gurobi optimization library (version 10.0.1). It is important to note that we can represent the variables in a more compact way to solve the linear program more efficiently. Instead of defining $x_{i,j}$ for all pairs of vertices, we only need to define $x_{i,j}$ for $j > i$, since $x_{i,i} = 0$ and $x_{j,i} = x_{i,j}$ for all $i, j \in \{1, \ldots, v\}$. This allows us to transform all the constraints accordingly. For instance, we can replace the k-regularity constraint (3.5) with the following constraint

$$\sum_{j=1}^{i-1} x_{j,i} + \sum_{j=i+1}^{v} x_{i,j} = k \text{ for } i \in \{1, \ldots, v\}.$$

This way we only use variables $x_{i,j}$ for which $j > i$. Similarly, all other constraints can be replaced. We can also represent the y variables in a more compact way. We only need to define $y_{i,j,m}$ for $j > i$ since $y_{i,i,m} = x_{i,m}x_{i,m} = x_{i,m}$ and $y_{j,i,m} = x_{j,m}x_{i,m} = x_{i,m}x_{j,m} = y_{i,j,m}$ for all $i, j, m \in \{1, \ldots, v\}$. We also do not need to define $y_{i,j,m}$ for $m = i$ or $m = j$, since we know that $y_{i,j,i} = x_{i,i}x_{j,i} = 0$ and similarly $y_{i,j,j} = x_{i,j}x_{j,j} = 0$ for $i, j \in \{1, \ldots, v\}$. Again, we can transform all our constraints as explained before. We do not go into too much detail. So for our implementation we use this more compact way to describe the variables instead. The full integer linear program with constraints can be found below. Here, constraints (3.34), (3.35) and (3.36) ensure the x variables, y variables and z variables are binary. Constraint (3.18) corresponds to constraint (3.3) and ensures the first s vertices form a clique. Constraint (3.19) corresponds to constraint (3.4) and ensures the clique formed by the first s vertices is e-regular. Constraint (3.20) corresponds to constraint (3.5) and ensures the graph is k-regular. Constraints (3.21) and (3.22) correspond to constraints (3.10) and (3.11) respectively and ensure that the graph is λ-edge-regular. Constraints (3.23), (3.24), (3.25), and (3.26) correspond to constraints (3.13), (3.14), (3.15), and (3.16) respectively and ensure the graph is not strongly regular. Constraints (3.27), (3.28), (3.29), (3.30), (3.31), (3.32), and (3.33) correspond to constraints (3.7), (3.8), and (3.9) and ensure the y variables represent products of the x variables. These constraints are divided into multiple cases depending on the value of m since we only have variables $x_{i,m}$ for $m > i$ and $x_{m,i}$ for $m < i$ and similarly for $x_{j,m}$ and $x_{m,j}$. The Python code for the integer linear program to find strictly Neumaier graphs with variables and constraints as described can be found in Appendix B.

$$\min 0 \tag{3.17}$$

$$x_{i,j} = 1 \qquad \text{for } i,j \in \{1,\dots,s\} \text{ s.t. } j > i \tag{3.18}$$

$$\sum_{j=1}^{s} x_{j,i} = e \qquad \text{for } i \in \{s+1,\dots,v\} \tag{3.19}$$

$$\sum_{j=1}^{i-1} x_{j,i} + \sum_{j=i+1}^{v} x_{i,j} = k \qquad \text{for } i \in \{1,\dots,v\} \tag{3.20}$$

$$\sum_{m\in\{1,\dots,v\}\setminus\{i,j\}} y_{i,j,m} \le \lambda + M(1 - x_{i,j}) \qquad \text{for } i,j \in \{1,\dots,v\} \text{ s.t. } j > i \tag{3.21}$$

$$\sum_{m\in\{1,\dots,v\}\setminus\{i,j\}} y_{i,j,m} \ge \lambda - M(1 - x_{i,j}) \qquad \text{for } i,j \in \{1,\dots,v\} \text{ s.t. } j > i \tag{3.22}$$

$$x_{a_n,b_n} \le 1 - z_n \qquad \text{for } n \in \{1,\dots,9\} \tag{3.23}$$

$$x_{c_n,d_n} \le 1 - z_n \qquad \text{for } n \in \{1,\dots,9\} \tag{3.24}$$

$$\sum_{m\in\{1,\dots,v\}\setminus\{a_n,b_n\}} y_{a_n,b_n,m} \le \sum_{m\in\{1,\dots,v\}\setminus\{c_n,d_n\}} y_{c_n,d_n,m} + M(1 - z_n) - 1 \qquad \text{for } n \in \{1,\dots,9\} \tag{3.25}$$

$$\sum_{n=1}^{9} z_n \ge 1 \tag{3.26}$$

$$y_{i,j,m} \ge x_{i,m} + x_{j,m} - 1 \qquad \text{for } i,j,m \in \{1,\dots,v\} \text{ s.t. } m > j > i \tag{3.27}$$

$$y_{i,j,m} \ge x_{i,m} + x_{m,j} - 1 \qquad \text{for } i,j,m \in \{1,\dots,v\} \text{ s.t. } j > m > i \tag{3.28}$$

$$y_{i,j,m} \ge x_{m,i} + x_{m,j} - 1 \qquad \text{for } i,j,m \in \{1,\dots,v\} \text{ s.t. } j > i > m \tag{3.29}$$

$$y_{i,j,m} \le x_{i,m} \qquad \text{for } i,j,m \in \{1,\dots,v\} \text{ s.t. } j > i, m > i, m \ne j \tag{3.30}$$

$$y_{i,j,m} \le x_{m,i} \qquad \text{for } i,j,m \in \{1,\dots,v\} \text{ s.t. } j > i > m \tag{3.31}$$

$$y_{i,j,m} \le x_{j,m} \qquad \text{for } i,j,m \in \{1,\dots,v\} \text{ s.t. } m > j > i \tag{3.32}$$

$$y_{i,j,m} \le x_{m,j} \qquad \text{for } i,j,m \in \{1,\dots,v\} \text{ s.t. } j > i, j > m, m \ne i \tag{3.33}$$

$$x_{i,j} \in \{0,1\} \qquad \text{for } i,j \in \{1,\dots,v\} \text{ s.t. } j > i \tag{3.34}$$

$$y_{i,j,m} \in \{0,1\} \qquad \text{for } i,j,m \in \{1,\dots,v\} \text{ s.t. } j > i, m \ne i, m \ne j \tag{3.35}$$

$$z_n \in \{0,1\} \qquad \text{for } n \in \{1,\dots,9\} \tag{3.36}$$

3.3 Results

For all results in this thesis the computations are conducted on a computer with an Intel Core i7-9750H processor, equipped with 16 GB of RAM. The time limit for all computations was 5 hours per instance. We were able to solve the linear program only for the smallest parameters that can be found in Table 2.1, namely $v = 16, k = 9, \lambda = 4, e = 2, s = 4$. Using these parameters, we obtained the strictly Neumaier graph shown in Figure 3.1, and its corresponding adjacency matrix A_1, which is presented below.

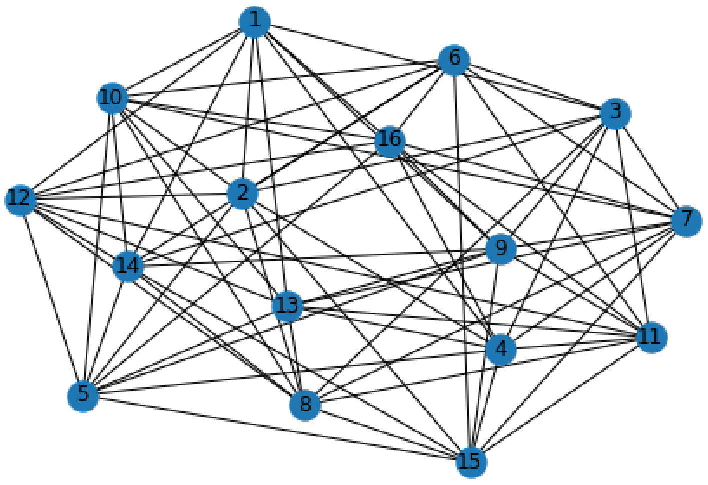


Figure 3.1: Strictly Neumaier graph with parameters $v = 16, k = 9, \lambda = 4, e = 2, s = 4$ found after solving linear program with adjacency matrix A_1.

$$A_1 = \begin{pmatrix}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{pmatrix}$$

This graph was found in 31.6 seconds. We hereby confirmed the existence of a strictly Neumaier graph with parameters $v = 16, k = 9, \lambda = 4, e = 2, s = 4$. This existence was previously established in literature. Specifically, we found a graph that is isomorphic to the strictly Neumaier graph mentioned in [9] with the same parameters. We say two graphs are isomorphic if they have the same underlying structure. For the next three parameter sets in Table 2.1, the program did not solve within the given time limit. We expect that the program would also not solve within the time limit for any higher parameters because the number of variables increases cubically with the number of vertices v.

3.4 Further constraints

The program only runs for the smallest parameters. We could instead solve the LP relaxation of our program. However, since we are not interested in determining the optimal value of the objective function, solving the LP relaxation of our program is not relevant in this situation. Instead we are looking for edges that make up a strictly Neumaier graph. We cannot conclude this from the solution to the LP relaxation as this has fractional values. We consider further techniques to improve the integer program. We introduce new constraints, known as cuts, which do not remove any feasible solutions from the original program. A valid cut is one that does not remove any integer solutions. We want to identify cuts that eliminate portions of the feasible region that contain fractional solutions. If the removed area includes the fractional solution obtained from the LP relaxation of the integer program, the cut is useful. However, determining in advance which cuts are definitely useful is challenging without knowledge of the LP relaxation solution [16]. We will add constraints to the integer linear program to exclude more non-integer solutions as we search for useful cuts. By doing so, we expect that the integer linear program will become easier to solve, enabling us to find whether larger Neumaier graphs exist and if so, find examples. It is worth noting that we only add constraints that eliminate non-integer solutions. Therefore, we still know that a strictly Neumaier graph exists for the given parameters v, k, λ, e, s if and only if the integer linear program is feasible.

3.4.1 Maximum clique size

As mentioned, we will try to add constraints that reduce the feasible region of the LP relaxation of the integer linear program as specified in Section 3.2. We find the following result on cliques in non-complete edge-regular graphs with an e-regular clique of size s. This is Theorem 1.1(ii) from [18].

Lemma 3.4.1. *Let G be an edge-regular graph which is not complete. If G has an e-regular clique of size s then every clique has size at most s.*

Since a strictly Neumaier graph with parameters v, k, λ, e, s is a non-complete edge-regular graph with an e-regular clique of size s, we know from Lemma 3.4.1 that every clique has size at most s. We now explain how we model this result as a constraint in our integer linear program. Since this result is satisfied for all strictly Neumaier graphs, and all feasible solutions to our integer linear program represent strictly Neumaier graphs, the constraint that models the result is valid. To model this result, we consider all subsets $S \subseteq \{1, \dots, v\}$ of $s+1$ vertices. For any such subset S, we require that there exists at least two vertices in S such that they are not connected in the graph. Note that if there is an edge between every two vertices in S, there are exactly $\frac{s(s+1)}{2}$ edges between all vertices in S. By enforcing at least one missing edge within each such subset, we ensure that the graph does not have a clique of size $s+1$ or larger. We get the following constraint for every subset S of $s+1$ vertices

$$\sum_{i,j \in S \text{ s.t. } i<j} x_{i,j} \leq \frac{s(s+1)}{2} - 1. \tag{3.37}$$

A question that now arises is whether the feasible region of the LP relaxation of our program is indeed reduced by adding constraint (3.37) for every set S, i.e. if this is a useful cut. We did some testing to see whether adding this constraint cuts of non-integer solutions found when running the LP relaxation of the linear program. To do this, we solve the LP relaxation of the integer linear program with variables and constraints as described in Section 3.2 and we check whether there is a subset S of size $s+1$ such that (3.37) is not satisfied. Namely, then this solution does not satisfy the constraint (3.37) and so it is cut off when adding this to the integer program. The corresponding Python code can be found in Appendix B. We have only tested this for some of the smaller parameters. We find that for the smaller parameters this constraint does not cut off any solutions found after running the LP relaxation. Also, by adding this constraint to the integer linear program, we still cannot solve it for higher parameters. Because iterating over all subsets S of size $s+1$ is not desirable, we choose not to use this constraint any further.

3.4.2 Constraints for $v = 3(k - \lambda) + 1$

Since adding the above constraints still does not help solving the problem for higher parameters, we try to add other constraints that reduce the feasible region of the LP relaxation of the integer linear program as described in Section 3.2. We find the following result on λ-edge-regular graphs with v vertices that are k-regular with parameters that satisfy $v = 3(k - \lambda) + 1$. This is Lemma 1 from [6].

