On the limits of Linear and Affine Integer Programming relaxations for Constraint Satisfaction Problems

by

Kimia Hashemi

B.Sc., Amirkabir University of Technology, 2021

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Summer 2024

Declaration of Committee

Name:	**Kimia Hashemi**
Degree:	**Master of Science**
Thesis title:	**On the limits of Linear and Affine Integer Programming relaxations for Constraint Satisfaction Problems**
Committee:	**Chair:** Matt Amy Assistant Professor, Computing Science
	Andrei Bulatov Supervisor Professor, Computing Science
	Igor Shinkar Committee Member Assistant Professor, Computing Science
	Qianping Gu Examiner Professor, Computing Science

Abstract

A Constraint Satisfaction Problem (CSP) asks whether values from a specified domain can be assigned to given variables subject to a set of constraints. The Promise Constraint Satisfaction Problem (PCSP) is a variant of the CSP in which the input is a pair of similar CSP instances, and the question is whether the stronger instance can be satisfied, or even the weaker one is unsatisfiable. Algorithms for the CSP and PCSP have been a major research direction for several decades. While the complexity of the CSP is largely understood, that of the PCSP is widely open.

One of the recent algorithmic approaches to both problems uses various combinations of Linear and Affine Programming relaxations of the problems. It has even been proposed that such a combination may serve as a universal efficient for all cases of tractable CSPs and PCSPs. In this thesis we refute this conjecture for some combinations of local algorithms, Linear, and Affine Integer Programming relaxations.

Keywords: constraint satisfaction problem, promise constraint satisfaction, linear programming, cohomology, algorithms

Acknowledgements

I would like to sincerely thank my supervisor, Dr. Andrei Bulatov, for his guidance and support throughout my M.Sc. degree. His expertise and encouragement made a significant difference in the completion of this work.

I am also deeply grateful to my family for their unwavering support. Their encouragement has been a constant source of strength for me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Constraint Satisfaction Problem (CSP) and its variants provide a fundamental and general framework in theoretical computer science and include a wide variety of computational problems including graph-coloring, k-SAT, and solving systems of linear equations. Indeed, since its formal inception in early 70's [37], the applications of the CSP in operational research, such as scheduling and timetabling has been known to the artificial intelligence community [9]. One can equivalently define the decision version of the CSP in different ways. For example, the CSP is the problem of deciding whether a homomorphism exists from one relational structure A to another one B, the latter structure is called a *template*. Another way to define it as the problem of deciding whether values from a defined domain can be assigned to some variables while keeping some constraints satisfied.

The decision CSP was naturally extended to a range of variants, including Quantified CSP (QCSP), Valued CSP (VCSP) and many others, to capture a range of related problems in the field. Recently, a new variant of the CSP, namely the Promise Constraint Satisfaction Problem (PCSP) was introduced, as certain open problems such as approximate graph coloring and variants of satisfiability cannot be expressed using the CSP definition. The PCSP extends the standard CSP: For two relational structures A and B with a homomorphism from A to B, given a third relational structure I, decide whether I has a homomorphism to A or does not have a homomorphism to B. The "promise" is that these are the only possible options for I. Equivalently, one can assume that in every instance of a PCSP, each constraint has a "strict" and a "weak" versions. The PCSP then asks to distinguish the case when an instance has a solution with respect to the strict constraints from when it does not have a solution even subject to the weak constraints [14], with the promise of no other case in between.

Just like the CSP, the PCSP has the search and decision versions. In fact, a PCSP where $A = B$ is equivalent to a CSP on this relational structure, therefore it is easy to see how the PCSP generalizes the CSP. Because of the close-knit connection between the two structures, it makes sense to consider the same questions that have been asked about

CSPs (e.g., classification with respect to complexity classes like P or NP) for PCSPs, with appropriate adjustments.

In terms of complexity, the general form of both the CSP and PCSP problems are computationally hard as they include NP-hard problems, some of which are mentioned earlier. That is why a major direction of research in this area focuses on identifying tractable cases and investigating the underlying mathematical properties and structures that enable their tractability. More specifically, the research focused on the so-called "non-uniform" CSPs, which means their template are fixed and we study the exact complexity of such CSPs. In fact, the famous Feder-Vardi dichotomy conjecture [22] on non-uniform CSPs, proved using algebraic approach in [18] and [41], states that CSP arose form each finite template is either in P or NP-complete. Similarly ,a dichotomy for symmetric boolean PCSPs have been established in [11, 13] for non-uniform PCSPs (i.e., both A and B are fixed relational structures). To better understand the complexity of PCSPs with the ultimate goal of recognizing what restrictions in the structure of these problems make them tractable, research continues.

The algebraic approach, which was the central tool in both proofs of the CSP Dichotomy, in its first layer uses the framework of so-called polymorphisms. In simple terms, polymorphisms are multivariate mappings between relational structures that form a PCSP templates. For example a simple polymorphism of a graph G is a homomorphism from some power of G to G [32]. Similarly, they can be viewed as high-dimensional “symmetries” of solution sets of a CSP or PCSP instance. Those CSPs which are solved efficiently happen to have such symmetries, while the lack of them in a CSP indicates the hardness of the corresponding problem [4]. On this same track, the focus of this thesis is to use algorithmic techniques to evaluate the applicability of some algorithms mentioned below to various types of CSPs and PCSPs.

There are two general methods for solving tractable CSPs, namely Local Consistency and Gaussian Elimination, the latter of which is out of scope of this thesis. It was conjectured [24] and later characterised [5, 6, 15], that CSPs with no "ability to count" are solvable by local consistency algorithms (or have bounded width), meaning all instances without a solution are recognizable by some form of local consistency. Bounded-width refers to the fact that in every iteration of a local consistency algorithm, it only deals with a bounded number of constraints and/or variables and the CSPs with bounded-width ensure the existence of a global solution. Some notions of local consistency methods include: Arc-Consistency, Path Consistency, (k, l)-consistency, and (k, l)-minimality.

Integer and Linear programming and their variants are among mathematical programming techniques which have a history of use cases in optimization and decision problems [42]. We particularly focus on them as a CSP can be reformulated as an integer linear programming problem. In fact, any instance of a decision CSP, and therefore PCSP, can be described as a canonical 0-1 integer program. The Basic Linear Programming relaxation

(BLP) and Affine Integer Programming relaxation (AIP) and their combination BLP+AIP [14] are among the standard polynomial-time relaxations for PCSPs [10].

These relaxations along with local consistency techniques, some newer cohomological algorithmic approaches, and their combinations open up possibilities to stronger algorithms to a point where it is believed they can solve all the polynomial-time CSPs and PCSPs. The work presented in this thesis is the result of an attempt to refute this idea by constructing carefully-designed algebras leading to CSP instances which do not have feasible solutions, yet they can deceive these algorithms, in particular combination of $(2-3)$-consistency + AIP and $(2-3)$-consistency + cohomology, into finding solutions. This will automatically rejects the conjecture and motivates searching for a more powerful algorithm to solve all polynomial-time CSPs and PCSPs.

Literature Review

One of the most well-studied problems in computer science and a fundamental example of CSPs is Graph Coloring and its variants. We say that a graph G is k-colorable if one can assign k colors to the vertices of G in a way that no two adjacent vertices have the same color. While there exist polynomial-time algorithms for deciding if a graph is k-colorable for $k \leq 2$, it is well-known that for $k \geq 3$ this problem is NP-hard. Such problems of the decision CSP type can be equivalently defined in other ways; for example, we can define k-coloring problem as the problem of deciding whether, for a fixed digraph H, a homomorphism from G to H exists. We are interested in such problems where the target structure, in this case H, is fixed and such problems are called non-uniform CSPs.

To demonstrate the earlier claim that PCSP is a natural generalization of CSP and their connection, it is noteworthy to introduce a primary example of a PCSP, which is the Approximate Graph Coloring, first introduced in [25]. Given a graph G, which is either k-colorable or not even c-colorable for some $c \geq k$, the approximate graph coloring is the problem of deciding between the two cases. It is proved that this problem is NP-hard for all $3 \leq k \leq c \leq 2k-1$ in [4]. For large k and $c = 2^{\Omega(k^{\frac{1}{3}})}$ this problem was proved to be in NP [27], and the bounds were later improved to $c = \binom{k}{\lfloor \frac{k}{2} \rfloor} - 1$ for $k \geq 5$ [40]. The approximate graph coloring problem is proved to be NP-hard for all $3 \leq k \leq c$ under some additional assumptions [21, 26].

Approximate Graph Coloring problem itself is a special case of Approximate Graph Homomorphism problem, where for two fixed graphs A and B with a homomorphism from A to B, given a graph G, it answers the question whether G has a homomorphism to A or does not have a homomorphism to B. For all non-bipartite loop-less graphs A and B, where there is a homomorphism from A to B, it has been conjectured that this problem is NP-hard [13]. Using some algebraic topological techniques in [31, 33] Approximate Graph Homomorphism problem is proved to be NP-hard for any non-bipartite 3-colorable graph A and $B = K_3$.

Aside from the occasional results for selected PCSP problems, the current knowledge of complexity classification of PCSP problems is quite limited. A systematic investigation of PCSPs (with fixed constraint languages) was initiated in [3, 11, 13] which suggested the use of the algebraic approach to the PCSPs, as it showed to be a powerful tool for proofs of CSP classification problems. The algebraic approach to CSPs, which includes using structural properties of finite universal algebras associated with instances of the problem, had a fundamental appearance in the independent proofs of the Feder-Vardi conjecture [32]. In [8], a full explanation of the algebraic approach in the CSPs, leading to many developments in both the CSP and universal algebra can be found. The algebraic approach to CSP has also led to uncovering notable connections with the theory of Maltsev conditions in universal algebra (e.g. [17, 16, 29])

The study of the complexity of the PCSP problems and the first link between the algebraic approach and PCSPs was originated in [3] by Austrin, Håstad, and Guruswami. In that paper, they introduced a natural promise variant of CNF-SAT and proved the following hardness result for it: Given a CNF-formula with each clause having width w and the promise that there is an assignment satisfying at least $g = \lceil \frac{w}{2} \rceil - 1$ literals in each clause, it is NP-hard to find a satisfying assignment to the formula (which sets at least one literal to true in each clause). On the other hand, when $g = \lceil \frac{w}{2} \rceil$, it is easy to find a satisfying assignment. Later in [10], they established a promising link between PCSPs and the CSP universal algebraic frameworks. Their primary motivation was to further understand the complexity of the PCSPs by focusing on the Boolean version and establishing a dichotomy. More specifically, they classified all tractable cases of Boolean PCSPs when the constraint predicates are symmetric. Following that, in [12], it is shown that the complexity of PCSPs are completely captured by their polymorphisms, meaning that two PCSPs with the same set of polymorphisms belong to the same complexity class. In [4], an extensive review of algebraic approach to PCSPs along with some new results is done.

To explain the tractability of PCSP problems, the idea of reducing them to tractable CSPs was introduced in [12] under the notion of homomorphic sandwiching. Those PCSPs that are reducible to tractable (finite-domain) CSPs are called finitely tractable. While not every tractable PCSP is finitely tractable as shown by the counter example of **1-in-3** vs **NAE** [4], it is conjectured in [12] that tractable finite-domain PCSPs can be reduced to tractable CSPs ,possibly over an infinite domain. Since the complexity of infinite-domain CSPs remains largely unexplored and therefore finite tractability does not capture all tractable PCSP problems, alternative algorithmic tools are necessary, potentially those that have been investigated and proven effective for CSPs such as consistency as discussed earlier. Furthermore, rather than employing, for instance, arc-consistency (the characterization of its power [23] has been lifted from CSP to PCSP in [4]) can be turned to convex relaxations [19].

Any instance of a CSP can be expressed as a Basic Linear Programming relaxation (BLP) and in fact a canonical analogue and the stronger version of arc-consistency [35]. The early results of attempting to characterize the power of this relaxation for CSPs in [34] was expanded to PCSPs in [4]. They captured the power of BLP for PCSPs equivalently in terms of the existence of symmetric polymorphisms of all arities, admitting a *minion* homomorphism consisting of rational stochastic vectors, and a certain *pp-constructability* concept.

Similar to BLP, another relaxation for PCSPs, first established in [12], is Affine Integer Programming relaxation (AIP). The power of AIP for PCSP was also characterized in [4] in terms of a minion consisting of integer affine vectors and polymorphisms of odd arities invariant under certain permutations. A prime example of a PCSP instance solved by AIP is **1-in-3** vs **NAE** [19].

Motivated to construct a stronger algorithm, in [14] BLP+AIP relaxation was introduced. In the same paper, the power of BLP+AIP was characterized in terms of the minion of the combination of AIP and BLP, and polymorphisms of odd arities invariant under permutations that only permute odd and even coordinates. However, BLP+AIP is unable to solve some instances of simple, tractable, non-Boolean PCSPs [14]. Therefore, building on the work of Brakensiek et. al, in [19] Ciardo et. al investigated algorithms stronger that BLP+AIP and introduced CLAP. Coming from the background of quantum foundations and category theory Conghaile in [20] introduced a novel cohomological algorithm which proved to have applications in solving CSPs and therefore PCSPs with power stronger than k-consistency.

Our Results

As was mentioned above, it was conjectured that a certain combination of local consistency algorithms and linear or affine integer relaxations suffices to solve any tractable PCSP and CSP.

Conjecture 1.1. *There is a combination of some level of consistency and the linear programming or cohomology solves all the polynomial-time CSPs and PCSPs.*

In this thesis we challenge this conjecture for some of those algorithms.

