Karlsruher Institut für Technologie

Bachelorthesis

# Recognition of Constraints in CNF Formulas

Robin Link

Date: 22. Oktober 2020

Reviewer: Prof. Dr. Carsten Sinz
Second Reviewer: Prof. Dr. Peter Sanders
Supervisor: Dr. Markus Iser

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

# Abstract

The tractability of the NP-complete satisfiability problem (SAT) can be greatly improved by algorithms that exploit knowledge about specific instances [7]. The observation that many SAT instances arise from encoded Boolean circuits motivated the reconstruction of such circuits from formulas, as effective algorithms are known for these instances [7]. Recent research [13] devised algorithms to efficiently recover gate structure from CNF formulas, including from encodings that have been simplified to reduce the number of clauses [18, 15]. Based on reconstructed Boolean circuits we attempt to recognize encoded cardinality constraints in CNF formulas using a graph algorithm in conjunction with a search algorithm for clauses in the constraint that are not distributed over the reconstructed gates. Furthermore we show that the gate recognition algorithm by Iser [13] is not confluent. We suggest a method to guide the reconstruction in a controlled manner such that graph based algorithms become feasible.

As a framework for this approach, we supplement the existing algorithms in Candy [1] with an encoder for cardinality constraints and implement an exporter to a common graph-visualizer [11].

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

# Contents

**Abstract** iii

**1 Introduction** 1
1.1 Motivation . . . 1
1.2 Structure of Thesis and Contribution . . . 2

**2 Fundamentals** 3
2.1 General Definitions . . . 3

**3 Sequential Counter Constraints** 7
3.1 Encoder . . . 9

**4 Related Work** 11
4.1 Encodings of Constraints and other Bitvector Operations . . . 11
4.2 Gate Recognition Procedures . . . 12
4.3 Subcircuit Recognition . . . 12

**5 Gate Recognition, Classification and Visualization** 13
5.1 Gate Recognition . . . 13
5.2 Classification . . . 19
5.3 Graph Representation of Boolean Circuits . . . 19
5.4 Visualization . . . 20

**6 Unification** 23
6.1 Formal Definition of Syntactical Unification . . . 23
6.2 Algorithms . . . 26
6.2.1 Syntactical Unification . . . 27
6.2.2 Syntactical Classification . . . 31

**7 Sequential Counter Recognition** 33
7.1 Preliminaries . . . 33
7.1.1 Soundness of Recognition Algorithms . . . 33
7.1.2 Data Structures and Auxiliary Algorithms . . . 33
7.1.3 The General Subcircuit . . . 34

7.2 Algorithms . . . . . 34
7.2.1 Graph-Based Recognition . . . . . 34
7.2.2 Syntactical Recognition . . . . . 42

**8 Implementation and Evaluation** **45**
8.1 Experimental Setup and Problem Instances . . . . . 45

**9 Discussion, Future Work and Conclusion** **49**
9.1 Discussion and Future Work . . . . . 49
9.2 Conclusion . . . . . 50

**Bibliography** **51**

# 1 Introduction

## 1.1 Motivation

The NP-complete satisfiability (SAT) problem has long been thought to be too impractical for practical applications [9]. However more recent developments have incrementally raised the size of tractable formulas. Even though some problems of (relatively) small size remain hard, a large number of cases arising from practical instances became solvable under real world circumstances. Key developments were the conflict driven clause learning algorithm, and the exploitation of structures in a formula that often allow heuristics to satisfy an instance. The latter approach can be further improved by choosing appropriate encodings, not necessarily in CNF, of the problem. For an in-depth history of the developments in satisfiability theory see [7].

Satisfiability modulo theories (SMT) and pseudo-Boolean (PB) solvers are examples that extend a CNF formula with more abstract terms in formulas [7]. PB solvers for example allow the direct representation of cardinality constraints and can possibly exploit these using cutting-plane algorithms to shrink the solution space, SAT4J [10] being a notable example.

Iser developed an algorithm to efficiently recover gate structure in CNF formulas [13]. This is of particular interest for two reasons. Firstly, a considerable amount of SAT-problems in practice arise from encoded gate structures. Secondly, even if the formula does not stem from an encoded gate structure, it may still encode one. This opens the possibility of using graph algorithms to gain insight into CNF formulas to possibly further improve the time needed to solve an instance by extracting information and employing non-pure SAT solvers.

We narrow down the wide range of structures to recognize. Of particular interest are structures that stem from encoded circuits since a more complete reconstruction back into gates can be expected. Additionally, the extraction should promise useful applications, for example by re-encoding the problem using the additional tools provided by SMT or PB solvers. For those reasons we choose Sinz' encoding [17] of at-most constraints as the principle subcircuit to recognize in this thesis.

## 1.2 Structure of Thesis and Contribution

We begin by revealing the graph structure in the gates found by the hierarchical gate recognition implemented in Candy [1] and export the graph to the dot format that serves as the input of Graphviz [11] to render images of Boolean circuits.

As a second step we implement an encoder for the sequential counter constraint by Sinz [17]. Our implementation takes a formula in the DIMACS format [3] as an input and gives an encoded cardinality constraint of the desired variables that can be added to the original formula.

Thirdly, we implement a gate classifier. As the hierarchical gate recognition algorithm is generic [13], i.e. it extracts clauses into gates, without explicit knowledge about the function encoded by those clauses, it is desirable to reveal the hidden function in the gates. We do this in two ways. The simple approach is using the properties known about the clauses in a gate and use this knowledge to match them to a set of predefined cases. The more elaborate way to classify gates is by unification. As the recognition of constraints cannot be done solely via gates, as we will see later, unification is necessary for a complete recognition. Incidentally, this also provides the basis for a different classification algorithm that can be easily extended by giving patterns of functions.

Lastly, we suggest an algorithm to recognize sequential counter constraints and explain why this algorithm can only be expected to work under special circumstances, as it assumes a fixed reconstruction. However, we prove that the gate recognition algorithm employed is not confluent. This questions the general applicability of the presented algorithm. To devise a recognition algorithm nonetheless, we suggest a unification based recognizer and show how to extend the hierarchical gate recognition algorithm by Iser [13] to complete the recognition on the graph.

