INSTITUT FÜR INFORMATIK
Algorithmische Bioinformatik

Universitätsstr. 1 D–40225 Düsseldorf

Heinrich Heine
Universität
Düsseldorf

A Framework for General Purpose Solvers with an Application to Cluster Editing

Jan Höckesfeld

Master's Thesis

Date of issue:	10.02.2022
Date of submission:	10.08.2022
Reviewers:	Prof. Dr. Gunnar W. Klau
	Prof. Dr. Michael Leuschel

Erklärung

Hiermit versichere ich, dass ich diese Master's Thesis selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 10.08.2022

Jan Höckesfeld

Abstract

General purpose methods play an important role in solving algorithmic problems. Instead of developing a complex algorithm for each problem, general-purpose methods solve problems automatically according to predefined rules. These rules can be expressed in different ways, e.g. logical statements (clauses) or linear inequalities (constraints). Each rule set is also called a formulation. Integer Linear Programming (ILP) is the most widely used tool, where the rules are defined with linear equations. Recent research shows that satisfiability solvers (SAT solvers) are faster in solving some problems, where the rules are defined as Boolean formulas in Conjunctive Normal Form (CNF). While the rules of each method may describe the same restriction, testing both methods still requires the development of the two different rule sets. We propose a framework that converts a common abstract language into ILP and SAT rule sets. The abstract language is closely related to predicate logic, but also includes important operations which are not trivial to express in logic. To evaluate our framework, we test it on the Cluster Editing problem. For this, we generate ILP and SAT formulations with the framework and solve them with well-known solvers. We show that the tool creates optimal comparable solutions for ILP and SAT. The generated ILP took a similar time as a manually created ILP. Some work remains with the SAT formulation, as it took significantly more time than a manually created SAT formulation.

Contents

1 Introduction

Many NP-hard problems can be reduced to rules that restrict the feasible solution space. These rules can be expressed in different ways, e.g. logical statements (clauses) or linear inequalities (constraints). Each rule set is also called a formulation. On one hand, classic approaches use these rules to give a baseline for an algorithm that can solve the problem. This requires research and allows different approaches with varying runtime performance. On the other hand, one could use general-purpose methods which solve the problem with a predefined set of rules. General-purpose methods often solve the NP-hard problems faster and often require less development time compared to an algorithmic approach.

Integer Linear Programming (ILP) is a widely-used tool in the research community. ILP optimizes many computationally hard problems efficiently and has a range of competitive commercial and free implementations. Here, the rules of the problem need to be expressed as linear equations along with a linear target function that is optimized. Variables in these equations can be bound to Integers and as a special case to 0 and 1. 0-1 bound variables take a major role since they can be interpreted as Boolean variables.

Another general-purpose method is Satisfiability-Solving abbreviated as SAT-Solving. This method takes a logic formula in Conjunctive Normal Form (CNF) and computes whether the formula is satisfiable or not. This means the SAT method solves the decision problem, as opposed to ILP which solves optimization problems.

In 2020, the paper "Comparing Integer Linear Programming to SAT-Solving for Hard Problems in Computational and Systems Biology" [1] promoted the strength of SAT solvers in computational biology. According to the paper, SAT solvers can compete with the most widely-used ILP solvers. They presented four computational biology problems, where their ILP implementation took more time with the ILP solver Gurobi [2] than a SAT formulation with the SAT solver CryptoMiniSat5 [3]. They developed SAT and ILP formulations for each problem and tested them on different datasets. In many cases, the SAT approach was faster or found better bounds before the process was terminated. Sometimes the SAT-Method was slower than the ILP counterpart, which resulted in the conclusion that SAT methods should be considered when ILP fails. Although this shows SAT is a viable strategy, in practice ILP is often superior without having a deeper understanding of why this is the case. Also, there are often multiple approaches to solving a problem. Some approaches might be faster for both ILP and SAT solving than others.

1.1 Motivation

A web tool that was developed in a preceding projektarbeit helps to compare these formulations. Users can upload scripts which create ILP or SAT formulations using instance data for a specified problem. The web tool runs these scripts with test instances and solves the resulting formulations with ILP or SAT solvers. Each formulation is compared on time and correctness afterwards. This allows the developer to directly compare ILP and SAT formulations for different problems, making the development process faster. But the main problem remains because the main development time goes into creating the

scripts that create ILP and SAT formulations.

Expressing ILP formulations as logic clauses is not always intuitive. Especially while working with integers, because numbers need to be defined as a set of Boolean variables. Each logic gate, like AND, OR and XOR can be expressed in linear equations, making it harder to work with them. When the gates are nested, e.g. $A \vee (B \wedge C)$, helper variables are needed (see Section 4.3). Understanding the translated equations is much harder than reading the logic it originated from. For this, the equation must be explained with Boolean logic. Additionally, when the documentation is unknown and one wants to create the SAT formulation counterpart, the ILP formulation needs to be reverse-engineered. Equations must be collected and interpreted as logic clauses, which can be difficult on complex formulations.

Developing SAT formulations is also time-intensive. Normal SAT solvers do not accept arbitrarily logic clauses, only CNF clauses. The CNF consists of the conjunction of clauses, also called AND. Each clause in turn is a literal or a disjunction of literals also called OR. A logic formula can be represented as an expression tree. In an expression tree, each node represents a Boolean operation with the arguments as its child node. The leaves of this tree consist of literals which describe a Boolean variable, a negated variable or a Boolean constant. An example expression tree is displayed in Figure 1. The benefit of the CNF representation is the limited depth of the tree so that it can be represented in a two-dimensional matrix. There is a simple method to transform a logic formula into an equivalent CNF formula which uses the basic laws of Boolean algebra. Sometimes the number of transformed clauses grows exponentially, then the Tseitin transformation [4] is a good alternative. The disadvantage of this method is the introduction of new helper variables, so the developer must take both methods into account. Introducing new variables and more transformed clauses lowers the readability of the formula. In general, these transformations are done by hand and any human error leads to a faulty result in the end.

An additional problem with the creation of formulation scripts is the repeated occurrence of often similar building blocks. One of those building blocks is counting. In many applications, it is needed to ensure that at least n variables are true (or 1 in the ILP). A helper variable matrix of size $n \times |variables|$ is necessary to encode this. For each formulation, this needs to be reimplemented, even though it may not be the main focus of the research. Reusing existing code lowers the effort but still requires adaptations to the variables.

Counting is also needed for optimization. Optimization with ILP is simple, as it already includes an optimization function. SAT solvers solve decision problems, meaning only the satisfiability of a problem. To optimize with SAT, the SAT solver must run multiple times on different target values. If a target is satisfiable and the next target (target+1, in maximization, target-1 in minimization problems) is not satisfiable, the optimal solution is found. Thus, an additional script must be created and run SAT formulations repeatedly to optimize the problem.

Lastly, traditional formulation scripts rely on loop and condition statements. They express the universal quantifier denoted by $\forall$ or existential quantifier $\exists$. Nested condition statements obscure the expression and make it harder to debug.

1.2 Goals

This thesis introduces a framework that creates ILP and SAT formulations from an abstract language. The abstract language is closely related to predicate logic, but also includes non-logic building blocks, like counting. It features universal and partly existential quantifiers to show expressions more clearly. An abstract formulation can be coded directly in Python [5] and printed as a mathematical term.

The goal of the thesis is that without understanding SAT and ILP formulations, an ILP and SAT formulation can be created with the abstract language framework. The framework has to be a viable option compared to creating formulations directly and must simplify the process. The creation of formulations must be as simple and understandable as possible. Everything that is not essential for problem-solving, has to be handled by the framework.

The abstract language can be converted to abstract ILP or CNF formulations. An abstract ILP formulation shows the concept in a linear equation. For this, each term of the abstract formulation is expressed in a linear equation. Similarly, CNF formulations express the abstract formulation in CNF. With an instance of the problem, these formulations can be converted to solver readable formats. Finally, a solver can solve the problem without the development of a script.

To evaluate the framework, we construct a formulation for the Cluster Editing (CE) problem. For this, an existing ILP formulation from [6] is expressed in the abstract language and then converted to ILP and SAT formulations. Afterwards, manually created ILP and SAT formulations are tested against the generated formulations. The results show that the generated formulations produce correct solutions. But, the manually created SAT formulation took less time than the generated SAT formulation. Also, both ILP formulations were solved faster than the SAT formulations. This shows that CE problems should be solved with ILP.

Even though the generated SAT formulation is solved slower, one could create formulations without knowledge about SAT and ILP. With basic programming skills, it is possible to create correct formulations. Additionally, these formulations are still a viable option in comparison to creating them manually.

1.3 Related Work

Many solvers introduce frameworks to reduce the effort in creating formulations.

A symbolic ILP formulation can be written with the Python library PuLP [7]. Symbolic equations with symbolic variables can be added and solved with different solvers. This gives some abstraction over multiple solvers but is restricted to ILP solvers. Another CryptoMiniSat5 Python library [3], allows the iterative addition of clauses to SAT formulations. Both libraries do not include universal quantifications or other building blocks.

A lot more advanced solvers are the satisfiability modulo theory (SMT) solvers. SMT is a generalization of SAT, where also numbers, arrays and other operations are allowed. The $Z3$ [8] is one of those solvers. It supports conversion to DIMACS and also includes quantifiers. The scope of this thesis is not to compete with these solvers. It focuses on

logic problems with simple arithmetics and on creating similar formulations for ILP and SAT.

Constraint programming (CP) solves combinatorial problems. For this, a valid arrangement of variables must be found that satisfies all constraints. A constraint can consist of linear equations, relational logic and other statements. CP solvers generally solve the problem internally and adapt ideas from SAT solving. For example, Gecode [9] is a CP modeller and solver written in C++.

Algebraic modeling languages (AML), like AMPL [10], model large scale mathematical problems. Besides other features, an AML can model ILP formulations with symbolic algebraic notation. Similar to the abstract language, it uses the same notations as the modeller would use. It is possible to quantify constraints and summate variables. AMLs, like AMPL, can be translated automatically to LP formats and solved with one of many supported solvers.

MiniZinc [11] is a constraint modeling language. Constraints can consist of equations and boolean logic. The constraints can include universal and existential quantifiers. MiniZinc is compatible with many CP solvers and Mixed-Integer Programming solvers (MIP, generalization of ILP). Similar to the abstract language, the MiniZinc language is converted and exported to an external solver.

2 Preliminaries

This section describes all necessary definitions. First, we describe the example problems and afterwards the basics of creating and solving formulations.

2.1 The Example Problems

We explain and test the proposed framework with two main problems, Max Clique and Cluster Editing.

Clique A $clique$ is a subset of vertices in a graph, where all vertices are adjacent to each other.

Max Clique Problem Given a graph $G = (V, E)$, the Max Clique problem describes the largest subgraph $G' = (V', E')$ of G, such that V' forms a clique.

Cluster Editing Problem Given a graph $G = (V, E)$, the Cluster Editing problem describes the minimum edge modifications (insertions and deletions), such that the modified graph $G' = (V, E')$ consists of disjoint cliques. A graph consisting of disjoint cliques is also called cliquen graph.

2.2 Formulation Preliminaries

This section defines some important terms around formulations and solving.

Propositional Logic Propositional logic consists of propositions, also referred to as variables, and logic operations connecting them. Propositions can be true or false.

Predicate Logic Predicate Logic uses predicates instead of propositions, which can be quantified. As opposed to propositional logic, it uses quantification and relations.

Conjunctive Normal Form (CNF) The Conjunctive Normal Form is the conjunction of one or more clauses. A clause is a literal or a disjunction of literals. A CNF example:

$$\bigwedge_{c \in C} \bigvee_{i \in c} (\neg) x_i$$

The variable C is a set of sets, where each set contains an arbitrary amount of variables. x_i describes a Boolean variable. The CNF exists for propositional as well as predicate logic. How quantification fits into the CNF, is explained later.

Integer Linear Programming (ILP) Opposed to CNF, ILP expresses rules with equations. An ILP formula consists of integer variables and has a linear optimization function:

$$\sum_i c_i x_i, c \in \mathbb{Z}$$

The function is either minimized or maximized. Additionally, it contains a set of linear equations in the form:

$$\sum_i a_{0,i} x_i \leq b_0, a_{0,i} \in \mathbb{Z}$$

$$\vdots$$

$$\sum_i a_{n,i} x_i \leq b_n, a_{n,i} \in \mathbb{Z}$$

Where $b_j \in \mathbb{Z}$. A more compact and usual definition is by the matrix product $Ax \leq b$. x is a vector of all variables, a is the matrix of all $a_{i,j}$ and b is the vector of all constants $b_{i,j}$. The definition only allows for the less than equal $\leq$ operation, but in this thesis, the other operations $=, <, \geq, >$ can also occur, as the LP format allows these operations. ILP solvers use transformations that convert them so that they satisfy the strict definitions.

Formulation To collectively describe the ILP and SAT rules, we call them formulations. A formulation has a definition section, which defines the properties of the atoms like variables. Secondly, a formulation can contain an optimization function and a set of rules is the body of the formulation. All rules are chained by a big logical AND ($\bigwedge$). If one rule is not fulfilled, the formulation is not fulfilled. These rules are often referred to as constraints.

CNF Formulation The CNF formulation is a special case of a formulation. The rules in a CNF formulation are called clauses. Here, every clause is a disjunction of literals which implies that also the conjunction of all clauses is in CNF. Additionally, the CNF formulation does not have an optimization function. The optimization problem is expressed as a decision problem. Instead of asking for the optimal value v, we ask if is there a satisfiable solution with $v \leq k$ or $v \geq k$ (depending if it is a maximization or minimization). The target k is a free variable in the CNF formulation. By asking this question multiple times with different k, the optimum can be found.