Note that Conjecture 1.1 (stated as Conjecture 2.6.3 in Section 2.6.2) allows for multiple combinations of consistency and relaxation algorithms. In a recent paper Lichter and Pago [36] refuted Conjecture 1.1 for some of those algorithms. In this thesis we consider a different collection of algorithms. More specifically, we test the combination of Basic Linear Programming (BLP) and Affine relaxation, as well as the combination of the $(2,3)$-consistency and cohomology algorithms. All the algorithms are described in detail in the thesis, and the exact way we deal with them is explained in Section 3.2. In order to refute the conjecture in those cases, we construct a small, 3-element algebra $\mathcal{A}$ which combines features that

are normally amenable to consistency algorithms, as well as features that require Gaussian elimination to solve. The idea is then that such a combination would fool the algorithms under consideration.

Next, we show that the algebra $\mathcal{A}$ satisfies the desired properties, show that the CSP over this algebra is not solvable by the consistency algorithms and linear/affine programming relaxation algorithms separately, and introduce several relations invariant with respect to $\mathcal{A}$ and that therefore can be use in instances of the CSP over $\mathcal{A}$. Finally, we construct two instances of that CSP, I_7 and I_{16} (the subscript here refers to the number of variables in the instance). In Lemmas 3.2.1 and 3.2.2 we prove that these two instances do not have a solution.

Next, we implement all the algorithms involved: the $(2,3)$-consistency algorithm, the Affine Integer Programming algorithm (AIP), and the cohomology algorithms. We then run those algorithms on instances I_7 and I_{16}. We find that the combination of the BLP and the AIP does not detect the inconsistency of I_7, while the combination of the $(2,3)$-consistency and AIP correctly detects that I_7 is inconsistent. This combination however does not detect the inconsistency of I_{16}. Since the CSP over the algebra $\mathcal{A}$ is known to be solvable in polynomial time, this refutes Conjecture 2.6.3 for these algorithms.

Our results do not fully refute refute Conjecture 2.6.3, since a number of (potentially more powerful) combinations of algorithms remain. In particular, we consider only consistency of a fairly low level — it is lower than the arity of the relations involved. It is possible that the consistency of a higher level, at least as high as the arity of the relation of the instance, may lead to a desired algorithm. However, an instance that may fool such an algorithm is quite complicated. It is based on graphs with high treewidth, see [2], and is not be feasible for empirical testing, as it would involve thousands of variables.

Chapter 2

Preliminaries

We use the notation $[n] = \{1, \ldots, n\}$ throughout this thesis.

2.1 Relational Structures

For sets $A_1, \ldots A_k$, their *Cartesian product* is the set of k-tuples $A_1 \times \ldots \times A_k = \{(a_1, \ldots, a_k) | a_i \in A_i, i \in [k]\}$. When $A_1, \ldots A_k = A$, we write A^k for $A_1 \times \ldots \times A_k$. A *relation* of arity k is then defined as a subset $R \subseteq A^k$. It is often convenient to represent relations by matrices. For example

$$R_{\neq} = \begin{pmatrix} 1 & 1 & 2 & 2 & 3 & 3 \\ 2 & 3 & 1 & 3 & 1 & 2 \end{pmatrix}$$

shows the disequality relation on the set $\{1, 2, 3\}$, where the tuples are written vertically. We denote the tuples by bold lowercase letters, such as $\mathbf{a}$. By $\mathbf{a}[i]$, we refer to the i-th entry of tuple $\mathbf{a}$. The projection of $\mathbf{a}$ on an index set I, denoted by $\mathrm{pr}_I \mathbf{a}$, is a tuple $(\mathbf{a}[i_1], \ldots, \mathbf{a}[i_l])$, where $I = \{i_1, \ldots, i_l\} \subseteq [k]$. The projection of a relation $R \subseteq A_1 \times \ldots \times A_k$ on I is defined as $\mathrm{pr}_I R = \{\mathrm{pr}_I \mathbf{a} \mid \mathbf{a} \in R\}$. Similarly for a single set A, an n-ary operation $p_i^{(n)} : A^n \to A$ is called a projection (or dictator) on A and has the form $p_i^{(n)}(x_1, \ldots, x_n) = x_i$. A *constraint language* Γ is an arbitrary finite set of relations, of possible different arity, on a set A. In this case, A is called the *domain* of Γ.

Definition 2.1.1. A *relational structure* is a tuple $\mathbf{A} = (A; R_1^{\mathbf{A}}, \ldots, R_k^{\mathbf{A}})$, where each $R_i^A \subseteq A^{ar(R_i)}$ is a relation on A of arity $ar(R_i) \geq 1$. For more clarity, we denote the relational structures by bold capital letters and their templates with capital letters.

When A is a finite set, we say $\mathbf{A}$ is finite. We assume that all relational structures are finite in this paper, unless specified otherwise. Two structures $\mathbf{A} = (A; R_1^{\mathbf{A}}, \ldots, R_k^{\mathbf{A}})$ and $\mathbf{B} = (B; R_1^{\mathbf{B}}, \ldots, R_k^{\mathbf{B}})$ are called *similar* if the number of their relations is the same and $ar(R_i^A) = ar(R_i^B)$ for each $i \in [k]$.

Example 2.1.1. *For a (directed) graph G, we denote the vertex set by $V(G)$ and the edge set by $E(G)$. Then G is a relational structure with $V(G)$ as its domain and $E(G)$ as its only relation of arity two, i.e., a binary relation.*

Definition 2.1.2. Let $\mathbf{A}$ and $\mathbf{B}$ be two similar relational structures. A *homomorphism* from $\mathbf{A}$ to $\mathbf{B}$, denoted by $h : \mathbf{A} \to \mathbf{B}$, is a mapping $h : A \to B$ such that for each i

$$\text{if } (a_i, \ldots, a_{ar(R_i)}) \in R_i^{\mathbf{A}} \text{ then } (h(a_i), \ldots, h(a_{ar(R_i)}) \in R_i^{\mathbf{B}}$$

We write $\mathbf{A} \to \mathbf{B}$ to denote that a homomorphism from $\mathbf{A}$ to $\mathbf{B}$ exists. If no such a homomorphism exists, we simply write $\mathbf{A} \not\to \mathbf{B}$.

2.2 Constraint Satisfaction Problem (CSP)

The Constraint Satisfaction Problem (CSP) is defined as a decision problem: is it possible to assign values to some variables such that all the pre-defined constraints are satisfied.

Definition 2.2.1. (CSP definition I) A CSP instance with constraint language Γ, denoted by CSP(Γ), is defined as a tuple $(V, A, \mathcal{C})$, where

- V is a set of variables,
- A, the *domain*, is a set of values, and
- $\mathcal{C}$ is a set of constraints $\{C_1, \ldots, C_n\}$, where each C_i itself is a pair and is denoted by $\langle s_i, R_i \rangle$. Each *constraint scope* s_i is a tuple of variables from V of length m_i and the corresponding R_i is an m_i-ary relation on A.

The goal is to find a satisfying assignment for the CSP instance; that is, a function φ from V to A such that for each $C_i \in \mathcal{C}$, $\varphi(s_i) \in R_i$. This means that the scope of each constraint, under φ, is mapped to a tuple of its corresponding relation.

We can also define CSP using the concepts of relational structures and homomorphisms.

Definition 2.2.2. (CSP definition II) Let $\mathbf{A}$ be a fixed relational structure. The decision problem of whether a given relational structure $\mathbf{I}$ admits a homomorphism to $\mathbf{A}$ is denoted by CSP($\mathbf{A}$). In this case, $\mathbf{A}$ is called the *template* for this problem.

CSP definitions I and II are equivalent and convertible. When we are given an instance $P = (V, A, \mathcal{C})$ from the first definition, we can construct an instance for the second definition and vice versa.

Example 2.2.1. *The Graph k-Colorability is the problem of deciding whether a coloring of the vertices of a graph G with k colors exists, such that no two adjacent edges of G have the same color, i.e., for any edge $e = uv \in E(G)$, u should have a different color than v.*

We call such a coloring a proper coloring. This problem can be formulated with Definition 2.2.2 as CSP($\boldsymbol{A}$), where $\boldsymbol{A} = \boldsymbol{K}_k$ is a k-clique and for an input graph $\boldsymbol{I}$, $\boldsymbol{I} \to \boldsymbol{A}$ is a graph k-coloring. To see how this problem can equivalently be described with tools from Definition 2.2.1, take

- $V = V(G)$,
- $A = \{1, \ldots, k\}$,
- $\mathcal{C} = \{\langle (u, v), \neq_2 \rangle \mid uv \in E(G)\}$, *where* $\neq_2$ *is the binary disequality relation on the domain* A. *More specifically, it consists of all the elements of* A_2 *except the tuples* (u_i, u_j) *where* $u_i = u_j$.

A satisfying assignment $\varphi : V \to A$ *correctly solves* $(V, A, \mathcal{C})$ *in this example if for each edge* (u, v), $\varphi(u) \neq \varphi(v)$ *holds, which is the same as* $\neq_2 (u, v)$.

Example 2.2.2. *As a generalization of Example 2.2.1, we can define the problem of H-Coloring, which has the same setting except instead of* $\boldsymbol{A} = \boldsymbol{K}_k$, $\boldsymbol{A}$ *is an arbitrary graph* H. *Similarly, we seek a proper coloring, which is again basically a homomorphism* $\boldsymbol{I} \to \boldsymbol{A}$.

Example 2.2.3. *Suppose we are given a set of Boolean variables,* $x_1, \ldots, x_n$, *and that they can each take the value of 0 or 1. We call these propositional variables and their negations literals. Any disjunction of literals is called a clause and a collection of clauses is a Conjunctive Normal Form (CNF). For a CNF* Φ, *a satisfying assignment makes every clause true by assigning 0,1 to variables. Boolean Satisfiability is the problem of deciding, given a CNF, whether such a satisfying assignment exists. Formally, given a CNF* $\Phi = \{c_1, \ldots, c_k\}$ *the corresponding instance* $(V, A, \mathcal{C})$ *is*

- $V = \{x_1, \ldots, x_n\}$
- $A = \{0, 1\}$
- $\mathcal{C} = \{\langle s_i, R_{c_i} \rangle \mid i \in [k]\}$, *where* s_i *is the tuple of all the literals in* c_i *and* R_{c_i} *consists of all the tuples that satisfy* c_i.

Example 2.2.4. *Let*

$$\begin{cases} e_1 : a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n & = b_1 \\ e_2 : a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n & = b_2 \\ \vdots & \vdots \\ e_m : a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n & = b_m \end{cases}$$

be a system of m *linear equations with* n *indeterminates, where* $x_1, \ldots, x_n$ *are the indeterminates,* $a_{11}, a_{12}, \ldots, a_{mn}$ *are the coefficients, and* $b_1, \ldots, b_m$ *are the constant terms. The*

matrix representation is then of the form $\boldsymbol{Ax=b}$

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix},$$

where $\boldsymbol{A}$ *is an* $m \times n$ *matrix and* $\boldsymbol{x}$ *and* $\boldsymbol{b}$ *are column vectors with* n *and* m *entries, respectively. A solution to a system of linear equations, over some field* $\mathbb{F}$*, is one that assigns values to each of the variables such that all the equations are satisfied. Therefore, the corresponding CSP instance* $(V, A, \mathcal{C})$ *for this problem asks whether such an assignment exists.*

- $V = \{x_1, \ldots, x_n\}$
- $A =$ *the elements of field* $\mathbb{F}$
- $\mathcal{C} = \{\langle s_i, R_{e_i} \rangle \mid i \in [m]\}$*, where* s_i *is the tuple of all the variables with non-zero coefficients in* e_i *and* R_{e_i} *consists of all the tuples that satisfy* e_i*. In short, the constrains are the equations.*

2.3 Promise Constraint Satisfaction Problem (PCSP)

PCSP is a fairly new and exciting variant of CSP which was introduced with the motivation of finding approximately good solutions to an instance of a typically hard problem with the promise that a good solution is guaranteed to exist. In a PCSP instance, every constraint is associated with a 'strict' and a 'relaxed' relation. Given a PCSP instance, the goal is to differentiate between the case when it has a solution subject to the strict constraint from the case when, even subject to the relaxed constraint, it does not have a solution. More formally, we define PCSP as follows.

Definition 2.3.1. Let $\mathbf{A}$ and $\mathbf{B}$ be a pair of similar relational structures such that $\mathbf{A} \to \mathbf{B}$. For a given input structure $\mathbf{I}$, the decision version of PCSP$(\mathbf{A}, \mathbf{B})$ outputs YES if $\mathbf{I} \to \mathbf{A}$ and NO if $\mathbf{I} \not\to \mathbf{B}$. In addition, it is always the case that either $\mathbf{I} \to \mathbf{A}$ or $\mathbf{I} \not\to \mathbf{B}$, which is the promise in the PCSP. The pair ($\mathbf{A}$, $\mathbf{B}$) is called the PCSP *template*.

Note that when the strict form and the relaxed form of each constraint coincide, the result is the standard CSP. This means that the PCSP framework generalizes CSP.

Remark 2.3.1. *PCSP(*$\boldsymbol{A}, \boldsymbol{A}$*) is the same as CSP(*$\boldsymbol{A}$*).*

Example 2.3.1. *The Approximate Graph Colorability, parameterized by two natural numbers* c *and* k*,* $k \leq c$*, is the problem of deciding whether a given graph* $\boldsymbol{I}$ *is* k*-colorable or is not* c*-colorable. This problem can be formulated as PCSP(*$\boldsymbol{A}, \boldsymbol{B}$*), where* $\boldsymbol{A} = \boldsymbol{K}_k$ *and* $\boldsymbol{B} = \boldsymbol{K}_c$*.*

Example 2.3.2. *[4] Take the following relational structures*

$$\boldsymbol{T} = (\{0,1\} : (1,0,0),(0,1,0),(0,0,1)),$$
$$\boldsymbol{H}_2 = (\{0,1\} : \{0,1\}^3 \setminus \{(0,0,0),(1,1,1)\}).$$

The classic NP-hard CSP problems of these relational structures, namely $CSP(\boldsymbol{T})$ and $CSP(\boldsymbol{H}_2)$ are called 1-in-3 and Not-all-equal-Sat, respectively. We can naturally define $PCSP(\boldsymbol{T}, \boldsymbol{H}_2)$, which is proven to be in P [13, 12].