Lemma 3.4.2. *Let G be a λ-edge-regular graph with v vertices that is k-regular for some v, k, λ. Then, if $v = 3(k - \lambda) + 1$, every triangle belongs to at most one K_4.*

Note that there are several open cases for strictly Neumaier graphs with parameters that satisfy $v = 3(k - \lambda) + 1$ that can be found in Table 2.1. All feasible parameters for Neumaier graphs up to 64 vertices that satisfy $v = 3(k - \lambda) + 1$ can be seen in Table 3.1.

v	k	λ	e	s	Exists?
16	9	4	2	4	Yes, [8]
22	12	5	2	4	No, [8]
28	15	6	2	4	
34	18	7	2	4	
40	21	8	2	4	
46	24	9	2	4	
52	27	10	2	4	
58	30	11	2	4	
64	33	12	2	4	

Table 3.1: Feasible parameters for strictly Neumaier graphs up to 64 vertices that satisfy $v = 3(k - \lambda) + 1$.

Therefore, it is relevant to see whether we can add a useful cut only for this parameter class. It is worth noting that strictly Neumaier graphs are edge-regular, so from Lemma 3.4.2 we know that for strictly Neumaier graphs with parameters listed in Table 3.1, each triangle belongs to at most one complete subgraph K_4. Note that every triangle belonging to at most one K_4 is equivalent to every triangle having at most one vertex outside the triangle connected to all vertices in the triangle. Specifically, for any three distinct vertices $i, j, m \in \{1, \dots, v\}$, if they are all connected, there can be at most one vertex that is connected to all three vertices i, j, m. All other $v - 4$ vertices within the graph can be connected to at most two of the vertices i, j, m. To include this result within our integer linear program, we can introduce the following constraint for every set of three distinct vertices $i, j, m \in \{1, \dots, v\}$

$$\sum_{t \neq i,j,m} (x_{t,j} + x_{t,i} + x_{t,m}) \leq 2(v-4) + 3 \text{ if } x_{i,j} = x_{i,m} = x_{j,m} = 1.$$

This is not a linear constraint. We again make use of the integer M that is large enough to indicate the corresponding constraints only have to be satisfied when the vertices form a triangle. So we can add the following constraints for all three distinct vertices $i, j, m \in \{1, \dots, v\}$ that is always satisfied if not all vertices i, j, m are connected

$$\sum_{t \neq i,j,m} (x_{t,j} + x_{t,i} + x_{t,m}) \leq 2(v-4) + 3 + M(1 - x_{i,j}) + M(1 - x_{i,m}) + M(1 - x_{j,m}). \quad (3.38)$$

Note that again $M = v$ is big enough. Now, as mentioned, it is interesting to see whether adding constraint (3.38) for any three distinct vertices $i, j, m \in \{1, \dots, v\}$ will cut off any non-integer solutions found after running the LP relaxation of the linear program, i.e. if this is a useful cut. To do this, we solve the LP relaxation of the integer linear program with variables and constraints as described in Section 3.2 and we check whether there are three distinct vertices $i, j, m \in \{1, \dots, v\}$ such that (3.38) is not satisfied. Namely, then this solution does not satisfy the constraint (3.38) and so it is cut off when adding this to the integer program. The corresponding Python code can be found in Appendix B. After some testing, we again find that for the parameters in Table 3.1, adding this constraint does not cut off any solutions found after running the LP relaxation of the linear program. Also, by adding this constraint to the integer linear program, we still cannot solve it for higher parameters. Because iterating over all triangles i, j, m is not desirable, we choose not to use this constraint any further. Instead, we can model the result of the lemma as a stronger constraint by introducing new variables w to represent whether a vertex outside the triangle is connected to all vertices in the triangle.

$$w_{i,j,m,t} \in \{0, 1\} \text{ for } i, j, m, t \in \{1, \dots, v\}$$

For any $i, j, m, t \in \{1, \dots, v\}$, $w_{i,j,m,t}$ should represent whether t is connected to all vertices i, j, m. For t to be connected to all vertices i, j, m, $x_{i,t}$, $x_{j,t}$ and $x_{m,t}$ must be equal to 1. The product $x_{i,t}x_{j,t}x_{m,t}$ equals 1 if and only if $x_{i,t}$, $x_{j,t}$ and $x_{m,t}$ are equal to 1. So we should have $w_{i,j,m,t} = x_{i,t}x_{j,t}x_{m,t}$ such that this variable represents whether t is connected to all i, j, m. In order to ensure that these w variables accurately represent these products, we can introduce additional linear constraints for all vertices $i, j, m, t \in \{1, \dots, v\}$. We again use the same methods as in [4] to linearize products of binary variables. We get the following constraints

$$w_{i,j,m,t} \leq x_{i,t}, \quad w_{i,j,m,t} \leq x_{j,t}, \quad w_{i,j,m,t} \leq x_{m,t}, \quad w_{i,j,m,t} \geq x_{i,t} + x_{j,t} + x_{m,t} - 2. \quad (3.39)$$

So $w_{i,j,m,t}$ represents whether t is a common neighbor of i, j and m for vertices $i, j, m, t \in \{1, \dots, v\}$. Now, if i, j, m are all connected, there should be at most one vertex outside the triangle connected to all i, j, m. We get the following constraints for every three distinct vertices $i, j, m \in \{1, \dots v\}$

$$\sum_{t \neq i,j,m} w_{i,j,m,t} \leq 1 \text{ if } x_{i,j} = x_{i,m} = x_{j,m} = 1.$$

This is not a linear constraint. Instead, again for M big enough, we can add the following constraints for all three distinct vertices $i, j, m \in \{1, \dots, v\}$ that is always satisfied if not all vertices i, j, m are connected

$$\sum_{t\neq i,j,m} w_{i,j,m,t} \leq 1 + M(1-x_{i,j}) + M(1-x_{i,m}) + M(1-x_{j,m}). \tag{3.40}$$

Also for this constraint, $M = v$ is big enough. We now prove that this is indeed a stronger constraint.

Lemma 3.4.3. *If the constraints* (3.39) *and* (3.40) *are satisfied for all three distinct vertices* $i, j, m \in \{1, \dots, v\}$*, then also the constraint* (3.38) *is satisfied for all three distinct vertices* $i, j, m \in \{1, \dots, v\}$.

Proof. We prove Lemma 3.4.3. We assume the constraints (3.39) and (3.40) are satisfied for all three distinct vertices i, j, m. Let $i, j, m \in \{1, \dots, v\}$ be three distinct vertices. Then

$$\begin{aligned}\sum_{t\neq i,j,m}(x_{t,j}+x_{t,i}+x_{t,m}) \leq \sum_{t\neq i,j,m}(w_{i,j,m,t}+2) &= \sum_{t\neq i,j,m} w_{i,j,m,t} + \sum_{t\neq i,j,m} 2 \\ &\leq 1 + M(1-x_{i,j}) + M(1-x_{i,m}) + M(1-x_{j,m}) + 2(v-3) \\ &= 2(v-4) + 3 + M(1-x_{i,j}) + M(1-x_{i,m}) + M(1-x_{j,m}).\end{aligned}$$

We find that also the constraints (3.38) are satisfied for all three distinct vertices i, j, m. □

Again, for this stronger constraint (3.40) it is interesting to see whether adding this constraint will cut off any non-integer solutions, i.e. if this is a useful cut. To do this, we add the w variables and we add the new constraints (3.39) to the integer linear program as explained in Section 3.2. We again use that we can represent w variables in a more compact way. Then we solve the LP relaxation of the linear program and we check whether there are three distinct vertices i, j, m such that (3.40) is not satisfied. The corresponding Python code can be found in Appendix B. We again did some testing and find that after solving for the parameters in Table 3.1, there are two parameter sets for which the found solution violates the constraint for some i, j, m. Specifically for $v = 34, k = 18, \lambda = 7, e = 2, s = 4$ and $v = 40, k = 21, \lambda = 8, e = 2, s = 4$. However, we also observe that the LP relaxation runs significantly slower as we are adding a lot of new variables to the linear program. In particular, the number of variables now increases with the fourth power in relation to the number of vertices v. Again, by adding this constraint to the integer linear program, we still cannot solve it for higher parameters. So we find that adding all these new w variables is not worth it. We decide that we do not use these constraints and variables any further.

3.4.3 Constraints for $v = 4(k+1) - 6\lambda$

We try to add other constraints that reduce the feasible region of the LP relaxation of the integer linear program as specified in Section 3.2 as adding the previous constraints still does not help solve the problem for larger parameters. We find the following result on λ-edge-regular graphs with v vertices that are k-regular and contain at least one K_4 with parameters that satisfy $v = 4(k+1) - 6\lambda$. This is Lemma 2 from [6].

Lemma 3.4.4. *Let G be a λ-edge-regular graph with v vertices that is k-regular for some v, k, λ. Assume G contains at least one K_4, then*

(i) $v = 4(k+1) - 6\lambda$ *if and only if every vertex of G is adjacent to either 1 or 2 vertices of every K_4 of which it is not an element;*

(ii) $v = 4(k+1) - 6\lambda$ *and* $\lambda = k/3 + 1$ *if and only if every vertex of G is adjacent to exactly 2 vertices of every K_4 of which it is not an element.*

Note that there are a several open cases for strictly Neumaier graphs with parameters that satisfy $v = 4(k+1) - 6\lambda$ that can be found in Table 2.1. All feasible parameters for Neumaier graphs up to 64 vertices that satisfy $v = 4(k+1) - 6\lambda$ can be seen in Table 3.2.

v	k	λ	e	s	Exists?
16	9	4	2	4	Yes, [8]
22	12	5	2	4	No, [8]
24	8	2	1	4	Yes, [8, 11, 13]
28	9	2	1	4	Yes, [8, 12]
28	15	6	2	4	
34	18	7	2	4	
36	11	2	1	4	
40	12	2	1	4	Yes, [8]
40	21	8	2	4	
46	24	9	2	4	
48	14	2	1	4	
52	15	2	1	4	Yes, [12]
52	27	10	2	4	
58	30	11	2	4	
60	17	2	1	4	
64	18	2	1	4	
64	33	12	2	4	

Table 3.2: Feasible parameters for strictly Neumaier graphs up to 64 vertices that satisfy $v = 4(k+1) - 6\lambda$.

Therefore, it is relevant to see whether we can add a useful cut only for this parameter class. It is worth noting that strictly Neumaier graphs are edge-regular, and for all parameters in Table 3.2, $s = 4$ and so the Neumaier graph contains a K_4. So by Lemma 3.4.4 we know that for strictly Neumaier graphs with parameters listed in Table 3.2, every vertex is adjacent to 1 or 2 vertices of every K_4 of which it is not an element and if moreover $\lambda = k/3 + 1$, every vertex is adjacent to exactly 2 vertices of every K_4 of which it is not an element. We can model item (i) as a constraint as follows. For every vertex l, and any subset K of 4 vertices that do not contain this vertex l, if they are all connected, l is connected to 1 or 2 of them. This gives the following constraint for every subset $K \subseteq \{1, \dots, v\}$ of 4 vertices and every vertex $l \in \{1, \dots, v\} \setminus K$

$$1 \leq \sum_{j \in K} x_{l,j} \leq 2 \text{ if } \sum_{i,j \in K \text{ s.t. } i<j} x_{i,j} = 6.$$

Namely, vertices in the subset K are all pairwise connected if and only if there are 6 edges between these vertices. This is not a linear constraint. Instead, for M large enough we can add the following constraint for every subset $K \subseteq \{1, \dots, v\}$ of 4 vertices and every vertex $l \in \{1, \dots, v\} \setminus K$

$$1 - M(6 - \sum_{i,j \in K \text{ s.t. } i<j} x_{i,j}) \leq \sum_{j \in K} x_{l,j} \leq 2 + M(6 - \sum_{i,j \in K \text{ s.t. } i<j} x_{i,j}). \tag{3.41}$$

Note that again $M = v$ is large enough. Similarly, we can model item (ii). We can add the following constraint for every subset $K \subseteq \{1, \ldots, v\}$ of 4 vertices and every vertex $l \in \{1, \ldots, v\} \setminus K$

$$2 - M(6 - \sum_{i,j \in K \text{ s.t. } i<j} x_{i,j}) \leq \sum_{j \in K} x_{l,j} \leq 2 + M(6 - \sum_{i,j \in K \text{ s.t. } i<j} x_{i,j}). \tag{3.42}$$

Note that this is a stronger constraint. Again, it is interesting to see whether adding these constraints will cut off any non-integer solutions found after running the LP relaxation of the linear program, i.e. if this gives a useful cut. To do this, we solve the LP relaxation of the integer linear program with variables and constraints as explained in Section 3.2 and for parameters that do not satisfy $\lambda = k/3 + 1$ we check whether there is a subset K of 4 vertices and a vertex $l \notin K$ such that the constraint (3.41) is not satisfied. For parameters that satisfy $\lambda = k/3 + 1$ we instead check whether there is a subset K of 4 vertices and a vertex $l \notin K$ such that the constraint (3.42) is not satisfied. The corresponding Python code can be found in Appendix B. We did some testing for all parameters in Table 3.2. We find there are two parameter sets for which the found solution to the LP relaxation violates the constraint for some K and l. Specifically for $v = 40, k = 21, \lambda = 8, e = 2, s = 4$ and $v = 64, k = 33, \lambda = 12, e = 2, s = 4$. Note that both these parameter sets satisfy $\lambda = k/3 + 1$. However, we can still not solve the linear program for parameters satisfying $v = 4(k + 1) - 6\lambda$ after adding this constraint to the integer linear program. Again, we decide that we do not use these constraints any further.