ILP Formulation A ILP formulation, on the other hand, has a similar structure but uses different rules. An ILP formulation has a definition section that describes the properties of the used variables. This could be whether it is of an integer or real type or its bounds. The ILP formulation also contains a section with an optimization function. Finally, the body of the formulation consists of rules in the form of linear equations. Each linear equation can be wrapped in an arbitrary many quantifiers, which will be explained later.

Instance An instance describes a dataset on which a given problem needs to be solved.

Model The model assigns the instance data to the variables. Every variable or set, that is not deterministically set within the formulation, must be assigned a value in the model. The model in combination with the formulation must give a deterministic solution to a given problem. As a short abbreviation, the model is a mapping between variables and their values.

Concrete Formulation While a formulation may need context, meaning an instance, a concrete formulation can be solved directly with a solver. This concrete formulation does not contain any quantifications. It returns whether it is satisfiable if it is a concrete CNF formulation or the optimal solution with a concrete ILP formulation.

DIMACS CNF Format Many SAT solvers require the CNF formula to be stored in the DIMACS CNF format (Center for Discrete Mathematics and Theoretical Computer Science Conjunctive Normal Form format). We will refer to this concrete CNF formulation as the DIMACS format. Here, each clause represents a line in the text file, consisting of literals separated by whitespaces and closed by a zero. Each Boolean variable gets assigned to a unique integer identifier between one and the number of variables. Literals, that represent these variables, are stored as integers in the file. Any literal that is a negated variable is stored with a preceding minus sign. The file can contain comments at the beginning lines, starting with a "c" character. Also, the number of variables and clauses must be specified before the clause lines with the marker "p cnf".

Example CNF:

$$(A \vee B) \wedge (\overline{B} \vee C)$$

This can be expressed in the DIMACS format:

```
c some comment
p cnf 3 2
1 2 0
-2 3 0
```

The first line shows a comment that is ignored by the solver, while the second line describes the number of variables (3) and the number of clauses (2). Each variable A, B and C are represented by the integers 1, 2 and 3. With this, the third line describes $A \vee B$. The last line describes $\overline{B} \vee C$. Since B is negated, the integer representation is negated (-2).

LP Format ILP solvers accept concrete formulations in Linear Program (LP) format. It consists of three main parts, the objective function, the constraints and the variables. The objective function is a maximization or minimization of a linear term. It is separated from the constraint part by a SUBJECT TO text. Each constraint consists of a linear term, followed by an operator and a constant. The operator is one of the standard comparators $\leq, <, \geq, >, =$. The final section of the format handles the domain of the variables. Following the keyword GENERAL, all variables in the set of real numbers are defined. 0-1 variables are defined after the BINARY keyword. Finally, the END keyword marks the end of the formulation. The LP format has many more features, which are left out because they are not needed in this thesis. Example:

```
MAX x1 + 2*x2
SUBJECT TO
x1 + x2 < 1
GENERAL
BINARY
x1
x2
END
```

Conversion The conversion describes the process of converting a formulation into a concrete formulation. This process is described in Section 4.5. An expression is resolvable at conversion time if it is a constant value in the concrete formulation. This can be an expression that resolves when further information by the instance is given. It is important to differentiate this from expressions solved by a solver. The majority of the computational load is done by the solver.

2.3 Tools

Python 3 [5] is a dynamic programming language, that can be developed while running. Creating formulations often involves trial and error. It is often necessary to test parts of the formulation on their own to track down errors or to make improvements. Being able to change parameters or check results is crucial in development. Many solvers include libraries that allow an incremental development of the formulation. Cryptominisat [3] for example includes a Python library, where one could add clauses and test them at any given point. Incremental development turns out to be especially useful together with display formatters like jupyter [12]. Jupyter is a tool to develop and present short Python code snippets. Each snippet can be executed individually, but they share the same space of variables. If configured correctly, it displays graphs, tables or LaTeX code. A formulation written in the abstract language is symbolic. Every operation that is used stays unevaluated and each introduced variable has no value attached. Python operates with values and has no native symbolic variables or operations. The abstract language cannot use the Python native operations but is based on a framework built on top of Python. Sympy [13], a Python library for symbolic mathematics, yields basic functionality for this idea. According to the authors [13], it is a fully featured computer algebra system (CAS). It has various algebraic applications, from simple additions to complex integrals. Next to the potential of expressing symbolic terms, its core purpose is to simplify, evaluate and solve difficult expressions. This could be, for example, a derivative of a matrix or the limits of a variable in an equation. Most of the numeric algebraic capabilities of the Sympy framework are not needed for the abstract language project. But, in the abstract language, predicates hold numeric terms and some functions like set operations need equations and inequations. The following section explains the numeric Sympy features used. Variables are defined with the *symbols* function. Each variable is stored as a symbol object with a name and a type, like an integer type.

```
a,b,c=symbols("a,b,c",integer=True)
```

Variables can be part of an operation. Each operation is also an object, for example, a^2+1:

```
Add(1,Pow(a,2))
```

Python allows method overloading of the operation symbols +,-,* and so on. For clearer writing, Sympy uses this feature. An alternative articulation of the previous example is:

```
(a**2)+1
```

Mathematical expressions are automatically simplified by the framework. Simplification means that any operations which do not include variables are resolved immediately. Sympy includes a substitute function that substitutes any given variable for another expression. Substituting a with 2, for example, yields the result 5:

```
((a**2)+1).subs(a,2)
```

This is not limited to numeric expressions but also equations and inequations. The result of an equation is of Boolean type. Logic is another important section of the Sympy framework. The Boolean domain consists of two constant objects, namely BooleanFalse and BooleanTrue. Sympy variables can also be defined in the Boolean domain.

The Python symbols, normally used for bitwise operations, | (OR), & (AND) and $\sim$ (NOT) can form simple logic expressions. Other operations like implication-, equivalence-, XOR- or ITE-gates are also available. However, the logic is limited to propositional logic. One cannot define predicates or quantifiers.

A CNF converter for propositional logic is included, which rearranges the operations with the basic laws of logic.

The Sympy project includes a solver that can test expressions on their satisfiability. Building and solving many expressions directly with Sympy turns out to be not performant. To show this, we compare the runtime of the Sympy evaluation against the direct interpretation with Python. Substituting and solving a variable x with 5 in an expression x^2 one million times took about 30 seconds with Sympy. Interpreting x^2 one million times directly with Python took about 0.3 seconds. Using the solver, or more generally, using Sympy logic together with a concrete formulation is impractical. Sympy is designed for small use cases, where humans read, understand and develop expressions.

The project includes some set operations, like complements, unions or intersections. Other composition sets, such as the condition set, are also included. This thesis uses the basic object structure but no simplifications or solvers. As explained earlier, Sympy expressions do not scale well, which means that implementing these representatives is within the scope of this thesis.

The print library provides a wide range of printers, which present internal expressions in different flavours. Next to simple ASCII and Unicode printers, LaTeX printers export the expression in LaTeX code. Table 1 displays example operations together with different printing methods. Sympy supports the earlier described Jupyter program, which can interpret printed LaTeX code. A Code printer translates an expression into a language-specific expression. Several languages are supported, including C, Javascript and Python.

Function	Python keyword	LaTeX representation
$Add(a, b)$	$a + b$	$a + b$
$Mul(2, a)$	$2 * a$	$2a$
$Pow(a, b)$	$a * *b$	a^b
$And(a, b)$	$a\&b$	$a \wedge b$
$Implies(OR(a, b), c)$	$(a\|b) >> c$	$(a \vee b) \Rightarrow c$

Table 1: Operations and their representations
Example operations with variables a, b and c differently displayed. To form an expression, both the functions and the Python keywords $(+, -, *, \ldots)$ can be used.

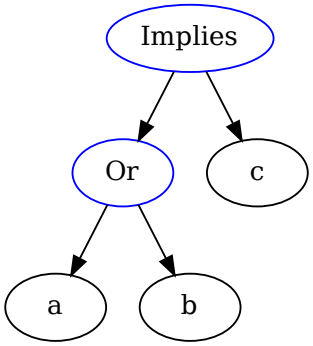

Figure 1: Expression tree
Expression $(a \vee b) \Rightarrow c$ displayed in a tree.

To be able to manipulate and extend the range of operations of Sympy, one must understand the structure of an expression. An expression is stored as an expression tree. Each node represents a Boolean operation with the arguments as its child nodes. Variables and constants are the leaf nodes in this tree, also called atoms. Figure 1 shows an example of an expression tree.

3 Abstract Language

In this section, the overall specifications of the abstract language are explained alongside the Max Clique problem. Given a graph $G = (V, E)$, the Maximum Clique (abbreviated Max Clique) is the biggest clique possible in G.

Predicates A predicate describes a property of objects. Each predicate can be defined with a unique name at the beginning of a formulation. It represents a specific feature of the problem. For example, in the given Max Clique problem, the predicate C indicates which nodes belong to the clique. An applied predicate is a predicate with arguments. In the Max Clique example, a node $v \in V$ can be inserted in the predicate, resulting in $C(v)$. v could be either a real value or a variable defined at the beginning of the formulation. The applied predicate $C(v)$ is true if and only if v is inside the resulting Max Clique.

The arguments of an applied predicate follow an order. A applied predicate $P(1, 2, 3)$ is not the same predicate as $P(3, 1, 2)$. Sometimes the order is hindering. If one would model an undirected graph edge $(1, 2)$ in predicate E, both $E(1, 2)$ and $E(2, 1)$ would represent the same meaning. $E(1, 2)$ and $E(2, 1)$ could have different values. Adding a rule $E(1, 2) \Leftrightarrow E(2, 1)$ would prevent this, but additional rules and equivalent predicates can be prevented. For this, ordered predicates ignore the order of the arguments. Variables are applied to these predicates as sets without order. This means $E(1, 2)$ and $E(2, 1)$ describe the same predicate.

The predicates had to be added to the abstract language framework, as the Sympy library does not support predicate logic.

ILP Predicates Instead of predicates, ILP predicates are used in ILP formulations. ILP Predicates are indexed integer variables. To differentiate them from predicates, they use angle brackets instead of round brackets, e.g. $P[i]$ instead of $P(i)$. The Python implementation of the ILP Predicate inherits from indexed variables which are used in the Sympy framework.

Variables Variables are the connection point between the formulation predicates and the real data. Since a variable v only describes a predicate P, these values are not limited to the integer type. Strings and other types are also accepted. These constants can be inserted directly in the formulation or in the model afterwards. Until now, for simplicity, only the insertion of variables into the predicates was mentioned. But at each index of an applied predicate, an arbitrary mathematical expression can be placed. This expression can contain multiple variables and constants. The expression can contain additions, multiplication and other basic operations. As an example, if the Max Clique problem additionally requires that the lexicographical median node is in the clique, this could be ensured by:

$$C(\left\lfloor \frac{|V|}{2} \right\rfloor)$$

Therefore the nodes V need to be represented and sorted by integers. Expressions in arguments of applied predicates are entirely handled by the Sympy framework.

Logic The core constraint is similar to predicate logic. An expression can be composed by all standard operations $\wedge, \vee, \neg, \Leftrightarrow, \Rightarrow, \oplus$. To ensure a clique in the Max Clique example, there must be an edge between two nodes in the clique. This rule can be enforced by preventing two non-adjacent nodes from being both in the clique. Given two nodes $u, v \in V$ with no common edge $(u, v) \notin E$:

$$\overline{C(u) \wedge C(v)}$$

The constraint ensures that all true $C(w)$ with $w \in V$ form a clique. Sympy does not include predicate logic, but by adapting the propositional logic of Sympy, it can be used for predicate logic.

Quantifier Until now, arguments in applied predicates only originate from the model. But these variables stay constant in a conversion. The universal quantifier with its notation $\forall$ allows for the scalability of datasets. In the predicate logic, a formula F is embedded in the quantifier as follows: $\forall x F$. Here formula F is quantified over all variables x in the domain, which could be an infinite set.

The abstract language uses a slightly different notation. Here, the domain is not bound to the variable but rather to the quantifier. With this, the domain of x can be chosen more freely and the variable can be used in other clauses. Additionally, the domain cannot be infinitely big, which would make any deterministic algorithm impossible. The better the restriction of the domain is, the shorter the concrete formulation and the faster the solver. Let S be a set and F a formula, describing for all x in S, applies F is denoted as:

$$\forall x \in S \colon F$$

S could also consist of tuples, triplets and so on. As an example, the Max Clique constraint from before must be aggregated over all edges $(u, v) \in \overline{E}$ (complement of the graph edges E). Ignoring the implementation notation of $\overline{E}$ for now, the complete clause is notated as follows:

$$\forall u, v \in \overline{E} \colon \overline{C(u) \wedge C(v)}$$

The universal quantifier is not bound to the outer layer of the constraint. The quantifier can be wrapped in another quantifier or with other predicate logic, e.g.

$$P(2) \Rightarrow \forall (i, j) \in S \colon P(i) \vee P(j)$$

With the introduction of the existential quantifier, denoted with $\exists$, the similarity to predicate logic diminishes. Because of technical limits, one can only use literals inside an existential quantifier. Also, the existential quantifier cannot be wrapped in logic other than universal quantifiers. For example, $\forall i \in S \exists i \in S'$ is allowed, but $P(i) \wedge \exists i \in S'$ is not. The reasoning behind this limitation is described in Section 4.2. This quantifier describes whether a satisfiable predicate $P(x)$ exists in the domain set S:

$$\exists x \in S \colon P(x)$$

An alternative notation would be $\bigvee_{x \in S} P(x)$ for the existential and $\bigwedge_{x \in S} P(x)$ for the universal quantifier. This helps to understand the meaning of the quantifier but is rarely used in formulation descriptions.