The CSP and PCSP constitute a broad range of problems and have played an important role in the development of computational complexity theory. Naturally for such a broad subject, multiple questions can be asked about a CSP instance. The original form of CSP, as we defined earlier in 2.2, is a *decision* problem, i.e., we need to decide whether an instance has a solution or not.

There are other versions of the CSP that include:

- The *search* problem, where we want to find a feasible solution for a given CSP instance.
- The *counting* problem, where we want to find how many solutions a given CSP instance has.
- The *Max-CSP*, where given a CSP instance, we want to find a solution that satisfies the most number of constraints as possible, even when the instance does not have a solution. Since this is an optimization problem, it requires using approximation approaches.
- The *valued* CSP, where given a CSP instance we assign values to tuples of constraint relations and seek a solution that maximizes (or minimizes) the weight.

2.4 Algebraic Tools

It was conjectured that for each finite constraint language Γ, the decision CSP(Γ) is either in P or NP-complete [23]. Both independent proofs of the conjecture, [18], [41], are heavily based on the algebraic approach to CSP which includes using *polymorphisms*, multivariate functions that preserve relations in a constraint language, and consequently solution sets of the corresponding CSPs. The concept of polymorphism links relations and operations and provides insight into complexity of constraint problems.

Definition 2.4.1. We define a polymorphism of a k-ary relation R on a set A to be a mapping $f : A^n \to A$ that preserves R; meaning that for any $\mathbf{a}_1, \ldots, \mathbf{a}_n \in R$ where each tuple $\mathbf{a}_i = (a_i[1], a_i[2], \ldots, a_i[k])$ we have

$$(f(a_1[1], \ldots, a_n[1]), \ldots, f(a_1[k], \ldots, a_n[k])) \in R.$$

To clarify, if we take a $k \times n$ matrix where the columns are $\mathbf{a}_1, \ldots, \mathbf{a}_n \in R$ and apply f on rows of this matrix, the resulting tuple is also a member of R. In this case, we call R to be *invariant* with respect to f.
A polymorphism of a relational structure $\mathbf{A} = (A; R_1^{\mathbf{A}}, \ldots, R_l^{\mathbf{A}})$ is a mapping $f : A^n \to A$ such that for each $i \leq l$ and all $\mathbf{a}_1, \ldots, \mathbf{a}_n \in R_i^{\mathbf{A}}$ where $\mathbf{a}_i = (a_i[1], a_i[2], \ldots, a_i[ar(R_i)])$

$$(f(a_1[1], \ldots, a_n[1]), \ldots, f(a_1[ar(R_i)], \ldots, a_n[ar(R_i)])) \in R_i^{\mathbf{A}}.$$

We denote the set of all polymorphisms of $\mathbf{A}$ by $Pol(\mathbf{A})$. For two similar relational structures $\mathbf{A}$ and $\mathbf{B}$, a polymorphism from $\mathbf{A}$ and $\mathbf{B}$ is a mapping $f : A^n \to B$, defined similarly as above, that is, for each $i \leq l$ and all $\mathbf{a}_1, \ldots, \mathbf{a}_n \in R_i^{\mathbf{A}}$ where $\mathbf{a}_i = (a_i[1], a_i[2], \ldots, a_i[ar(R_i)])$

$$(f(a_1[1], \ldots, a_n[1]), \ldots, f(a_1[ar(R_i)], \ldots, a_n[ar(R_i)])) \in R_i^{\mathbf{B}}.$$

The set of all such polymorphisms is denoted by $Pol(\mathbf{A}, \mathbf{B})$.

Example 2.4.1. *Take the relational structures from example 2.3.2. We can verify that* $Pol(\boldsymbol{T})$ *consists of the projections on the set* $\{0, 1\}$ *and* $Pol(\boldsymbol{H}_2)$ *consists of operations* $\pi(p_i^{(n)})$ *in which* $p_i^{(n)}$ *is a projection and* π *is a permutation on the set* $\{0, 1\}$.

Definition 2.4.2. An L-ary polymorphism f is said to be *symmetric* if for any permutation α in the symmetric group of size L), we have $f(x_1, ..., x_L) = f(x_{\alpha(1)}, ..., x_{\alpha(L)})$.

Example 2.4.2. *Take the system of linear equations in example 2.2.4 over the field* $\mathbb{Z}_p$ *of integers modulo p. Then the* affine *operation* $g(x, y, z) = x - y + z$ *is an example of a polymorphism, since for any* $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}$ *that are solutions to the linear system, so is* $\boldsymbol{u} - \boldsymbol{v} + \boldsymbol{w}$. *This is because u is a solution if and only if* $\boldsymbol{A}\boldsymbol{u} = \boldsymbol{b}$ *and we need to show that* $\boldsymbol{u} - \boldsymbol{v} + \boldsymbol{w}$ *is also a solution, meaning* $\boldsymbol{A}(\boldsymbol{u} - \boldsymbol{v} + \boldsymbol{w}) = \boldsymbol{b}$, *which is trivial to prove. However, g, which satisfies the Mal'tsev identities* $g(x, y, y) = g(y, y, x) = x$ *for all* $x, y \in R$, *is not symmetric.*

There exists a similar operation, which is interreplaceable with the Mal'tsev operation, and is symmetric. This $p+1$*-ary mapping over the same field is defined as* $f(x_1, \ldots x_{p+1}) = x_1 + x_2 + \ldots + x_{p+1}$ *and, as promised, can be used instead of the Mal'tsev operation as follows:*
Without loss of generality, take $p = 3$. *With the setting we have described in this example, we show that* $x_1 + x_2 + x_3 + x_4$ *can be transformed to* $x - y + z$ *and vise versa by means of substitutions, which implies that they have the same invariant relations. Starting with*

$x_1 + x_2 + x_3 + x_4 = f(x_1, x_2, x_3, x_4)$, *we define* $h(x, y, z) = f(x, y, y, z)$. *Then*

$$\begin{aligned} h(x,y,z) = f(x,y,y,z) &= x + y + y + z \\ &= x + 2y + z \\ &\equiv x - y + z \pmod 3 \\ &= g(x,y,z). \end{aligned}$$

Conversely, take $g(x, y, z) = x - y + z$. *With substitution of variables, introducing* $v \in R$, *we have,*

$$\begin{aligned} g(g(x,y,z),y,v) = (x - y + z) - y + v &= x - y + z - y + v \\ &\equiv x + 2y + z + 2y + v \pmod 3 \\ &\equiv x + y + z + v \pmod 3 \\ &= f(x,y,z,v). \end{aligned}$$

An extension of symmetric polymorphisms is the class of block-symmetric polymorphisms, where a mapping f is said to be *block-symmetric* if its coordinates can be partitioned into several blocks such that f is invariant under permutations within each block.

Definition 2.4.3. A mapping $f : A^n \to B$ is said to be *block-symmetric* if there exists a partition of the coordinates of f into blocks $B_1 \cup \ldots \cup B_k = [L]$ such that f is invariant under permutations within each block B_i, meaning that the mapping output remains unchanged regardless of the arrangements of the coordinates within each block. In addition, the minimum size of any block is defined as the *width* of f.

Example 2.4.3. *An example of a block-symmetric polymorphism is the alternating threshold operation, defined as*

$$AT(x_1, \ldots, x_L) = 1[x_1 - x_2 + x_3 - \ldots \pm x_L \geq 1],$$

where $AT(x_1, \ldots, x_L)$ *equals* 1 *if* $x_1 - x_2 + x_3 - \ldots \pm x_L \geq 1$ *and* -1 *otherwise. In this case, the blocks are formed by the odd and even coordinates and the width is not bounded.*

Next, we introduce a closure operator on the set of mappings.

Definition 2.4.4. For an n-ary function $f : A^n \to B$ and an m-ary function $g : A^m \to B$ and a given map $\pi : [m] \to [n]$, we call f a *minor* of g, and we write $g = f^\pi$, if for all $x_1, \ldots, x_n \in A$

$$f((x_1, \ldots, x_n) = g((x_{\pi(1)}, \ldots, x_{pi(m)})$$

As is easily seen, $Pol(\mathbf{A})$ is closed under composition of functions for any $\mathbf{A}$. However, $Pol(\mathbf{A}, \mathbf{B})$ is generally not closed under composition, but it is always closed under taking minors.

Definition 2.4.5. Let $\mathcal{O}(A, B) = \{f : A^n \to B \mid n \geq 1\}$ for the pair of sets (A, B). Then, a *(function) minion* $\mathcal{M}$ on this pair is a non-empty subset of $\mathcal{O}(A, B)$ which is also closed under taking minors. In other words, if $f \in \mathcal{M}$ then $g = f^\pi \in \mathcal{M}$ for any $\pi : [ar(f)] \to [ar(g)]$. Furthermore, we denote the set of n-ary functions of $\mathcal{M}$ by $\mathcal{M}^{(n)}$ for $n \geq 1$.

When studying the CSP and PCSP, we make use of polymorphisms, which offers a compact representation of extensive sets of relations. However, we also leverage the language of universal algebra, which provides an even more structured and expansive framework. We start by introducing some definitions.

Definition 2.4.6. Given a set A, called the *universe*, and F, a set of operations over A which are called *basic*, a *(universal) algebra* is a pair $\mathcal{A} = (A, F)$.

We will need some basic constructions on algebras.

Definition 2.4.7. Let $\mathcal{A} = (A, F)$ be an algebra and B a subset of A such that for any (k-ary) operation $f \in F$ and any $b_1, \dots b_k \in B$ we have $f(b_1, \dots b_k) \in B$. Then the algebra $\mathcal{B} = (B, F|_B)$ is a *subalgebra* of A. If $A \neq B$, then $\mathcal{B}$ is said to be *proper*.

An example of an algebra is a *semilattice*, which is defined as below.

Definition 2.4.8. An algebra $\mathcal{S} = (S, \{\cdot\})$, where $\cdot$ is a semilattice operation, which means it satisfies the equations $x \cdot x = x$, $x \cdot y = y \cdot x$, and $x \cdot (y \cdot z) = (x \cdot y) \cdot z$, is called a semilattice.

Another fundamental concept used later on is *congruence*, which is defined as the following.

Definition 2.4.9. For an algebra $\mathcal{A} = (A, F)$, let $\mathrm{Inv}(F)$ denote the the set of all relations on A that are preserved by every operation from F. A congruences is an equivalent relation $\theta \in \mathrm{Inv}(F)$, meaning that for any operation $f \in F$ and any $(a_1, b_1), \dots, (a_k, b_k) \in \theta$ it holds that $(f(a_1, \dots, a_k), f(b_1, \dots, b_k)) \in \theta$.

Let $\mathcal{A}$ be any algebra. Then the equality relation $\underline{0}_{\mathcal{A}}$ and the full binary relation $\underline{1}_{\mathcal{A}}$ on $\mathcal{A}$ are congruences of $\mathcal{A}$.

2.5 Methods of Solving CSP

We start by introducing some well-known methods of solving CSPs, the decision problem in particular.

2.5.1 Local Consistency

A large group of algorithms that were suggested to solve CSP rely on some kind of local propagation. These algorithms execute a propagation procedure that can either prove that a given instance is unsatisfiable or modify it into a state that is locally consistent, while

keeping the set of solutions unchanged. Here, locally refers to the fact that at each step of such algorithms we consider only a bounded number of variables or constraints. Then, we can say that the CSP instance has a solution if the locally consistent instance has a solution. We start by defining *arc-consistency*.

Definition 2.5.1. Given a CSP instance $\mathcal{P} = (V, A, \mathcal{C})$, we say it is arc-consistent if the following condition is satisfied: For any $\langle s_1, R_1\rangle$, $\langle s_2, R_2\rangle$, $I = s_1 \cap s_2$, and any $\mathbf{a} \in R_1$, there exists $\mathbf{b} \in R_2$ such that $\mathrm{pr}_I \mathbf{a} = \mathrm{pr}_I \mathbf{b}$.

The following algorithm takes a CSP instance $\mathcal{P}$ and return an arc-consistent instance $\mathcal{P}'$ with the same set of solutions as $\mathcal{P}$ (if any exists).

Algorithm 1 Arc-consistency algorithm

Require: A CSP instance $\mathcal{P} = (V, A, \mathcal{C})$
Ensure: An arc-consistent instance $\mathcal{P}' = (V, A, \mathcal{C}')$ with the same set of solutions as $\mathcal{P}$
1: $\mathcal{P}' = (V, A, \mathcal{C}') \coloneqq \mathcal{P}$ where $\mathcal{C}' = \{\langle s, R'\rangle | \langle s, R\rangle \in \mathcal{C}, R' = R\}$
2: **repeat**
3: **for** $\langle s_1, R_1\rangle, \langle s_2, R_2\rangle \in \mathcal{C}'$ such that $I = s_1 \cap s_2 \neq \emptyset$ and any $\mathbf{a} \in R_1$ **do**
4: **if** there exists no $\mathbf{b} \in R_2$ such that $\mathrm{pr}_I \mathbf{a} = \mathrm{pr}_I \mathbf{b}$ **then**
5: Remove $\mathbf{a}$ from R_1
6: **end if**
7: **end for**
8: **until** $\mathcal{P}'$ is arc-consistent

Arc-consistency can be generalized by considering tuples (instead of pairs) of variables (instead of constraints). This leads us to another version of local consistency, namely (k, l)-consistency. We start with the notion of *weak partial solution*.