Adding extra constraints does not seem to improve the performance time of the linear program. There are more results on Neumaier graphs that we have not considered modeling as extra constraints. For example Lemma 1.5 in [18] which is a result on regular cliques in edge-regular graphs. This is stated below as Lemma 3.4.5.

Lemma 3.4.5. *Let G be an edge-regular graph which is not complete and which has e-regular cliques of size s. Then two distinct regular cliques have at most e common points.*

This is just an example, there are various other results we have not looked at. However, we expect this approach to not be effective and so we will not pursue it further.

Chapter 4

Boolean satisfiability problem approach

Unfortunately, our current integer linear program fails to find any new strictly Neumaier graphs. So we try a different approach to find strictly Neumaier graphs. We model the problem as a boolean satisfiability problem.

Similar as for the integer program, we label the vertices of the Neumaier graph from 1 up to v. We define binary variables $x_{i,j}$ for $i, j \in \{1, \dots, v\}$ to represent whether there is an edge in the resulting graph between vertices i and j. If there is such an edge, $x_{i,j} = 1$, otherwise $x_{i,j} = 0$. We can interpret $x_{i,j}$ as true when it equals 1 and false when it equals 0. We then define a graph $G = (V, E)$ with vertices $V = \{1, \dots, v\}$ and edges $E = \{\{i, j\} \mid i, j \in \{1, \dots, v\}$ and $x_{i,j} = 1\}$. We now introduce conditions that should be satisfied by these variables x to make sure the resulting graph is a strictly Neumaier graph.

We make use of the following symbols: $\neg, \wedge, \vee$. Firstly, the expression $\neg x_{i,j}$ represents the negation of $x_{i,j}$ which means $x_{i,j}$ should be false. Next, the expression $x_{i,j} \wedge x_{i,m}$ represents the conjunction of $x_{i,j}$ and $x_{i,m}$ which means that both $x_{i,j}$ and $x_{i,m}$ should be true. Moreover, we use the following notation $\wedge_{j \in \{1,\dots,v\}}(x_{i,j})$ to denote the conjunction of all $x_{i,j}$ for $j \in \{1, \dots, v\}$, implying all $x_{i,j}$ should be true for $j \in \{1, \dots, v\}$. Lastly, the expression $x_{i,j} \vee x_{i,m}$ represents the disjunction of $x_{i,j}$ and $x_{i,m}$ which means that $x_{i,j}$ or $x_{i,m}$ should be true. Again we use the following notation $\vee_{j \in \{1,\dots,v\}}(x_{i,j})$ to denote the disjunction of all $x_{i,j}$ for $j \in \{1, \dots, v\}$, implying at least one $x_{i,j}$ should be true for $j \in \{1, \dots, v\}$.

4.1 Constraints

4.1.1 Preliminary constraints

We first introduce conditions on these variables to make sure they represent edges in a graph. We do not allow self loops in the graph. So there should be no edges between two the same vertices. For any vertex i, $x_{i,i}$ represents whether the vertex is connected to itself, so this should be false. We get the following condition

$$\wedge_{i \in \{1,\dots,v\}}(\neg x_{i,i}). \tag{4.1}$$

Edges are symmetric, meaning there is an edge between vertices i and j for two vertices $i, j \in \{1, \dots, v\}$ if and only if there is also an edge between j and i. Either there is an edge between vertex i and j and so there is also an edge between vertex j and i, or there is no edge between vertex i and j and so there is also no edge between vertex j and i. So for any 2 vertices i, j, either both $x_{i,j}$ and $x_{j,i}$ are false or they are both true. We get the following condition

$$\wedge_{i,j\in\{1,\ldots,v\}}((x_{i,j} \wedge x_{j,i}) \vee (\neg x_{i,j} \wedge \neg x_{j,i})). \tag{4.2}$$

With these conditions we have made sure that the variables indeed represent edges in a graph. Now we still have to make sure that the resulting graph is a strictly Neumaier graph.

4.1.2 s-clique

The resulting graph needs to have a clique of size s. Similar as in our integer linear program we can assume without loss of generality that the first s vertices form such a clique. This means there is an edge between every two distinct vertices in the first s vertices. So $x_{i,j}$ should be true for any two distinct vertices $i, j \in \{1, \ldots, s\}$. We get the following condition

$$\wedge_{i,j\in\{1,\ldots,s\}\text{s.t.}i\neq j}(x_{i,j}). \tag{4.3}$$

Note that assuming the first s vertices form this clique makes the condition relatively easy, this will also be helpful when adding the condition to ensure this is an e-regular clique.

4.1.3 e-regular clique

There should be an e-regular s-clique in the graph, meaning that every vertex outside the clique is connected to e vertices in the clique. Note that we can use that the first s vertices, 1 up to s, form a clique of size s. We will ensure this is an e-regular clique by imposing extra conditions on these vertices. We can split this into two conditions. Each vertex outside the clique should not be connected to more than e vertices in the clique and each vertex outside the clique should not be connected to less than e vertices in the clique. The first condition can be expressed using the following formula

$$\wedge_{i\in\{s+1,\ldots,v\}}(\wedge_{I\subseteq\{1,\ldots,s\}\text{s.t.}|I|=e+1}(\vee_{l\in I}(\neg x_{i,l}))). \tag{4.4}$$

This formula states that for any vertex i outside the clique, for all subsets I of $e+1$ vertices in the clique, at least one edge should be missing between vertex i and a vertex $l \in I$. This implies all vertices i outside the clique cannot have more than e neighbors in the clique. Similarly, the second condition can be expressed using the following formula

$$\wedge_{i\in\{s+1,\ldots,v\}}(\wedge_{I\subseteq\{1,\ldots,s\}\text{s.t.}|I|=s-e+1}(\vee_{l\in I}(x_{i,l}))). \tag{4.5}$$

This formula states that for any vertex i outside the clique, for all subsets I of $s-e+1$ vertices in the clique, at least one edge should be present between vertex i and a vertex $l \in I$. This implies all vertices i outside the clique cannot have less than e neighbors in the clique. Together, these formulas imply that all vertices i outside the clique have exactly e neighbors in the clique. For this we used that the first s vertices form a clique.

4.1.4 k-regular

The graph should be k-regular, meaning each vertex in the graph is connected to k other vertices. We can again split this into two conditions. Each vertex should not be connected to more than k other vertices and each vertex should not be connected to less than k other vertices. The first condition can be expressed using the following formula

$$\wedge_{i\in\{1,\ldots,v\}}(\wedge_{I\subseteq\{1,\ldots,v\}\text{s.t.}|I|=k+1}(\vee_{l\in I}(\neg x_{i,l}))). \tag{4.6}$$

This formula states that for any vertex i, for all subsets I of $k+1$ vertices, at least one edge should be missing between vertex i and a vertex $l \in I$. This implies all vertices i cannot have more than k neighbors. Similarly, the second condition can be expressed using the following formula

$$\wedge_{i\in\{1,\ldots,v\}}(\wedge_{I\subseteq\{1,\ldots,v\}\text{s.t.}|I|=v-k+1}(\vee_{l\in I}(x_{i,l}))). \tag{4.7}$$

This formula states that for any vertex i, for all subsets I of $v-k+1$ vertices, at least one edge should be present between vertex i and a vertex $l \in I$. This implies all vertices i cannot have less than k neighbors. Together, these formulas imply that all vertices have exactly k neighbors and thus guarantee that the graph is k-regular.

4.1.5 λ-edge-regular

The resulting graph should moreover be λ-edge-regular, meaning the graph is regular and all connected vertices have λ common neighbors. We already made sure the resulting graph is k-regular with conditions (4.6) and (4.7). We can make sure all adjacent vertices have exactly λ common neighbors by splitting this into two conditions. The first condition is that for every two vertices, they are either not connected or they do not have more than λ common neighbors. The second condition is that for every two vertices, they are either not connected or they do not have less than λ common neighbors. The first condition can be expressed using the following formula

$$\wedge_{i,j\in\{1,\ldots,v\}}(\neg x_{i,j} \vee (\wedge_{I\subseteq\{1,\ldots,v\}\text{s.t.}|I|=\lambda+1}(\vee_{m\in I}(\neg x_{i,m} \vee \neg x_{j,m})))). \tag{4.8}$$

This formula states that for any two vertices i and j, either $x_{i,j}$ is false, which means that vertices i and j are not connected, or for all subsets I of $\lambda+1$ vertices, there is at least one vertex $m \in I$ that is not a common neighbor of vertices i and j. So either $x_{i,m}$ is false or $x_{j,m}$ is false for some $m \in I$. This condition implies that all vertices i and j are either not adjacent, or they cannot have more than λ common neighbors. Similarly, the second condition can be expressed using the following formula

$$\wedge_{i,j\in\{1,\ldots,v\}}(\neg x_{i,j} \vee (\wedge_{I\subseteq\{1,\ldots,v\}\text{s.t.}|I|=v-\lambda+1}(\vee_{m\in I}(x_{i,m} \wedge x_{j,m})))). \tag{4.9}$$

This formula states that for any two vertices i and j, either $x_{i,j}$ is false, which means that vertices i and j are not connected, or for all subsets I of $v-\lambda+1$ vertices, there is at least one vertex $m \in I$ that is a common neighbor of vertices i and j. So both $x_{i,m}$ and $x_{j,m}$ are true for some $m \in I$. This condition implies that all vertices i and j are either not adjacent, or they cannot have less than λ common neighbors. Together, these formulas imply that all vertices i and j are either not connected or they have exactly λ common neighbors.

4.1.6 Strictly Neumaier graph

We now have all formulas to ensure the graph is a Neumaier graph. We are however looking for srictly Neumaier graphs. For the graph to be a strictly Neumaier graph, it should moreover not be co-edge-regular. This means that not all pairs of non-adjacent vertices have the same number of common neighbors. So there should at least be two pairs of non-adjacent vertices in the graph such that they have a different number of common neighbors. In Section 3.1.6 we found all 9 cases with vertices a, b, c, d we need to ensure the graph is not co-edge-regular. We saw for at least one such case we need that vertices a and b are not connected, vertices c and d are not connected, and a and b have less common neighbors than c and d. For given a, b, c, d, the following formula ensures these properties

$$(\neg x_{a,b}) \wedge (\neg x_{c,d}) \wedge (\vee_{K\in\{1,\dots,k\}}((\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=K}(\vee_{m\in I}(\neg x_{a,m} \vee \neg x_{b,m}))) \\ \wedge (\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=v-K+1}(\vee_{m\in I}(x_{c,m} \wedge x_{d,m}))))). \quad (4.10)$$

This formula states that for given vertices a, b, c, d, $x_{a,b}$ and $x_{c,d}$ are both false. This means there is no edge between vertices a and b and there is no edge between vertices c and d. Moreover, there is a number $K \in \{1, \dots, k\}$ such that for all subsets I of K vertices, there is at least one vertex $m \in I$ that is not a common neighbor of vertices a and b and for all subsets I of $v - K + 1$ vertices, there is at least one vertex $m \in I$ that is a common neighbor of c and d. This means that vertices a and b have less than K common neighbors and vertices c and d have at least K common neighbors for some $K \in \{1, \dots, k\}$. So this implies that vertices a and b have fewer common neighbors than vertices c and d. Note that we already ensured that all vertices have exactly k neighbors, so there exist no two vertices that have more than k common neighbors. Therefore we know that there exists no $K > k$ such that vertices a and b have less than K common neighbors and vertices c and d have at least K common neighbors. We need for at least one case with values a_n, b_n, c_n, d_n that the formula is true. So we can add the disjunction of (4.10) for all cases. This gives the following formula

$$\vee_{n\in\{1,\dots,9\}} ((\neg x_{a_n,b_n}) \wedge (\neg x_{c_n,d_n}) \wedge (\vee_{K\in\{1,\dots,k\}}((\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=K}(\vee_{m\in I}(\neg x_{a_n,m} \vee \neg x_{b_n,m}))) \\ \wedge (\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=v-K+1}(\vee_{m\in I}(x_{c_n,m} \wedge x_{d_n,m})))))). \quad (4.11)$$

We have successfully established all the necessary conditions to ensure that the resulting graph from our boolean satisfiability problem $G = (V, E)$ is a strictly Neumaier graph with parameters v, k, λ, e, and s. Here $V = \{1, \dots, v\}$ and $E = \{\{i, j\} \mid i, j \in \{1, \dots, v\} \text{ and } x_{i,j} = 1\}$. This graph consists of v vertices. It is non-complete as this follows from the parameters given in Table 2.1. It is edge regular as it is non-empty, k-regular, and any two adjacent vertices have λ common neighbors. Additionally, there exists a clique of size s, namely $\{1, \dots, s\}$, that is e-regular. Moreover, the resulting graph is not strongly regular.

We now know that a strictly Neumaier graph exists for the given parameters v, k, λ, e, s if and only if the corresponding boolean satisfiability problem is satisfiable because we have not imposed any additional assumptions on these graphs, aside from the usual 'without loss of generality'.