The Sympy framework does not include any predicate logic. The quantifiers had to be added to the Sympy library.

Set operations With enough preprocessing of the instance, it is possible to only use simple sets, so-called symbolic sets. A symbolic set is a variable with a name that is replaced by a set of values in the conversion phase. The Max Clique clause described above uses a symbolic set E. This means the model must contain this variable with a mapping to the values.

The Sympy framework features a wide range of set operations such as union, intersection, the cartesian product or image sets. The abstract language framework only uses these set operations symbolically. Resolving these set operations had to be implemented in the abstract language (see Section 4.5).

Some important set operations are missing in the Sympy framework. Often a specific subset of a set is needed. For example, a set containing all neighbours of node v in a graph $G = (V, E)$ is necessary. One could provide an additional set S in the instance which was precomputed. This makes it less readable, the data in the instance is redundant and the instance needs additional preprocessing. With condition sets, displayed with the set-builder notation, these issues can be avoided. To achieve the goal, every node u is in the set, such that $(v, u) \in E$:

$$\{u \mid u \in V \wedge \text{contains}((u, v), E)\}$$

In Python code:

```
ConditionSet(u, V, contains((u,v), E))
```

where the Boolean function *contains* implements $(v, u) \in E$. $u \in V$ describes the domain of u. Theoretically, this could be derived from the condition $(v, u) \in E$. For technical reasons, the domain always needs to be described. The condition itself can consist of any possible Boolean expression, as long as it is resolvable at conversion time. The condition set of Sympy could not be used, as it has compatibility problems with the other sets.

Lastly, a very useful set operation for graphs is the combinations set. Given a set V and a value k, this set describes all subsets of V which have k distinct elements. Since this is equal to the binomial coefficient, the set uses its notation. Listing all possible edges in a graph (V, E) with product and condition operations is possible but complicated. Self-looping edges need to be avoided, as well as semantically identical edges, e.g. (u, v) and (v, u). The combination operation is perfect for this case. With Python it is expressed as:

```
Combinations(V, 2)
```

with the LaTeX representation:

$$\binom{V}{2}$$

Order function The abstract language often handles sets which have no internal order. Sometimes values need to be ordered in an arbitrary matter. This is often used for sorting or counting variables. Another example is the chaining of applied predicates. If the predicates $P(1)$, $P(2)$ and $P(3)$ should be chained with some operation, one could refer

to them using $P(i)$ and $P(i+1)$ with $i \in \{1, 2\}$. But, since sets are unordered, in one moment $P(1)$ could be chained to $P(3)$, whereas in the other it is chained to $P(2)$. Another problem is the type of i. If i is a string, there is no existing method to get a successor by adding 1. We defined the $order_P$ bijective function for this purpose. It randomly maps a set of variables to ascending integers. Values of text type can be mapped to integer identifiers which enables access to predecessors and successors. The previous example would for example order the predicates:

$$order_P(2) = 1$$
$$order_P(3) = 2$$
$$order_P(1) = 3$$

With $P(order_P(i))$ and $P(order_P(i)+1)$ the values can be chained reliable and deterministic. The predicate P in $order_P$ defines which predicates should be ordered. This ensures that the function can be used for other predicates separately.

Functions There are occasions where expressing functions with the known functionality is limited. As an example, given a graph $G = (V, E)$ and three variables $u, v, w \in V$, write $P(u, v) \vee P(v, w)$ if $(u, v) \in E$. With the previously described methods, one could write the following:

$$\forall i \in \{x \mid x \in \{u\} \wedge (x, v) \in E\} \colon P(u, v) \vee P(v, w)$$

Here, a single term is expressed, if (u, v) is in E, otherwise nothing. The example shows that it is already possible to achieve these goals. But this is obscuring and complicates the formulation which defeats the purpose of the abstract language. Boolean Python functions solve these issues. Functions can be defined with an identifier name and an implemented Python function. This way, the realization is handled by Python and is not part of the abstract language. With a describing function name, this abstraction does not obstruct the readability of the constraint. The example from before now looks like this:

$$P(u, v) \vee P(v, w) \vee \overline{hasEdge(u, v, E)}$$

$hasEdge(u, v, E)$ returns true if and only if (u, v) is in E. If $(u, v) \notin E$, $\overline{hasEdge(u, v, E)} \Leftrightarrow True$, and therefore the entire term is true. Adding true constraints is the same as adding no constraints, fulfilling the requirement. Otherwise, the term is equivalent to $P(u, v) \vee P(v, w)$, because $\overline{hasEdge(u, v, E)}$ is false and can be neglected in a disjunction (OR). In Python, this could look like this:

```
def edge_in_E(u, v, Edges):
    return (u,v) in Edges or (v, u) in Edges
hasEdge = abl.BooleanPythonFunction("hasEdge",edge_in_E)
```

Boolean functions are Boolean constants after all variables are substituted for constants. Therefore, Boolean functions are literals. The symbolic functions are based on a Sympy function class. This class is not compatible with Boolean expressions and had to be extended. Also, to create a function object in Sympy, it must be defined as a class that inherits from the Sympy function class. Additionally, the function must implement some methods. This is simplified in the abstract language framework so that a function can be created using a Python lambda and a function name.

If-then-else (ITE) The If-then-else operator is a logic operation which decides with variable a, whether b or c must be true. Its abstract language notation is:

$$\text{ITE}(a, b, c)$$

ITE clauses are a convenient way to express the conventional logic:

$$(\overline{a} \vee b) \wedge (a \vee c)$$

If the a is constant by the conversion time, meaning it is true or false, depending on its value, b or c is ignored. Depending on the complexity of the expressions b and c, part of the expression might stay in the final concrete formulation. Also, removing the ignored part increases the computation time. The Sympy framework has an integrated ITE function.

Constant ITE As seen in the ITE description, adding ITEs with a constant a expression can be costly to the formulation creation as well as the solving time. We added constant ITE clauses to Sympy which does not have this problem. Expression a must be constant in this clause and a and b are limited to literals. This means that a and b can be either true, false, a predicate or a negated predicate. Why this limitation is needed is described in Section 4.2. One special case would be the Constant-Negate-If clause. A constant expression a decides whether a literal b should be negated:

$$\text{ConstantNegateIf}(a, b) \Leftrightarrow \text{ITE}(a, \overline{b}, b)$$

A constant ITE is a literal because at conversion time it only contains an applied predicate or its negation.

3.0.1 Building Blocks

The so-called building blocks, add non-logical statements to the language. They give a level of abstraction to the formulation and contain a hidden ILP and SAT interpretation. If no LaTeX representation is available, the notation is similar to the function notation. They cannot be combined with other logical expressions other than quantifiers. This part is extendable and other blocks could be easily added. The main focus is on the most used building blocks. The Sympy framework does not include any support for building blocks.

OneHot Encoding Sometimes exactly one distinct value of a set must be picked. The OneHot encoding ensures that a set of applied predicates are connected by an XOR gate. Depending on the set size, different implementations may be chosen. For example default implementation with predicate $P(i)$ over set S:

$$\text{OneHot}_{i \in S}(P(i))$$

Another implementation is the BinaryOneHot implementation which looks similar. The advantages and disadvantages of both implementations are described in Section 4.4.4.

At-Most-One Encoding The At-Most-One Encoding is similar to the OneHot encoding without the requirement that at least one applied predicate must be true.

Counter Every optimization method needs a possibility to count or summate variables. The counter block counts true predicates $P(i)$ with $i \in S$. For this, we interpret $P(i)$ as 1 if it is true, else 0. This makes the notation more comprehensible:

$$\sum_{i \in S} P(i)$$

It is not possible to interleave multiple sets, e.g. to include another variable j. For this, the two sets need to be combined, meaning S is a set of tuples. Another big drawback is the limitation of the counter body. It can only contain literal and constant ITEs, but no other expressions. The counter block is not a constraint and cannot be used on its own. But it is a useful tool together with other operators.

Comparison Operators Sometimes there is a need for a *at-most-n* or *at-least-n* constraint. With a generalized n, there is no efficient implementation, like for the $n = 1$ special case (OneHot, At-Most-One-Encoding). Comparison operators, namely $\leq, <, \geq, >$ and $=$, compare a counter object with a constant n. The formulation is satisfiable if the expression is. With a count of predicates $P(i)$, a set S and a constant n, this can look like this:

$$\sum_{i \in S} P(i) \leq n$$

Optimize For optimization problems, a minimization or maximization of certain predicates is needed. The operator's min and max take a counter $\sum_{i \in S} P(i)$. The Max Clique problem can use this. As stated before, the predicate $C(v)$ describes if node v is within a solution clique. If we maximize the amount of true $C(v)$ predicates over all nodes $v \in V$, the result is the optimal solution:

$$\max \sum_{v \in V} C(v)$$

If some reduction rules were applied beforehand, the optimal target could be not correct. To solve the Max Clique problem, one could use a simple reduction method. In the reduction, every node is ignored which has an edge over every other node. Every clique without one of these nodes v can be expanded by v. This means they are always contained in the optimal solution and can be ignored. But, if the reduction removes m nodes and the optimal target of the formulation is n, the correct solution is $n + m$. This motivates the additional arguments to the optimization, constants a and b. a is a constant multiplier to the counter of the optimization and b is added to that. Using a and b is purely for aesthetic reasons since the constants do not influence the optimal solution. The complete notation of an optimization constraint, in this case maximization, looks like this:

$$\max a * \sum_{i \in S} P(i) + b$$

```
import abstractlanguage as abl

n, i, j = abl.symbols('n, i, j')
V = abl.Range(0, n)                          # Nodes in the graph
V.label = 'V'
Ecompl = abl.SymbolicSet("\\overline{E}") # complement of E
C = abl.Predicate('C')                       # Nodes in the clique

f = abl.Formulation()
f.name = "maxclique"
f.add(abl.ForAll(i,j, Ecompl, ~(C(i) & C(j))))
f.set_optimization_function(abl.Maximize(abl.Counter(C(i), V, n)))
```

Figure 2: Max Clique Python formulation
Abstract language implementation of the abstract Max Clique formulation in Python.

3.1 Max Clique Formulation

To complete the Max Clique formulation example, the full formula is shown here. Given a Graph (V, E) and a predicate C, the optimal solution can be computed with:

$$\max \sum_{v \in V} C(v)$$
$$\forall v, u \in \overline{E} \colon \overline{C(u) \wedge C(v)}$$

Figure 2 shows the implementation in Python.

4 Methods

The abstract language is a tool to represent a formulation and has no other uses. To implement the functionality of that formulation, a script must be written. This thesis proposes a converter which can create solver-accepted instances from the abstract language. Each conversion and solving step is also shown in Figure 3.

Initially, the focus lies on the symbolic representation in Python. For this, the Sympy framework needs to be extended by predicate logic and other special functions defined in the abstract language. The second step describes the conversion to a CNF formulation. We show two techniques which use different approaches to create a logically equivalent CNF expression from any expression. Similarly, we show the transformation to an ILP formulation. In both conversions, the building blocks are replaced by actual logic or equations respectively. The Section 4.4 discusses possible implementations for the building blocks. Subsequently, the following section describes how the resulting formulation is translated into the desired format using the data from an instance. This section is mainly concerned with the efficient quantification of clauses, but also looks at possible post-processing optimizations. The outcome is written to a file. Applied predicates must be cast to integer identifiers for the DIMACS format or to valid LP format variables. Section 4.6 describes the iterative solving process. The optimal result must be cast back into the defined applied predicates and returned to the caller. Also, a faster formulation creation is presented.

To evaluate the abstract language framework, we create a formulation for the Cluster Editing problem with the developed tool. Additionally, we create a formulation traditionally, to be able to compare the correctness and runtime with the automated framework.