Definition 2.5.2. For a CSP instance $\mathcal{P} = (V, A, \mathcal{C})$ and $W \subseteq V$, a mapping $\phi : W \to A$ is a weak partial solution of $\mathcal{P}$ on W if for every $\langle s, R\rangle \in \mathcal{C}$ such that $s \subseteq W$, it holds that $\phi|_s \in R$.

Next, we define (k, l)-consistency.

Definition 2.5.3. Let $\mathcal{P} = (V, A, \mathcal{C})$ be a CSP instance and $k \leq l$ be natural numbers. Then $\mathcal{P}$ is said to be (k, l)-consistent if for any $U \subseteq W \subseteq V$ such that $|U| \leq k$, $|W| \leq l$, any weak partial solution of $\mathcal{P}$ on U can be extended to a weak partial solution on W. Plus, we say Instance $\mathcal{P}$ is k-consistent if it is $(k, k+1)$-consistent.

The following algorithm takes a CSP instance and returns a (k, l)-consistent instance with the same set of solutions.

Example 2.5.1. *Let $\mathcal{P} = (V, A, \mathcal{C})$ be a CSP instance given by*

- $V = \{x, y, z\}$

Algorithm 2 (k, l)-consistency algorithm

Require: A CSP instance $\mathcal{P} = (V, A, \mathcal{C})$
Ensure: A (k, l)-consistent instance $\mathcal{P}' = (V, A, \mathcal{C}')$ with the same set of solutions as $\mathcal{P}$

```
for every U ∈ V, U ≤ k do create a constraint ⟨U, R_U⟩, where R_U is the set of all weak partial solutions of P on U
end for
P' := (V, A, C') where C' = C ∪ {⟨U, R_U⟩ | U ⊆ V, |U| ≤ k}
repeat
    for U, W ⊆ V, |U| ≤ k, |W| = l, U ⊆ W and any a ∈ R_U do
        if there exists no weak partial solution φ of P' on W such that φ|_U = a then
            Remove a from R_U
        end if
    end for
until P' is (k, l)-consistent
```

- $A = \{0, 1\}$
- $\mathcal{C} = \{\langle (x, y, z), R_1\rangle, \langle (x, y), R_2\rangle, \langle (x, y, z), R_3\rangle\}$, *where*

$$R_1 = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \; R_2 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}, \; R_3 = A^3.$$

Then $\mathcal{P}$ is not arc-consistent because for $\langle (x, y, z), R_1\rangle, \langle (x, y), R_2\rangle$, $(0, 1, 1) \in R_1$ there is no tuple $\boldsymbol{b} \in R_2$ such that $pr_{\{x,y\}}(0, 1, 1) = (0, 1) = pr_{\{x,y\}}\boldsymbol{b}$. However, $\mathcal{P}$ is (2,3)-consistent since we can extend every assignment on a 2-element set to a weak solution.

For a wide range of constraint languages the existence of any set of local consistent solutions implies the existence of a global solution. Such constraint languages are said to have *bounded width*. It was first conjectured and later proved that CSP that lack bounded width property can simulate relations to be encoded as linear equations over finite field [7]. Atserias et.al in [1] proved another limitation of this type of methods,: the k-consistency algorithm is not always correct when the template of a CSP instance does not have tree-width of at most k.

2.5.2 Gaussian Elimination

Another method of solving CSP is called Gaussian elimination, which comes from basic linear algebra. Unlike the already-mentioned algorithms which cannot solve linear equations, Gaussian elimination can solve them. An appropriate extension of linear algebra problems are CSPs over relations admitting a compact representation that can be solved by generalizations of Gaussian elimination. A compact representation of a relation is its subset that has cardinality polynomial in the arity of the relation, and uniquely determining the relation

given its polymorphisms. For an algebra $\mathcal{A} = (A, F)$, this property can be formalized using *few subpowers* property.

Definition 2.5.4. Let $\mathcal{A} = (A, F)$ be an algebra. Then $\mathcal{A}$ is said to be an algebra with few subpowers if there is a polynomial p such that for every $k \in \mathbb{N}$ and every k-ary relation R invariant under F, there is a *generating set* Q of R such that $|Q| \leq p(k)$. A set $Q \subseteq A^k$. is a generating set of a k-ary relation R over A invariant under F, if Q generates R as a subalgebra of $\mathcal{A}^k$.

This definition is followed by the theorem below.

Theorem 2.5.1. *There is a $k+1$-ary operation that for any $x, y \in \mathcal{A}$ satisfies the following identities:*

$$\begin{aligned}
f(x, x, y, y, y, \ldots, y, y) &= y, \\
f(x, y, x, y, y, \ldots, y, y) &= y, \\
f(y, y, y, x, y, \ldots, y, y) &= y, \\
f(y, y, y, y, x, \ldots, y, y) &= y, \\
&\ldots \\
f(y, y, y, y, y, \ldots, y, x) &= y.
\end{aligned}$$

Algebras with few subpowers property have been completely characterized by Idziak et. al [28]. While it is notable to recognize the method that solves the linear equation CSP problem, further discussion on the Gaussian elimination method is out of the scope of this thesis.

2.5.3 Basic Linear and Integer Programming

Convex relaxations are among the powerful methods of designing exact and approximate algorithms for the CSP. The general idea is to formulate a CSP instance as an integer problem and later relax it to a convex problem such as linear (LP) or semidefinite program (SDP). We start by introducing the foundations of basic integer and linear programming.

An instance $\mathcal{P} = (V, A, \mathcal{C})$, $V = \{v_1, \ldots, v_n\}, \mathcal{C} = \{C_1, \ldots, C_m\}$, of CSP($\Gamma$) can be equivalently described as a *canonical 0-1 integer program*, denoted by IP($\mathcal{P}$). For an assignment $V \to A$ of a variable, we introduce a variable $w_i(a)$, where $a \in A$ and $i \in [n]$ is the index of the variable. Since we are establishing an integer programming problem where the involved variables can take value 0 or 1, when $w_i(a) = 1$, we interpret that the variable v_i is assigned the value a. We also consider variables $p_j(\mathbf{y})$, where $j \in [m]$ is the index of a constraint $C_j = \langle s_j, R_j \rangle$ and $\mathbf{y} \in R_j^A$ is a potential assignment to s_j. More formally, the resulting basic

integer program $\mathrm{IP}(\mathcal{P})$ with the described variables is as follows:

$$\begin{array}{rcll} w_i(a) & \in & \{0,1\} & \forall i \in [n]\,, \quad a \in A \\ p_j(y) & \in & \{0,1\} & \forall j \in [m]\,, \quad y \in R_j^{\mathbf{A}} \\ \sum\limits_{a \in A} w_i(a) & = & 1 & \forall i \in [n] \\ \sum\limits_{y \in R_j^{\mathbf{A}}} p_j(y) & = & 1 & \forall j \in [m] \\ \sum\limits_{y|_i = a} p_j(y) & = & w_i(a) & \forall i \in [n]\,, \quad j \in [m] \end{array} \tag{2.1}$$

We can relax $\mathrm{IP}(\mathcal{P})$ by letting the variables to take any value in the range $[0,1]$ instead of $\{0,1\}$ to obtain $\mathrm{LP}(\mathcal{P})$, the *basic linear program*. This way, $w_i(a)$ is interpreted as a probability distribution for the assignment $V \to A$ of a variable and still sums up to 1. Similarly, a probability distribution over the satisfying assignments to each constraint is $p_j(\mathbf{y})$. Just like in the basic integer programming setup, for every i, a, $w_i(a)$ equals the marginal probability distribution of variable v_i in every constraint. The formal basic linear programming $\mathrm{LP}(\mathcal{P})$ with the described variables is the following:

$$\begin{array}{rcll} w_i(a) & \geq & 0 & \forall i \in [n]\,, \quad a \in A \\ p_j(y) & \geq & 0 & \forall j \in [m]\,, \quad y \in R_j^{\mathbf{A}} \\ \sum\limits_{a \in A} w_i(a) & = & 1 & \forall i \in [n] \\ \sum\limits_{y \in R_j^{\mathbf{A}}} p_j(y) & = & 1 & \forall j \in [m] \\ \sum\limits_{y|_i = a} p_j(y) & = & w_i(a) & \forall i \in [n]\,, \quad j \in [m] \end{array} \tag{2.2}$$

Sherali and Adams proposed [38] a hierarchy of linear programming relaxations that captures the process of enhancing convex relaxations by incorporating additional constraints that are fulfilled by an integer solution. These constraints, although larger in scope, contribute to generating more robust relaxations. Nevertheless, even the stronger versions of convex relaxations have limitations as to the CSP problems they can solve or approximate [39]. Yet another kind of relaxations, affine relaxations, are also a newly found method of solving CSPs. They are introduced in Section 2.6.1.

2.6 Methods of Solving PCSP

We now turn to the PCSP. We introduce some methods of solving the PCSP. Note first that the all the techniques that solve PCSP also apply to CSPs. To some extent the converse is also true: the relaxations that solve CSPs are often applicable to PCSPs and inspire methods of solving PCSP instances.

2.6.1 Basic Linear Programming + Affine Relaxation

This algorithm, developed by Brakensiek and Guruswami in [14] and inspired by the well-studied methodology in the CSP, consists of two main parts. In the first part, we consider the canonical linear programming relaxation and for a CSP instance $\mathcal{P}$ it is defined exactly like the system of constraints defined in Equation 2.2. We let $LP_{\mathbb{Q}}(\mathcal{P}, \mathbf{A})$ represent the rational polytope of the solutions.

The second part of the method is the affine relaxation of a PCSP. For the affine relaxation the system of a linear constraints is the same as above, but we replace the condition that each variable is a non-negative rational number with the condition that it is an integer number (positive or not). Therefore, we let $r_i(a) \in \mathbb{Z}$ replace $w_i(a)$ and $q_j(y) \in \mathbb{Z}$ replace $p_j(y)$. The explicit system is shown below and, similarly, we let $Aff_{\mathbb{Z}}(\mathcal{P}, \mathbf{A})$ denote the set of solutions:

$$\begin{aligned} \sum_{a \in A} r_i(a) &= 1 \qquad \forall i \in [n] \\ \sum_{y \in R_j^{\mathbf{A}}} q_j(y) &= 1 \qquad \forall j \in [m] \\ \sum_{y|_i = a} q_j(y) &= r_i(a) \quad \forall i \in [n]\,, \quad j \in [m] \end{aligned} \tag{2.3}$$

The algorithm runs in three steps and involves finding solutions in the mentioned sets (polytopes). Both linear systems can be solved in polynomial time via the algorithm proposed in [30]. In certain cases the algorithm correctly solves the input PCSP instance $\mathcal{P}$ in polynomial time.

Algorithm 3 BLP + Affine algorithm

Require: An instance $\mathcal{P}$ of PCSP($\mathbf{A}$, $\mathbf{B}$)
Ensure: Accepts instance $\mathcal{P}$ satisfiable in $\mathbf{A}$ and rejects it if unsatisfiable in $\mathbf{B}$

1: **if** There exists a relative interior point in $LP_{\mathbb{Q}}(\mathcal{P}, \mathbf{A})$ exists **then** Find it
2: **else** Reject
3: **end if**
4: Refine $Aff_{\mathbb{Z}}(\mathcal{P}, \mathbf{A})$ to $Aff'_{\mathbb{Z}}(\mathcal{P}, \mathbf{A})$ by discarding assignments to constraints which have weight 0 according to the relative interior point. Namely, let $r_i(a)$ be 0 whenever $w_i(a)$ is, and requiring $q_i(y)$ to be 0 whenever $p_i(y)$ is.
5: **if** $Aff'_{\mathbb{Z}}(\mathcal{P}, \mathbf{A})$ is empty **then** Reject
6: **else** Accept
7: **end if**

Theorem 2.6.1 (Theorem 3.2 of [14]). *Let($\mathbf{A}$, $\mathbf{B}$) be a promise template (over any finite domains) such that Pol($\mathbf{A}$, $\mathbf{B}$) has symmetric polymorphisms of arbitrarily large arities. Then, the BLP+Affine algorithm correctly solves the decision PCSP($\mathbf{A}$, $\mathbf{B}$).*

Furthermore, the algorithm's exact effectiveness is established by the presence of *block-symmetric* polymorphisms, as mentioned in Definition 2.4.3, which are both necessary and sufficient for its functioning.

Theorem 2.6.2 (Theorem 4.1 of [14]). *Let $(\boldsymbol{A}, \boldsymbol{B})$ be a promise template (over any finite domain) such that Pol$(\boldsymbol{A}, \boldsymbol{B})$ has block-symmetric polymorphisms of arbitrarily large width. Then, the BLP+Affine algorithm correctly solves the decision PCSP$(\boldsymbol{A}, \boldsymbol{B})$.*

Going back to one of the main examples of this thesis, systems of linear equations have the right structure for BLP+Affine algorithm to be applied on.

Example 2.6.1. *From Example 2.4.1, we know that the system of linear equations admits a symmetric polymorphism. Furthermore, it has at least one block-symmetric polymorphism. Therefore, it is easy to verify that it is solvable by BLP+Affine algorithm.*

2.6.2 Cohomology

In [20], Conghaile introduced a novel algorithm based on concepts of cohomology and inspired by quantum foundations, category theory, and proved that it is stronger than k-consistency. The paper has established a setting and formulation that is beyond the scope of this thesis but the essence of the approach is close to Affine Integer Programming (AIP), introduced earlier. Additionally, the input requires to be k-consistent for some k. The proposed cohomological k-consistency method is equivalent to the following algorithm, which takes a k-consistent CSP instance and constructs an Affine Integer Programming (AIP) relaxation by introducing some additional marginal conditions. Then it continues to check that for each of the variables in the relaxation there exists an integer solution with the condition that that variable equals 1. If yes, the instance passes the cohomology test and is accepted; otherwise the added constraint is removed and the process continues until the instance is cohomologically consistent.