# 2 Fundamentals

## 2.1 General Definitions

At the center of this thesis are Boolean functions and circuits. We begin by giving central definitions of these concepts, terms related to these, and introduce formalities to connect them. As this thesis largely depends on gate recognition and requires similar terminology, we hold the definitions (mostly) coherent with the ones in [13].

**Definition 2.1.1.** *Over the Boolean domain $\mathbb{B} := \{0, 1\}$ we define the following sets.*

- *Let $\Phi_k := \{\phi : \mathbb{B}^k \to \mathbb{B} \mid k \in \mathbb{N}\}$ be the set of $k$-ary Boolean functions.*
- *Let $\Phi := \bigcup_{i=1}^{\infty} \Phi_i$ be the set of all finite Boolean functions.*
- *Call a finite subset $\kappa$ of $\Phi$ a set of connectives.*
- *Any element of a countably infinite set $\mathbb{X}$ can serve as a variable symbol. Since for every such set $\mathbb{X} \cong \mathbb{N}$, we can use without restriction $\mathbb{X} = \{x_i \mid i \in \mathbb{N}\}$ as the set of variable symbols. However more suitable namings may be employed.*
- *The set of $k$ variables over which a function $F \in \Phi_k$ is defined, is denoted by $\mathbb{X}_F \subset \mathbb{X}$.*
- *The set of variables actually occurring in a formula is denoted by $vars(F) \subseteq \mathbb{X}_F$.*
- *A literal is either the variable itself, or the negation of the variable.*
- *The set of literals actually occurring in $F$ is denoted by $lits(F)$.*

Let $x$ be a variable. Continuing from here on, we call literals $\neg x$ *signed* or *negative*, and $x$ *unsigned* or *positive*. The *norm* $|\cdot|$ maps the negative literal $\neg x$ to $x$ and a positive literal to $x$. The complement of a literal is the negated literal. For a literal $l$ we denote the complement by $\bar{l}$.

Now we can start introducing terms fundamental to logic.

**Definition 2.1.2.** *(Interpretation Functions and Models). Let $F \in \Phi$ be a formula and let $\alpha : \mathbb{X}_F \to \mathbb{B}$ be a function. We call $\alpha$ a variable assignment. Furthermore, we define $M_\alpha$ as the set of all literals satisfied under $\alpha$, i.e. $M_\alpha := \{l \mid l \in lits(F) \text{ and } \mathcal{I}_\alpha(l) = \top\}$. We call $\mathcal{I}_\alpha : \Phi \to \{\top, \bot\}$ the interpretation function.*

*The assignment $\alpha$ is a model for $F$, iff $\mathcal{I}_\alpha(F) = \top$, in this case we write $M_\alpha \models F$. Lastly, we define the set of all models of $F$ by $\mathcal{M} := \{M_\alpha \mid \mathcal{I}_\alpha = \top\}$.*

**Definition 2.1.3.** *(Logical Equivalence). Let $F$ and $G$ be two formulas in $\Phi$. Iff $F$ has the same models as $G$, thus $\mathcal{M}(F) = \mathcal{M}(G)$. We write $F \equiv G$ and call $F$ logically equivalent to $G$.*

In most examples we are concerned with formulas containing additional encoding variables, the sequential counter encoding being the most relevant example in this thesis. Motivated by this we define the following two terms.

**Definition 2.1.4.** *(Projection of Models). Let $F \in \Phi$ be a formula over the variables $\mathbb{X}_F$. For a non-empty subset of variables $X \subseteq \mathbb{X}_F$ we define the projection of models of $F$ to $X$ by*

$$\mathcal{M}_X(F) := \{M \cap lits(X) \mid M \in \mathcal{M}(F)\}.$$

**Definition 2.1.5.** *(Equivalence under Projection). Let $F, G \in \Phi$ be two formulas and let $I = \mathbb{X}_F \cap \mathbb{X}_G$ be non-empty. We call these formulas equivalent under projection, iff $\mathcal{M}_I(F) = \mathcal{M}_I(G)$, we denote this with $F \cong_I G$.*

We remark that equivalence under projection is a generalization of logical equivalence, as the following shows.

$$\text{If } \mathbb{X}_F = \mathbb{X}_G = I = \mathbb{X}_F \cap \mathbb{X}_G, \text{ and } F \cong_I G \text{ then } F \equiv G.$$

We apply these definitions in the following example.

**Example 2.1.6.** *(At Most One Constraint and Projections). The models of the formula $F = (x_1 \vee \neg a) \wedge (\neg x_2 \vee \neg a)$ can be constructed as follows. If $a = \bot$ then every assignment of $x_1, x_2$ satisfies the formula. If $a = \top$ then the formula is equivalent to $x_1 \wedge \neg x_2$. This formula is satisfied if at most one of the variables is assigned to $\top$. Now the projection $\mathcal{M}_{\{x_1,x_2\}}(F)$ can be interpreted as a constraint which is satisfied iff no more than one of the variables $x_1, x_2$ is satisfied. The assignment of $a$ is ignored in the interpretation of the function under this projection. I.e. this variable contributes only to the construction of the formula, but not to its interpretation.*

Next, we give a formal definition of gates and how they are connected to Boolean formulas.

**Definition 2.1.7.** *(Gates). Call a tuple $G = (o, P, g)$ a gate over a finite, non-empty set of variables $\mathbb{X}_G \subset \mathbb{X}$, iff $\mathbb{X}_G = \{o\} \cup P$, and call $g : \mathbb{B}^{|P|} \to \mathbb{B}$ the characteristic function of the gate.*

**Definition 2.1.8.** *(Gate Semantics). Let $G = (o, P, g)$ be a gate with $P = (p_1, \ldots, p_n)$ and $\alpha : \mathbb{X}_G \to \mathbb{B}$ a variable assignment. Iff the interpreting function $\mathring{\mathcal{I}}_\alpha = \top$, then $\alpha$ is a model of the gate. The interpreting function is defined as follows*

$$\mathring{\mathcal{I}}_\alpha := \begin{cases} \top, \text{ if } \alpha(o) = g(\alpha(p_1), \ldots, \alpha(p_n)) \\ \bot, \text{ otherwise} \end{cases}.$$

In this thesis it is of primary interest how gate structure can be exploited. Later, the following problem will be revealed: not all formulas of interest are given in a form that can currently be described solely in terms of gates; i.e supplementing gate structures with Boolean formulas is necessary. We consolidate both concepts in a single definition to provide suitable language for the coming algorithms.