4.2 Implementation

We implement the boolean satisfiability problem in Python (version 3.7.6) using the Z3-solver library (version 4.12.1.0). Similar to our integer linear program, it is important to note that we can represent the variables in a more compact way to solve the SAT problem more efficiently. Instead of defining $x_{i,j}$ for all pairs of vertices, we only need to define $x_{i,j}$ for $j > i$, since $x_{i,i} = 0$ and $x_{j,i} = x_{i,j}$ for all $i, j \in \{1, \dots, v\}$. This allows us to transform all conditions accordingly. For instance, we can replace the k-regularity condition (3.5) with the following condition

$$\wedge_{i\in\{1,\dots,v\}}(\wedge_{I\subseteq\{1,\dots,v\}\setminus\{i\}\text{s.t.}|I|=k+1}((\vee_{l\in I\text{s.t.}l>i}(\neg x_{i,l})) \vee (\vee_{l\in I\text{s.t.}l<i}(\neg x_{l,i}))).$$

This way we only use variables $x_{i,j}$ for which $j > i$. Similarly, all other conditions can be replaced. We do not go into too much detail. So for our implementation we use this more compact way to describe the variables instead. The full boolean satisifiability problem can be found below. Here, condition (4.12) corresponds to condition (4.3) and ensures the first s vertices form a clique. Conditions (4.13) and (4.14) correspond to conditions (4.4) and (4.5) respectively and ensure the clique formed by the first s vertices is e-regular. Conditions (4.15) and (4.16) correspond to conditions (4.6) and (4.7) respectively and ensure the graph is k-regular. Conditions (4.17) and (4.18) correspond to conditions (4.8) and (4.9) respectively and ensure that the graph is λ-edge-regular. Condition (4.19) corresponds to condition (4.11) and ensures the graph is not strongly regular. The Python code for the boolean satisfiability problem to find strictly Neumaier graphs with variables and constraints as described can be found in Appendix B.

$$\wedge_{i,j\in\{1,\dots,s\}\text{s.t.}j>i}\ (x_{i,j}) \tag{4.12}$$

$$\wedge_{i\in\{s+1,\dots,v\}}\ (\wedge_{I\subseteq\{1,\dots,s\}\text{s.t.}|I|=e+1}(\vee_{l\in I}(\neg x_{l,i}))) \tag{4.13}$$

$$\wedge_{i\in\{s+1,\dots,v\}}\ (\wedge_{I\subseteq\{1,\dots,s\}\text{s.t.}|I|=s-e+1}(\vee_{l\in I}(x_{l,i}))) \tag{4.14}$$

$$\wedge_{i\in\{1,\dots,v\}}\ (\wedge_{I\subseteq\{1,\dots,v\}\setminus\{i\}\text{s.t.}|I|=k+1}((\vee_{l\in I\text{s.t.}l>i}(\neg x_{i,l})) \vee (\vee_{l\in I\text{s.t.}l<i}(\neg x_{l,i})))) \tag{4.15}$$

$$\wedge_{i\in\{1,\dots,v\}}\ (\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=v-k+1}((\vee_{l\in I\text{s.t.}l>i}(x_{i,l})) \vee (\vee_{l\in I\text{s.t.}l<i}(x_{l,i})))) \tag{4.16}$$

$$\begin{aligned}&\wedge_{i,j\in\{1,\dots,v\}\text{s.t.}j>i}\ (\neg x_{i,j} \vee (\wedge_{I\subseteq\{1,\dots,v\}\setminus\{i,j\}\text{s.t.}|I|=\lambda+1}((\vee_{m\in I\text{s.t.}m<i}(\neg x_{m,i} \vee \neg x_{m,j}))\vee\\ &\qquad(\vee_{m\in I\text{s.t.}i<m<j}(\neg x_{i,m} \vee \neg x_{m,j})) \vee (\vee_{m\in I\text{s.t.}m>j}(\neg x_{i,m} \vee \neg x_{j,m})))))\end{aligned} \tag{4.17}$$

$$\begin{aligned}&\wedge_{i,j\in\{1,\dots,v\}\text{s.t.}j>i}\ (\neg x_{i,j} \vee (\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=v-\lambda+1}((\vee_{m\in I\text{s.t.}m<i}(x_{m,i} \wedge x_{m,j}))\vee\\ &\qquad(\vee_{m\in I\text{s.t.}i<m<j}(x_{i,m} \wedge x_{m,j})) \vee (\vee_{m\in I\text{s.t.}m>j}(x_{i,m} \wedge x_{j,m})))))\end{aligned} \tag{4.18}$$

$$\begin{aligned}&\vee_{n\in\{1,\dots,9\}}\ ((\neg x_{a_n,b_n}) \wedge (\neg x_{c_n,d_n}) \wedge (\vee_{K\in\{1,\dots,k\}}((\wedge_{I\subseteq\{1,\dots,v\}\setminus\{a_n,b_n\}\text{s.t.}|I|=K}(\\ &\qquad(\vee_{m\in I\text{s.t.}m<a_n}(\neg x_{m,a_n} \vee \neg x_{m,b_n})) \vee (\vee_{m\in I\text{s.t.}a_n<m<b_n}(\neg x_{a_n,m} \vee \neg x_{m,b_n}))\vee\\ &\qquad\qquad(\vee_{m\in I\text{s.t.}m>b_n}(\neg x_{a_n,m} \vee \neg x_{b_n,m})))) \wedge (\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=v-K+1}(\\ &\qquad\qquad\qquad(\vee_{m\in I\text{s.t.}m<c_n}(x_{m,c_n} \wedge x_{m,d_n})) \vee (\vee_{m\in I\text{s.t.}c_n<m<d_n}(x_{c_n,m} \wedge x_{m,d_n}))\vee\\ &\qquad\qquad\qquad\qquad(\vee_{m\in I\text{s.t.}m>d_n}(x_{c_n,m} \wedge x_{d_n,m})))))))\end{aligned} \tag{4.19}$$

4.3 Results

We were able to solve the SAT problem only for the smallest parameters that can be found in Table 2.1, namely $v = 16, k = 9, \lambda = 4, e = 2, s = 4$. Using these parameters, we obtained the strictly Neumaier graph shown in Figure 4.1, and its corresponding adjacency matrix A_2, which is presented below.

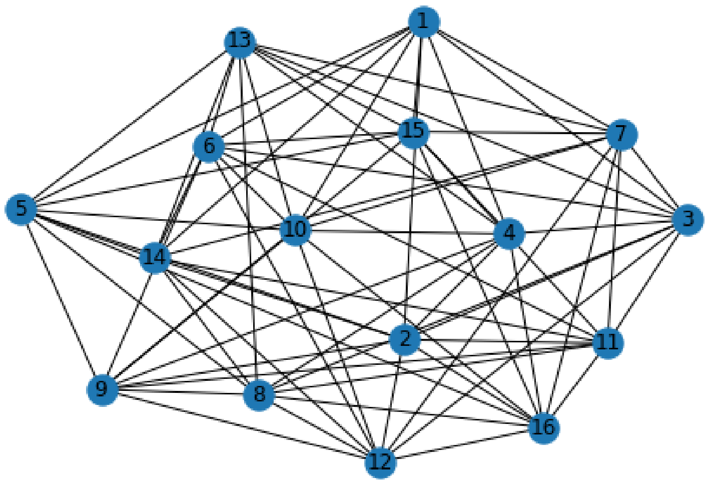


Figure 4.1: Strictly Neumaier graph with parameters $v = 16, k = 9, \lambda = 4, e = 2, s = 4$ found after solving SAT problem with adjacency matrix A_2.

$$A_2 = \begin{pmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0
\end{pmatrix}$$

We already found a strictly Neumaier graph with parameters $v = 16, k = 9, \lambda = 4, e = 2, s = 4$ using our integer linear program as seen in Chapter 3. Specifically, we found a graph that is isomorphic to the strictly Neumaier graph we found in Chapter 3 and therefore also isomorphic to the strictly Neumaier graph mentioned in [9] with the same parameters. It took 20 minutes and 14.1 seconds to find this graph by running our implementation of the SAT problem. This is significantly slower than the integer linear program as seen in Chapter 3. For the next three parameter sets in Table 2.1, the program did not solve within the given time limit of 5 hours. We expect that the program would also not solve within the time limit for any higher parameters as the number of variables increases quadratically with the number of vertices v.

Chapter 5

How close is a graph to be a Neumaier graph

We saw that both the integer linear program as specified in Section 3.2 and the boolean satisfiability problem as specified in Section 4.2 fail to find new strictly Neumaier graphs. Therefore we will now look into whether we can find graphs close to Neumaier graphs. We first define what it means for a graph to be close to a Neumaier graph.

5.1 Distance to Neumaier graph

We did some experimenting and found that for both the integer program and the SAT problem, the constraints that ensure all adjacent vertices in the graph have exactly λ common neighbors make the problem particularly difficult. For the integer program these are constraints (3.21) and (3.22). For the SAT problem these are conditions (4.17) and (4.18). We therefore define the distance between a Neumaier graph and another graph based on how many vertices do not satisfy this λ-edge-regularity constraint. As mentioned, $NG(v, k, \lambda, e, s)$ is the collection of Neumaier graphs with parameters v, k, λ, e, s. We define the distance between a k-regular graph $G = (V, E)$ with v vertices and an e-regular s-clique from a Neumaier graph with parameters v, k, λ, e, s in two different ways. These two distances correspond to the L_0-norm and the L_1-norm respectively. The L_0 norm of a vector is the number of non-zero entries while the L_1-norm of a vector is the sum of the absolute values of the entries. For the first distance we count the number of pairs of adjacent vertices $i, j \in \{1, \ldots, v\}$ that do not have exactly λ common neighbors. We let $N(i)$ denote the neighborhood of vertex i, i.e. the set of all vertices connected to i and similarly we let $N(j)$ denote the neighborhood of vertex j. We define the first distance below.

$$d_0(G, NG(v, k, \lambda, e, s)) = \mid \{\{i, j\} \in E \text{ s.t. } |N(i) \cap N(j)| \neq \lambda\} \mid$$

This distance d_0 equals the L_0-norm of the difference between the vector that consists of the number of common neighbors for all adjacent pairs of vertices and the vector for which each entry is λ. Namely, the L_0-norm of this vector is exactly the number of pairs of adjacent vertices which do not have λ common neighbors. For the second distance, we sum over each pair of adjacent vertices $i, j \in \{1, \ldots, v\}$ and calculate the difference between the number of common neighbors and λ. This distance is defined below.

$$d_1(G, NG(v,k,\lambda,e,s)) = \sum_{\{i,j\}\in E} ||N(i)\cap N(j)| - \lambda|$$

This distance d_1 equals the L_1-norm of the difference between the vector that consists of the number of common neighbors for all adjacent pairs of vertices and the vector for which each entry is λ. Namely, the L_1-norm of this vector is exactly the sum of all the differences between the number of common neighbors of pairs of adjacent vertices and λ. Note that for both distances, if the graph G is a Neumaier graph with parameters v, k, λ, e, s, the distance to $NG(v,k,\lambda,e,s)$ is 0.

5.2 Lagrangian relaxation of the integer linear program

As mentioned, we can still only solve the linear program for the smallest number of vertices, $v = 16$. We tried removing some constraints from the integer linear program with variables and constraints as specified in Section 3.2 and find that removing constraints (3.21) and (3.22) make the the program run significantly faster. We repeat these constraints here.

$$\sum_{m\in\{1,\dots,v\}\setminus\{i,j\}} y_{i,j,m} \leq \lambda + M(1-x_{i,j}) \text{ for } i,j\in\{1,\dots,v\} \text{ s.t. } j>i$$
$$\sum_{m\in\{1,\dots,v\}\setminus\{i,j\}} y_{i,j,m} \geq \lambda - M(1-x_{i,j}) \text{ for } i,j\in\{1,\dots,v\} \text{ s.t. } j>i$$

These are the constraints that guarantee the resulting graph of the linear program is λ-edge-regular. Now, instead of solving the original integer linear program, we can remove these difficult constraints and move them to the objective function and solve the program with the following objective function

$$\min \sum_{i=1}^{v}\sum_{j=i+1}^{v} \max\{\sum_{m\in\{1,\dots,v\}\setminus\{i,j\}} y_{i,j,m} - \lambda - M(1-x_{i,j}), \lambda - M(1-x_{i,j}) - \sum_{m\in\{1,\dots,v\}\setminus\{i,j\}} y_{i,j,m}, 0\}. \tag{5.1}$$

So for given parameters, a strictly Neumaier graph exists if and only if this new integer program, with objective function (5.1) and variables and constraints as specified in Section 3.2 except for the λ-edge-regularity constraints, has an optimal solution with value 0. Namely, then for all $i, j \in \{1,\dots,v\}$ such that $j > i$, $\sum_{m\in\{1,\dots,v\}\setminus\{i,j\}} y_{i,j,m} \leq \lambda + M(1-x_{i,j})$ and $\sum_{m\in\{1,\dots,v\}\setminus\{i,j\}} y_{i,j,m} \geq \lambda - M(1-x_{i,j})$, so each value in the sum is 0. We can linearize this objective function (5.1) and solve the new integer linear program. We implement this in Python, the code can be found in Appendix B. This still does not give new insights as the program only solves for the smallest parameter set in Table 2.1. Again, we find a graph isomorphic to the graphs seen in Chapter 3 and Chapter 4. It is worth noting that the smallest graph was found in 13.5 seconds which is faster than the original integer program which solved in 31.6 seconds. The program however still does not solve for the next three parameter sets in Table 2.1 within a time limit of 5 hours. Instead, we can penalize violations of the constraints (3.21), (3.22) in the objective function using the Lagrangian relaxation method as described in [4]. This method may be useful to solve an integer program that consists of both nice constraints and complicating constraints, which is indeed the case in our problem. However, the Lagrangian approach only provides a bound on the optimal value of the original integer linear program. We are not interested in the optimal value of the integer program as the objective function is constant and we only want to find a feasible solution. In hindsight, this approach does not make sense as we are interested in whether the problem is feasible and if so what a feasible solution would be, instead of the optimal value. We therefore do not go into more detail. We have nevertheless tested the Lagrangian relaxation approach and implemented it in Python. The code can be found in Appendix B. Indeed it appears using the method of Lagrangian relaxation is not the right way to tackle our problem as we do not find graphs close to Neumaier graphs according to the distances as defined before.