4.1 Python Representation

Sympy does not include predicate logic or predicates without quantifiers. But by inheriting from the Boolean class, we create a base class called FirstOrderLogic for this purpose. The predicate class inherits directly from this class. By creating a predicate object, we can define a name and if the arguments of the applied predicate have an order. Internally, this object also memorizes if it was generated automatically. The solver should only return predicates defined by the user since generated predicates are not part of the wanted solution set. With a given predicate P and arguments i, j, an applied predicate object can be build: $P(i, j)$. This object contains a reference to the predicate P and the arguments. Applied predicates can use the main Boolean operations like $P(i)\&P(j)$ because they inherit from $sympy.Boolean$. ILP formulations, on the other hand, rely on numeric expressions, e.g. adding two applied predicates. This is where the ILPPredicate comes into play. It inherits from $sympy.IndexedBase$, which is already a base class for indexed variables. The ILPPredicate has a reference to its Boolean version, if it exists, to inherit its properties. Additionally, it stores whether it is a $0-1$ variable. This is because ILP predicates can also be unbounded, i.e., they can take other values besides 0 and 1. The applied predicate counterpart is the ILPAppliedPredicate. It inherits from $sympy.Indexed$ and is created by indexing an ILPPredicate ($P[i]$). The abstract quantifier class expects indices,

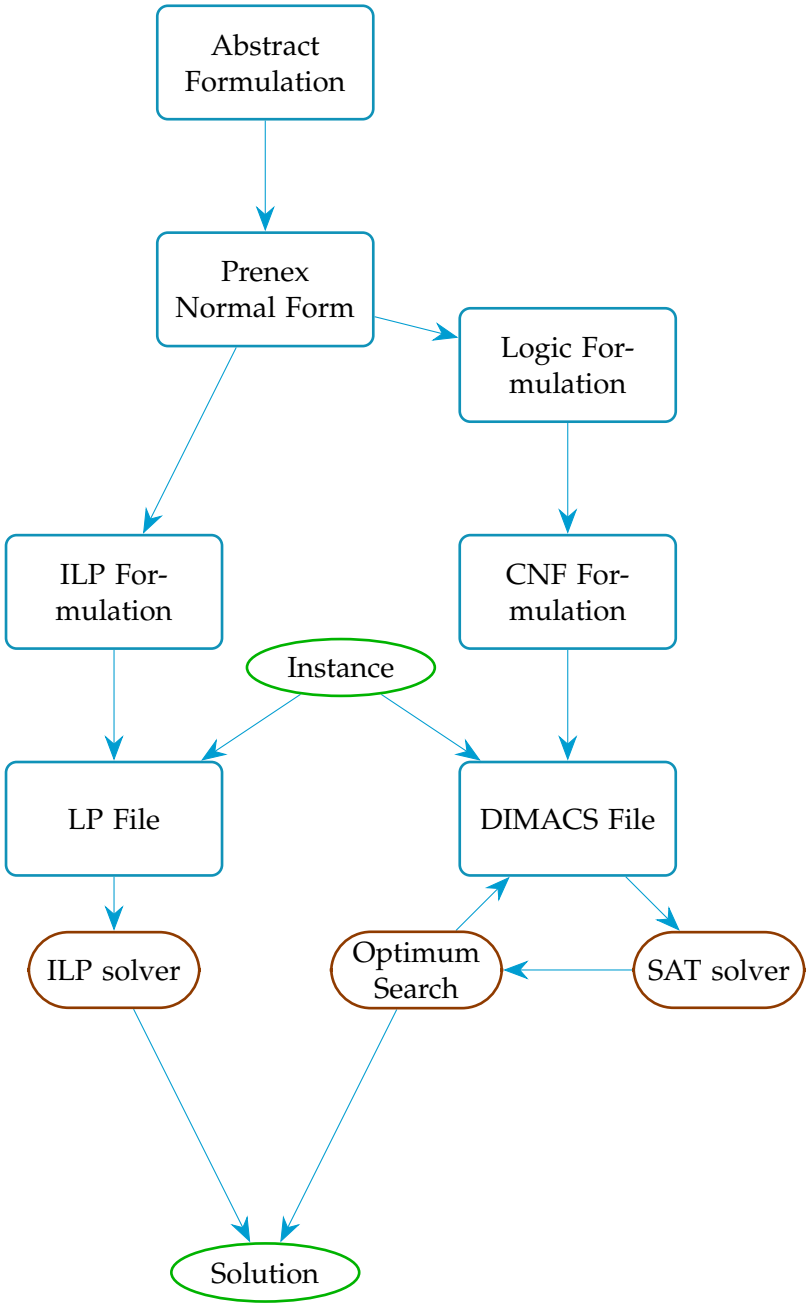

Figure 3: Conversion and solving steps
This figure shows all the internal steps of the framework. The blue vertices show the format of the formulation. They represent the modelling phase. Separated from that, is the solving phase (brown) of the framework. The green ellipses show the input and output of a formulation solving process.

a set and an expression. The classes ForAll and Exists inherit from the quantifier.

The building block base class is easily extendable. A building block class must include methods that convert the block to ILP constraints or SAT clauses. The arguments can be set individually as well as their printed representation. Onehot, Compare and Optimize are already included.

There are four different ways to create a formulation object. The most general formulation allows expression in the abstract language. It is called *Formulation* and is defined with a name. A clause can be added to the formulation f by creating a *Clause* object which contains an expression. A *LogicFormulation* inherits from *Formulation* but disallows building blocks in the clauses. The *CNFformulation* inherits from the *LogicFormulation*. In *CNFformulation* objects each clause must be in CNF format. The *ILPFormulation* class is structured similarly and only allows equations as clauses. Each equation consists of a linear term on one side and a constant on the other side. Each formulation uses predicates in form of the class AppliedPredicates, except the *ILPFormulation* which uses ILPPredicates with the ILPAppliedPredicates class.

4.2 CNF conversion

The conversion of any propositional logic formula into CNF can be done in two main ways. The naive method first creates an equivalent expression that only includes the operations AND, OR and NOT. Implications such as $a \Rightarrow b$ can be expressed equivalently as $\overline{a} \vee b$. An ITE gate ITE(a, b, c) can be expressed as $(a \wedge b) \vee (\overline{a} \wedge c)$. Similar translations exist for other operations such as the XOR or the Equivalence operator. For example, removing implications in the expression

$$(a \vee b) \Rightarrow c$$

gives the following expression:

$$\overline{a \vee b} \vee c$$

Another intermediate step includes the Negation Normal Form (NNF). In this form, the negation operator is only applied to variables. Also, the expression only contains conjunctions (AND) and disjunctions (OR) besides the negation. Using the De Morgan's Laws, the negations are shifted inwards. In our example, the remaining negation of the implication is moved inwards:

$$(\overline{a} \wedge \overline{b}) \vee c$$

Finally, to create an expression in CNF, the conjunctions are distributed over the disjunctions. This is done by repeatedly replacing $(x \wedge y) \vee z$ with $(x \vee z) \wedge (y \vee z)$ for any variables x, y, z. In the example from before, the CNF looks like this:

$$(c \vee \overline{b}) \wedge (c \vee \overline{a})$$

This propositional CNF conversion is already part of the Sympy library. But this naive approach has a major flaw. For larger expressions, the amount of disjunctions increases exponentially compared to the original expression. Converting an expression with 4 conjunctions in a disjunction leads to an exponential blowup of $2^4 = 16$ clauses:

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3) \vee (x_4 \wedge y_4)$$

to CNF:

$$
\begin{aligned}
&(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3 \vee y_4) \wedge (x_1 \vee x_2 \vee x_4 \vee y_3) \\
\wedge\, &(x_1 \vee x_2 \vee y_3 \vee y_4) \wedge (x_1 \vee x_3 \vee x_4 \vee y_2) \wedge (x_1 \vee x_3 \vee y_2 \vee y_4) \\
\wedge\, &(x_1 \vee x_4 \vee y_2 \vee y_3) \wedge (x_1 \vee y_2 \vee y_3 \vee y_4) \wedge (x_2 \vee x_3 \vee x_4 \vee y_1) \\
\wedge\, &(x_2 \vee x_3 \vee y_1 \vee y_4) \wedge (x_2 \vee x_4 \vee y_1 \vee y_3) \wedge (x_2 \vee y_1 \vee y_3 \vee y_4) \\
\wedge\, &(x_3 \vee x_4 \vee y_1 \vee y_2) \wedge (x_3 \vee y_1 \vee y_2 \vee y_4) \wedge (x_4 \vee y_1 \vee y_2 \vee y_3) \\
\wedge\, &(y_1 \vee y_2 \vee y_3 \vee y_4)
\end{aligned}
$$

The naive method is part of the Sympy library. The method is modified in the abstract language framework to include the proposed functionality.

The Tseitin transformation named after its inventor in 1970 [4] takes any logical expression and produces a smaller CNF formula. Instead of duplicating subexpressions in the naive approach, each is equivalent to a newly assigned variable. This allows the substitution of subexpressions in expressions by their assigned variable. The previous example has one relevant subexpression $(a \vee b)$. Assigning new variables to literals, c in this case, is not necessary.

$$(a \vee b) \Rightarrow c$$

Giving the subexpression an equivalent variable d:

$$d \Leftrightarrow (a \vee b)$$

The subexpression in the example can be replaced, but the new equivalence must be added:

$$(d \Rightarrow c) \wedge (d \Leftrightarrow (a \vee b))$$

Now we use the naive approach to convert this formula to a CNF formula:

$$(\overline{d} \vee c) \wedge (\overline{d} \vee a \vee b) \wedge (d \vee \overline{a}) \wedge (d \vee \overline{b})$$

The resulting formula is not equivalent to the original formula. But if and only if the original formula is satisfiable, this one is. This is called equisatisfiable.

For example, the expression with 4 conjunctions in a disjunction leads to an exponential blowup in the naive approach:

$$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3) \vee (x_4 \wedge y_4)$$

The Tseitin transformation only uses 13 clauses:

$$
\begin{aligned}
&(z_1 \vee z_2 \vee z_3 \vee z_4) \\
\wedge\, &(z_1 \vee \overline{x_1} \vee \overline{y_1}) \wedge (\overline{z_1} \vee x_1) \wedge (\overline{z_1} \vee y_1) \\
\wedge\, &(z_2 \vee \overline{x_2} \vee \overline{y_2}) \wedge (\overline{z_2} \vee x_2) \wedge (\overline{z_2} \vee y_2) \\
\wedge\, &(z_3 \vee \overline{x_3} \vee \overline{y_3}) \wedge (\overline{z_3} \vee x_3) \wedge (\overline{z_3} \vee y_3) \\
\wedge\, &(z_4 \vee \overline{x_4} \vee \overline{y_4}) \wedge (\overline{z_4} \vee x_4) \wedge (\overline{z_4} \vee y_4)
\end{aligned}
$$

The Tseitin transformation is not included in the Sympy framework but is part of the abstract language.

To summarize, the naive approach might be better in smaller samples, while the Tseitin transformation works well with large expressions. The framework has the option to choose the method which produces fewer clauses. For this, both methods must be computed. The naive approach possibly takes exponential time compared to the Tseitin transformation. Because of this, the option should not be chosen for large expressions. If the setting is not chosen, the Tseitin transformation produces the clauses.

Comparing and choosing the formula depending on the clause count is not scientifically backed. One could also take the variables or the clause length into account. In the paper Successful SAT Encoding Techniques [14], a few SAT solver experts were interviewed on their experiences with SAT formulations. On the topic of SAT transformation, the authors favoured the Tseitin transformation [14]:

> "There is consensus that these extra variables impose a much smaller overhead than the exponential blowup in size."

A minor improvement to the Tseitin transformation is the representation of equivalences. A formula $a \vee (b \Leftrightarrow (c \vee d))$ does not need a new variable because b can be reused. This leads to the clauses: $(a \vee b) \wedge (b \Leftrightarrow (c \vee d))$.

Lets look at arbitrary expressions $\phi_1, \phi_2 \dots \phi_n$ and formula

$$\phi_1 \Leftrightarrow \phi_2 \Leftrightarrow \dots \Leftrightarrow \phi_n$$

By just creating a variable v for ϕ_1, we save $n - 1$ variables:

$$\begin{aligned} v &\Leftrightarrow \phi_1 \\ v &\Leftrightarrow \phi_2 \\ &\dots \\ v &\Leftrightarrow \phi_n \end{aligned}$$

This alters the Tseitin method because normally each ϕ_i is assigned to a variable.

Another improvement is the caching of variables. In case an expression appears twice in a formula, the creation of two analogue variables is prevented. This also reduces the number of logically redundant clauses.

The abstract language cannot be converted directly to CNF using the Tseitin transformation. It uses predicates instead of variables and contains quantifiers. To create CNF from predicate logic, the constraints must be converted into the NNF. The NNF in predicate logic is similar to the NNF in propositional logic. The only difference is the negation of the quantifiers. The formula is in NNF if negations are only used on predicates. For this, applied predicates behave the same as variables and the same De Morgan rules apply. But, negating $\forall i \in S \colon \phi$ turns into a $\exists i \in S \colon \overline{\phi}$ and a $\neg(\exists i \in S \colon \phi)$ is equivalent to a $\forall i \in S \colon \overline{\phi}$. The Sympy framework has a function to convert an expression to NNF. The abstract language extends this method to also include predicates and quantifiers.

The next step is to transform the expression into the Prenex normal form (PNF). In the PNF, $\forall$ quantifiers become the prefix of the formulation. Every quantifier is bound to

the beginning of an expression, while the rest contains the quantifier-free operations. To prevent the redefining of variables, they need to be standardized first. If a quantifier uses the same variable as another quantifier, one must be renamed. For example:

$$(\forall x \in S\colon A(x)) \vee (\forall x \in S'\colon B(x+1))$$

Since x is defined at two places, one is redefined to y:

$$(\forall x \in S\colon A(x)) \vee (\forall y \in S'\colon B(y+1))$$

To move a quantifier outwards, a few rules can be applied. If a universal quantifier is inside an AND gate, their places can be swapped:

$$\begin{aligned}&(\phi \wedge \forall x \in S\colon \psi)\\ \Leftrightarrow\ &\forall x \in S\colon \phi \wedge \psi\end{aligned}$$

The same holds for OR gates. This can be explained by the interpretation of a $\forall$ quantifier as a big AND gate. In this interpretation $A(1) \vee (\forall x \in \{1,2\}\colon B(x))$ can be expressed as:

$$A(1) \vee (B(1) \wedge B(2))$$

Which is equivalent to:

$$(A(1) \vee B(1)) \wedge (A(1) \vee B(2))$$

On the other hand, existential quantifiers must be moved outwards. Here, $(\phi \wedge \exists x \in S\colon \psi)$ is replaced by $\exists x \in S\colon \phi \wedge \psi$ and the same holds for OR gates. The abstract language framework implements the PNF conversion.