The mentioned problem can be observed for example in 14 by comparing the clauses given by the sequential counter encoding in 3 with the clauses found in the gates.

**Definition 2.1.9.** *(Structural Formulas). Let $F$ be a finite set of clauses and $\Gamma$ a finite set of gates. We call $(F, \Gamma)$ a structural formula over the variables $\mathbb{X}_S := \mathbb{X}_F \cup \{v \mid G \in \Gamma$ and $v \in \mathbb{X}_G\}$.*

**Definition 2.1.10.** *(Structural Formula Semantics). An assignment $\alpha : \mathbb{X}_S \to \mathbb{B}$ is a model of $S$ iff $\mathcal{I}_\alpha(F) = \top$ and $\forall G \in \Gamma : \mathring{\mathbb{X}}_\alpha = \top$.*

**Definition 2.1.11.** *(Gate Encodings). A formula $F$ encodes a gate $G$ iff $\mathbb{X}_F = \mathbb{X}_G$ and $(F, \emptyset) \cong (\emptyset, \{G\})$.*

## Notation

Commonly, SAT solvers take Boolean formulas in conjunctive normal form (CNF). We define this form and propositional formulas formally and introduce the CNF set notation.

**Example 2.1.12.** *(Propositional Formulas, CNF and DNF). It is common to construct formulas over a fixed set of connectives. By requiring $\kappa = \{\wedge, \vee, \to, \leftrightarrow, \neg\}$ we obtain propositional formulas. Alternatively, we can require $\kappa = \{\wedge, \vee, \neg\}$ and demand the formula to be $(a)$ a conjunction of disjunctions or $(b)$ a disjunction of conjunctions. In case $(a)$ we get the conjunctive normal form (CNF), in case $(b)$ the disjunctive normal form (DNF).*
*For propositional formulas we write ${}^{PROP}F$ and for formulas in CNF we write ${}^{CNF}F$. The latter are of particular interest in this thesis.*

For many applications it is useful to rewrite a formula in CNF to a set, and use set-theoretical operations.

**Definition 2.1.13.** *(CNF Set Notation). Let $F = \bigwedge_{i=1}^{n} D_i$ be a conjunction of disjunctions $D_i = \bigvee_{j=1}^{m_i} l_j$ with literals $l_j$. The corresponding set representation is then defined as*

$$\Big\{ \underbrace{\{l_1, \ldots, l_{m_1}\}}_{D_1}, \ldots, \underbrace{\{l_n, \ldots, l_{m_n}\}}_{D_n} \Big\} = F.$$

*Although less precise, we choose to write the last equality to simplify notation.*

# 3 Sequential Counter Constraints

When modeling problems as SAT instances cardinality constraints are frequently employed.

**Definition 3.0.1.** *(Cardinality Constraint). Let* $\circ \in \{<, \leq, =, \geq, >\}$ *be a relation. Then we call* $\circ k(x_1, \ldots, x_n)$ *a cardinality constraint on* $x_1, \ldots, x_n$*. Let there be further a variable assignment* $\alpha$*, then this constraint evaluates to true iff* $\sum_{i=1}^{n} \alpha(x_i) \circ k$*.*

To suggest the power of cardinality constraint, we formulate the well-known coloring problem solely in terms of cardinality constraints. This construction could also serve as a benchmark test for the number of constraints recognized and the possible performance improvement by different problem encodings.

**Example 3.0.2.** *(Vertex Coloring and Constraints). Given a graph* $G$ *with vertices* $V$ *and edges* $E \subseteq \{(v_i, v_j) \mid v_i, v_j \in V, v_i \neq v_j\}$ *we can define the* $k$*-coloring problem on* $G$ *as follows. Any vertex has to have a color assigned to it by* $c : V \to \{1, \ldots, k\}$*, such that for every edge* $(v_i, v_j) \in E$ *it holds that* $c(v_i) \neq c(v_j)$*.*

*This problem can be encoded in a formula by introducing variables* $c_{i,j}$ *which indicate whether vertex* $v_i$ *has been assigned color* $j$*. To ensure exactly one color is assigned to each vertex we include the constraints* $=1(c_{i,1}, \ldots, c_{i,k})$ *for every vertex* $v_i$ *. To furthermore ensure that connected vertices have different colors, we include for every edge* $(v_i, v_j) \in E$*, and for every color* $l$*, a constraint* $\leq 1(c_{i,l}, c_{j,l})$*.*

*We now have a formula completely constructed from cardinality constraints, for which every model is a valid* $k$*-coloring of* $G$*.*

In this thesis constraints of the form $\leq k(x_1, \ldots, x_n)$ are of particular interest.