5.3 Gradually solving the boolean satisfiability problem

We can only solve the boolean satisfiability problem for the smallest number of vertices, $v = 16$. Similar as to the integer linear program, we tried removing some conditions from the SAT problem as specified in Section 4.2 and we find that the conditions (4.17), (4.18), and (4.19) take the longest to add to the program. We repeat these conditions here.

$$\begin{aligned}
&\wedge_{i,j\in\{1,\dots,v\}\text{s.t.}j>i} (\neg x_{i,j} \vee (\wedge_{I\subseteq\{1,\dots,v\}\setminus\{i,j\}\text{s.t.}|I|=\lambda+1}((\vee_{m\in I\text{s.t.}m<i}(\neg x_{m,i} \vee \neg x_{m,j}))\vee \\
&\qquad (\vee_{m\in I\text{s.t.}i<m<j}(\neg x_{i,m} \vee \neg x_{m,j})) \vee (\vee_{m\in I\text{s.t.}m>j}(\neg x_{i,m} \vee \neg x_{j,m})))))
\end{aligned}$$

$$\begin{aligned}
&\wedge_{i,j\in\{1,\dots,v\}\text{s.t.}j>i} (\neg x_{i,j} \vee (\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=v-\lambda+1}((\vee_{m\in I\text{s.t.}m<i}(x_{m,i} \wedge x_{m,j}))\vee \\
&\qquad (\vee_{m\in I\text{s.t.}i<m<j}(x_{i,m} \wedge x_{m,j})) \vee (\vee_{m\in I\text{s.t.}m>j}(x_{i,m} \wedge x_{j,m})))))
\end{aligned}$$

$$\begin{aligned}
&\vee_{n\in\{1,\dots,9\}} ((\neg x_{a_n,b_n}) \wedge (\neg x_{c_n,d_n}) \wedge (\vee_{K\in\{1,\dots,k\}}((\wedge_{I\subseteq\{1,\dots,v\}\setminus\{a_n,b_n\}\text{s.t.}|I|=K}(\\
&\qquad (\vee_{m\in I\text{s.t.}m<a_n}(\neg x_{m,a_n} \vee \neg x_{m,b_n})) \vee (\vee_{m\in I\text{s.t.}a_n<m<b_n}(\neg x_{a_n,m} \vee \neg x_{m,b_n}))\vee \\
&\qquad\qquad (\vee_{m\in I\text{s.t.}m>b_n}(\neg x_{a_n,m} \vee \neg x_{b_n,m})))) \wedge (\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=v-K+1}(\\
&\qquad\qquad\qquad (\vee_{m\in I\text{s.t.}m<c_n}(x_{m,c_n} \wedge x_{m,d_n})) \vee (\vee_{m\in I\text{s.t.}c_n<m<d_n}(x_{c_n,m} \wedge x_{m,d_n}))\vee \\
&\qquad\qquad\qquad\qquad (\vee_{m\in I\text{s.t.}m>d_n}(x_{c_n,m} \wedge x_{d_n,m})))))))
\end{aligned}$$

The first two formulas guarantee that the resulting graph of the SAT problem is λ-edge-regular while the third formula guarantees that it is not strongly regular. To simplify the problem-solving process, we can remove these difficult conditions and repeatedly solve the SAT problem while adding more conditions based on which ones are not satisfied. For the λ-edge-regular conditions, we can for example solve the problem, then check which pairs of vertices do not satisfy the condition and for such pairs we can introduce extra conditions so they satisfy the λ-edge-regular property and then solve the problem again. We can repeat this until we find a λ-edge-regular graph which also satisfies the other conditions. However, dealing with the condition that ensures the graph is not strongly regular is more complex. Therefore we will not add this condition. Instead, we can continue solving the problem until we obtain a graph that is λ-edge-regular and hoping for a favorable outcome not strongly regular.

So we can solve the SAT problem as defined in Section 4.2, and add conditions (5.2) and (5.3) for a random edge $\{i, j\}$ such that vertices i and j do not have λ common neighbors in the solution of the SAT problem. We will continue this process until we find a graph that is λ-edge-regular and, ideally, not strongly regular.

$$\neg x_{i,j} \vee (\wedge_{I\subseteq\{1,\dots,v\}\setminus\{i,j\}\text{s.t.}|I|=\lambda+1}((\vee_{m\in I\text{s.t.}m<i}(\neg x_{m,i} \vee \neg x_{m,j}))\vee (\vee_{m\in I\text{s.t.}i<m<j}(\neg x_{i,m} \vee \neg x_{m,j})) \vee (\vee_{m\in I\text{s.t.}m>j}(\neg x_{i,m} \vee \neg x_{j,m})))) \tag{5.2}$$

$$\neg x_{i,j} \vee (\wedge_{I\subseteq\{1,\dots,v\}\text{s.t.}|I|=v-\lambda+1}((\vee_{m\in I\text{s.t.}m<i}(x_{m,i} \wedge x_{m,j}))\vee (\vee_{m\in I\text{s.t.}i<m<j}(x_{i,m} \wedge x_{m,j})) \vee (\vee_{m\in I\text{s.t.}m>j}(x_{i,m} \wedge x_{j,m})))) \tag{5.3}$$

We expect this approach to solve faster than adding all conditions at once as we did in Section 4.2. Namely, removing the difficult conditions makes the problem easier to solve. Then in each iteration we expect that the solver remembers its previous solutions and thus solve the problem more efficiently. Moreover, by gradually adding conditions, we can stop the process as soon as we find a strictly Neumaier graph, rather than adding more unnecessary conditions. Also, we can stop the process as soon as the SAT problem is not satisfiable anymore as then we know there exists no Neumaier graph with the given parameters.

We have implemented this way of solving the SAT problem in Python. The code can be found in Appendix B. It solves in 45 iterations for the smallest parameters as found in Table 2.1, which are $v = 16, k = 9, \lambda = 4, e = 2, s = 4$. So the λ-edge-regularity condition is added for 45 pairs of vertices instead of all $\binom{16}{2} = 120$ pairs of vertices. Here $\binom{16}{2}$ represents the binomial coefficient, which equals the number of ways to choose 2 vertices out of a set of 16 vertices. We again find a strictly Neumaier graph isomorphic to the graphs found before in Chapter 3 and Chapter 4 with these parameters. It only took 4 minutes and 54.0 seconds to find this graph by gradually adding more conditions compared to the 20 minutes and 14.1 seconds which it took to find this graph with the regular SAT problem.

We already found this graph. Instead we now try this method for the parameters $v = 21, k = 14, \lambda = 9, e = 4, s = 7$. This is the second parameter set in Table 2.1. For these parameters it is known that no strictly Neumaier graph exists. We are however wondering whether we can find that this graph does not exist using the SAT problem. We run it for 50 iterations. The adjacency matrices corresponding to the first solution and the solution after 50 iterations can be found in Appendix A, these are A_3 and A_4 respectively. Note that both graphs of these adjacency matrices have v vertices, are k-regular and contain an e-regular s-clique. But they may not be λ-edge-regular and they may be co-edge-regular. Let G_3 denote the graph with adjacency matrix A_3, similarly let G_4 denote the graph with adjacency matrix A_4. Then $d_0(G_3, NG(21, 14, 9, 4, 7)) = 95$ and $d_0(G_4, NG(21, 14, 9, 4, 7)) = 78$. So we see that for graph G_3 there are more adjacent vertices that do not have exactly λ common neighbors than for graph G_4. Also, $d_1(G_3, NG(21, 14, 9, 4, 7)) = 125$ and $d_1(G_4, NG(21, 14, 9, 4, 7)) = 88$. So we see that for graph G_3 the sum of the difference of number of common neighbors with λ is larger than for graph G_4. We see that graph G_4 is closer to be a Neumaier graph than graph G_3 according to both defined distances. So we find that by using this method, we indeed gradually get closer to finding a Neumaier graph. However, we are looking for strictly Neumaier graphs.

A disadvantage of this method is that we have not taken into account that the graph should not be strongly regular. So it may be the case that we would eventually find a Neumaier graph with parameters $v = 21, k = 14, \lambda = 9, e = 4, s = 7$ if such a graph exists, and so we could still not conclude that no strictly Neumaier graph exists with these parameters. Moreover, for parameters for which a strictly Neumaier graph exists, it may be the case that we would find a strongly regular graph that is λ-edge-regular at some point during the process. Then, in the next iteration no more conditions are added since there are no pairs of vertices $\{i, j\}$ in the graph such that i and j do not have λ common neighbors. The problem is solved again and we expect it finds the same graph since no conditions are added. This would mean that we would keep finding the same Neumaier graph that is not a strictly Neumaier graph. Another disadvantage of this method is that it still takes a long time to add all conditions. This is mainly because we iterate over all subsets of sizes $k + 1$, $v - k + 1$, $\lambda + 1$, and $v - \lambda + 1$ of a set of size v.

We find that using this method, we get closer towards a Neumaier graph according to both distances as defined before. So we could possibly find whether a strictly Neumaier graph exists with parameters in Table 2.1. However, since adding all conditions takes a long time, by gradually solving the SAT problem we have not managed to find any other strictly Neumaier graphs except for the smallest graph.

Chapter 6

Conclusions and future work

The goal of this thesis was to identify new methods to find strictly Neumaier graphs with parameters in Table 2.1. We have thus formulated the problem of finding a strictly Neumaier graph as an integer linear program and as a boolean satisfiability problem. We implemented and tested both methods. We found that for both of these methods, we are able to find the smallest strictly Neumaier graph. However, this graph was already known to exist. We also found that for larger parameters, both programs do not solve within the given time limit of 5 hours. This is because both problems are NP-complete. We then tried to add constraints to the integer program to cut off more non-integer solutions, however this approach did not lead to promising results. So instead we tried to find methods with which we could approach a strictly Neumaier graph. We implemented the Lagrangian relaxation of the integer program and found that this approach did not work in our case since the integer program has a constant objective function. We also tried to incrementally solve the boolean satisfiability problem by gradually adding more conditions. We found that even though this is faster than the original boolean satisfiability problem and it does work, we can still not find other graphs within the time limit of 5 hours.

Our research has certain limitations that should be acknowledged. First off all, we have exclusively used Python for solving the programs, in particular we have only considered Gurobi as integer program solver and Z3-solver as SAT solver. We have not explored any other solvers that may offer alternative approaches. Additionally, we have focused on linearizing all constraints in the integer program rather than exploring the possibility of developing a nonlinear integer program. These limitations highlight potential directions for further investigation and expansion of our research. Moreover, even though we have found two methods with which we can find whether strictly Neumaier graphs exist for given parameters, we were not able to find new results on Neumaier graphs. This was due to the limitations of time and computational power. Since the integer linear program solved significantly faster for the smallest strictly Neumaier graph, we expect that this method is more likely to solve faster for larger parameters as well. For future research both methods should be tested for a longer period of time to see whether this gives new outcomes.

These methods can be explored more beyond finding new strictly Neumaier graphs as well. Similar integer programs or SAT problems could for example be developed to find other graphs that satisfy given constraints. As seen, strictly Neumaier graphs are a family of graphs that are very constrained. If the methods we explored would be used for graphs that have to satisfy fewer constraints, they might be more efficient.

Additionally, for future research, it should be investigated whether either the integer program or the SAT problem can be solved more efficiently. One idea worth investigating is the application of symmetry breaking techniques, which we have not yet explored. For example, in both problems all vertices in the clique are interchangeable and all vertices outside the clique are interchangeable. Similar methods as in [3] could be used to reduce the search space by applying symmetry breaking techniques both in the integer linear program and the SAT problem. By incorporating symmetry breaking into the programs, it may be possible to significantly improve the efficiency of the methods and find for other parameters in Table 2.1 whether a strictly Neumaier graph exists.