Later, a process is applied that is called Skolemization. It was named after the mathematician Albert Thoralf Skolem and has the purpose to obtain a formulation without existential quantifiers. Here, each existential quantifier is removed by substituting its variable with a Skolem function. The Skolem function depends on all free variables, which are in the scope of the removed quantifier. These free variables are all variables from $\forall$ quantifiers, which appear in front of the given quantifier. Skolemizing $\forall i \in S\ \exists j \in S'\colon A(i,j)$, gives a expression $\forall i \in S\colon A(i, f(i))$ with Skolem function f. f maps j to i. This expression is no longer in predicate logic. The function f cannot be precomputed and is assigned to a value in the solving process. It is not possible to determine the correct i with j before knowing a satisfiable $A(i,j)$. This states a problem because functions are not allowed within the desired solvers. The abstract language was constrained for this reason. Recall that given an arbitrarily long chain of universal quantifiers ϕ and a literal $P(i)$, an existential quantifier is only allowed if:

$$\phi\, \exists i \in S\colon P(i)$$

This restriction allows bypassing the Skolemization. The existential quantifier can be interpreted as a big OR gate, which is already in CNF format. Nevertheless, Skolemization is included in the code and can be used by breaking the existential quantifier rule of the abstract language. This could be useful to understand the concept, but cannot be converted into a solver readable format.

Finally, the expression inside the prefix, meaning the inner expression that has no quantifiers, is converted to CNF. The expression is already in NNF, so the ORs must be distributed inwards over the ANDs. Alternatively, the Tseitin transformation can be used for the last step.

The required conversion to NNF at the start may lower the efficiency of the Tseitin transformation. The NNF may need more clauses and variables, for example, equivalences are optimized as shown before. Additionally, a helpful feature would be an intermediate step, where the quantifiers are on the outside and the inside is not limited to AND, NOT and OR gates. This intermediate step is stored in the LogicFormulation class. It intends to show the untangled formulation, where the quantifier definition part is separated from the inner expression. To preserve as much original logic as possible, untouched subexpressions are substituted with placeholder variables. Before the conversion process begins, the expression is parsed, substituting each subexpression that does not contain a quantifier with a placeholder variable. Later, after the NNF conversion and moving the quantifiers outwards, the variables are substituted back to the original expression. The NNF conversion is only necessary to move the quantifiers outward so that no laws are violated. As an example, a universal quantifier inside another operation:

$$A(1) \vee \forall i \in S\colon A(i) \Rightarrow B(i)$$

$A(i) \Rightarrow B(i)$ can be substituted by a variable v:

$$A(1) \vee \forall i \in S\colon v$$

After transforming to NNF and moving the quantifiers outwards:

$$\forall i \in S\colon A(1) \vee v$$

Now the expression can be substituted back from v:

$$\forall i \in S\colon A(1) \vee (A(i) \Rightarrow B(i))$$

The expression could preserve its original operators, making the formulation more comprehensible.

This adapted NNF conversion is implemented with the help of Sympy substitutions. Substitution is an important feature of Sympy, after integrating the quantifier and predicate logic, it is also a great tool for the abstract language. Generating placeholder variables is also possible with Sympy. But, there is no tool to automatically generate predicates. The predicates for the Tseitin transformation had to be generated with the name $xgen_i$ with an increasing i. This forces the language to never allow predicates that end on gen, as they are reserved for generated values.

4.3 ILP transformation

This section explains the conversion of logic expressions to linear constraints. In the following, the applied predicates $P(i)$ and $Z(i)$ are used to explain the transformations. For simplicity, only one argument i is used.

First off, all clauses except the building blocks are converted to Prenex Normal Form. This separates the quantifiers from the core expression. Initially, only the core expression is transformed to ILP. Also, constraints with an existential quantifier must be ignored.

Then, each applied predicate is assigned to an applied ILP predicate equivalent. By differentiating between Boolean and integer types, logic operations like AND or OR are not entangled with arithmetic operations $(+, -, *)$. For example, the applied predicate $P(i)$ has an ILP version $P[i]$, which is visually distinguishable by the square brackets. The new ILP predicate now represents true if its value is 1 and else 0.

The ILP transformation of operators in this thesis is based on the observations in the paper "Formulating Integer Linear Programs: A Rogues Gallery" [15]. They describe the process of expressing logic operators as ILP constraints.

Each logic operation $f(P(i), \ldots)$ can be transformed in two ways. When the operation must hold, meaning it must be satisfiable, we denote this with $f(P(i), \ldots)$. Else it is part of another operation g:

$$g(f(P(i), \ldots), \ldots)$$

Then the operation is substituted with a new predicate $Z(i)$. Therefore the operation must be the equivalent to $Z(i)$:

$$Z(i) \Leftrightarrow f(P(i), \ldots)$$

and

$$g(Z(i), \ldots)$$

This is similar to the Tseitin transformation, where each operation is substituted with a generated predicate. The top operation of a constraint must be satisfiable and is not substituted.

Transforming an AND operation with n predicates, such as $P(1) \wedge P(2) \wedge \ldots \wedge P(n)$, is ensured by constraining the sum of the predicates to be greater or equal to n. This is only the case if all predicates are 1:

$$\sum_{i \in [1,n]} P[i] \geq n$$

This transformation step is not needed for the conversion to ILP. The constraints of a formulation can be viewed as a conjunction of constraints. If one constraint is infeasible, the entire formulation is infeasible. This shows that the constraints are linked by an AND gate. Given expressions ϕ and ψ, a logic formulation with constraints $c_1, \ldots, c_m$ and constraint $\phi \wedge \psi$ can be viewed as $(\phi \wedge \psi) \wedge \bigwedge c_i$. With the associativity of the AND gate, ϕ and ψ can be viewed as individual constraints. By recursively applying this rule, the explained AND transformation is never used.

But, AND gates still occur inside other operations. The substitution $Z1(1) \Leftrightarrow (P(1) \wedge P(2) \wedge \ldots \wedge P(n))$ is transformed to:

$$0 \leq -n * Z1[1] + \sum_{i \in [1,n]} P(i) \leq n - 1$$

$Z1[1]$ must be equal to 0 if there exists a i so that $P[i] = 0$ because the sum must be smaller than n. Also, $Z1[1]$ must be equal to 1 if for all i $P[i]$ is equal to 1, because the sum must be greater (or equal) to 0.

The OR gate substitution is very similar to the AND gate substitution.

$$0 \leq n * Z1[1] - \sum_{i \in [1,n]} P(i) \leq n - 1$$

These equations are the ILP equivalent of the OR gate substitution $Z(1) \Leftrightarrow (P(1) \vee P(2) \vee \ldots \vee P(n)$. $Z1[1]$ must be equal to 1 if there exists a i so that $P[i] = 1$, because the sum must be bigger than (or equal) 0. Also, $Z1[1]$ must be equal to 0 if for all i $P[i]$ is equal to 0 because the sum must be smaller than n.

An OR gate $(P(1) \vee P(2) \vee \ldots \vee P(n))$ without substitution is realized by ensuring that the sum of all $P[i]$ is greater than 1:

$$\sum_{i \in [1,n]} P[i] \geq 1$$

If a NOT operation $\overline{P(1)}$ must hold in ILP, $P[1]$ must be restricted to 0:

$$P[1] \leq 0$$

In case the NOT operation was substituted, $Z1(1) \Leftrightarrow \overline{P(1)}$ is transformed to:

$$Z1[1] = 1 - P[1] \Leftrightarrow Z1[1] + P[1] = 1$$

Here, $1 - P[1]$ is 1 if $P[1] = 0$, else it is 0. Substituting the NOT gate is not necessary. Instead of using $Z1[1]$ in its parent operation, one could also use $(1 - P[1])$. For example, the expression $\overline{P(1)} \vee \overline{P(2)}$ can be transformed to:

$$\begin{aligned} Z1[1] + P[1] &= 1 \\ Z2[1] + P[2] &= 1 \\ Z1[1] + Z2[1] &\geq 1 \end{aligned}$$

But also much shorter:

$$(1 - P[1]) + (1 - P[2]) = 2 - P[1] - P[2] \geq 1$$

An implication $P(1) \Rightarrow P(2)$ is equivalent to $\overline{P(1)} \vee P(2)$. It is converted and handled by the OR transformation. In ILP terms, it is interpreted as $P[2]$ is greater or equal to $P[1]$:

$$(1 - P[1]) + P[2] \geq 1 \Leftrightarrow P[1] \leq P[2]$$

The substituted implication is processed similarly in an OR statement.

Other special operators, like XOR and the implication, are not implemented. The expression converts to NNF, before applying the transformations. This makes the implementation easier, as only AND and OR clauses need to be added. NOT gates only appear at leaves of the expression tree, which additionally simplifies the process.

But this has some drawbacks. While the ILP constraints of an implication gate are equivalent to its NNF, the XOR gate is different. The XOR gate can be implemented so that only one extra predicate $Z1[1]$ is needed for the expression $P(1) \veebar P(2)$:

$$\begin{aligned} Z1[1] &\leq P[1] + P[2] \\ P[1] - P[2] &\leq Z1[1] \\ -P[1] + P[2] &\leq Z1[1] \\ Z1[1] &\leq 2 - P[1] - P[2] \end{aligned}$$

But in the NNF conversion $Z1(1) \Leftrightarrow (P(1) \veebar P(2))$ becomes $Z1(1) \Leftrightarrow ((P(1) \vee P(2)) \wedge (\overline{P(1)} \vee \overline{P(2)}))$. This means, two more auxiliary variables and constraints are needed:

$$\begin{aligned} 0 \leq 2 * Z2[1] - P[1] - P[2] &\leq 1 \\ 0 \leq 2 * Z3[1] - (1 - P[1]) + (1 - P[2]) &\leq 1 \\ 0 \leq Z2[1] + Z3[1] - 2 * Z1[1] &\leq 1 \end{aligned}$$

A better transformation would include all possible conversions, for all possible special operators. Since these operators are rarely used (from experience), this is not in the scope of this thesis.

4.3.1 Generating Substitute Predicates

The substitution predicates, previously described as $Zi(1)$, must generate automatically in the transformation phase. The process uses the reserved predicate space $XGEN_i$. With every new substitution, i is increased and $XGEN_i$ is the new substitution predicate. If a subexpression appears twice in a formulation, the required constraints are only generated once with the same substitution variable.

Until now, the substitution predicate only contained the argument 1. After adding the quantifiers back into the ILP constraint, this becomes a problem. Given a constraint $\forall i \in Range(1,3) : P(i-1) \wedge (P(i) \vee P(i+1))$ that was transformed into ILP constraints and applied to the quantifiers:

$$\begin{aligned} &\forall i \in Range(1,3) \colon 0 \leq 2 * Z1[1] - P[i] - P[i+1] \leq 1 \\ &\dots \end{aligned}$$

The substitution variable $Z1[1]$ applies to $i = 1$ and $i = 2$. To prevent this behaviour, each substitution predicate holds all free variables from the quantifiers. In the above case, this is $Z1[i]$.

Lastly, each equation is reshaped to a specific format. All predicates are on the left of the equation, while the right side holds the constant value. This is needed, as some ILP solvers require this format. More precisely, the constraint is bought into the format:

$$c_1 * x_1 + c_2 * x_2 + \ldots + c_n * x_n \triangle C$$

Here, $x_1 \ldots x_n$ are predicates, $c_1 \ldots c_n$ are constant multipliers, C is a constant and $\triangle$ is a placeholder for any comparator.

Additionally, the equation is rearranged, so that the constant is always positive. This step is purely cosmetic and intends to increase the readability of the formulation.

4.3.2 Optimizing Constraints

Sometimes predicates appear which are trivially 1 or 0, e.g. $P(1) \leq 0$. The transformation process tracks these constants and substitutes them later with their constant value. This reduces the number of predicates and the complexity to understand the meaning.

Another optimization removes trivially satisfied clauses. For this, the linear term is minimized or maximized by varying the predicate values. For example, the equation $2 * x - y - z \leq 2$ for predicates x, y, z can be discarded. If the maximum value of the left term is less or equal to 2, this constraint is always fulfilled. The maximization of a linear term is the same as maximizing each summand:

$$max(2 * x - y - z) = max(2 * x) + max(-1 * y) + max(-1 * z) = 2 * 1 + -1 * 0 + -1 * 0 = 2$$

Here, the maximum value does not exceed 2. The clause is always fulfilled. This trivial optimization is only possible when constant predicates are substituted to 0 or 1. The other comparison operators use a similar optimization technique.

The optimization is intended to improve readability. ILP solvers use more sophisticated optimization practices.

4.4 Building Block Realization

Building blocks are an important tool in abstract language. They can abstract away complicated logic and simplify the development. Each building block object has a method called *to_lp_formulation*, which returns linear equations that implement the behaviour. In the same way, the method *dissolve_pseudo_boolean* returns logic constraints that implement the building block. These methods get called in the conversion steps from the abstract formulation to the ILP or SAT formulation. Building blocks can also contain LaTeX printers for better readability. The following sections describe the implementation of these building blocks.

4.4.1 Counting

To count true predicates, the block $Counter$ is used. Let i be an index, S a set, n the size of the set and $P(i)$ a predicate. The block is defined with the expression $Counter(i, S, L(i), n)$ and is expressed in LaTeX as:

$$\sum_{i \in S} P(i)$$

The counter itself is not a Boolean function and can only be used inside other building blocks.