**Lemma 3.0.3.** *Let* $X = \{x_1, \ldots, x_n\}$ *and* $X' \subseteq X$*. Then it holds under any variable assignment that* $\leq k(X) \implies \leq k(X')$*.*

We adapt the definition of clausal encodings from Sinz [17] to our definitions:

**Definition 3.0.4.** *(Clausal Encodings of At Most Constraints). A clause set* $E$ *over the variables* $V = \{x_1, \ldots, x_n, s_1, \ldots, s_m\}$ *is a clausal encoding of* $\leq k(x_1, \ldots, x_n)$*, iff* $\mathcal{M}_{\{x_1, \ldots, x_n\}}(E) = \mathcal{M}(\leq k(x_1, \ldots, x_n))$*.*

Sinz also presents a clausal encoding $LT_{SEQ}^{n,k}$ of $\leq k(x_1, \ldots, x_n)$ in [17] based on sequential counters. We recite the formula:

**Definition 3.0.5.** *(Sequential Counter Encoding of At Most Constraints).*

$$
\begin{aligned}
LT_{SEQ}^{n,k} =& (\neg x_1 \vee s_{1,1}) \\
& (\neg s_{1,j}) && \textit{for } 1 < j \leq k \\
& (\neg x_i \vee s_{i,1})(\neg s_{i-1,1} \vee s_{i,1})(\neg x_i \vee \neg s_{i-1,k}) && \textit{for } 1 < i < n \\
& (\neg x_i \vee \neg s_{i-1,j-1} \vee s_{i,j})(\neg s_{i-1,j} \vee s_{i,j}) && \textit{for } 1 < j \leq k, 1 < i < n \\
& (\neg x_n \vee \neg s_{n-1,k}).
\end{aligned}
$$

The origin of this encoding is a gate structure with its size depending on $n$ and $k$. For formulas constructed in this manner we expect to be able to find a recursive form. In the case of sequential counters this is possible. This form makes it straightforward to prove properties of this encoding. These insights and the recursive form itself will later be central to our constraint recognition algorithm.

**Lemma 3.0.6.** *(Recursive Forms of the Sequential Counter Encoding). The base case is*

$$LT_{SEQ}^{2,1} = (\neg x_1 \vee s_{1,1})(\neg x_2 \vee \neg s_{1,1}).$$

*The recursive forms are*

$$
\begin{aligned}
LT_{SEQ}^{n+1,k} =& LT_{SEQ}^{n,k} \\
& (\neg x_n \vee s_{n,1})(\neg s_{n-1,1} \vee s_{n,1}) \\
& (\neg x_n \vee \neg s_{n-1,j-1} \vee s_{n,j})(\neg s_{n-1,j} \vee s_{n,j}) && \textit{for } 1 < j \leq k \\
& (\neg x_{n+1} \vee \neg s_{n,k})
\end{aligned}
$$

$$
\begin{aligned}
LT_{SEQ}^{n,k+1} =& LT_{SEQ}^{n,k} \\
& (\neg s_{1,k+1}) \\
& (\neg x_i \vee \neg s_{i-1,k} \vee s_{i,k+1})(\neg s_{i-1,k+1} \vee s_{i,k+1}) && \textit{for } 1 < i < n
\end{aligned}
$$

$$
\begin{aligned}
LT_{SEQ}^{n+1,k+1} =& LT_{SEQ}^{n,k} \\
& (\neg x_n \vee s_{n,1})(\neg s_{n-1,1} \vee s_{n,1}) \\
& (\neg s_{1,k+1}) \\
& (\neg x_n \vee \neg s_{n-1,j-1} \vee s_{n,j})(\neg s_{n-1,j} \vee s_{n,j}) && \textit{for } 1 < j \leq k \\
& (\neg x_i \vee \neg s_{i-1,k} \vee s_{i,k+1})(\neg s_{i-1,k+1} \vee s_{i,k+1}) && \textit{for } 1 < i < n
\end{aligned}
$$

It holds that $LT_{SEQ}^{n+1,k+1} \equiv LT_{SEQ}^{n+1,k} LT_{SEQ}^{n,k+1}$, and after removing duplicate clauses the formulas are equal. Similarly to writing $\leq k(x_1, \ldots, x_n)$, we write $LT_{SEQ}^{n,k}(x_1, \ldots, x_n)$ when constraining the variables . It is important to note that $\leq k$ is commutative, while $LT_{SEQ}^{n,k}$ is not; i.e. changing the order in which variables are presented to the sequential counter encoding changes the resulting clauses.

The following observations lead to a requirement on recognition algorithms that must hold, because the result is not useful otherwise. A precise discussion is provided later.

**Lemma 3.0.7.** *Every formula that contains the clauses of a sequential counter encoding of $\leq k(x_1, \ldots, x_n)$ also contains the clauses of a sequential counter encoding of $\leq k'(x_1, \ldots, x_n)$ for every $k' \leq k$.*

*Proof.* Checking the recursive forms above, we see $LT_{SEQ}^{n,k} \subset LT_{SEQ}^{n,k+1}$. □

Similarly, the recursive form also suggests the following lemma.

**Lemma 3.0.8.** *Every formula that contains an encoding of $\leq k(X)$ also contains an encoding of $\leq k(X')$ for any $X' \subsetneq X$.*

*Proof.* We assume otherwise: there is an assignment of $\top$ to more than $k$ variables in $X'$ that doesn't falsify $\leq k(X')$. However, this assignment falsifies $\leq k(X)$. □

We formulate a last definition that encapsulates the desired properties the result of any constraint recognition algorithm should satisfy.

**Definition 3.0.9.** *(Maximality of Constraints). An encoded constraint $\leq k(X')$ is maximal, iff there is no superset $X \supsetneq X'$ for which the formula contains an encoding of $\leq k(X)$. I.e. a maximal set of variables is constrained in a single encoding.*

## 3.1 Encoder

We conclude this chapter with a straight-forward algorithm that encodes $LT_{SEQ}^{n,k}$ in the DIMACS format [3]. Practical usages in mind, our algorithm takes a formula in the DIMACS format as its input and additionally a set of numbers that represent the variables to constrain, and lastly the number $k$ of variables that can be satisfied at most. The result is a string of numbers that can be appended to the formula, as the auxiliary variables use a seamless numbering. I.e. the number corresponding to the first auxiliary variable is one larger than the largest variable occurring in the original formula.

To this end we define $m$ as the number of variables in the formula. Now given $n$, the number of variables constrained, and $k$, the upper bound, we can define the following map. The auxiliary variables are mapped to positive integers by $s_{i,j} \mapsto m + 1 + (i-1) \cdot k + j$.

The string representation of the result of this mapping shall be denoted by $s'_{i,j}$. Similarly the string representation of any variable $x$ shall be denoted by $x'$.

Based on these definitions we give the following algorithm.