Recall that any instance $\mathcal{P} = (V, A, \mathcal{C})$ of CSP(Γ) can be similarly described as a *canonical integer program*, denoted by IP$(\mathcal{P})$. For an assignment $V = \{v_1, \ldots, v_n\} \to A$, as in Section 2.6.1, we again introduce $r_i(a) \in \mathbb{Z}$, where, $a \in A$ and $i \in [n]$. Furthermore, we consider variable $q_j(\mathbf{y}) \in \mathbb{Z}$, where $j \in [m]$ is the index of constraint $C_j = \langle s_j, R_j \rangle$ and $\mathbf{y} \in R_j^A$ is a potential assignment to s_j. What differentiates this system of linear equations from the previous settings is what follows:

1. The algorithm cycles through the introduced variables and creates an additional constraint by assigning one of the introduced variables (for all $r_i(a)$ and $q_j(\mathbf{y})$) to 1 separately and adding it to the system. Then it attempts to find a solution for the new system. If no feasible solution is found, the algorithm removes the a (or $\mathbf{y}$) from the set of possible values of v_i ($\mathbf{y}$ from the corresponding constraint relation) and continues with adding the next constraint. If, despite adding the new constraint, a solution is found, we remove the added constraint and repeat the process by setting the next variable to 1 and creating a new system with an additional constraint. Finally, the algorithm ensures the modified solution without cohomological obstractions

is found for each of the systems created or declares inconsistency if for at least one system no solution is found.

2. There is no requirement of $r_i(a)$ and $q_j(y)$ to sum to 1.

For each round of the algorithm the explicit system is shown below and, similarly, we let $Aff_{\mathbb{Z}}(\mathbf{I}, \mathbf{A})$ denote the submodule of solutions:

$$\begin{array}{rcll} \sum\limits_{y|_i=a} q_j(y) & = & r_i(a) & \forall i \in [n]\,, \quad j \in [m] \\ r_k(a) & = & 1 & \forall k \in [n] \\ (or\ q_k(y) & = & 1 & \forall k \in [m]) \end{array} \tag{2.4}$$

The cohomology-based algorithm described above is as follows:

Algorithm 4 Cohomology algorithm

Require: A k-consistent instance $\mathcal{P}$ of CSP($\mathbf{A}$)
Ensure: An updated instance $\mathcal{P}'$ without cohomological obstractions

```
for each s ∈ r_i(a) ∪ q_j(y) do
    Add s = 1 to Aff_ℤ(P, A)
    if There exists a solution in LP_ℚ(P, A) then Find it
    else remove s = 1 from constraints
    end if
    if Aff_ℤ(P, A) is empty then Reject
    else Continue
    end if
end for
Accept
```

Aside from promise templates which admit symmetric or block-symmetric polymorphisms, there exists a tractable template which is solvable in polynomial time but does not admit any trivial block-symmetric polymorphism and, thus, is not solvable by BLP + Affine algorithm[14]. Therefore, the search for algorithms that solve all or at least a larger group of promise templates must continue. A natural alternative for the BLP+AIP algorithm would be to replace the BLP section of this approach with local consistency methods or Cohomology. The power of these combinations have not been thoroughly studied and we also do not know either they would solve all tractable PCSPs, but there exists a general belief over the power of the combination of such linear programming and consistency approaches.

Conjecture 2.6.3. *The combination of some level of consistency and the linear programming or cohomology solves at least all the polynomial-time CSPs or PCSPs.*

So far, no known CSP or PCSP instance has been suggested as a counterexample of this conjecture. Therefore, a majority of the community believe that this conjecture is true. In Chapter 3, we introduce the main contribution of this work and provide a counter example that disproves this conjecture in some cases.

Chapter 3

Where Cohomology Fails

As mentioned in section 2.5.3, it is generally believed that all polynomial time PCSP templates, or any polynomial time solvable CSP problems, are solvable by the algorithms mentioned there. These methods include a combination of checking the consistency of the instance followed by finite relaxation or cohomology methods. The main contribution of this thesis is to provide a counter example that disproves this for some of those combinations. More specifically, in the next section we introduce an algebra which gives rise to a CSP instance, which eventually refutes conjecture 2.6.3.

3.1 The Counter-example Algebra

Let $\mathcal{A} = (A, F)$ be an algebra with universe $A = \{0, 1, 2\}$ and two operations: a ternary operation f such that $f(x, y, z) = x + y + z \pmod 2$ if $x, y, z \in \{0, 1\}$ and $f(x, y, z) = x$ otherwise; and a binary operation $g(x, y) = 0$ if $\{x, y\} = \{0, 2\}$, $g(2, 2) = 2$, and $g(x, y) = 1$ if $\{x, y\} = \{1, 2\}$.

Let $R, T, S, Q \subseteq A^k$, $k \in \{1, 5\}$ be defined as below.

$$R = \begin{pmatrix} 1&0&0&0&1&1&1&1&1&0&0&0&1&1&1&0&2 \\ 0&1&0&0&1&1&0&0&0&1&0&0&1&1&0&1&2 \\ 0&0&1&0&1&0&1&0&0&0&1&0&1&0&1&1&2 \\ 0&0&0&1&0&1&1&0&0&0&0&1&0&1&1&1&2 \\ 0&0&0&0&0&0&0&0&2&2&2&2&2&2&2&2&2 \end{pmatrix},$$

$$T = \begin{pmatrix} 0&2&0&0&0&2&2&2&0&0&0&2&2&2&0&0&2&0&0&0&2&2&2&0&0&0&2&2&2&0&2 \\ 0&0&2&0&0&2&0&0&2&2&0&2&2&0&2&0&0&2&0&0&2&0&0&2&2&0&2&2&0&2&2 \\ 0&0&0&2&0&0&2&0&2&0&2&2&0&2&2&0&0&0&2&0&0&2&0&2&0&2&2&0&2&2&2 \\ 0&0&0&0&2&0&0&2&0&2&2&0&2&2&2&0&0&0&0&2&0&0&2&0&2&2&0&2&2&2&2 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&2&2&2&2&2&2&2&2&2&2&2&2&2&2&2&2 \end{pmatrix},$$

$$S = \begin{pmatrix} 0&2&0&0&0&2&2&2&0&0&0&2&2&2&0 \\ 0&0&2&0&0&2&0&0&2&2&0&2&2&0&2 \\ 0&0&0&2&0&0&2&0&2&0&2&2&0&2&2 \\ 0&0&0&0&2&0&0&2&0&2&2&0&2&2&2 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \end{pmatrix}$$

$$Q = \begin{pmatrix} 0 \end{pmatrix}.$$

We verify that the relations R, T, S are invariant under the operations f, g.

Lemma 3.1.1. *The relation R is invariant under the operations f and g.*

Proof. Let $\mathbf{x}, \mathbf{y}, \mathbf{z} \in R$. To show that R is invariant under f, it is sufficient to show that $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) \in R$.

- Case 1: $(2,2,2,2,2) \in \{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$
  In this case, by definition of f, $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (2,2,2,2,2) \in R$ if $(\mathbf{x}) = (\mathbf{y}) = (\mathbf{z}) = (2,2,2,2,2)$, or $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = (\mathbf{x}) \in R$ otherwise.
- Case 2: $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \{0,1\}^4 \times \{0,2\}$
  In this case, for the first four coordinates of $\mathbf{x}, \mathbf{y}, \mathbf{z}$,

$$\begin{aligned} f(\mathrm{pr}_{[4]}\mathbf{x}, \mathrm{pr}_{[4]}\mathbf{y}, \mathrm{pr}_{[4]}\mathbf{z}) &= \mathrm{pr}_{[4]}\mathbf{x} + \mathrm{pr}_{[4]}\mathbf{y} + \mathrm{pr}_{[4]}\mathbf{z} \\ &= (\mathbf{x}[1] + \mathbf{y}[1] + \mathbf{z}[1], \mathbf{x}[2] + \mathbf{y}[2] + \mathbf{z}[2], \\ &\quad \mathbf{x}[3] + \mathbf{y}[3] + \mathbf{z}[3], \mathbf{x}[4] + \mathbf{y}[4] + \mathbf{z}[4]) \\ &\in \mathrm{pr}_{[4]}R \end{aligned}$$

  By the structure of relation R, the parity of every tuple is odd and there are odd number of tuples in the input of f. Therefore, the total parity is odd and regardless of the fifth coordinate, $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) \in R$.

Similarly, to show R is invariant under g, it is sufficient to show that for $\mathbf{x}, \mathbf{y} \in R$ it holds that $g(\mathbf{x}, \mathbf{y}) \in R$. Firstly, let $(2,2,2,2,2) \in \{\mathbf{x}, \mathbf{y}\}$. It is easy to check that in this case $g(\mathbf{x}, \mathbf{y}) = \mathbf{x}$ if $\mathbf{y} = (2,2,2,2,2)$ and $g(\mathbf{x}, \mathbf{y}) = \mathbf{y}$ if $\mathbf{x} = (2,2,2,2,2)$. In either case, $g(\mathbf{x}, \mathbf{y}) \in R$. In all the other cases, we can verify that $g(\mathbf{x}, \mathbf{y}) = \mathbf{x} \in R$. □

Lemma 3.1.2. *The relations T and S are invariant under the operations f and g.*

Proof. Let $\mathbf{x}, \mathbf{y}, \mathbf{z} \in T$. To show that T is invariant under f, it is sufficient to show that $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) \in T$, which is obvious, since by definition $f(\mathbf{x}, \mathbf{y}, \mathbf{z}) = \mathbf{x}$.

As for operation g, let $\mathbf{x}, \mathbf{y} \in T$. Note that $g(x, y) = 1$ never occurs. On the other hand, since tuples of T cover all the possible permutations of $\{0,2\}^5$ except $(2,2,2,2,0)$, regardless of $\mathbf{x}$ and $\mathbf{y}$ and by definition of g, $g(\mathbf{x}, \mathbf{y}) \in T$. Indeed, in order to obtain $(2,2,2,2,0)$, one of $\mathbf{x}$ and $\mathbf{y}$ has to be that tuple.

For S the argument is similar. □

Observation 3.1.1. The partition $\{0,1\},\{2\}$, denoted as $\alpha = 01|2$, is a congruence for algebra $\mathcal{A}$ with the following properties:

1. $\mathcal{A}/\alpha$ is a semilattice, and therefore is solvable by local consistency.
2. The α-block $B = \{0,1\}$ is an affine algebra, i.e., its operations are linear idempotent and every relation is given by a system of linear equations.

Lemma 3.1.3. *The algebra $\mathcal{A}$ does not have symmetric polymorphisms.*

Proof. First, take the relations $R' = \binom{0\ 1\ 2}{1\ 0\ 2}$ and $Q' = \binom{0\ 0\ 1\ 1\ 0\ 1\ 2\ 2}{0\ 1\ 0\ 1\ 2\ 2\ 0\ 1}$. Noticing the similar structures of these relations to the relations introduced earlier in this sections, it is easy to verify that R' and Q' are both invariant under operations of $\mathcal{A}$, defined in 3.1, and therefore they are also invariant under all the operations of $\mathcal{A}$. Suppose $h(x_1,\ldots,x_n)$ is a symmetric polymorphism of $\mathcal{A}$ with one block B, where permuting the inputs produces the same output. In particular, h is a polymorphism of R' and Q'. We branch over the parity of n.

Case 1 : n is odd.

In this case, take $h\binom{0\ 1\ 0\ 1\ \cdots\ 0\ 1\ 2}{1\ 0\ 1\ 0\ \cdots\ 1\ 0\ 2} \in R'$. We can easily verify that the second row is a permutation of the first row and since h is a symmetric polymorphism, the result equals $\binom{a}{a}$ for some $a \in \{0,1,2\}$. If $a \in \{0,1\}$, the tuple $\binom{a}{a} \notin R'$. Thus for $\binom{a}{a}$ to be in R' and consequently for h to be a symmetric polymorphism of $\mathcal{A}$, $a \notin \{0,1\}$, and we must have $a = 2$.

Now consider $h\binom{0\ 1\ \cdots\ 0\ 1\ 2\ 1\ 0}{1\ 0\ \cdots\ 1\ 0\ 0\ 2\ 1} \in Q'$. Since h in this case is a permutation of itself from before, $h\binom{0\ 1\ \cdots\ 0\ 1\ 2\ 1\ 0}{1\ 0\ \cdots\ 1\ 0\ 0\ 2\ 1} \in R'$. Similarly the second row of h is a permutation of its first row and because of symmetricity of h, the result equals $\binom{a}{a}$. Then for $\binom{a}{a}$ to be in Q' and consequently for h to be a symmetric polymorphism of $\mathcal{A}$ in this case, a must belong to $\{0,1\}$ is possible while $a \neq 2$. Therefore, no feasible value for a exists, and accordingly h cannot be a polymorphism of $\mathcal{A}$.

Case 2 : n is even.

In this case, take $h\binom{0\ 1\ 0\ 1\ \cdots\ 0\ 1\ 2\ 2}{1\ 0\ 1\ 0\ \cdots\ 1\ 0\ 2\ 2}$. Similarly, the second row is a permutation of the first row and therefore it the resulting tuple is equal to $\binom{a}{a}$ for some $a \in \{0,1,2\}$. With similar reasoning in Case 1, we can verify that $a \notin \{0,1\}$ for h to be invariant under R'. Now, take $h\binom{0\ 1\ \cdots\ 0\ 1\ 2\ 0\ 2\ 1}{1\ 0\ \cdots\ 1\ 0\ 0\ 2\ 1\ 2}$. Similarly for to be invariant under Q', $a \neq 2$. Therefore, no feasible value for a exists, and accordingly h cannot be a polymorphism of $\mathcal{A}$.