Appendices

Appendix A

Adjacency Matrices

$$A_3 = \begin{pmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{pmatrix}$$

$$A_4 = \begin{pmatrix}
0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0
\end{pmatrix}$$

Appendix B

Code

The code can additionally be found *here*.

```python
# import packages
from gurobipy import *
import networkx as nx
import numpy as np
from itertools import combinations
import matplotlib.pyplot as plt
from z3 import *
import random


def find_adjacencymatrix(edges, v):
    """"Find adjacency matrix of graph given its edges and the number of vertices."""

    adjacency_matrix = np.zeros((v, v), dtype=int)

    for (i,j) in edges:
        adjacency_matrix[i][j] = 1
        adjacency_matrix[j][i] = 1
    return adjacency_matrix


def find_edges(adjacency_matrix):
    """Find edges of graph given its adjacency matrix."""

    edges = []
    v = len(adjacency_matrix)

    for i in range(v):
        for j in range(i):
            if adjacency_matrix[i][j] != 0:
                edges.append((i, j))

    return edges


def is_Neumaier_graph(A, v, k, l, e, s):
    """"Check whether graph with adjacency matrix A is a Neumaier graph with
    parameters v, k, l, e, s."""

    G = nx.from_numpy_matrix(A) # graph with adjacency matrix A

    if nx.is_directed(G):
        return False    # directed graph

    if nx.number_of_selfloops(G) != 0:
        return False    # graph has self loops

    if nx.number_of_nodes(G) != v:
        return False    # not v vertices

    if nx.degree_histogram(G)[k] != v:
        return False    # not k-regular

    for (i,j) in nx.edges(G):
        if len(list(nx.common_neighbors(G, i, j))) != l:
            return False    # not l-edge-regular

    for clique in nx.find_cliques(G):
        nr_nodes = 0
        if len(clique) == s:    # only consider s-cliques
            for i in set(range(v)) - set(clique):
                if sum(A[i][j] for j in clique) == e:
                    nr_nodes += 1
            if nr_nodes == v - s:
                return True # e-regular clique

    return False    # no e-regular clique


def is_strictlyNeumaier_graph(A, v, k, l, e, s):
    """"Check whether graph with adjacency matrix A is a strictly Neumaier graph with
    parameters v, k, l, e, s."""

    if not is_Neumaier_graph(A, v, k, l, e, s):
        return False    # not Neumaier graph
    G = nx.from_numpy_matrix(A)
    if nx.is_strongly_regular(G):
        return False    # strongly regular
    return True # strictly Neumaier graph


def print_graph(edges, parameters):
    """Print edges and plot the graph with these edges."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters

    print(edges)    # print edges

    G = nx.Graph()
    G.add_nodes_from(range(1, v + 1))
    G.add_edges_from([(i + 1,j + 1) for (i,j) in edges])
    nx.draw(G, with_labels=True)    # plot the graph
```

3.2 Implementation of the integer linear program

```
def initialize_ILP(parameters):
    """Create integer program with binary variables x, y, z."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters

    model = Model() # create model

    x = {}  # create x variables
    for i in range(v):
        for j in range(i):
           x[i,j] = model.addVar(vtype = GRB.BINARY, name = "x_%i_%i" % (i, j))

    y = {}  # create y variables
    for i in range(v):
        for j in range(i):
            for m in set(range(v)) - {i, j}:
                y[i,j,m] = model.addVar(vtype = GRB.BINARY, name = "y_%i_%i_%i" % (i, j, m))

    z = model.addVars(9, vtype = GRB.BINARY, name = 'z')    # create z variables

    model.setObjective(0, GRB.MINIMIZE) # set objective to 0

    return model, x, y, z

```

```python

def initialize_relaxed_ILP(parameters):
    """Create relaxed program with variables x, y, z."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters

    model = Model() # create model

    x = {}  # create x variables
    for i in range(v):
        for j in range(i):
           x[i,j] = model.addVar(lb = 0, ub = 1, name = "x_%i_%i" % (i, j))

    y = {}  # create y variables
    for i in range(v):
        for j in range(i):
            for m in set(range(v)) - {i, j}:
                y[i,j,m] = model.addVar(lb = 0, ub = 1, name = "y_%i_%i_%i" % (i, j, m))

    z = model.addVars(9, lb = 0, ub = 1, name = 'z')    # create z variables

    model.setObjective(0, GRB.MINIMIZE) # set objective to 0

    return model, x, y, z


def add_Neumaier_constraints(model, x, y, z, parameters):
    """Add linear constraints to the integer program so the resulting graph is a Neumaier graph."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters
    M = v   # big M

    # the first s vertices form a clique
    model.addConstrs((x[i,j] == 1 for i in range(s) for j in range(i)))

    # the graph is k regular
    model.addConstrs((quicksum(x[i,j] for j in range(i)) +
        sum(x[j,i] for j in range(i + 1, v)) == k for i in range(v)))

    # the clique is e regular
    model.addConstrs((quicksum(x[i,j] for j in range(s)) == e for i in range(s, v)))

    # the y variables are products of the x variables
    model.addConstrs((y[i,j,m] >= x[i,m] + x[j,m] - 1 for i in range(v) for j in range(i) for m in range(j)))
    model.addConstrs((y[i,j,m] >= x[i,m] + x[m,j] - 1 for i in range(v) for j in range(i) for m in range(j+1, i)))
    model.addConstrs((y[i,j,m] >= x[m,i] + x[m,j] - 1 for i in range(v) for j in range(i) for m in range(i+1, v)))

    model.addConstrs((y[i,j,m] <= x[i,m] for i in range(v) for j in range(i) for m in set(range(i)) - {j}))
    model.addConstrs((y[i,j,m] <= x[m,i] for i in range(v) for j in range(i) for m in range(i+1,v)))
    model.addConstrs((y[i,j,m] <= x[j,m] for i in range(v) for j in range(i) for m in range(j)))
    model.addConstrs((y[i,j,m] <= x[m,j] for i in range(v) for j in range(i) for m in set(range(j+1,v)) - {i}))

    # the graph is l edge regular
    model.addConstrs((quicksum(y[i,j,m] for m in set(range(v)) - {i, j}) <= l + M * (1 - x[i,j])
        for i in range(v) for j in range(i)))
    model.addConstrs((quicksum(y[i,j,m] for m in set(range(v)) - {i, j}) >= l - M * (1 - x[i,j])
        for i in range(v) for j in range(i)))


def add_strictlyNeumaier_constraint(model, x, y, z, parameters):
    """Add linear constraints to the integer program so the resulting graph is not co-edge-regular."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters
    M = v   # big M

    cases = [(0, s, 1, s + 1), (0, s, 0, s + 1), (0, s, 1, s), (0, s, s + 1, s + 2), (0, s, s, s + 1),
    (s + 1, s + 2, 0, s), (s, s + 1, 0, s), (s, s + 1, s + 2, s + 3), (s, s + 1, s + 1, s + 2)]
    model.addConstr(quicksum(z[h] for h in range(9)) >= 1)  # at least one of the cases holds

    for i in range(9):  # z variables represent whether a case holds
```

```
        a, b, c, d = cases[i]
        model.addConstr(x[b,a] <= (1 - z[i]))
        model.addConstr(x[d,c] <= (1 - z[i]))
        model.addConstr(quicksum(y[b,a,m] for m in set(range(v)) - {a, b}) -
            sum(y[d,c,m]for m in set(range(v)) - {c, d}) <= M * (1 - z[i]) - 1)


def solve_model_ILP(model, x, y, z, parameters):
    """Solve the integer program and return x variables."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters

    model.optimize()    # optimize model
    if model.status == GRB.Status.OPTIMAL:   # feasible
        x_values = model.getAttr('x', x)   # get values of x corresponding to edges in resulting Neumaier graph
    else:   # infeasible
        x_values = None
    return x_values

def solve_ILP_NG(v, k, l, e, s):
    """Find whether a strictly Neumaier graph exists with given parameters v, k, l, e, s
    by solving corresponding integer linear program and print the graph if it exists."""

    parameters = v, k, l, e, s

    model, x, y, z = initialize_ILP(parameters) # create model
    add_Neumaier_constraints(model, x, y, z, parameters)    # Neumaier graph
    add_strictlyNeumaier_constraint(model, x, y, z, parameters) # not strongly regular
    x_values = solve_model_ILP(model, x, y, z, parameters)  # solve the program and get x variables
    if x_values is None:   # infeasible
        print('There exists no strictly Neumaier graph with the given parameters:
            v = {}, k = {}, l = {}, e = {}, s = {}.'.format(v, k, l, e, s))
    else:   # feasible
        edges = [(i,j) for i in range(v) for j in range(i) if x_values[i, j] >= 0.99]
        print_graph(edges, parameters)  # strictly Neumaier graph
```

3.4 Adding further constraints to the integer program

```
def check_maximumclique_constraint(v, k, l, e, s):
    """Check whether any solutions to the LP relaxation of the integer program that finds a strictly Neumaier graph
    with parameters v, k, l, e, s are cut off by adding the constraint on the maximum clique size."""

    parameters = v, k, l, e, s

    model, x, y, z = initialize_relaxed_ILP(parameters)   # create LP relaxation
    model.setParam('OutputFlag', 0)
    model.setParam('Method', 0)

    add_Neumaier_constraints(model, x, y, z, parameters)    # Neumaier graph
    add_strictlyNeumaier_constraint(model, x, y, z, parameters) # not strongly regular

    x_values = solve_model_ILP(model, x, y, z, parameters) # solve LP relaxation and get x values

    maximum = 0 # initialize maximum to 0
    for subset in combinations(range(v), s+1):  # iterate over subsets S
        sum_value = sum(x_values[i,j] for i in subset for j in subset if i > j)
        if sum_value > maximum:
            maximum = sum_value

    return maximum > s * ((s + 1) / 2) - 1  # True if solution is cut off, False if not


def check_triangle_constraint(v, k, l, e, s):
    """Check whether any solutions to the LP relaxation of the integer program that finds
    a strictly Neumaier graph with parameters v, k, l, e, s are cut off by adding the
    constraint that every triangle belongs to at most one K4."""

    assert v == 3 * (k - l) + 1, 'This constraint only holds for parameters such that v = 3(k - l) + 1.'
    parameters = v, k, l, e, s
    M = v

    model, x, y, z = initialize_relaxed_ILP(parameters)   # create LP relaxation
    model.setParam('OutputFlag', 0)
    model.setParam('Method', 0)

    add_Neumaier_constraints(model, x, y, z, parameters)    # Neumaier graph
    add_strictlyNeumaier_constraint(model, x, y, z, parameters) # not strongly regular

    x_values = solve_model_ILP(model, x, y, z, parameters)  # solve LP relaxation and get x values

    for i in range(v):  # copy x values for j > i
        for j in range(i+1, v):
            x_values[i,j] = x_values[j,i]

    for triangle in combinations(range(v), 3):  # iterate over all triangles
        if (sum(x_values[i,j] for i in triangle for j in (set(range(v)) - set(triangle))) >
            2 * (v - 4) + 3 + M * (3 - (sum(x_values[i,j] for i,j in combinations(triangle, 2))))):
            return True # solution is cut off
    return False    # solution is not cut off


def check_strongertriangle_constraint(v, k, l, e, s):
    """Check whether any solutions to the LP relaxation of the integer program that finds
    a strictly Neumaier graph with parameters v, k, l, e, s are cut off by adding the stronger
    constraint that every triangle belongs to at most one K4."""

    assert v == 3 * (k - l) + 1, 'This constraint only holds for parameters such that v = 3(k - l) + 1.'
    parameters = v, k, l, e, s
    M = v

    model, x, y, z = initialize_relaxed_ILP(parameters)   # create LP relaxation
    model.setParam('OutputFlag', 0)
    model.setParam('Method', 0)

    w = {}  # add w variables
    for i in range(v):
        for j in range(i):
            for m in range(j):
                for t in set(range(v)) - {i,j,m}:
                        w[i,j,m,t] = model.addVar(lb = 0, ub = 1, name = "w_%i_%i_%i_%i" % (i, j, m, t))

    add_Neumaier_constraints(model, x, y, z, parameters)    # Neumaier graph
    add_strictlyNeumaier_constraint(model, x, y, z, parameters) # not strongly regular

    for triangle in combinations(range(v), 3):  # add constraints on w variables
        m,j,i = triangle
        model.addConstrs(w[i,j,m,t] <= x[i,t] for t in (set(range(i)) - {j,m}))
        model.addConstrs(w[i,j,m,t] <= x[t,i] for t in (set(range(i+1,v))))
        model.addConstrs(w[i,j,m,t] <= x[j,t] for t in (set(range(j)) - {m}))
        model.addConstrs(w[i,j,m,t] <= x[t,j] for t in (set(range(j+1,v)) - {i}))
        model.addConstrs(w[i,j,m,t] <= x[m,t] for t in (set(range(m))))
        model.addConstrs(w[i,j,m,t] <= x[t,m] for t in (set(range(m+1,v)) - {i,j}))

        model.addConstrs(w[i,j,m,t] >= x[i,t]+x[j,t]+x[m,t] - 2 for t in range(m))
        model.addConstrs(w[i,j,m,t] >= x[i,t]+x[j,t]+x[t,m] - 2 for t in range(m+1, j))
        model.addConstrs(w[i,j,m,t] >= x[i,t]+x[t,j]+x[t,m] - 2 for t in range(j+1, i))
        model.addConstrs(w[i,j,m,t] >= x[t,i]+x[t,j]+x[t,m] - 2 for t in range(i+1, v))

    x_values = solve_model_ILP(model, x, y, z, parameters)  # solve LP relaxation and get x values
    for i in range(v):  # copy x values for j > i
        for j in range(i+1, v):
            x_values[i,j] = x_values[j,i]
    w_values = model.getAttr('x', w)    # get w values

    for triangle in combinations(range(v), 3):  # iterate over all triangles
        m,j,i = triangle
        if (sum(w_values[i,j,m,t] for t in (set(range(v)) - {i,j,m})) >
            1 + M * (3 - x_values[i,j] - x_values[i,m] - x_values[j,m])):
            return True # solution is cut off
    return False    # solution is not cut off