**Algorithm 1:** Encode $LT^{n,k}_{SEQ}$

```
Input: Formula F in the DIMACS format, a subset of variables
       V = {x_1, ..., x_n} ⊆ vars(F) and k
Assert n ≥ 2 and k ≥ 1
Start with an empty string encoding
Append "−x'_1  s'_{1,1}  0" to encoding in a new line
for j = 2, ..., k do
    Append "−s'_{1,j}  0" to encoding in a new line
for i = 2, ..., n − 1 do
    Append "−x'_i  s'_{i,1}  0" to encoding in a new line
    Append "−s'_{i−1,1}  s'_{i,1}  0" to encoding in a new line
    for j = 2, ..., k do
        Append "−x'_i  −s'_{i−1,j−1}  s'_{i,j}  0" to encoding in a new line
        Append "−s'_{i−1,j}  s'_{i,j}  0" to encoding in a new line
    Append "−x'_i  −s'_{i−1,k}  0" to encoding in a new line
Append "−x'_n  −s'_{n−1,k}  0" to encoding in a new line
return encoding
```

Here $x_i$ simply is $V[i]$, where the variables in $V$ are ordered arbitrarily but are fixed in place.

# 4 Related Work

The tractability of SAT problems is largely determined by heuristics exploiting known or assumed properties of SAT instances. Various solvers have been developed that allow to express constraints in different ways. For example a pure SAT solver needs a clausal encoding of a constraint, however an SMT solver possibly allows to directly express the constraint as an (in)equality and then cuts the search space by applying a cutting-planes algorithm [10]. This more abstract representation and the exploitation of the knowledge about constraints in the solving algorithms allows SMT-solvers to possibly solve instances more efficiently. Thus, a promising path to further improve the efficiency of solvers is to extract information from formulas that can be exploited by existing algorithms. Conversely, if an efficient and promising extraction of knowledge from formulas has been found, new algorithms may be developed.

Three topics have been the interest of past research: (clausal) encodings of constraints and bitvector operations based on Boolean circuits, gate recognition procedures, and the exploitation of the extracted knowledge. In this chapter we will summarize key developments, relate them to subcircuit recognition, and briefly assess advantages and disadvantages of the presented methods. In the following chapters we discuss concrete examples of these topics and employ them for the described ends.

## 4.1 Encodings of Constraints and other Bitvector Operations

Due to their wide applicability different realizations of constraints and their effect on performance have been evaluated. For example, the sequential counter encoding introduced by Sinz [17] has improved the naive, pairwise encoding of such constraints (at least in the number of clauses required). Other encodings, and strengthenings of those, have been presented and compared, for example, in [20]. We give a subset of encodings of at most constraints:

- Pairwise Encoding,
- Sequential Counter Encoding [17],
- Parallel Counter Encoding [17],
- Pigeon Hole Encoding [20],
- Sort Based Encoding [20],
- Tree-Based Encoding [20],
- Bailleux & Boufkhad [5],
- Warners Encoding [19].

However, it is subject to experimentation whether encoding a constraint or employing an SMT-solver is more efficient, as Abío et al. showed in their paper "To Encode or to Propagate? The Best Choice for each Constraint in SAT" [4].

## 4.2 Gate Recognition Procedures

Developments of gate recognition procedures include the following. Ostrowski et al. suggested a procedure in [14] to recognize AND-, OR-, EQUIV-gates. Roy et al. [16] devised a procedure that additionally recognizes NAND-, NOR-, NOT-, XOR-, XNOR-, and MAJ3-gates. A comparison of these methods shows a fundamental difference in the foundation of the recognition procedures. Ostrowski et al. use the resolution graph representation of SAT instances in addition to the fact that gates appear in blocked sets; in contrast Roy et al. extract gates from a formula using known patterns of gate encodings. Iser presents the most recent approach in [13], which will receive a more in-depth introduction in the following chapters. While this approach shines in genericity, the presented algorithm requires solving multiple SAT instances to decide right-uniqueness.

Since the performance improvement expected by the exploitation of structural knowledge is dependent on the complexity of the gate recognition procedure, more restricted but more efficient gate recognition procedures may be of practical value.

## 4.3 Subcircuit Recognition

Alan Mishchenko et al. [8] proposed a structural recognition approach based cut enumeration to find half- and full-adders. They based their algorithms on the assumption that the boundaries of arithmetic components are clear-cut and can be traced to these borders. By detecting which components belong to the same adder tree they are able to extract arithmetic components.

# 5 Gate Recognition, Classification and Visualization

In this chapter we introduce the general concept of gate recognition and take a closer look on the confluence of two gate recognition procedures. The second procedure introduced hides the characteristic function of the recognized gates behind a clausal encoding. We take such gate structures and suggest a first algorithm to reveal the underlying characteristic functions of the gates. To conclude this, we present how we render reconstructed and classified Boolean circuits to images.

## 5.1 Gate Recognition

Given a formula $F$, the process of finding encodings of a gate in $F$ is called gate recognition. Iterating this process incrementally reveals *a possible* gate structure in the formula.

**Definition 5.1.1.** *(Gate Recognition Procedure). Let $(F, \Gamma)$ be a structural formula. A process that applies a transformation under the condition $C$ for which $(F \setminus E, \Gamma \cup \{G\}) \equiv (F, \Gamma)$ holds, is denoted by $C \mid (F, \Gamma) \xrightarrow{R} (F', G')$, and $R$ is a procedure that describes which $E \subseteq F$ is transformed to the gate $G$.*

The definition of a gate requires a non-empty set, thus $\emptyset \neq E \subseteq F$ in the above definition. This implicitly guarantees that any gate recognition rule applied repeatedly results in either a violation of $C$ or an empty formula $F$ after a finite number of applications, if there are no new clauses introduced during the procedure. Otherwise, the rules need to be constructed such that the procedure terminates.