The equivalent LP term for this is $P[1] + P[2] + \ldots + P[n]$ if $S = \{1, 2, \ldots, n\}$. This quantification cannot be resolved because S could be a variable set at this point. S is resolved in a later stage of the process when the ILP formulation is translated to an LP file. This means the expression is stored in another counter object:

$$\sum_{i \in S} P[i]$$

Creating a variable counter in logic is a more difficult task. This thesis implements a sequential counter that was used in a paper by Hannah Brown, Lei Zuo and Dan Gusfield [1]. For demonstration reasons, imagine S is a list $1, \ldots, n$ (not a set) and $\sum_{i \in 1,\ldots,n} P(i)$ has to be counted. The sequential counter counts partial sums $\sum_{i \in 1,\ldots,j'} P(i)$ for increasing j, until $j = n$ is reached.

This dynamic approach introduces a new predicate $T(i, d)$. $T(i, d)$ describes that at least d predicates $P(j)$ with $j < i$ are true. This means that $T(n, k)$ describes that at least k predicates are true. $\overline{T(n, k+1)}$ implies that at most k predicates are true. k is also referred to as the target value. Since at most n predicates can be counted, the upper bound of k is n.

The meaning of $T(i, d)$ can be expressed in some constraints. First, at least 0 predicates are always true at any given point:

$$\forall i \in 1, \ldots, n \colon T(i, 0)$$

Also, if d predicates are true before position i, also d predicates are true at position $i + 1$. This holds for all d where $d \leq k$:

$$\forall i \in 1, \ldots, n \, \forall d \in 1, \ldots, k \colon T(i, d) \Rightarrow T(i + 1, d)$$

When $P(i)$ is true, it must be added to the previous partial counts $T(i, d)$. Meaning $T(i + 1, d + 1)$ must be true, if $C(i)$ and $T(i, d)$ are true:

$$\forall i \in 1, \ldots, n \, \forall d \in 1, \ldots, k \colon (T(i, d) \wedge P(i)) \Rightarrow T(i + 1, d + 1)$$

With these constraints, $T(i, d)$ must be true if d predicates of $P(1) \ldots P(i - 1)$ are true. But, $T(i, d)$ is not restricted to return false when this is not the case. This is done by an implication in the other direction:

$$\forall i \in 1, \ldots, n \, \forall d \in 1, \ldots, k \colon T(i + 1, d) \Rightarrow (T(i, d) \vee (P(i) \wedge T(i, d - 1)))$$

This avoids $T(i, i)$, where more predicates are counted than available. For this special case, another constraint is needed:

$$\forall i \in 1, \ldots, n \colon T(i + 1, d) \Rightarrow (P(i) \wedge T(i, d - 1))$$

This sequential counter uses $k * n$ T predicates. This also includes predicates T(i,d), where $d \geq i$. Counting higher than available predicates $P(1), \ldots, P(i - 1)$ will never happen. Because of this, the proposed sequential counter is slightly modified. The sets of the previous constraints can be more restrictive if this constraint is added:

$$\forall i \in 1, \ldots, k \colon \overline{T(i, i)}$$

The predicate $P(i)$ was chosen for demonstration purposes, but any other literal, like $\overline{P(i)}$ or ConstantNegateIf$(f, P(i))$ could take its place.

For the explanation of counters, the set S was interpreted as a list of numbers 1 to n. Sequential counters need access to the successor $(i+1)$ and predecessor $(i-1)$ of a number i. But the set S has no order and is not limited to numbers from 1 to n. By definition, S

Comparison	Additional Constraint
$\sum > c$	$T[n+1, c+1]$
$\sum \geq c$	$T[n+1, c]$
$\sum < c$	$\overline{T[n+1, c]}$
$\sum \leq c$	$\overline{T[n+1, c+1]}$
$\sum = c$	$T[n+1, c] \wedge \overline{T[n+1, c+1]}$

Table 2: CNF translation to compare a sum with a constant c

should contain arbitrary values. The function $order$ maps all values from S to the values 1 to n. We defined and created the $order$ function in the abstract language for this case. The above constraints just slightly change. Each list $1, \ldots, n$ is replaced with an arbitrary set S and each index i in the counter predicates T is replaced by $order_P(i)$. For example, $T[i+1]$ is substituted to $T[order_P(i)+1]$.

These constraints are converted to CNF. They replace the original counter building block in the formulation. Additionally, the predicate T is generated, such that multiple counters can occur in the same formulation.

To express different comparisons with a constant c, additional constraints are necessary. The Table 2 shows these.

4.4.2 Comparing

The comparison block compares a counter block to a constant c. The allowed comparisons are $<, \leq, <, \geq, =$.

After the counter block is converted to ILP, the comparison block is a valid ILP formula.

To convert a comparison block to a CNF formula, the counter is converted to CNF and the corresponding expressions from the Table 2 are added.

4.4.3 Optimization

The minimizing and the maximizing block determine the optimization direction. The blocks consist of a counter block and a constant.

The ILP formulation can contain an optimization. To convert the optimization block from the formulation to the LP formulation, only the counter block argument must be converted.

CNF formulations only solve decision problems and not optimization problems. The optimum for CNF formulations is found by trying different values for a target t.

To optimize CNF formulations, the optimization is split into multiple decision problems.

Instead of asking $max \sum_{i \in S} P(i)$, one could ask for different targets t:

$$\sum_{i \in S} P(i) \geq t$$

Or for minimization problems $min \sum_{i \in S} P(i)$:

$$\sum_{i \in S} P(i) \leq t$$

To convert the optimization to a CNF formula, replace the optimization with a corresponding comparator block. The resulting building block can then be turned into a CNF formula.

4.4.4 OneHot

The OneHot encoding ensures that exactly one predicate $P(i)$ for $i \in S$ is true, where S is some set. An example OneHot encoding has the form:

$$\text{OneHot}(i, S, P(i))$$

The ILP equivalent for this is:

$$\sum_{i \in S} P[i] = 1$$

$P[i]$ is 0 or 1, so exactly one $P[i]$ is set to 1.

A CNF formula for OneHot encodings is presented in [14]. The formula consists of two parts. First, it must be ensured that at least one predicate is true:

$$\exists_{i \in S} \colon P(i)$$

Secondly, at most one $P(i)$ can be true. For a true $P(i)$ there exists no $P(j)$ that is true:

$$\forall i, j \in Combinations(S, 2) \colon \overline{P(i)} \vee \overline{P(j)}$$

This constraint contains a quadratic number of clauses.

A BinaryOneHot encoding can improve this second constraint to $n \log_2 n$ clauses. To simplify the explanation of the encoding, S is equal to $\{1, \ldots, n\}$.

The encoding needs an additional helper predicate B. $B(j)$ describes if the digit j in a binary sequence is set to 1.

To represent an integer $i \in 1, \ldots, n$ in binary, the binary sequence must consist of $\lceil \log_2 n \rceil$ digits. This binary sequence is unique to that integer. By encoding a binary sequence in a list of predicates $B(0), \ldots, B(\lceil \log_2 n \rceil - 1)$, a unique integer can be expressed.

Each predicate $P(i)$ can be represented by the binary representation of its index i. For example, if $n = 3$ and $i = 2$, the binary sequence has $\lceil \log_2 3 \rceil = 2$ digits. With this information, i has the binary representation 10. If expressed with predicates, the first digit must evaluate to false, while the second digit evaluates to true: $\overline{B(0)} \wedge B(1)$.

Onehot	**Onehot+binary**
$o_0 \vee o_1 \vee o_2 \vee o_3$	$o_0 \vee o_1 \vee o_2 \vee o_3$
$\bigwedge_{i \neq j} \neg o_i \vee \neg o_j$	$o_0 \rightarrow \neg b_0 \wedge \neg b_1$
	$o_1 \rightarrow \ \ b_0 \wedge \neg b_1$
	$o_2 \rightarrow \neg b_0\wedge \ \ b_1$
	$o_3 \rightarrow \ \ b_0 \wedge \ \ b_1$

Figure 4: OneHot and OneHotBinary
This figure from [14] shows two implementations of onehot encoded binary variables o_0, o_1, o_2, o_3. The *at least one* constraint is in both methods encoded in an OR gate.

The requirement that at most one predicate $P(i)$ is true, is ensured by the uniqueness of the binary sequence:

$$P(2) \Rightarrow (\overline{B(0)} \wedge B(1)).$$

Figure 4 shows a full example. The predicates $B(0), \ldots, B(\lceil \log_2 n \rceil - 1)$ encode one predicate $P(i)$, every other predicate must be false.

This is defined and implemented with tools from the abstract language. Let $f(i, m)$ be true, if the digit m is 1 in the binary sequence of i. Then, the following constraint ensures that at most one predicate $P(i)$ is true:

$$\begin{aligned} &\forall i \in \mathit{Range}(0, n): \\ &\quad P(i) \Rightarrow \forall j \in \mathit{Range}(0, \lceil \log_2 n \rceil): \text{ConstantNegateIf}(f(i, j), B(j)) \end{aligned}$$

To complete the BinaryOnehot definition, the at-least-one constraint is defined by:

$$\exists i \in \mathit{Range}(0, n): P(i)$$

In the implementation, the set S is not bound to $0, \ldots, n$. By using the $order$ function, the set S is mapped to $0, \ldots, n$. This is necessary to create a unique binary identifier for each item in S.

To convert an encoding to a CNF formula, the presented constraints are converted to CNF.

4.4.5 At-Most-One

The At-Most-One block allows only one true predicate in a set of predicates.

The CNF version of this block is similar to the OneHot block but without an existential constraint (at-least-one constraint). The ILP version of At-Most-One is simply:

$$\sum_{i \in S} P[i] \leq 1$$

4.5 Format Parsing

An ILP or CNF formulation holds free variables and quantifiers. These must be resolved to create LP or DIMACS files. Sympy has printers that convert expressions to Python code. This printer is extended with abstract language functionality so that entire formulation scripts can be created automatically. These formulation scripts, if supplied with data, then create the LP and DIMACS files.

The script takes a model and returns a finished LP or DIMACS formulation. The function $print(\phi)$ creates this script recursively. To do this, each operation implements a translation of itself and its arguments. To resolve a universal quantifier $\forall i, j \in \psi \colon \phi$, a Python for loop is created:

```
phipython for i,j in psipython
```

The code *phipython* and *psipython* represent the already translated expressions ϕ and ψ.

ψ is a set in this case. A set is either supplied by the model or is a composition of set operations. Each set operation has a translation to Python. For example, given a set S, a Boolean function $f(x)$ and a condition set $\{x \mid f(x) \wedge S\}$. After translating $f(x)$ to *fpython* and S to *Spython*, the condition set is translated to:

```
filter(fpython(x), Spython)
```

A Python translation exists for every symbolic operation, including the expressions inside a predicate, e.g. $P(i+j)$. Also, each introduced building block contains a translation from ILP to Python. A translation from CNF is not needed, as these building blocks are already resolved.

The inner AND and OR operations of a CNF formulation constraint are translated according to the DIMACS specifications. This means the arguments of an AND gate are separated by a newline. Arguments of an OR gate are separated by whitespaces and ended with a trailing 0.

To convert the applied predicates to an integer DIMACS variable, a generator assigns integers to the individual predicates. If the applied predicate is inside a negation, it must be preceded by a minus sign. For example, the constraint:

$$\forall i \in S \colon P(i) \vee P(i+1)$$

Is converted to:

```
(f"{GEN[P(i)]} {GEN[P(i+1)]} 0" for i in S)
```

Where GEN is the DIMACS variable generator.

The equations in ILP constraints translate to similar Python code, that parses the equations according to the LP format. It is not necessary to rename the ILP predicates like the CNF predicates, because they already conform to the naming conventions.

4.5.1 Postprocessing

Debugging DIMACS files is complicated. For example, if a DIMACS file is unsatisfiable, even though it was expected to be satisfiable, one sets certain DIMACS variables to true or false. By setting certain variables to true or false, other variables may also become constant. After manually removing clauses with this technique, the unsatisfiable core of the DIMACS formulation can be isolated.

A variable can be set to true by adding a clause that only contains the variable. The formulation is only satisfiable if the clause and therefore the variable is false.

Postprocessing recursively removes constant predicates, leaving only the core logic. For example, the predicate $P(1)$ must be true:

$$P(1) \wedge (\overline{P(1)} \vee P(2))$$

Evaluating $P(1)$ to true shows that $P(2)$ must be also true.

The postprocessing module recursively removes constant variables from the DIMACS file, making debugging easier.

4.6 The Solving Process

The process includes a solving module which automatically solves a formulation with an SAT or ILP solver. Afterwards, it returns the optimal value and the optimal solution set.

In the ILP solving process, the LP file is created and solved with an ILP solver. Afterwards, the solution integer variables are interpreted as Boolean predicates and returned to the caller.

The solving process of CNF formulations takes more steps. SAT only models decision problems instead of optimization problems. Instead of "maximize x" it asks "is there an assignment of x such that $x \geq k$". If this question is fulfilled with $k = n$, but not $k = n+1$, the optimal solution is n. By searching the domain of k to find the optimal solution, the decision problem solves an optimization problem. For each k, a DIMACS formulation must be built and tested with a SAT solver.

In linear search, k is first set to a lower bound that is known to be satisfiable. Then k iteratively increased and tested for satisfiability. If $k + 1$ is not satisfiable, k is the optimal target.

Alternatively, the binary search halves the search space with each run. k is set to the median integer of the search space. If k is satisfiable, the solution lies in the upper half of the search space, else in the lower one. This requires fewer solving steps but is often not advised. While the linear search stops at the first unsatisfiable formulation, the binary search could run into multiple. Unsatisfiable formulations take in general more time to solve than satisfiable formulations. This often outweighs the benefits of fewer solving steps.