```

```python

def check_K4_constraint(v, k, l, e, s):
    """Check whether any solutions to the LP relaxation of the integer program that finds
    a strictly Neumaier graph with parameters v, k, l, e, s are cut off by adding the
    constraint that every vertex is adjacent to 1 or 2 vertices of every K4."""

    assert v == 4 * (k + 1) - 6 * l, 'This constraint only holds for parameters such that v = 4(k + 1) - 6l.'
    parameters = v, k, l, e, s
    M = v

    model, x, y, z = initialize_relaxed_ILP(parameters)   # create LP relaxation
    model.setParam('OutputFlag', 0)
    model.setParam('Method', 0)

    add_Neumaier_constraints(model, x, y, z, parameters)    # Neumaier graph
    add_strictlyNeumaier_constraint(model, x, y, z, parameters) # not strongly regular

    x_values = solve_model_ILP(model, x, y, z, parameters)  # solve LP relaxation and get x values

    for i in range(v):  # copy x values for j > i
        for j in range(i+1, v):
            x_values[i,j] = x_values[j,i]

    if l == k/3 + 1:
        for i in range(v):
            for subset in combinations(set(range(v)) - {i}, 4): # iterate over all subsets of size 4
                if sum(x_values[i,j] for j in subset) >
                    2 + M * (6 - sum(x_values[t,j] for (t,j) in combinations(subset, 2))):
                    return True # solution is cut off
                elif sum(x_values[i,j] for j in subset) <
                    2 - M * (6 - sum(x_values[t,j] for (t,j) in combinations(subset, 2))):
                    return True # solution is cut off
    else:
        for i in range(v):
            for subset in combinations(set(range(v)) - {i}, 4): # iterate over all subsets of size 4
                if sum(x_values[i,j] for j in subset) >
                    2 + M * (6 - sum(x_values[t,j] for (t,j) in combinations(subset, 2))):
                    return True # solution is cut off
                elif sum(x_values[i,j] for j in subset) <
                    1 - M * (6 - sum(x_values[t,j] for (t,j) in combinations(subset, 2))):
                    return True # solution is cut off
    return False    # solution is not cut off
```

4.2 Implementation of the boolean satisfiability problem

```python
def initialize_SAT_easyconditions(parameters):
    """Create SAT problem with variables x and add conditions such that the resulting graph
    has an e-regular s-clique and is k-regular."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters

    x = [[Bool("x_%i_%i" % (i, j)) for j in range(i)] for i in range(v)]    # create literals

    solver = Solver()   # create solver instance

    # the first s vertices form a clique
    solver.add(
        And([x[i][j] for i in range(s) for j in range(i)])
    )

    # the graph is k-regular
    solver.add(
        And([
            And([
                Or([
                    Not(x[i][m]) if m < i else Not(x[m][i]) if m > i else True for m in I
                ])
                for I in combinations(range(v), k+1)    # iterate over all subsets of size k + 1
            ])
            for i in range(v)   # iterate over all vertices
        ])
    )
    solver.add(
        And([
            And([
                Or([
                    x[i][m] if m < i else x[m][i] if m > i else False for m in I
                ])
                for I in combinations(range(v), v - k + 1)  # iterate over all subsets of size v - k + 1
            ])
            for i in range(v)   # iterate over all vertices
        ])
    )

    # the clique is e-regular
    solver.add(
        And([
            And([
                Or([
                    Not(x[i][m]) for m in I
                ])
                for I in combinations(range(s), e + 1)  # iterate over all subsets of size e + 1
            ])
            for i in range(s, v)    # iterate over all vertices outside the clique
        ])
    )
    solver.add(
        And([
            And([
                Or([
                    x[i][m] for m in I
                ])
                for I in combinations(range(s), s - e + 1)  # iterate over all subsets of size s - e + 1
            ])
            for i in range(s, v)    # iterate over all vertices outside the clique
        ])
    )

    return solver, x


def add_SAT_difficultconditions(solver, x, parameters):
    """Add difficult conditions to the SAT problem so the resulting graph is
    l-edge-regular and not strongly regular."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters
    cases = [(0, s, 1, s + 1), (0, s, 0, s + 1), (0, s, 1, s), (0, s, s + 1, s + 2), (0, s, s, s + 1),
            (s + 1, s + 2, 0, s), (s, s + 1, 0, s), (s, s + 1, s + 2, s + 3), (s, s + 1, s + 1, s + 2)]

    # the graph is l-edge-regular
    solver.add(
        And([
            Or(Not(x[i][j]), And([
                Or([
                    Or(Not(x[i][m]), Not(x[j][m])) if m < j else
                    Or(Not(x[i][m]), Not(x[m][j])) if j < m < i else
                    Or(Not(x[m][i]), Not(x[m][j])) if m > i else
                    True for m in I
                ])
                for I in combinations(range(v), l + 1)  # iterate over all subsets of size l + 1
            ]))
            for i in range(v) for j in range(i) # iterate over all pairs of vertices
        ])
    )
    solver.add(
        And([
            Or(Not(x[i][j]), And([
                Or([
                    And(x[i][m], x[j][m]) if m < j else
                    And(x[i][m], x[m][j]) if j < m < i else
                    And(x[m][i], x[m][j]) if m > i else
                    False for m in I
                ])
                for I in combinations(range(v), v - l + 1)  # iterate over all subsets of size v - l + 1
            ]))
            for i in range(v) for j in range(i) # iterate over all pairs of vertices
        ])
    )

    # the graph is not strongly regular
    solver.add(
        Or([
            And(Not(x[b][a]), Not(x[d][c]), Or([
                And(And([
                    Or([
                        Or(Not(x[a][m]), Not(x[b][m])) if m < a else
                        Or(Not(x[m][a]), Not(x[b][m])) if a < m < b else
                        Or(Not(x[m][a]), Not(x[m][b])) if m > b else
                        True for m in I
                    ])
                    for I in combinations(range(v), K)  # iterate over all subsets of size K
                ]),
                And([
                    Or([
                        And(x[c][m], x[d][m]) if m < c else
                        And(x[m][c], x[d][m]) if c < m < d else
                        And(x[m][c], x[m][d]) if m > d else
                        False for m in I
                    ])
                    for I in combinations(range(v), v - K + 1)  # iterate over all subsets of size v - K + 1
                ]))
                for K in range(1, k + 1)    # iterate over all values for K
            ]))
            for (a, b, c, d) in cases   # iterate over all cases
        ])
    )


def solve_model_SAT(solver, x, parameters):
    """Solve SAT problem and return the edges."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters

    if str(solver.check()) == 'sat':    # feasible
        model = solver.model()
        edges = []
        for i in range(v):
            for j in range(i):
                if model.evaluate(x[i][j]) == True:
                    edges += [(i,j)]    # get edges of resulting graph
    else:   # infeasible
        edges = None

    return edges


def solve_SAT_NG(v, k, l, e, s):
    """Find whether a strictly Neumaier graph exists with given parameters v, k, l, e, s
    by solving corresponding SAT problem and print the graph if it exists."""

    parameters = v, k, l, e, s

    solver, x = initialize_SAT_easyconditions(parameters) # create model and add easy conditions
    add_SAT_difficultconditions(solver, x, parameters)  # add difficult conditions

    edges = solve_model_SAT(solver, x, parameters)
    if edges is None:   # infeasible
        print('There exists no strictly Neumaier graph with the given parameters:
            v = {}, k = {}, l = {}, e = {}, s = {}.'.format(v, k, l, e, s))
    else:   # feasible
        print_graph(edges, parameters)  # print resulting graph
```

5.1 Distance to Neumaier graph

```python
def d0(A, v, k, l, e, s):
    """Return distance between graph with adjacency matrix A and NG(v, k, l, e, s) based on number
    of pairs of adjacent vertices that do not satisfy l-edge-regularity."""

    neighbors = np.zeros((v), dtype=set)
    for i in range(v):
        neighbors[i] = [j for j in range(v) if A[i][j] == 1]   # get neighbors of each node

    distance0 = 0    # initialize distance to 0
    for i in range(v):
        for j in range(i + 1, v):
            if A[i][j] == 1:    # only consider adjacent vertices
                common_neighbors = len(set(neighbors[i]).intersection(neighbors[j]))    # number of common neighbors
                if common_neighbors != l:
                    distance0 += 1
    return distance0


def d1(A, v, k, l, e, s):
    """Return distance between graph with adjacency matrix A and NG(v, k, l, e, s) based on distances
    between the number of common neighbors of adjacent pairs of vertices and l."""

    neighbors = np.zeros((v), dtype=set)
    for i in range(v):
        neighbors[i] = [j for j in range(v) if A[i][j] == 1]   # get neighbors of each node

    distance1 = 0   # initialize distance to 0
    for i in range(v):
        for j in range(i + 1, v):
            if A[i][j] == 1:    # only consider adjacent vertices
                common_neighbors = len(set(neighbors[i]).intersection(neighbors[j]))    # number of common neighbors
                distance1 += abs(common_neighbors - l)  # distance between number of common neighbors and l
    return distance1
```

5.2 Lagrangian relaxation of the integer linear program

```python
def initialize_model_obj(parameters):
    """Create integer program with binary variables x, y, z, w and objective function."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters

    model = Model(name = 'neumaier linear program') # create model

    x = {}  # create x variables
    for i in range(v):
        for j in range(i):
           x[i,j] = model.addVar(vtype = GRB.BINARY, name = "x_%i_%i" % (i, j))

    y = {}  # create y variables
    for i in range(v):
        for j in range(i):
            for m in set(range(v)) - {i, j}:
                y[i,j,m] = model.addVar(vtype = GRB.BINARY, name = "y_%i_%i_%i" % (i, j, m))

    z = model.addVars(9, vtype = GRB.BINARY, name = 'z')    # create z variables

    w = {}  # create max variables
    for i in range(v):
        for j in range(i):
           w[i,j] = model.addVar(vtype = GRB.BINARY, name = "w_%i_%i" % (i, j))

    model.setObjective(quicksum(w[i,j] for i in range(v) for j in range(i)), GRB.MINIMIZE)  # set objective

    return model, x, y, z, w