**Definition 5.1.2.** *(Gate Recognition). We call any non-empty set $\mathcal{R}$ of gate recognition rules a gate recognition procedure. The application of any single gate recognition rule in $\mathcal{R}$ is denoted by $\xrightarrow{\mathcal{R}}$.*

Applying $k$-many gate recognition rules in a procedure is denoted by $\xrightarrow{\mathcal{R}^k}$. The reflexive, transitive closure is denoted by $\xrightarrow{\mathcal{R}^*}$. For a given sequence of rules $(R_1, \ldots, R_n) \in \mathcal{R}^n$ we write

$$(F^{(0)}, \Gamma^{(0)}) \xrightarrow{R_1 \ldots R_n} (F^{(n)}, \Gamma^{(n)})$$

as a shorthand for

$$(F^{(0)}, \Gamma^{(0)}) \xrightarrow{R_1} (F^{(1)}, \Gamma^1) \xrightarrow{R_2} \dots \xrightarrow{R_n} (F^{(n)}, \Gamma^{(n)}).$$

**Definition 5.1.3.** *(Irreducibility and Terminating Sequences). Let $\mathcal{R}$ be a gate recognition procedure and let $S = (F, \Gamma)$ be a structural formula. We call $S$ irreducible under $\mathcal{R}$ iff no rule in $\mathcal{R}$ can be applied. A sequence $(R_1, \dots, R_n) \in \mathcal{R}^n$ of rules is called terminating, iff the last structural formula in $(F, \Gamma) \xrightarrow{R_1 \dots R_n} (F^{(n)}, \Gamma^{(n)})$ is irreducible.*

**Definition 5.1.4.** *(Confluence). Let $\mathcal{R}$ be a gate recognition procedure and let $\mathcal{A} = (R_1, \dots, R_n) \in \mathcal{R}^n$ and $\mathcal{A}' = (R'_1, \dots, R'_k) \in \mathcal{R}^k$ be two terminating sequences of rules over this set. We call $\mathcal{R}$ confluent iff for every structural formula $(F, \Gamma)$ the resulting structural formulas are equal, thus*

$$(F, \Gamma) \xrightarrow{R_1 \dots R_n} (F', \Gamma') \xleftarrow{R'_1 \dots R'_n} (F, \Gamma).$$

The first gate recognition procedure we present chiefly serves as an example for a confluent gate recognition procedure. The second example, hierarchical gate recognition, is one of the major developments contributed by Iser in [13] and is the go-to procedure in this thesis.

**Definition 5.1.5.** *(Trivial Gate Recognition). As a conjunction of disjunctions we can easily convert the formula to a Boolean circuit consisting only of OR- and AND-Gates. To do this we complete each disjunction with an output $o_C$ to an OR-gate. After completion, we introduce a single AND-gate with the outputs of the OR-gates as its inputs and a new output $o_F$.*

*Formally, $\mathcal{R}_T := \{R_1, R_2\}$, with the rules defined by*

$$\exists C \in F : |C| > 1 \mid (F, \Gamma) \xrightarrow{R_1} \big((F \setminus C) \cup \{\{o_C\}\}, \Gamma \cup \{(o_C, vars(C), OR)\}\big)$$
$$\forall C \in F : |C| = 1, |F| > 1 \mid (F, \Gamma) \xrightarrow{R_2} \big(\{\{o_F\}\}, \Gamma \cup \{(o_F, vars(F), AND)\}\big)$$

**Theorem 5.1.6.** *$\mathcal{R}_T$ is confluent.*

*Proof.* The conditions of $R_1, R_2$ are called $K_1, K_2$, respectively. Now, $K_2$ can only be applied after all clauses of size $>1$ have been extracted from $F$ to an OR-gate in $\Gamma$. If $K_1$ applies, then the clauses corresponding to OR-gates can be chosen in any order without altering the reconstructed circuit which is obtained when $(F, \Gamma)$ is irreducible under $\{R_1\}$. Furthermore, exactly one AND-gate is introduced by $R_2$, and the set $vars(F)$ is invariant under $R_1$.

Thus, there is exactly one possible Boolean circuit that can be obtained by $\mathcal{R}_T$. □

Of interest are gate recognition procedures that recognize gates other than AND-/OR-gates as well. Trivial gate recognition reveals no new information about the formula, as the result is obvious from the formula itself, and any other encoding of possible gates is similarly hidden in the reconstructed circuit, e.g. encodings of XOR-gates are still hidden in the gate structure. Thus, we are interested in more powerful recognition procedures that promise the possibility of useful subcircuit recognition.

Multiple approaches present methods to extract specific sets of different gates [14, 16]. The approach taken by Iser in [13] is different in that it is based on the idea of blocked clauses and a right-uniqueness prove to identify clauses that define a gate. The resulting algorithm is generic in the sense that this recognition procedure is not limited by a specific set of predefined gates. Before continuing to describe the algorithm itself, we introduce further theory to formally root the approach to make an analysis of the resulting problems possible.

**Definition 5.1.7.** *(Resolution). Let $C_1$ and $C_2$ be clauses and let $l$ be a literal for which $l \in C_1$ and $\neg l \in C_2$. We define the resolvent by $C_1 \otimes_l C_2 := (C_1 \cup C_2) \setminus \{l, \neg l\}$.*

**Definition 5.1.8.** *(Set of Literal Occurrences). Let $F$ be a formula and $l$ a literal. We define $F[l] := \{C \in F \mid l \in C\}$.*

**Definition 5.1.9.** *(Blocked Clauses). Let $F$ be a formula. We call $F$ blocked on a literal $l$ iff either of the following conditions holds:*

*(i)* $F[\neg l] = \emptyset$*, or*

*(ii)* $\forall D \in F[\neg l] : \mathcal{I}_\alpha(C \otimes_l D) = \top$ *for every assignment $\alpha$ of variables.*

The mathematical concept of a function is well-known. For a relation to be a function it needs to be well-defined: any input is mapped to at least one value. Additionally, for a map to be a function, there has to be at most one value an input is mapped to. Iser applies this idea to a definition of a relation over Boolean tuples, which in turn is more suitable.