We can conclude that regardless of the parity of the symmetric operation $h(x_1,\ldots,x_n)$, it cannot be a polymorphism of the relations invariant in $\mathcal{A}$. Therefore, $\mathcal{A}$ does not have a symmetric polymorphism. □

Lemma 3.1.4. *The algebra $\mathcal{A}$ does not have block-symmetric polymorphisms.*

Proof. The proof proceeds much like that of Theorem 3.1.3. Let $h : A^{|B_1 \cup \ldots \cup B_k|} \to A$ be a block-symmetric polymorphism of $\mathcal{A}$ such that each block B_b, with $b \in [k]$, has size $l_b \geq 3$. Recall that $R' = \left(\begin{smallmatrix} 0 & 1 & 2 \\ 1 & 0 & 2 \end{smallmatrix}\right)$ and $Q' = \left(\begin{smallmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 2 & 2 \\ 0 & 1 & 0 & 1 & 2 & 2 & 0 & 1 \end{smallmatrix}\right)$ are two relations invariant under the operations of $\mathcal{A}$. Take $h(B_1^*, \ldots, B_k^*)$, where each B_b^* is a block of even or odd parity of the form $\left(\begin{smallmatrix} 0 & 1 & \ldots & 1 & 2 \\ 1 & 0 & \ldots & 0 & 2 \end{smallmatrix}\right)$ or $\left(\begin{smallmatrix} 0 & 1 & \ldots & 1 & 2 & 2 \\ 1 & 0 & \ldots & 0 & 2 & 2 \end{smallmatrix}\right)$. Clearly in each block, the second row is a permutation of the first row and based on the definition of a block-symmetric polymorphism, each B_b^* is symmetric, regardless of its parity. Therefore, $h = (B_1^*, \ldots, B_k^*)$ equals $\left(\begin{smallmatrix} a \\ a \end{smallmatrix}\right)$ for some $a \in \{0, 1, 2\}$. If $a \in \{0, 1\}$, we can see that $h(B_1^*, \ldots, B_k^*) \in R$, but if $a = 2$, h cannot be invariant under R.

Now consider $h(B_1^\dagger, \ldots, B_k^\dagger)$ where each $B_b^\dagger$ is a block of even or odd parity of the form $\left(\begin{smallmatrix} 0 & 1 & \ldots & 0 & 1 & 2 & 1 & 0 \\ 1 & 0 & \ldots & 1 & 0 & 0 & 2 & 1 \end{smallmatrix}\right)$ or $\left(\begin{smallmatrix} 0 & 1 & \ldots & 0 & 1 & 2 & 0 \\ 1 & 0 & \ldots & 1 & 0 & 0 & 2 \end{smallmatrix}\right)$. Similarly, these blocks, regardless of their parity are symmetric. Again, $h(B_1^\dagger, \ldots, B_k^\dagger)$ equals $\left(\begin{smallmatrix} a \\ a \end{smallmatrix}\right)$ for some $a \in \{0, 1, 2\}$, where $a \in \{0, 1\}$ is the only possibility if $h(B_1^\dagger, \ldots, B_k^\dagger)$ belongs to Q'. Therefore, we can see that the block-symmetric operation h cannot be a polymorphism of the relations invariant in $\mathcal{A}$. Therefore, $\mathcal{A}$ cannot have block-symmetric polymorphisms. □

3.2 Experiments

In this section we conduct experiments that put the methods of solving CSP instances that was explained in Section 2.6 to test. These experiments start by building up a CSP instance by carefully constructing its set of constraints and then solving this instance with the algorithms discussed earlier, namely consistency + affine relaxation and consistency + cohomology. The goal is to determine whether the CSP instance that arises from the counter-example algebra in Section 3.1 is solvable by the these algorithms, and if so, to find patterns in the solution in order to gain further insight.

3.2.1 The setup

The coding environment was first implemented in Python but because of the lack of proper linear equation to matrix transformation functions and other matrix operations, which was a necessary key in our experiment, we switched to Matlab. In our experiments, Matlab provides rich toolkit when it comes to building systems of linear equations and optimization without using external libraries. The integral part of the code is solving the systems of linear equations that are built up, which can have up to 200 equations with more than 1700 variables, for which we tried to use in-built Matlab functions. It transpired that these functions were unable to handle large systems of linear equations to find integer solutions and they turned out to be impractically slow. Therefore, we decided to take advantage of an industrial optimization solver, named Gurobi.

The code starts with the input of the counter-example CSP, including the relations invariant under the operations of the algebra $\mathcal{A}$ along with the universe. Initially the con-

sistency constraints were added manually to the system. This step was later programmed so that the consistency for any instance is added to the code automatically.

3.2.2 Evaluation of the linear programming + affine relaxation algorithm

For the final system of linear equations to be formed, we need to add two sets of equations:

1. The first set of equations contains variables, which we call universe-variables and denote by $r\_i\_BE\_a$, represent the main variables having each of the universe values. These are the same variables as $r_i(a) \in \mathbb{Z}$, where $a \in A$ and $i \in [n]$ is the index of the variable, introduced in Subsection 2.6.1. The corresponding equation is that their sum across all universe values is equal to 1, formally $\sum_{a \in A} r_i(a) = 1 \ \ \forall i \in [n]$
2. The second set of equations contain variables, which we call column-variables and denote by $q\_j\_BE\_y$, represent each constraint in the CSP to be assigned the value of each of the columns from their corresponding relation. These are the same variables as $q_j(y) \in \mathbb{Z}$, where where $j \in [m]$ is the index of constraint $C_j = \langle s_j, R_j \rangle$ and $y \in R_j^A$ is a potential assignment to s_j, introduced in Subsection 2.6.1. The corresponding equation is equivalent to $\sum_{y_i = a} q_j(y) = r_i(a) \ \ \forall i \in [n], j \in [m]$.

The resulting system of linear equations is then transformed to a sparse matrix to be ready for the Gurobi solver. The result of solving our system of linear equations with Gurobi solver is a table with two columns: the left column are all the variables, including the original ones and those that were created, and on the right column are the corresponding integer values for variables.

3.2.3 Evaluation of the consistency + cohomology algorithm

Similarly, to create the final system of linear equations, we need to add two sets of equations:

1. Similar to above, the first set of variables, which we call universe-variables and show with $r\_i\_BE\_a$, represent the main variables having each of the universe values. These are the same variables as $r_i(a) \in \mathbb{Z}$, where $\in A$ and $i \in [n]$ is the index of the variable, introduced in Subsection 2.6.2. Unlike the evaluation of linear programming + affine relaxation algorithm above, we do not require these variables to sum up to 1.
2. Same as above, the second set of variables, which we call column-variables and show with $q\_j\_BE\_y$, represent the consistency constraints having each of the column values of their corresponding relation. These are the same variables as $q_j(y) \in \mathbb{Z}$, where $j \in [m]$ is the index of constraint $C_j = \langle s_j, R_j \rangle$ and $y \in R_j^A$ is a potential assignment to s_j, introduced in Subsection 2.6.2. The corresponding equation is equivalent to $\sum_{y_i = a} q_j(y) = r_i(a) \ \ \forall i \in [n], j \in [m]$.

In the next step, two FOR loops run consequently. First, the code loops through variable $r_k(a)$ for all $a \in A$ and for all $k \in [n]$ and sends it to a function $FindAnIntSol$ which returns a flag and an equation. The function checks if the parameter $r_k(a)$ is a valid assignment. If so, the flag stays zero and the function returns the equation $r_k(a) = 1$. In that case, the new

returned equation is added to the system of linear equations and is solved with the Gurobi solver, as explained above. If the parameter is invalid, the flag changes to one and the code prints "Function teminated early due to unsatisfied condition" and moves to $r_{i+1}(a)$.

Next, the same steps above is repeated for $q_k(y)$ $\forall k \in [m]$ and y from the corresponding constraint.

3.2.4 The instances

In this section we introduce our CSP instances. The input instances represent the CSPs that arise from the counter-example algebra from Section 3.1. These instances are unique and has a carefully-designed structure and they are meant to refute Conjecture 2.6.3.

3.2.4.1 Instance I_7

The basic form of the instance, including the variables and the relations, can be visualized in Figure 3.1. Each vertex represents a variable and there exists curves between the vertices that are under some relation, together. The relations and the constraints, consisting of tuples where each tuple represents the variables and the relation they are applied on is as following.

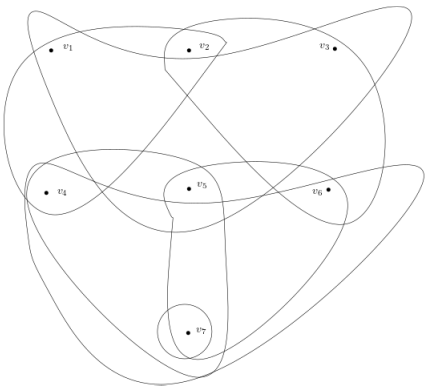

Figure 3.1: The structure of instance I_7

$$R_3 = \begin{pmatrix} 1\ 0\ 1\ 0\ 2 \\ 0\ 1\ 0\ 1\ 2 \\ 0\ 0\ 2\ 2\ 2 \end{pmatrix}, T_3 = \begin{pmatrix} 0\ 2\ 0\ 0\ 2\ 0\ 2 \\ 0\ 0\ 2\ 0\ 0\ 2\ 2 \\ 0\ 0\ 0\ 2\ 2\ 2\ 2 \end{pmatrix}, Q = \begin{pmatrix} 0 \end{pmatrix}$$

$$\begin{aligned} constraints = \{&[(v_1, v_2, v_4), R_3], [(v_1, v_3, v_5), R_3], [(v_2, v_3, V_6), R_3] \\ &, [(v_4, v_5, v_7), T_3], [(v_4, v_6, v_7), T_3], [(v_5, v_6, v_7), T_3], \\ &[(v_7), Q], \} \end{aligned}$$

Lemma 3.2.1. *The CSP instance I_7 does not have a solution.*

Proof. Firstly, since Q is applied on v_7, the value of v_7 must be 0. Next, from the first three constraints, we can conclude that on $\{0,1\}$, $v_1 \neq v_2, v_2 \neq v_3, v_1 \neq v_3$, which is impossible. Therefore, since no solution exists on $\{0,1\}$, we must have $(v_1, v_2, v_3) = (2,2,2)$. Next, as it was discussed, $v_7 = 0$, and thus, we have the following cases for v_4, v_5, v_6:

$$v_4 = v_5 = 0 \ or \ v_4 \neq v_5$$
$$v_4 = v_6 = 0 \ or \ v_4 \neq v_6$$
$$v_5 = v_6 = 0 \ or \ v_5 \neq v_6,$$

where we can conclude $0 \in v_4, v_5, v_6$. This would mean that at least one of $\{v_1, v_2, v_3\}$ must have a 0 or 1 value. Consequently, $\{v_1, v_2, v_3\} \in \{0,1\}$ but we proved earlier that it is impossible. Therefore the CSP does not have a solution. □

3.2.4.2 Instance I_{16}

We introduce another instance with 16 variables but similar characteristics. This instance was introduced to challenge the cohomology approach to the CSP. The relations and the constraints, consisting of tuples where each tuple represents the variables and the relation they are applied on is as following.

$$R_5 = \begin{pmatrix} 1&0&0&0&1&1&1&0&1&0&0&0&1&1&1&0&2 \\ 0&1&0&0&1&1&0&1&0&1&0&0&1&1&0&1&2 \\ 0&0&1&0&1&0&1&1&0&0&1&0&1&0&1&1&2 \\ 0&0&0&1&0&1&1&1&0&0&0&1&0&1&1&1&2 \\ 0&0&0&0&0&0&0&0&2&2&2&2&2&2&2&2&2 \end{pmatrix},$$

$$T_5 = \begin{pmatrix} 0&0&0&0&2&2&2&0&0&0&2&2&2&0&0&0&0&0&2&2&2&0&0&0&2&2&2&0&2 \\ 0&2&0&0&2&0&0&2&2&0&2&2&0&2&0&2&0&0&2&0&0&2&2&0&2&2&0&2&2 \\ 0&0&2&0&0&2&0&2&0&2&2&0&2&2&0&0&2&0&0&2&0&2&0&2&2&0&2&2&2 \\ 0&0&0&2&0&0&2&0&2&2&0&2&2&2&0&0&0&2&0&0&2&0&2&2&0&2&2&2&2 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&2&2&2&2&2&2&2&2&2&2&2&2&2&2&2 \end{pmatrix},$$

$$S_5 = \begin{pmatrix} 0&2&0&0&0&2&2&2&0&0&0&2&2&2&0 \\ 0&0&2&0&0&2&0&0&2&2&0&2&2&0&2 \\ 0&0&0&2&0&0&2&0&2&0&2&2&0&2&2 \\ 0&0&0&0&2&0&0&2&0&2&2&0&2&2&2 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \end{pmatrix},$$

$$Q = \begin{pmatrix} 0 \end{pmatrix}$$

$$
\begin{aligned}
\mathit{constraints} = \{&[(v_1, v_2, v_3, v_4, v_6), R_5], [(v_1, v_2, v_3, v_5, v_7), R_5], [(v_1, v_2, v_4, v_5, v_8), R_5] \\
&, [(v_1, v_3, v_4, v_5, v_9), R_5], [(v_2, v_3, v_4, v_5, v_{10}), R_5], [(v_6, v_7, v_8, v_9, v_{11}), T_5] \\
&, [(v_6, v_7, v_8, v_{10}, v_{12}), T_5], [(v_6, v_7, v_9, v_{10}, v_{13}), T_5], [(v_6, v_8, v_9, v_{10}, v_{14}), T_5] \\
&[(v_7, v_8, v_9, v_{10}, v_{15}), T_5], [(v_{11}, v_{12}, v_{13}, v_{14}, v_{16}), S_5], \\
&[(v_{11}, v_{12}, v_{13}, v_{15}, v_{16}), S_5], [(v_{11}, v_{12}, v_{14}, v_{15}, v_{16}), S_5], [(v_{11}, v_{13}, v_{14}, v_{15}, v_{16}), S_5], \\
&[(v_{12}, v_{13}, v_{14}, v_{15}, v_{16}), S_5], [(v_{16}), Q], \}
\end{aligned}
$$

The proof of the following lemma is fairly straightforward and similar to the proof for Lemma 3.2.1.