```

```python
def add_Neumaier_constraints_obj(model, x, y, z, w, parameters):
    """Add linear constraints to the integer program so the resulting graph is a Neumaier graph."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters
    M = v   # big M

    # the first s vertices form a clique
    model.addConstrs((x[i,j] == 1 for i in range(s) for j in range(i)))

    # the graph is k regular
    model.addConstrs((quicksum(x[i,j] for j in range(i)) +
        sum(x[j,i] for j in range(i + 1, v)) == k for i in range(v)))

    # the clique is e regular
    model.addConstrs((quicksum(x[i,j] for j in range(s)) == e for i in range(s, v)))

    # the y variables are products of the x variables
    model.addConstrs((y[i,j,m] >= x[i,m] + x[j,m] - 1 for i in range(v) for j in range(i) for m in range(j)))
    model.addConstrs((y[i,j,m] >= x[i,m] + x[m,j] - 1 for i in range(v) for j in range(i) for m in range(j+1, i)))
    model.addConstrs((y[i,j,m] >= x[m,i] + x[m,j] - 1 for i in range(v) for j in range(i) for m in range(i+1, v)))

    model.addConstrs((y[i,j,m] <= x[i,m] for i in range(v) for j in range(i) for m in set(range(i)) - {j}))
    model.addConstrs((y[i,j,m] <= x[m,i] for i in range(v) for j in range(i) for m in range(i+1,v)))
    model.addConstrs((y[i,j,m] <= x[j,m] for i in range(v) for j in range(i) for m in range(j)))
    model.addConstrs((y[i,j,m] <= x[m,j] for i in range(v) for j in range(i) for m in set(range(j+1,v)) - {i}))

    # the w variables represent max values
    model.addConstrs((w[i,j] >= 0 for i in range(v) for j in range(i)))
    model.addConstrs((w[i,j] >= quicksum(y[i,j,m] for m in set(range(v)) - {i,j})- l -
        M * (1 - x[i,j]) for i in range(v) for j in range(i)))
    model.addConstrs((w[i,j] >= l - M * (1 - x[i,j]) -
        quicksum(y[i,j,m] for m in set(range(v)) - {i,j}) for i in range(v) for j in range(i)))


def solve_model_obj(model, x, y, z, w, parameters):
    """Solve the integer program and return the x variables and the optimal value."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters

    model.optimize()    # optimize model
    if model.status == GRB.Status.OPTIMAL:   # feasible
        x_values = model.getAttr('x', x)   # get values of x corresponding to edges in resulting Neumaier graph
        value = model.getAttr('ObjVal')
    else:   # infeasible
        x_values = None
        value = 0

    return x_values, value


def solve_ILP_with_objective_NG(v, k, l, e, s):

    parameters = v, k, l, e, s

    model, x, y, z, w = initialize_model_obj(parameters)    # create model
    add_Neumaier_constraints_obj(model, x, y, z, w, parameters) # Neumaier graph
    add_strictlyNeumaier_constraint(model, x, y, z, parameters) # not strongly regular
    x_values, value = solve_model_obj(model, x, y, z, w, parameters)    # get x variables and optimal value
    if x_values is None:   # infeasible
        print('There exists no strictly Neumaier graph with the given parameters:
            v = {}, k = {}, l = {}, e = {}, s = {}.'.format(v, k, l, e, s))
    elif value > 0:   # feasible but not a strictly Neumaier graph
        print('There exists no strictly Neumaier graph with the given parameters:
            v = {}, k = {}, l = {}, e = {}, s = {}.'.format(v, k, l, e, s))
        edges = [(i,j) for i in range(v) for j in range(i) if x_values[i, j] >= 0.99]
        print_graph(edges, parameters)
        A = find_adjacencymatrix(edges, v)
        distance0 = d0(A, v, k, l, e, s)
        distance1 = d1(A, v, k, l, e, s)
        print('d0 to NG{}: {}'.format(parameters, distance0))
        print('d1 to NG{}: {}'.format(parameters, distance1))
    else:   # feasible and strictly Neumaier graph
        edges = [(i,j) for i in range(v) for j in range(i) if x_values[i, j] >= 0.99]
        print_graph(edges, parameters)


def initialize_model_Lagrangian(u_1, u_2, parameters):
    """Create linear program with binary variables x, y, z with objective function
    to approach edge-regular graph."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters
    M = v

    model = Model() # create model

    x = {}  # create x variables
    for i in range(v):
        for j in range(i):
           x[i,j] = model.addVar(vtype = GRB.BINARY, name = "x_%i_%i" % (i, j))

    y = {}  # create y variables
    for i in range(v):
        for j in range(i):
            for m in set(range(v)) - {i,j}:
                y[i,j,m] = model.addVar(vtype = GRB.BINARY, name = "y_%i_%i_%i" % (i, j, m))

    z = model.addVars(9, vtype = GRB.BINARY, name = 'z')    # create z variables

    # set objective for lagrangian relaxation
    model.setObjective(quicksum((u_1[i,j] * (l + M * (1 - x[i,j]) -
        quicksum(y[i,j,m] for m in set(range(v)) - {i,j})) + u_2[i,j] *
        (quicksum(y[i,j,m] for m in set(range(v)) - {i,j}) - l + M *
        (1 - x[i,j]))) for i in range(v) for j in range(i)), GRB.MAXIMIZE)

    return model, x, y, z


def add_Neumaier_constraints_Lagrangian(model, x, y, z, parameters):
    """Add constraints to the integer program except for edge-regularity constraints."""

    assert len(parameters) == 5, '5 parameters are expected.'
    v, k, l, e, s = parameters
    M = v

    # the first s vertices form a clique
    model.addConstrs((x[i,j] == 1 for i in range(s) for j in range(i)))

    # the graph is k regular
    model.addConstrs((quicksum(x[i,j] for j in range(i)) +
        sum(x[j,i] for j in range(i + 1, v)) == k for i in range(v)))

    # the clique is e regular
    model.addConstrs((quicksum(x[i,j] for j in range(s)) == e for i in range(s, v)))

    # the y variables are products of the x variables
    model.addConstrs((y[i,j,m] >= x[i,m] + x[j,m] - 1 for i in range(v) for j in range(i) for m in range(j)))
    model.addConstrs((y[i,j,m] >= x[i,m] + x[m,j] - 1 for i in range(v) for j in range(i) for m in range(j+1, i)))
    model.addConstrs((y[i,j,m] >= x[m,i] + x[m,j] - 1 for i in range(v) for j in range(i) for m in range(i+1, v)))

    model.addConstrs((y[i,j,m] <= x[i,m] for i in range(v) for j in range(i) for m in set(range(i)) - {j}))
    model.addConstrs((y[i,j,m] <= x[m,i] for i in range(v) for j in range(i) for m in range(i+1,v)))
    model.addConstrs((y[i,j,m] <= x[j,m] for i in range(v) for j in range(i) for m in range(j)))
    model.addConstrs((y[i,j,m] <= x[m,j] for i in range(v) for j in range(i) for m in set(range(j+1,v)) - {i}))


def subgradient_method(v, k, l, e, s, nr_iterations):
    """Run the subgradient algorithm for given number of iterations hoping to approach a Neumaier graph."""

    parameters = v, k, l, e, s
    M = v   # big M

    u_1 = {}    # initialize Lagrangian multipliers
    u_2 = {}
    for i in range(v):
        for j in range(i):
            u_1[i,j] = 1
            u_2[i,j] = 1

    t = 1   # iteration counter
    g_best = 1000000    # initialize g_best to high value

    while t <= nr_iterations:
        model, x, y, z = initialize_model_Lagrangian(u_1, u_2, parameters)  # initialize model

        add_Neumaier_constraints_Lagrangian(model, x, y, z, parameters) # Neumaier graph except edge-regular
        add_strictlyNeumaier_constraint(model, x, y, z, parameters) # not strongly regular
        model.setParam('OutputFlag', 0)

        x_star = solve_model_ILP(model, x, y, z, parameters) # solve the program and get x variables
        y_star = model.getAttr('x', y)  # get y variables
        g = model.getAttr('ObjVal')
        print(g)    # print objective value

        if g < g_best:  # update g_best if necessary
            g_best = g
            x_values = x_star
        t += 1

        s_1 = {}    # find a subgradient
        s_2 = {}
        for i in range(v):
            for j in range(i):
                s_1[i,j] = l + M * (1 - x_star[i,j]) - sum(y_star[i,j,m] for m in set(range(v)) - {i,j})
                s_2[i,j] = sum(y_star[i,j,m] for m in set(range(v)) - {i,j}) - l + M * (1 - x_star[i,j])

        alpha = 1/t # stepsize

        for i in range(v):  # update lagrangian multipliers
            for j in range(i):
                u_1[i,j] = max(0, u_1[i,j] - alpha * s_1[i,j])
                u_2[i,j] = max(0, u_2[i,j] - alpha * s_2[i,j])

    print(g_best)   # print best value
    edges = [(i,j) for i in range(v) for j in range(i) if x_values[i, j] >= 0.99]
    print_graph(edges, parameters)
    A = find_adjacencymatrix(edges, v)
    distance0 = d0(A, v, k, l, e, s)
    distance1 = d1(A, v, k, l, e, s)
    print('d0 to NG{}: {}'.format(parameters, distance0))
    print('d1 to NG{}: {}'.format(parameters, distance1))
```

5.3 Gradually solving the boolean satisfiability problem

```
def add_edgeregularity_constraint(solver, x, i, j, parameters):
    """Add edge regularity condition to SAT problem for vertices i and j."""

    v, k, l, e, s = parameters

    solver.add(
        Or(Not(x[i][j]), And([
            Or([
                Or(Not(x[i][m]), Not(x[j][m])) if m < j else
                Or(Not(x[i][m]), Not(x[m][j])) if j < m < i else
                Or(Not(x[m][i]), Not(x[m][j])) if m > i else
                True for m in I
            ])
            for I in combinations(range(v), l + 1)  # iterate over all subsets of size l + 1
        ]))
    )
    solver.add(
        Or(Not(x[i][j]), And([
            Or([
                And(x[i][m], x[j][m]) if m < j else
                And(x[i][m], x[m][j]) if j < m < i else
                And(x[m][i], x[m][j]) if m > i else
                False for m in I
            ])
            for I in combinations(range(v), v - l + 1)  # iterate over all subsets of size v - l + 1
        ]))
    )


def check_edgeregular(G, l):
    """Check whether the graph G is l-edge-regular."""

    for (i,j) in nx.edges(G):
        if len(list(nx.common_neighbors(G, i, j))) != l:
            return False    # graph is not l-edge-regular
    return True # graph is l-edge-regular


def check_notcoedgeregular(G):
    """Check whether the graph G is not co-edge-regular."""

    number_common_neighbors = []
    for (i, j) in nx.non_edges(G):
        number_common_neighbors.append(len(list(nx.common_neighbors(G, i, j))))
    return len(set(number_common_neighbors)) > 1


def solve_graduallySAT_NG(v, k, l, e, s, nr_iterations):
    """Find whether a Neumaier graph exists with given parameters v, k, l, e, s by solving
    corresponding SAT problem and gradually adding conditions and print the graph if it exists."""

    parameters = v, k, l, e, s
    t = 0   # iteration counter
    solver, x = initialize_SAT_easyconditions(parameters)   # initialize model and add easy constraints
    G = nx.Graph()  # initialize graph

    while t <= nr_iterations:
        print(t)    # print iteration
        if str(solver.check()) == 'sat':    # feasible
            model = solver.model()
            edges = []
            for i in range(v):
                for j in range(i):
                    if model.evaluate(x[i][j]) == True:
                        edges += [(i,j)]    # edges of found solution
            if t == nr_iterations:  # print results in last iteration
                print_graph(edges, parameters)
                A = find_adjacencymatrix(edges, v)
                distance0 = d0(A, v, k, l, e, s)
                distance1 = d1(A, v, k, l, e, s)
                if distance0 == 0 and distance1 == 0:   # Neumaier graph
                    if check_notcoedgeregular(G):   # strictly Neumaier graph
                        print('A strictly Neumaier graph is found in {} iterations'.format(t))
                    else:
                        print('A Neumaier graph is found in {} iterations,
                            but it is not a strictly Neumaier graph'.format(t))
                else:
                    print('d0 to NG{}: {}'.format(parameters, distance0))
                    print('d1 to NG{}: {}'.format(parameters, distance1))
            else:   # not last iteration
                G.clear()
                G.add_edges_from(edges) # graph with edges of found solution
                if check_edgeregular(G, l) and check_notcoedgeregular(G):   # strictly Neumaier graph
                    print('A strictly Neumaier graph is found in {} iterations'.format(t))
                    print_graph(edges, parameters)
                    break
                if not check_edgeregular(G, l): # add condition for two random vertices
                    vertex_1, vertex_2 = random.choice([(i, j) for (i, j) in edges if
                        len(list(nx.common_neighbors(G, i, j))) != l])
                    add_edgeregularity_constraint(solver, x, vertex_1, vertex_2, parameters)
            t += 1
        else:   # infeasible
```

```
            print('There exists no Neumaier graph with the given parameters:
                v = {}, k = {}, l = {}, e = {}, s = {}.'.format(v, k, l, e, s))
            break
```

Bibliography

[1] A. Abiad, M. de Boeck, W. Castryck, J.H. Koolen, and S. Zeijlemaker. An infinite class of Neumaier graphs and non-existence results. *Journal of Combinatorial Theory, Series A*, 193, 2023.

[2] A. Biere, M. J. H. Heule, H. Van Maaren, and T. Walsh. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, 2 edition, 2021.

[3] M. Codish, A. Miller, P. Prosser, and P. J. Stuckey. Breaking symmetries in graph representation. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 510–516, 2013.

[4] M. Conforti, G. Cornuéjols, and G. Zambelli. *Integer Programming*. Springer, 2014.

[5] Stephen D. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[6] K. Coolsaet, P. D. Johnson, K. J. Roblee, and T. D. Smotzer. Some Extremal Problems for Edge-Regular Graphs. *Ars Combinatoria*, 105:411–418, 2012.

[7] F. de Ruiter and N. Biggs. Applications of integer programming methods to cages. *The Electronic Journal of Combinatorics*, 2015.

[8] R. D. Evans, S. Goryainov, and D. Panasenko. The Smallest Strictly Neumaier Graph and its Generalisations. *Electronic Journal of Combinatorics*, 26(2), 2019.

[9] R. J. Evans. *On regular induced subgraphs of edge-regular graphs*. PhD thesis, Queen Mary University of London, 2020.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability*. 1979.

[11] S.V. Goryainov and L.V. Shalaginov. Cayley–Deza graphs with fewer than 60 vertices. *Sib. Èlektron. Mat. Izv.*, 11:268–310, 2014.

[12] G. R. W. Greaves and J. H. Koolen. Edge-regular graphs with regular cliques. *European Journal of Combinatorics*, 71:194–201, 2018.

[13] G. R. W. Greaves and J. H. Koolen. Another construction of edge-regular graphs with regular cliques. *Discrete Mathematics*, 342(10):2818–2820, 2019.

[14] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research*. McGraw-Hill, 9 edition, 2009.

[15] A. Ignatiev, A. Morgado, and J. Marques-Silva. *On Reducing Maximum Independent Set to Minimum Satisfiability*. Springer, 2014.

[16] E. Klotz and A. M. Newman. Practical guidelines for solving difficult mixed integer linear programs. *Surveys in Operations Research and Management Science*, 18(1-2):18–32, 2013.

[17] L. Lovász. Graph Theory and Integer Programming. *Annals of Discrete Mathematics*, 4:141–158, 1979.

[18] A. Neumaier. Regular cliques in graphs and special $1\frac{1}{2}$-designs. In *Finite Geometries and Designs*, pages 244–259. Cambridge University Press, 1981.