The solving process was described using maximization. Minimization searches in the opposite direction. After finding an optimal solution with a search method, the solution

```
fcnf = abl.CNFFormulation(f)

# retrive_data
# V_inst, E_inst describe a graph
# lb and ub are known lower and upper bounds
V_inst, E_inst, lb, ub = retrive_data()
model = {Ecompl: E_inst.complement(), n: len(V_inst)}
instance = abl.Instance(fcnf, model, lowerbound=lb, upperbound=ub)

result = abl.Solver(fcnf, instance).run()
```

Figure 5: Max Clique solving in Python
The Max Clique formulation *f* (see Figure 2) is converted to a CNF formulation *fcnf*. Afterwards, the formulation is optimized using a supplied instance.

variables are returned to the caller. Figure 5 shows how this process can be started using the Max Clique problem as an example.

4.6.1 Succeeding Runs

A DIMACS formulation for target k has many identical constraints to a formulation for target $k+1$. The succeeding runs method uses this to lower the formulation creation time. If a clause in a CNF formulation does not contain the variable k, it will not change the DIMACS outcome on different k. This means that these DIMACS clauses can be saved for later runs and do not need to be recalculated.

Additionally, counter blocks in maximizations can also store clauses. The counter block is built dynamically, which means the block counting to $k+1$ can be added on top of the block adding to k. Fewer clauses need to be computed and clauses from a previous run can be reused.

These additions decrease the building time of DIMACS formulations.

4.7 The Cluster Editing Example

Cluster Editing can be solved by an ILP formulation proposed by Martin Klümpen [6]. This section shows an application of the abstract language by generalizing the ILP formulation to an abstract formulation. After that, the abstract formulation can be used to solve the problem with SAT and ILP solvers. The complete implementation of the abstract language and the CE example can be found on GitLab [1].

Recall that a cliquen graph is a graph consisting of non-adjacent cliques. Given a graph (V, E), the ILP formulation returns a modified graph (V, E') that is a cliquen graph. For this, each possible edge $(u, v) \in V \times V$ is represented by a variable $x_{u,v}$ with $x_{u,v} \in \{0, 1\}$.

[1] https://gitlab.cs.uni-duesseldorf.de/hoeckesfeld/abstractlanguage

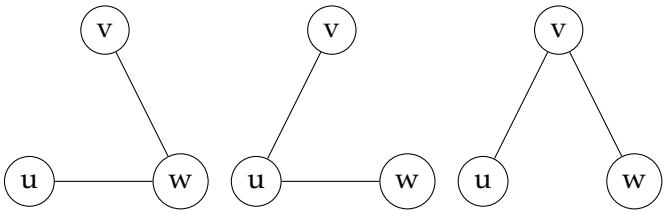

Figure 6: Conflict triplets
This figure shows all possible conflict triplets for nodes u, v, w.

This variable $x_{u,v}$ reflects whether edge (u, v) occurs in the modified graph:

$$x_{u,v} = \begin{cases} 1, & \text{if } (u,v) \in E' \\ 0, & \text{otherwise} \end{cases}$$

Similar to variable $x_{u,v}$, we use predicate $EDGE(u, v)$ to represent edges in the modified graph. The predicate $EDGE$ has the setting "not ordered" so that $EDGE(u, v)$ and $EDGE(v, u)$ represent the same edge.

To ensure that only non-adjacent cliques can be created, so-called conflict triples are forbidden. A conflict triplet consists of three nodes $(u, v, w) \in V$ with exactly two edges between them (see Figure 6). The paper "Going weighted: Parameterized algorithms for cluster editing" [16] shows that no conflict triplets exist if and only if the graph is a cliquen graph.

The ILP formulation from [6] has a rule that prohibits these conflict triples for all nodes $\{u, v, w\} \in \binom{V}{3}$:

$$\begin{aligned} x_{u,v} + x_{u,w} - x_{v,w} &\leq 1 \\ x_{u,v} - x_{u,w} + x_{v,w} &\leq 1 \\ -x_{u,v} + x_{u,w} + x_{v,w} &\leq 1 \end{aligned}$$

The first condition states that the edge (u, v) and the edge (u, w) imply that the edge (v, w) cannot exist. The other two conditions do the same for other cases. This can be directly translated to the abstract language:

$$\begin{aligned} \forall u, v, w \in \binom{V}{3}: \\ & EDGE(u,v) \wedge EDGE(u,w) \Rightarrow EDGE(v,w) \\ & EDGE(u,v) \wedge EDGE(v,w) \Rightarrow EDGE(u,w) \\ & EDGE(u,w) \wedge EDGE(v,w) \Rightarrow EDGE(u,v) \end{aligned}$$

With these constraints, both formulations produce *any* cliquen graph with $|V|$ nodes. To find a cliquen graph with minimal modification cost, the ILP formulation uses the

optimization function:

$$minimize \sum_{e \in E} c(e) - \sum_{(u,v) \in \binom{V}{2}} x_{uv} c(e)$$

The function $c(e)$ models the modification cost of edge e. In the CE problem it is defined as follows:

$$c(e) = \begin{cases} 1, & \text{if } (u,v) \in E \wedge (u,v) \notin E' \\ -1, & \text{if } (u,v) \notin E \wedge (u,v) \in E' \\ 0, & \text{otherwise} \end{cases}$$

Alternatively, you could build an ILP function that counts the modifications directly. For this x_{uv} is added if the edge (u, v) is not in E. However, if $(u, v) \in E$, the modification $1 - x_{uv}$ is added:

$$minimize \sum_{(u,v) \in \binom{V}{2}} \text{modification}(u,v)$$

with the function modification:

$$\text{modification}(u,v) = \begin{cases} x_{uv}, & \text{if } (u,v) \in E \\ 1 - x_{uv}, & \text{otherwise} \end{cases}$$

With this objective function, it is easier to create an optimization function with the given abstract language tools. An optimization function for the abstract formulation looks similar:

$$minimize \sum_{(u,v) \in \binom{V}{2}} \text{ConstantNegateIf}(inE(u,v), EDGE(u,v))$$

The function $inE(u, v)$ returns true when $(u, v) \in E$. This completes the abstract formulation. After conversion to ILP or SAT, it can yield the same optimal results as the original ILP formulation. To create such a formulation, the preceding constraints must be written in Python. The code in Figure 7 shows the implementation of the formulation.

Another idea is the maximization of not modified edges. For this, the opposite of the minimization is maximized:

$$((n * n) - n)/2 - maximize \sum_{(u,v) \in \binom{V}{2}} \overline{\text{ConstantNegateIf}(inE(u,v), EDGE(u,v))}$$

where $((n * n) - n)/2$ is the number of all possible edges. Counting performs better on lower targets, so maximizing this is likely to take more time.

```
import abstractlanguage as abl

n, u, v, w = abl.symbols('n, u, v, w')
V = abl.SymbolicSet("V")                    # Nodes in the graph
E = abl.SymbolicSet("E")                    # Edges in the graph
# Edges in the solution graph:
EDGE = abl.Predicate('EDGE', ordered=False)

def edge_in_E(node1, node2, Edges):
    return (node1,node2) in Edges or (node2, node1) in Edges
had_edge = abl.BooleanPythonFunction("hadedge",edge_in_E)


f = abl.Formulation("MinimizeCE")
f.set_optimization_function(
    abl.Minimize(
        abl.Counter(
            abl.ConstantNegateIf(had_edge(u,v, E), EDGE(u,v)),
            abl.Combinations(V,2),
            ((n*n)-n)/2                    # number of combinations
            )
        )
    )

uv = EDGE(u,v)
uw = EDGE(u,w)
vw = EDGE(v,w)
f.add(
    abl.ForAll(u,v,w,abl.Combinations(V,3),
        ((uv & uw) >> vw)
        & ((uv & vw) >> uw)
        & ((uw & vw) >> uv)
        )
    )
f_cnf = abl.CNFFormulation(f)
f_ilp = abl.ILPFormulation(f)
```

Figure 7: CE Python formulation
Python implementation of the CE formulation.

5 Results

The following section describes the tests and results of the abstract language tool. A Cluster Editing formulation is run with test instances with both ILP and SAT solvers. Additionally, we compare the results with a manually created solver and lastly test the formulation creation and its postprocessing.

5.1 Cluster Editing

A binomial graph $G_{n,p}$ is a random graph with n nodes built with the algorithm from [17]. Each edge is randomly chosen with probability p from all possible edges. Instances generated with this method can be used as Cluster Editing instances. Running the proposed ILP CE formulation with a random graph $G_{50,0.9}$ yielded an editing cost of 129. The Gurobi solver took about 4.011 seconds on a high-performance computer with 10 cores used. Another run with a binomial graph $G_{200,0.9}$ finished with the editing cost of 2033 and took 93 seconds.

The quality of the solution is ensured by a script that checks the correctness of the solution. It checks whether all nodes in the modified graph are in a clique and whether the cliques are not connected by edges.

5.1.1 A CE Solver Testrun

The bachelor thesis by Martin Klümpen [6] describes an ILP formulation for CE. The ILP is reimplemented in Python because the original method is embedded in another program. We can use this as a reference to prove the optimality of the solutions, assuming that the implementation is correct. The manual ILP run with the previously described instance $G_{50,0.9}$ returned the same results. It took about 3.5 seconds to complete. Comparing the LP files of both methods reveals that they use the same amount of constraints. Except for the different variable naming, the structure of the clauses for instance $G_{50,0.9}$ is also the same. In summary, the CE formulation can be written with the abstract language without sacrificing performance.

Additionally, we tested the abstract language CE formulation with the SAT solver CryptoMiniSat5. We chose the linear search algorithm, without preprocessing and without the successive run option. The upper bound of the run was set to 201. The process took about 348 *minutes* to process the $G_{50,0.9}$ instance on the high-performance setup.

The instance $G_{200,0.9}$ even ran out of memory. This can be explained with the counter block. The ILP method uses n^2 variables, to sum all possible edges in the modified graph. For $n = 200$ this is about 40.000 variables.

The CNF counting must additionally count these variables in a two-dimensional matrix. For all n^2 variables we need $target$ many counting variables. The variable $target$ describes the number of allowed edge modifications which can be much larger than n. With the larger example, the ILP found the optimal target at 2033 modifications. In the best case, where the SAT solver immediately picks the correct target, $(200^2)*2033 = 81.320.000$

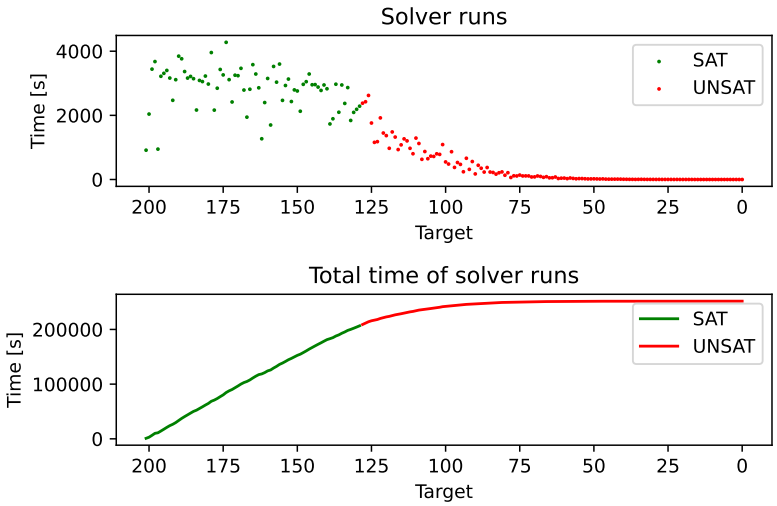

Figure 8: Abstract Language test with the CE problem
Computation of all possible targets between 300 and 0. Satisfiable targets mostly compute fast. Unsatisfiable targets, on the other hand, take longer to compute closer to the optimal target of 129.

variables must be solved. The amount of variables is not the only indicator of the complexity of the problem. Clause length and clause count also play a role, but the variable amount indicates that SAT solver may only compete with ILP if the target is known to be small.

Figure 8 shows all SAT runs for all possible targets. It deviates somewhat from what is expected from SAT solvers. In standard runs, satisfiable formulations solve much faster than unsatisfiable formulations. Here we observe that unsatisfiable runs performed quicker than satisfiable solutions.

Still, for the smaller instance $G_{50,0.9}$, the SAT solver found an optimal solution with the same modification cost as its ILP counterpart.

5.1.2 Comparison To The Conventional Way

We created a CE formulation script in Python by manually converting the constraints to CNF and writing a DIMACS parser. It was a challenge to implement the variable matrix $EDGE(u, v)$ into the counting system. The counting system is not optimized and uses a similar implementation to the method used in the abstract language.