Lemma 3.2.2. *The CSP instance I_{16} that arises from the counter-example algebra from Section 3.1 and is described in Section 3.2.4.2 does not have a solution.*

3.2.5 Results

The experiment includes running each of the two algorithms, i.e., linear programming + affine relaxation and consistency + cohomology, on instances I_7 and I_{16}, a total of four separate tests. The goal is to show that in each of these tests, the corresponding algorithm would find a solution for its respective instance in that test. We know from Lemmas 3.2.1 and 3.2.2 that neither of the instances have a solution. The algorithms finding solutions for the instances would demonstrate that the structure of the instance, as desired, deceives the algorithm, therefore, the Cojecture 2.6.3 is refuted.

For the instance I_7, linear programming + affine relaxation algorithm finds a feasible solution, as desired. A table containing the results by this algorithm on this instance can be found below. On the other hand, when run on the instance, the consistency + cohomology algorithm does not find a solution. This means that this algorithm can recognise that the instance has no solution.

Table 3.1: Variable Assignments for I_7

Variable	Assignment
q1BE010	0
q1BE010	0
q1BE100	0
q1BE102	0
q1BE222	1
q2BE010	0
q2BE012	0
q2BE100	0
q2BE102	0
q2BE222	1
...	...
r1BE0	0
r1BE1	0
r1BE2	1
r2BE0	0
r2BE1	0
r2BE2	1
...	...
r7BE2	0

Similarly for instance I_{16}, linear programming + affine relaxation algorithm finds a feasible solution. However, unlike the case for instance I_7, the consistency + cohomology algorithm finds a solution. This therefore shows that the structure of I_{16} deceives this algorithm.

Conclusion In conclusion, we considered two combinations of consistency and relaxation algorithms, and in both cases found an instance of a polynomial time solvable CSP that are not solved by those methods, thus, partially refuting Conjecture 2.6.3. As was mentioned in the Introduction, Our results do not fully refute Conjecture 2.6.3, since a number of (potentially more powerful) combinations of algorithms remain. In particular, we consider only consistency of a fairly low level — it is lower than the arity of the relations involved. It is possible that the consistency of a higher level, at least as high as the arity of the relation of the instance, may lead to a desired algorithm. However, an instance that may fool such an algorithm is quite complicated. It is based on graphs with high treewidth, see [2], and is not be feasible for empirical testing, as it would involve thousands of variables.

Bibliography

[1] Albert Atserias, Andrei Bulatov, and Victor Dalmau. On the power of k-consistency. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki, editors, *Automata, Languages and Programming*, pages 279–290, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[2] Albert Atserias, Andrei A. Bulatov, and Víctor Dalmau. On the power of k -consistency. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 279–290. Springer, 2007.

[3] Per Austrin, Venkatesan Guruswami, and Johan Håstad. $(2+\varepsilon)$-sat is np-hard. *SIAM Journal on Computing*, 46(5):1554–1573, 2017.

[4] Libor Barto, Jakub Bulín, Andrei A. Krokhin, and Jakub Oprsal. Algebraic approach to promise constraint satisfaction. *J. ACM*, 68(4):28:1–28:66, 2021.

[5] Libor Barto and Marcin Kozik. Constraint satisfaction problems of bounded width. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 595–603, 2009.

[6] Libor Barto and Marcin Kozik. Constraint satisfaction problems solvable by local consistency methods. *J. ACM*, 61(1), jan 2014.

[7] Libor Barto and Marcin Kozik. Constraint satisfaction problems solvable by local consistency methods. *J. ACM*, 61(1), jan 2014.

[8] Libor Barto, Andrei Krokhin, and Ross Willard. *Polymorphisms, and How to Use Them.* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.

[9] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[10] Joshua Brakensiek and Venkatesan Guruswami. *Promise Constraint Satisfaction: Structure Theory and a Symmetric Boolean Dichotomy*, pages 1782–1801.

[11] Joshua Brakensiek and Venkatesan Guruswami. Promise constraint satisfaction: Algebraic structure and a symmetric boolean dichotomy, 2017.

[12] Joshua Brakensiek and Venkatesan Guruswami. An algorithmic blend of lps and ring equations for promise csps, 2018.

[13] Joshua Brakensiek and Venkatesan Guruswami. *Promise Constraint Satisfaction: Structure Theory and a Symmetric Boolean Dichotomy*, page 1782–1801. Society for Industrial and Applied Mathematics, January 2018.

[14] Joshua Brakensiek, Venkatesan Guruswami, Marcin Wrochna, and Stanislav Živný. The power of the combined basic linear programming and affine relaxation for promise constraint satisfaction problems. *SIAM Journal on Computing*, 49(6):1232–1248, 2020.

[15] Andrei Bulatov. Bounded relational width. 2009.

[16] Andrei Bulatov and Víctor Dalmau. A simple algorithm for mal'tsev constraints. *SIAM Journal on Computing*, 36(1):16–27, January 2006.

[17] Andrei Bulatov, Peter Jeavons, and Andrei Krokhin. Classifying the complexity of constraints using finite algebras. *SIAM Journal on Computing*, 34(3):720–742, January 2005.

[18] Andrei A. Bulatov. A dichotomy theorem for nonuniform csps. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 319–330, 2017.

[19] Lorenzo Ciardo and Stanislav Živný. Clap: A new algorithm for promise csps. *SIAM Journal on Computing*, 52(1):1–37, January 2023.

[20] Adam Ó Conghaile. Cohomology in constraint satisfaction and structure isomorphism, 2022.

[21] Irit Dinur, Elchanan Mossel, and Oded Regev. Conditional hardness for approximate coloring, 2005.

[22] Tomás Feder and Moshe Y. Vardi. The computational structure of monotone monadic snp and constraint satisfaction: A study through datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, 1998.

[23] Tomás Feder and Moshe Y. Vardi. The computational structure of monotone monadic snp and constraint satisfaction: A study through datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, 1998.

[24] Tomás Feder and Moshe Y. Vardi. The computational structure of monotone monadic snp and constraint satisfaction: A study through datalog and group theory. *SIAM Journal on Computing*, 28(1):57–104, January 1998.

[25] M. R. Garey and D. S. Johnson. The complexity of near-optimal graph coloring. *Journal of the ACM*, 23(1):43–49, January 1976.

[26] Venkatesan Guruswami and Sai Sandeep. d-to-1 hardness of coloring 3-colorable graphs with o(1) colors. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[27] Sangxia Huang. Improved hardness of approximating chromatic number, 2013.

[28] Paweł Idziak, Petar Marković, Ralph McKenzie, Matthew Valeriote, and Ross Willard. Tractability and learnability arising from algebras with few subpowers. *SIAM Journal on Computing*, 39(7):3023–3037, 2010.

[29] PaweŁ Idziak, Petar Marković, Ralph McKenzie, Matthew Valeriote, and Ross Willard. Tractability and learnability arising from algebras with few subpowers. *SIAM Journal on Computing*, 39(7):3023–3037, January 2010.

[30] Ravindran Kannan and Achim Bachem. Polynomial algorithms for computing the smith and hermite normal forms of an integer matrix. *SIAM Journal on Computing*, 8(4):499–507, November 1979.

[31] Andrei Krokhin and Jakub Opršal. The complexity of 3-colouring h-colourable graphs. 04 2019.

[32] Andrei Krokhin and Jakub Opršal. An invitation to the promise constraint satisfaction problem, Aug 29 2022. Copyright - © 2022. This work is published under http://creativecommons.org/licenses/by-sa/4.0/ (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2022-08-31.

[33] Andrei Krokhin, Jakub Opršal, Marcin Wrochna, and Stanislav Živný. Topology and adjunction in promise constraint satisfaction. 2020.

[34] Gabor Kun, Ryan O'Donnell, Suguru Tamaki, Yuichi Yoshida, and Yuan Zhou. Linear programming, width-1 csps, and robust satisfaction. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 484–495. ACM, 2012.

[35] Gábor Kun and Mario Szegedy. A new line of attack on the dichotomy conjecture. *European Journal of Combinatorics*, 52:338–367, February 2016.

[36] Moritz Lichter and Benedikt Pago. Limitations of affine integer relaxations for solving constraint satisfaction problems. *CoRR*, abs/2407.09097, 2024.

[37] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, February 1977.

[38] Hanif D. Sherali and Warren P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal on Discrete Mathematics*, 3(3):411–430, 1990.

[39] Johan Thapper and Stanislav Živný. The power of sherali–adams relaxations for general-valued csps. *SIAM Journal on Computing*, 46(4):1241–1279, 2017.

[40] Marcin Wrochna and Stanislav Živný. Improved hardness for h-colourings of g-colourable graphs. 2019.

[41] Dmitriy Zhuk. A proof of csp dichotomy conjecture. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 331–342, 2017.

[42] H.J. Zimmermann and Angelo Monfroglio. Linear programs for constraint satisfaction problems. *European Journal of Operational Research*, 97(1):105–123, 1997.

Appendix A

Code

A.0.1 Consistency

```
    clc;
clear all;
%******************INPUTS************************%
num_of_variables = 16;
Dom = {'0', '1', '2'};
%Here goes the relations, constraints, and consistency
constraints as described in The Instance Section.%
%******************ENDofINPUTS*******************%
%Preliminary Variable Definition%
dummy_variables = sym('x', [1 num_of_variables]);
%Number of the constraints
len_cons = length(cons);
%Number of Original Variables
len_vars = num_of_variables;
len_Dom = length(Dom);
len_Dom_times_V = len_Dom*len_vars;


 current_cons_firstCell = {};
 for i = 1:len_cons
     current_cons_firstCell{end+1} = cons{i}{1};
 end

cons_1 = cell(size(cons));
for i = 1:numel(cons)
    cons_1{i} = cons{i}{1};
end
cons_2 = cell(size(cons));
for i = 1:numel(cons)
    cons_2{i} = cons{i}{2};
```

```
end

vars_vector = 1:1:num_of_variables;

CC = nchoosek(vars_vector, 2);
CCV = num2cell(CC,2);

conss = {};
for i = 1:length(CCV)
    new_cons = make_new_cons(CCV(i), cons);
    conss{end+1} = new_cons;
end

updated_conss = cons_postprocess(conss);

cons_after_phase1 = horzcat(cons, updated_conss);

%Phase 2

%Seperating the S and R of the updated Constraint Set after Phase1
cons_after_phase1_S = cell(size(cons_after_phase1));
for i = 1:numel(cons_after_phase1)
    cons_after_phase1_S{i} = cons_after_phase1{i}{1};
end
cons_after_phase1_R = cell(size(cons_after_phase1));
for i = 1:numel(cons_after_phase1)
    cons_after_phase1_R{i} = cons_after_phase1{i}{2};
end


CC2 = nchoosek(vars_vector, 3);
CCV2 = num2cell(CC2,2);

tuple = CCV2(1);

PrintingCons(cons_after_phase1_S, cons_after_phase1_R, len_cons)

function consistancy(tuple)
    uv = tuple{1}(1:2); % Extracts [u v]
    uw = [tuple{1}(1), tuple{1}(3)]; % Extracts [u w]
    vw = tuple{1}(2:3); % Extracts [v w]

end

function add_new_cons = make_new_cons(uv, cons)%%uv is in the shape of {[u,v]}
    w = {};%constraints that have both uv in them
    R_uv = {};
```

```
    uv_as_vector = cell2mat(uv);
    for p = 1:length(cons)
    intersecting_variable = intersect(uv_as_vector, cons{p}{1});
        if length(intersecting_variable) == 2
            [~, loc] = intersect(cons{p}{1}, uv_as_vector);
            intersect_u_loc = loc(1);
            intersect_v_loc = loc(2);
            R_uv_first_row = relation_splitter(intersect_u_loc, cons{p}{2});
            R_uv_second_row = relation_splitter(intersect_v_loc, cons{p}{2});
            R_uv_composed = relation_compose(R_uv_first_row, R_uv_second_row);
            R_uv{end+1} = R_uv_composed;
        end
    end

    new_cons_s = uv_as_vector;
    new_cons_R = R_uv;

    add_new_cons = {new_cons_s, new_cons_R};
end

function V = relation_splitter(intersect_loc, R)
    % Split each string into a cell array of substrings
    S = cellfun(@(x) strsplit(x), R, 'UniformOutput', false);

    % Extract the first substring from each cell array
    V = cellfun(@(x) x{intersect_loc}, S);
end


function c = relation_compose(a, b)
% Convert the elements of a and b to strings
% Combine the corresponding characters of a and b with a space in between
% Convert the input strings to arrays of characters
a_chars = num2cell(a);
b_chars = num2cell(b);

% Concatenate the characters from a and b with a space character in between
c_chars = strcat(a_chars, {' '}, b_chars);

% Convert the result to a cell array
c0 = c_chars;
c = unique(c0);


end

function displayWholeObj(obj)
```

```
% Display the contents of a custom object
    if iscell(obj)
        for ii = 1:numel(obj)
            displayWholeObj(obj{ii});
        end
    else
        disp(obj);
    end
end

function updated_cons = cons_postprocess(conss)
    temp_conss = {};

    %Deleting relations that are zeros
    for i = 1:length(conss)
        if isempty(conss{i}{2}) == 0
            temp_conss{end+1} = conss{i};
        end
    end
    length(temp_conss)
    for i = 1:length(temp_conss)
        temp_conss{i}{1};
        temp_conss{i}{2} = temp_conss{i}{2}{1};
    end

    updated_cons = temp_conss;
end



function PrintingCons(cons_after_phase1_S, cons_after_phase1_R, len_cons)
    for i = len_cons+1:numel(cons_after_phase1_S)
        fprintf('{');
        fprintf('[');
        fprintf('%d ', cons_after_phase1_S{i});
        fprintf('],');
        fprintf('{');
        for j = 1:numel(cons_after_phase1_R{i})
            if j < numel(cons_after_phase1_R{i})
                fprintf("'%s', ", cons_after_phase1_R{i}{j});
            else
                fprintf("'%s'", cons_after_phase1_R{i}{j});
            end
        end
        fprintf('}');
        fprintf('},...\n');
    end
```