**Definition 5.1.10.** *(Functional Relation). Call a relation $R \subseteq \mathbb{B}^{n+1}$ functional iff the following holds.*

$$\forall I \in \mathbb{B}^n \exists o \in \mathbb{B} : (i_1, \ldots, i_n, o) \in R \qquad \textit{(left-totality)}$$
$$\forall I \in \mathbb{B}^n \exists o \in \mathbb{B} : (i_1, \ldots, i_n, o) \notin R \qquad \textit{(right-uniqueness)}$$

**Theorem 5.1.11.** *(Gates and Functional Relations). Let $G$ be a gate. There is a functional relation and a bijection between its elements and models of $G$.*

Iser also proofs the following theorem.

**Theorem 5.1.12.** *(Left-Totality of Gate Encodings). Let $G$ be a gate with output $o$ and encoding $E$. For every clause $C \in E$ either $o \in C$ or $\overline{o} \in C$. Furthermore, all resolvents in $E[o] \otimes_o E[\overline{o}]$ are tautologic.*

Thus, blockedness on the output literal is a sufficient criterion for left-totality. To detect right-uniqueness of candidate encodings, Iser reformulates the criterion above and transforms this to a SAT instance. See [13] for details.

Now given a formula $F$ and a literal $o$, Iser verifies that $F[\overline{o}]$ and $F[o]$ are blocked on $o$ and solve the SAT instance mentioned, to conclude that $F$ encodes a gate $(o, vars(F[\overline{o}]) \setminus \{o\}, f)$, where $f$ is the function defined by the clauses in $F[\overline{o}] \cup F[o]$. This procedure is called `decodeGate`.

**Definition 5.1.13.** *(The Hierarchical Gate Recognition Rule). There is a single, general gate recognition rule*

$$\exists E \subseteq F : (E, \emptyset) \cong (\emptyset, \{(o, I, g)\}) \mid (F, \Gamma) \xrightarrow{R} (F \setminus E, \Gamma \cup \{(o, I, g)\})$$

This rule is implemented as a breadth-first search as follows. For a more in-depth discussion see [13].

**Algorithm 2:** Hierarchical Gate Recognition

```
Input: Structural Formula (F, Γ), roots ⊆ lits(F)
for r ∈ roots do setAsInput(r)
while roots ≠ ∅ do
    L ← ∅
    for r ∈ roots do
        monotonic ← ¬isSetAsInput(r) ∨ isSetAsInput(¬r)
        (r, P, g) ← decodeGate(F, r)
        if (r, P, g) ≠ ⊥ then
            Γ ← Γ ∪ (r, P, g)
            F ← F \ (F[¬r] ∪ F[r])
            L′ ← lits(F[¬o]) \ {r, ¬r}
            L ← L ∪ L′
            for l ∈ L′ do
                setAsInput(l)
                if ¬monotonic then setAsInput(¬l)
```

Notably, the concrete set of roots is unspecified so far. Furthermore, any actual implementation presents the root candidates and the clauses in arbitrary order. These points of freedom allow for different reconstructions.

The following formulas provide a starting point to evaluate possible choices in gate recognition procedures and reveal a deficit of optimized encodings. Additionally and informally speaking, a Boolean function can contain symmetries, while any (reconstructed) Boolean circuit is a directed acyclic graph, and thus a linear ordering of gates exists.

We construct two formulas from

$$F = (x_1 \leftrightarrow x_2 \leftrightarrow x_3) \ \wedge \ \left[ \leq 1(x_1, x_2) \right].$$

By encoding the equivalences in CNF we obtain

$$\begin{aligned} {}^{CNF}F' = &(\neg x_1 \vee x_2)(\neg x_2 \vee x_1) && (I) \\ &(\neg x_2 \vee x_3)(\neg x_3 \vee x_2) && (II) \end{aligned}$$

For the constraint $\leq 1(x_1, x_2)$ we use the sequential counter encoding from definition 3. We remark that $LT^{2,1}_{SEQ}(x_1, x_2) \neq LT^{2,1}_{SEQ}(x_2, x_1)$ and construct a formula for each case. Now

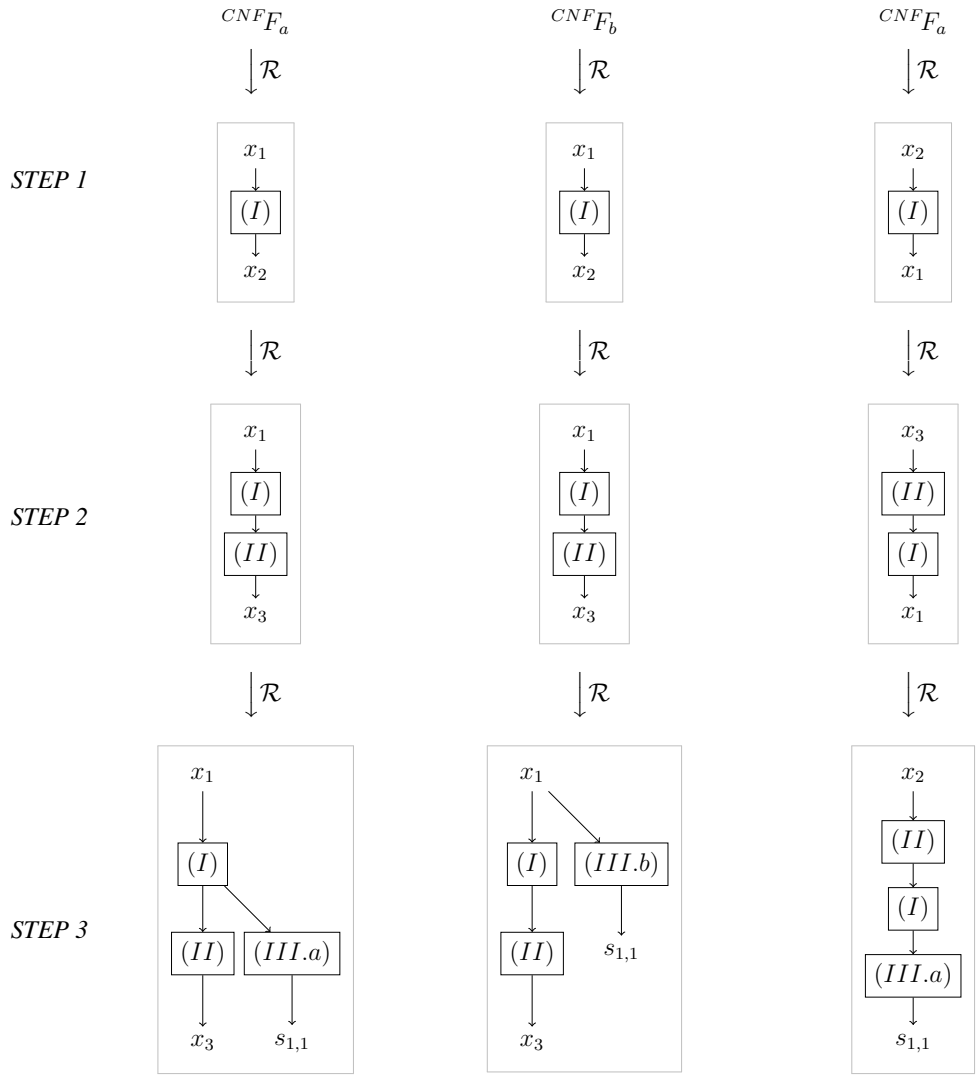
$$\begin{aligned} {}^{CNF}F_a = &(I)(II) \\ &(\neg x_1 \vee s_{1,1}) && (III.a) \\ &(\neg x_2 \vee \neg s_{1,1}) && (IV.a) \end{aligned}$$