The method took 123 minutes to solve the CE instance. This is a lot faster than the 348 minutes from the abstract formulation. Interestingly, the unsatisfiable runs seen in Figure 9 take similar time to compute than the abstract formulation (see Figure 8). Although opposed to the abstract formulation, the satisfiable runs were computed much faster than the unsatisfiable ones. Both methods return valid results and both methods use a similar counting approach. Either the counting is implemented more efficiently or the order

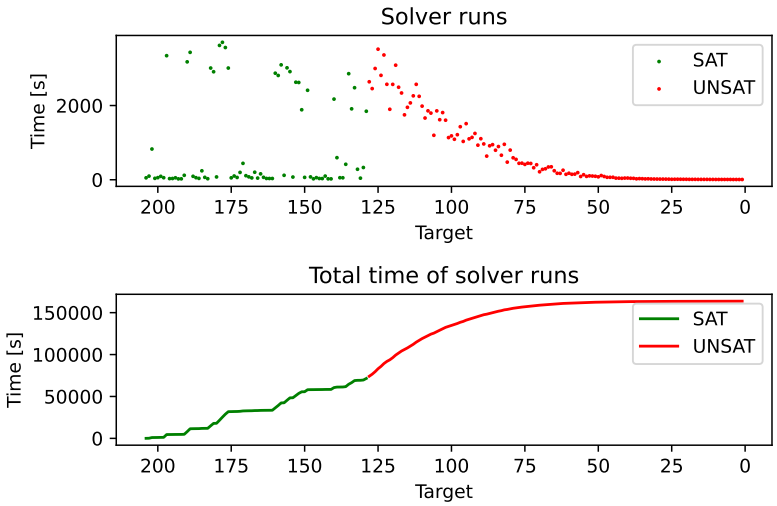

Figure 9: Manual formulation test with the CE problem
Computation of all possible targets between 300 and 0 with a manually created script.

	$G_{10,0.9}$	$G_{40,0.9}$	$G_{80,0.9}$	$G_{400,0.9}$
manual ILP	0.01	1.37	4.02	970.88
abstract ILP	0.06	1.45	3.35	911.02
manual SAT	0.47	670.19	31319.76	-
abstract SAT	0.68	310.57	10911.63	-

Table 3: Runs on random graphs
This table shows the different runtimes (in seconds) for random graphs. The runs are rounded to the nearest hundredth of a second. The upper bound for the linear search is set to $t + 10$, where t is the number of minimal modifications. This means both methods ran the SAT solver 11 times to compute the optimal solution.

in naming the DIMACS variables is more important than expected. The manual implementation uses the basic counting functionality with $1.504.301$ variables and $4.564.352$ clauses. The abstract formulation on the other hand uses just 308.107 variables and 971.586 clauses. The main difference between the two is that the abstract formulation uses as few clauses as possible. This might be more compact but is harder to solve.

As we can see in Figure 9, the time-intensive computations are the targets close to the optimal modifications k. By setting an upper bound of possible modifications to $k + 10$, we can compare the different methods for these time-intensive computations. Table 3 shows that the manual ILP formulation and the abstract language ILP formulation take similar time to compute these instances. But, the SAT formulation from the abstract language was faster than the manually created SAT formulation in all instances. This shows that the SAT formulation constructed by the abstract language is faster if a good upper bound is found. In the initial instance $G_{50,0.9}$ (see Figure 8), the upper bound was set to 201 modifications, which is far apart from the optimal modification cost of 129.

Until now, everything is tested on random graphs. A more realistic instance is a modified

	3 modifications	40 mod.	160 mod.	500 mod.
manual ILP	1.8	1.81	1.76	258.91
abstract ILP	3.06	1.61	1.47	261.85
manual SAT	297.59	467.15	7091.05	-
abstract SAT	9.19	111.42	5270.23	-

Table 4: Runs on modified cliquen graphs
This table shows the different runtimes (in seconds) for a cliquen graph with random edge modifications. The described cliquen graph consists of 4 cliques with different sizes. In total, the graph has 60 nodes. The runs are rounded to the nearest hundredth of a second. The upper bound for the linear search is set to $t + 10$, where t is the number of minimal modifications. This means both methods ran the SAT solver 11 times to compute the optimal solution.

cliquen graph. Here, we create a cliquen graph by joining different cliques. Then we modify random edges in the cliquen graph to create a modified cliquen graph. Table 4 shows runs with the different methods on a cliquen graph with varying modifications. The upper bound for the linear search is set to $t + 10$, where t is the optimal modification cost. In Table 3 you can see that the runtime is higher when you increase the number of nodes. With the modified cliquen graph (Table 4) you can now see that the runtime is also higher when there are more modifications.

5.1.3 Formulation Creation

To test the performance of the DIMACS conversion, we compare a formulation written with the abstract language against a manually coded script. If created manually, the formulation script is in general neglectable for large instances. Solving the formulation is considerably more time-consuming. Only in small instances, the formulation creation time influences the whole process. To show that the formulation creation time of the abstract language tool is reasonable, we compare the time to a manually created formulation script. We create formulations for the Max Clique problem using our methods, as well as with a manually created script. For each possible target, one formulation per procedure is generated and monitored. The manually created script is not optimized and creates the entire counter matrix for each iteration. The DIMACS representations are computed using a random graph $G_{200,0.9}$.

The results show that the standard abstract language DIMACS creation is slower than the other methods. But, this does not significantly change the whole solving process. As Figure 10 shows, the successive runs method performs a lot better than the manual approach. The manual approach needs a constant creation time because it always computes the entire counter-matrix. If one would increase the number of nodes, the time ratio of the different methods would look similar. It is a misconception that the standard run increases exponentially against infinity. By increasing the number of nodes, all three methods take more time.

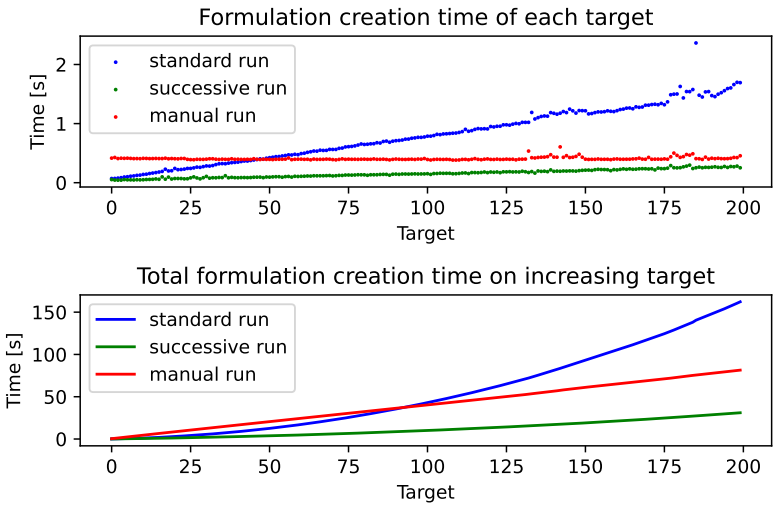

Figure 10: Formulation creation times with the Max Clique problem
The figure shows the formulation creation times of a simple script, as well as a standard and a successive run with the abstract language. Each dot in the upper graph is a formulation creation. The lower graph shows the time used until the target x.

5.2 Postprocessing

The idea of postprocessing is to eliminate simple logic conclusions and to simplify the debugging of DIMACS files directly. This is implemented as intended and works for small debuggable files. But it would also be interesting to see if the overall running time could decrease with this. To test this, we run a formulation with and without preprocessing. Given a formulation f with predicate A, a variable n and a constraint:

$$\forall i \in Range(1, n) \colon A(i) \Rightarrow A(i+1)$$

After adding constraint $A(1)$, the problem is entirely solved by the postprocessing if enabled. When $A(1)$ is true and is removed from $A(1) \Rightarrow A(2)$, A(2) becomes true. The postprocessor clears all constraints iteratively without using the solver. With the postprocessor, it took 7.14 seconds with a personal computer until it reported a solution. By disabling the postprocessor, we achieved the same with a CPU time of about 3.57 seconds. In this case, CryptoMiniSat5 was used, which probably uses a DIMACS preprocessing method. This method outperforms our postprocessing algorithm. With this observation, it is still possible that the solver guessed that all variables are true. Randomly replacing applied predicates with their negation changes the solution:

$$\begin{aligned}\forall i \in Range(1, n) \colon \\ &\text{ConstantNegateIf}(randBool(i), A(i)) \\ &\Rightarrow \text{ConstantNegateIf}(randBool(i+1), A(i+1))\end{aligned}$$

This degrades the postprocessing to about 12.87 CPU seconds, while without this feature it took 4.88 seconds. CryptoMiniSat5 is better optimized and written in the considerably faster C++ language. This shows that the postprocessing method only has a use for debugging and understanding DIMACS formulations.

6 Conclusions

The abstract language is a versatile and simple tool to create formulations. With its LaTeX representation, the formulation can be modelled directly in Python. The implementation phase is done automatically, which makes inventing and testing new formulations faster. Gusfield, Brown and Zuo showed in their paper [1] that sometimes SAT solving is faster than ILP solving. With the abstract language, the additional effort to try SAT is minimal because converting to SAT and ILP is done automatically.

The abstract language framework supports a wide range of ILP and SAT solvers because it uses the standard LP and DIMACS format. This has the advantage that different solvers can be compared without additional development. ILP and SAT solvers can be compared on their performance but also a SAT solver with another SAT solver. Additionally, the search method for SAT formulations can be chosen. With this, binary search can be compared to linear search without effort.

Debugging a DIMACS file is much simpler with features like postprocessing. This enables the formulation developer to focus on the core problem. The formulation creation time is slower with our process, but the succeeding runs option can be faster on a large scale.

The DIMACS files produced by our program have a similar structure to manually created files and share the same solutions. But the manually developed files are solved faster. In the future, one could investigate DIMACS variable naming. Maybe, variables which occur together in a formulation should have integer identifiers close to each other. Also, only the Tseitin transformation was implemented, other research improved upon this. The Plaisted-Greenbaum encoding [18] for example requires fewer constraints for the Tseitin variables.

The building blocks also simplify the development process. Designing and implementing these building blocks is research in itself and abstracting them is a good idea. Additions like the order function enable the counting of predicates over any sets, which decreases development time. The counter block has other possible implementations that would be a helpful addition to the abstract language. Each one has an advantage in certain areas and the developer could choose an option.

In the framework, building blocks are very restrictive. They cannot be combined with other logic or building blocks. For example, a summation block would add multiple counter blocks together. With this, the counting of two different predicates P and Q could be possible (S and S' are arbitrary sets):

$$\sum_{i \in S} P(i) + \sum_{j \in S'} Q(j)$$

The OneHot encoding also has limitations. It can only quantify over one index. A predicate $M(i, j)$ that models a matrix, cannot require exacly one $M(i, j)$ $\forall i, j$ is true.

The ILP conversion of the framework is not as efficient as possible. The XOR gate and other special operations are converted to NNF before being translated to ILP. Implementing each operation manually could improve the conversion step.

Using the existence quantifier $\exists$ in the abstract language is misleading. The idea was to

mimic predicate logic as much as possible. But the quantifier is restricted to a special case and could be replaced by an OR quantifier $\bigvee$.

To summarize, the abstract language and its attached converter is a good tool to create ILP and SAT formulations. Results show that some work remains to make it fully competitive with a manual conversion.

In future work, one could add more solvers and other building blocks. In related work, modelling languages also use constraint programming (CP) solvers. The conversion to the CP format could be added to the abstract language framework. Additionally, adding better conversions like the Plaisted-Greenbaum encoding [18] could improve the quality. The differences between SAT solving with and without the abstract language should be investigated in the future. The impact of the DIMACS variable naming and the implementation of the counting block must be checked.

References

[1] Hannah Brown, Lei Zuo, and Dan Gusfield. "Comparing Integer Linear Programming to SAT-Solving for Hard Problems in Computational and Systems Biology". In: *International Conference on Algorithms for Computational Biology*. Springer. 2020, pp. 63–76.

[2] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2022.

[3] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT Solvers to Cryptographic Problems". In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*. Ed. by Oliver Kullmann. Vol. 5584. Lecture Notes in Computer Science. Springer, 2009, pp. 244–257.

[4] Grigori S Tseitin. "On the complexity of derivation in propositional calculus". In: *Automation of reasoning*. Springer, 1983, pp. 466–483.

[5] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, 2009.

[6] Martin Klümpen. "Cluster Editing mit ganzzahliger linearer Programmierung: eine Open-Source-Lösung". In: ().

[7] Stuart Mitchell, Michael OSullivan, and Iain Dunning. "PuLP: a linear programming toolkit for python". In: *The University of Auckland, Auckland, New Zealand* 65 (2011).

[8] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[9] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. "Modeling and programming with gecode". In: *Schulte, Christian and Tack, Guido and Lagerkvist, Mikael* 1 (2010).

[10] Robert Fourer, David M Gay, and Brian W Kernighan. *AMPL: A mathematical programming language*. AT & T Bell Laboratories Murray Hill, NJ, 1987.

[11] Nicholas Nethercote et al. "MiniZinc: Towards a standard CP modelling language". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 529–543.

[12] Thomas Kluyver et al. "Jupyter Notebooks – a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. Ed. by F. Loizides and B. Schmidt. IOS Press. 2016, pp. 87–90.

[13] Aaron Meurer et al. "SymPy: symbolic computing in Python". In: *PeerJ Computer Science* 3 (Jan. 2017), e103.

[14] Magnus Björk. "Successful SAT encoding techniques". In: *Journal on Satisfiability, Boolean Modeling and Computation* 7.4 (2011), pp. 189–201.

[15] Gerald G Brown and Robert F Dell. "Formulating integer linear programs: A rogues' gallery". In: *INFORMS Transactions on Education* 7.2 (2007), pp. 153–159.

[16] Sebastian Böcker et al. “Going weighted: Parameterized algorithms for cluster editing”. In: *Theoretical Computer Science* 410.52 (2009), pp. 5467–5480.

[17] Erds Paul and Rényi Alfréd. “On random graphs I”. In: *Publicationes Mathematicae (Debrecen)* 6 (1959), pp. 290–297.

[18] David A Plaisted and Steven Greenbaum. “A structure-preserving clause form translation”. In: *Journal of Symbolic Computation* 2.3 (1986), pp. 293–304.

List of Figures

List of Tables