end

A.0.2 Consistency + AIP

```
clc;
clear all;
%**********************INPUTS**********************%
num_of_variables = 7;
Dom = {'0', '1', '2'};
%Here goes the relations, constraints, and consistency
constraints as described in The Instance Section.%
%**********************END of INPUTS**********************%
dummy_variables = sym('x', [1 num_of_variables]);

eqn = {};
%constraints on R: A, B, C are the column variables
respecting each constraint
alphabets = sym('A', [1 1000]);
%Number of the constraints
len_cons = length(cons);
%Number of Original Variables
len_vars = num_of_variables;
len_Dom = length(Dom);
len_Dom_times_V = len_Dom*len_vars;


varss = {};
Ms = {};
vars_structs = {};
for i = 1:len_vars
    vars_structs{i}.alphabet = char(dummy_variables(i));
    temp = sym(vars_structs{i}.alphabet,[1 length(Dom)]);

    for j = 1:length(temp)%To make the variable names unique
        y0 = [vars_structs{i}.alphabet, 'BE', Dom{j}];
        yy0 = str2sym(y0);
        temp(j) = yy0;
    end

    vars_structs{i}.alphabet_str = sym2cell(temp);
    M = containers.Map(Dom, vars_structs{i}.alphabet_str);
    vars_structs{i}.map = M;

    varss{i} = i;
    Ms{i} = vars_structs{i}.map;
```

```
    eqn = eqn_maker_sumTo1(eqn, Dom, vars_structs{i}.
       alphabet_str);
end

MMM = containers.Map(varss,Ms);

cons_structs = {};
for i = 1:len_cons
    cons_structs{i}.id = i;
    cons_structs{i}.var = cons{i}{1};
    cons_structs{i}.relation = cons{i}{2};
    cons_structs{i}.alphabet = char(alphabets(i));
    tempo = sym(cons_structs{i}.alphabet,
    [1 length(cons_structs{i}.relation)]);

    for j = 1:length(tempo)%To make the variable names unique
        y = [cons_structs{i}.alphabet, 'BE', cons_structs{i}.
           relation{j}];
        yy = str2sym(y);
        tempo(j) = yy;
    end

    cons_structs{i}.alphabet_str = sym2cell(tempo);
    M = containers.Map(cons_structs{i}.relation, cons_structs{i
       }.alphabet_str);
    cons_structs{i}.map = M;


    eqn = eqn_maker(eqn, cons_structs{i}.var, MMM, Dom,
       cons_structs{i}.relation, cons_structs{i}.map);
end

all_vars = [];
for i = 1:length(eqn)
    temppp = symvar(eqn{i:i});
    all_vars = cat(2, all_vars, temppp);
    all_vars = unique(all_vars);
end


[wow1,wow2] = equationsToMatrix(eqn{:}, all_vars);
format rat;
wow1 = double(wow1);
wow2 = double(wow2);
f = zeros(size(wow1, 2) ,1);
f = double(f);
intcon = 1:size(wow1, 2);

wow1 = sparse(wow1);
```

```
model.A = wow1;
model.obj = f;
model.rhs = wow2;
model.lb = -inf*ones(size(wow1,2),1);
model.ub = +inf*ones(size(wow1,2),1);
model.sense = '=';
model.vtype = 'I';


% Set the type of optimization problem to MIP
params.outputflag = 1;
params.method = 0; % primal simplex

% Solve the MIP using Gurobi
result = gurobi(model, params);

SolutionTable = table(all_vars',result.x);
displayWholeObj(SolutionTable)

% Print the solver runtime and number of nodes explored, and
   status
status = result.status;
fprintf('Solver status: %s \n', status)
fprintf('Solver runtime: %f seconds\n', result.runtime);
fprintf('Number of nodes explored: %d\n', result.nodecount);

function eqn = eqn_maker(eqn, r1, MMM, Dom, R, MA)
index = 1;
for k = r1
    whichVar = MMM(k);
for i = Dom
    tempp = {};
    for j = R
        j;
        temp = split(j);
        temp;
        if(ismember(temp(index), i))
            tempp{end+1} = MA(j{1});

        end
    end
    eqn{end+1} = (sum([tempp{:}]) - whichVar(i{1}) == 0);

end
index = index + 1;
end
end

function eqn = eqn_maker_sumTo1(eqn, Dom, alpha_str)
```

```
    temp = {};
    for j = 1:length(Dom)
        temp{end+1} = alpha_str{j};
    end
    eqn{end+1} = (sum([temp{:}]) == 1);

end
```

A.0.3 Consistency + Cohomology

```
    diary("Coho_16var_withConsistency_Result.txt")

clc;
clear all;
%**********************INPUTS************************%
num_of_variables = 22;
Dom = {'0', '1', '2'};
%Here goes the relations, constraints, and consistency
constraints as described in The Instance Section.%
%*****************END OF INPUTS*****************%
dummy_variables = sym('x', [1 num_of_variables]);
eqn = {};
%constraints on R: A, B, C are the column variables
respecting each constraint
alphabets = sym('A', [1 100000]);
%Number of the constraints
len_cons = length(cons);
%Number of Original Variables
len_vars = num_of_variables;
len_Dom = length(Dom);
len_Dom_times_V = len_Dom*len_vars;


varss = {};
Ms = {};
vars_structs = {};
for i = 1:len_vars
    vars_structs{i}.alphabet = char(dummy_variables(i));
    temp = sym(vars_structs{i}.alphabet,[1 length(Dom)]);

    for j = 1:length(temp)%To make the variable names unique
        y0 = [vars_structs{i}.alphabet, 'BE', Dom{j}];
        yy0 = str2sym(y0);
        temp(j) = yy0;
    end
```

```
    vars_structs{i}.alphabet_str = sym2cell(temp);
    M = containers.Map(Dom, vars_structs{i}.alphabet_str);
    vars_structs{i}.map = M;

    varss{i} = i;
    Ms{i} = vars_structs{i}.map;

    %eqn = eqn_maker_sumTo1(eqn, Dom, vars_structs{i}.alphabet_str);
end
MMM = containers.Map(varss,Ms);

All_A_variables_symbolic = {};
cons_structs = {};
for i = 1:len_cons
    cons_structs{i}.id = i;
    cons_structs{i}.var = cons{i}{1};
    cons_structs{i}.relation = cons{i}{2};
    cons_structs{i}.alphabet = char(alphabets(i));
    tempo = sym(cons_structs{i}.alphabet, [1 length(cons_structs{i}.relation)]);%

    for j = 1:length(tempo)%To make the variable names unique
        y = [cons_structs{i}.alphabet, 'BE', cons_structs{i}.relation{j}];
        yy = str2sym(y);
        tempo(j) = yy;
        All_A_variables_symbolic{end+1} = yy;
    end

    cons_structs{i}.alphabet_str = sym2cell(tempo);
    M = containers.Map(cons_structs{i}.relation, cons_structs{i}.alphabet_str);
    cons_structs{i}.map = M;

    eqn = eqn_maker(eqn, cons_structs{i}.var, MMM, Dom, cons_structs{i}.relation
end

all_vars = [];
for i = 1:length(eqn)
    temppp = symvar(eqn{i:i});
    all_vars = cat(2, all_vars, temppp);
    all_vars = unique(all_vars);
end

%%%For X#BE# variables
for i = 1:len_vars
    for j = 1:length(Dom)
        temp = vars_structs{i}.alphabet_str{j};
        %eqn = eqn_coho(eqn, temp);
        [eqn, flag] = eqn_coho(eqn, temp);
```

```
        if flag == 1
            % Handle the case where the function returned nothing
            fprintf("For %s = 1, function terminated early due to unsatisfied co
            fprintf('----------------------------------------------------
            ---------\n')

        %temp = hi;
        else
            [intsol, exitflag] = FindAnIntSol(eqn, all_vars);
            if exitflag == 'OPTIMAL'
                SolutionTable = table(all_vars',intsol);
                displayWholeObj(SolutionTable)
                fprintf('%s = 1 passed, solution printed above, continue\n', tem
                fprintf('----------------------------------------------------
                ----------\n')
                eqn(end) = [];
            else
                fprintf('%s = 1 failed, terminate\n', temp);
                return
            end
        end
    end
end

%For A#### variables
for i = All_A_variables_symbolic
        temp = cell2sym(i);
        %eqn = eqn_coho(eqn, temp);
        [eqn, flag] = eqn_coho(eqn, temp);
        if flag == 1
            % Handle the case where the function returned nothing
            fprintf("For %s = 1, function terminated early due to unsatisfied co
            fprintf('----------------------------------------------------
            ---------\n')
        else
            [intsol, exitflag] = FindAnIntSol(eqn, all_vars);
            if exitflag == 'OPTIMAL'
                SolutionTable = table(all_vars',intsol);
                displayWholeObj(SolutionTable)
                fprintf('%s = 1 passed, solution printed above, continue\n', tem
                fprintf('----------------------------------------------------
                ----------\n')
                eqn(end) = [];
            else
                fprintf('%s = 1 failed, terminate\n', temp);
                return
            end
```

```
        end
end

diary off


function [intSol, exitflag] = FindAnIntSol(eqn, all_vars)
    [wow1,wow2] = equationsToMatrix(eqn{:}, all_vars);
    format rat;
    wow1 = double(wow1);
    wow2 = double(wow2);
    f = zeros(size(wow1, 2) ,1);
    f = double(f);

    wow1 = sparse(wow1);
    model.A = wow1;
    model.obj = f;
    model.rhs = wow2;
    model.lb = -inf*ones(size(wow1,2),1);
    model.ub = +inf*ones(size(wow1,2),1);
    model.sense = '=';
    model.vtype = 'I';

    % Set the type of optimization problem to MIP
    params.outputflag = 1;
    params.method = 0; % primal simplex

    % Solve the MIP using Gurobi
    result = gurobi(model, params);

    % Print the solver runtime and number of nodes explored, and status
    status = result.status;
    fprintf('Solver status: %s \n', status)
    fprintf('Solver runtime: %f seconds\n', result.runtime);
    fprintf('Number of nodes explored: %d\n', result.nodecount);

    intSol = result.x;
    exitflag = status;
end

%%AIP eqn maker
function eqn = eqn_maker(eqn, r1, MMM, Dom, R, MA)
    index = 1;
    for k = r1
        whichVar = MMM(k);
    for i = Dom
        tempp = {};
```

```
        for j = R
            j;
            temp = split(j);
            temp;
            if(ismember( temp(index), i))
                tempp{end+1} = MA(j{1});

            end
        end
        eqn{end+1} = (sum([tempp{:}]) - whichVar(i{1}) == 0);

    end
    index = index + 1;
    end
end


%%Sum to 1 eqn maker
function eqn = eqn_maker_sumTo1(eqn, Dom, alpha_str)
    temp = {};
    for j = 1:length(Dom)
        temp{end+1} = alpha_str{j};
    end
    eqn{end+1} = (sum([temp{:}]) == 1);

end


%%Cohomology eqn maker_single step
function [eqn, flag] = eqn_coho(eqn, alph_str)
    a = 'x8BE1';
    b = 'x9BE1';
    c = 'x10BE1';
    d = 'x11BE1';
    e = 'x12BE1';
    f = 'x13BE1';
    g = 'x14BE1';
    h = 'x15BE1';
    i = 'x16BE1';
    j = 'x17BE1';
    k = 'x18BE1';
    l = 'x19BE1';
    m = 'x20BE1';
    n = 'x21BE1';
    o = 'x22BE1';
    p = 'x22BE2';
    aa = str2sym(a);
```

```
    bb = str2sym(b);
    cc = str2sym(c);
    dd = str2sym(d);
    ee = str2sym(e);
    ff = str2sym(f);
    gg = str2sym(g);
    hh = str2sym(h);
    ii = str2sym(i);
    jj = str2sym(j);
    kk = str2sym(k);
    ll = str2sym(l);
   mm = str2sym(m);
    nn = str2sym(n);
    oo = str2sym(o);
    pp = str2sym(p);

    if alph_str == aa || alph_str == bb || ...
       alph_str == cc || alph_str == dd || ...
       alph_str == ee || alph_str == ff || ...
       alph_str == gg || alph_str == hh || ...
       alph_str == ii || alph_str == jj || ...
       alph_str == kk || alph_str == ll || ...
       alph_str == mm || alph_str == nn || ...
       alph_str == oo || alph_str == pp
        flag = 1;
        eqn = eqn;

    else
        flag = 0;
        eqn{end+1} = (alph_str == 1);
    end

end
```