$$\begin{aligned} {}^{CNF}F_b = &(I)(II) \\ &(\neg x_2 \vee s_{1,1}) && (III.b) \\ &(\neg x_1 \vee \neg s_{1,1}) && (IV.b) \end{aligned}$$

Both formulas have the same models when considering the projection to $X = \{x_1, x_2, x_3\}$, more precisely ${}^{CNF}F_a \cong_X {}^{CNF}F_b$.

The clauses $(IV.a)$ and $(IV.b)$ never appear in any reconstructed gate resulting from hierarchical gate recognition. However, the remaining parts $(I), (II), (III.a), (III.b)$ all define valid gates. Thus for each formula there are three choices for the first step of hierarchical gate recognition. The concrete order in which these are selected is irrelevant, other than it possibly affects which literal the algorithm picks as the inputs of the gates. In all three of the following examples we select the clauses in the same order, only changing the input of the first gate. The first two examples are possible executions on the two formulas above. The third example is a possible alternative to the first, where the input variable has been switched with the output variable.

**Example 5.1.14.** *(Non-Confluence of Hierarchical Gate Recognition).*




We compare the Boolean circuits in STEP 3; all three are the result of a terminating sequence. Firstly, we see that none of the three graphs is isomorphic to any other graph. Furthermore, there is no bijection between the literals in the clauses in the gates reconstructed from ${}^{CNF}F_a$ such that the circuits are isomorphic. Secondly, simply switching the order in which the variables are encoded also possibly affects the reconstruction. Thirdly, any recognition algorithm on a Boolean circuit that is the result of hierarchical gate recognition can not find a complete encoding in the circuit alone, as there is a subset of clauses that remains in the formula. Namely, we miss the clause $(\neg x_2 \vee \neg s_{1,1})$ in the first Boolean circuit and $(\neg x_1 \vee \neg s_{1,1})$ in the second. We will later return to these problems.

## 5.2 Classification

The result of hierarchical gate recognition is a set of gates with their characteristic function given by a set of clauses. We present a simple method to recognize a limited set of functions. This approach is based on the pattern check implemented in Candy [1] and has been extended to also include other functions than $k$-ary ANDs and ORs, and binary EQUIVs in non-monotonic gates.

Throughout this chapter we fix the meaning of the following symbols.

**Definition 5.2.1.** *(Forward and Backward Clauses). Let $G = (o, P, g)$ be a gate recognized by hierarchical gate recognition with its characteristic function encoded by clauses in $E$. Then $F = E[\overline{o}]$ is the set of forward clauses, and $B = E[o]$ the set of backward clauses.*

Now, as the gate is the result of hierarchical gate recognition, we know that $F$ and $B$ are blocked on the output $o$ of the gate. The algorithm implemented in [1] also provides the possibility to check whether the given gate is monotonic. Based on this we classify the gates by the following checks:

- If $G$ is not monotonic, then check the following.
  - If $|B| = 1$ and $\forall c \in F : |c| = 2$ then $g$ is an $|P|$-ary AND.
  - If $|F| = 1$ and $\forall c \in B : |c| = 2$ then $g$ is an $|P|$-ary OR.
  - If $|B| = 1 = |F|$ and $c \in F$, $d \in B : |c| = 2 = |d|$ then $g$ is a binary equivalence.
  - If $|B| = 2 = |F|$ and $|P| = 2$ then $g$ is a binary XOR.
- If $G$ is monotonic, then $|B| = 0$, and it remains to check the following.
  - If $\forall c \in F : |c| = 2$ then $g$ is an $|P|$-ary AND.
  - If $|F| = 1$ then $g$ is an $|P|$-ary OR.

## 5.3 Graph Representation of Boolean Circuits

We present a definition to construct a graph from the set of gates returned by a gate recognition procedure.

We begin by formalizing the result of a gate recognition procedure.

**Definition 5.3.1.** *(Reconstructed Boolean Circuits). Let $\mathcal{R}$ be a gate recognition procedure. By $\mathcal{G}_{\mathcal{R}}(F)$ we denote the Boolean circuit that is the result of $\mathcal{R}$ applied to the formula $F$. If $\mathcal{R}$ is clear from the context, we omit the subscript.*

Informally, we called a Boolean circuit, which is a set $\mathcal{G} = \{G_1, \ldots, G_n\}$, a graph. Formally a graph $(V, E)$ has a set of vertices and a set of edges. The construction

$$\Big(\mathcal{G}, \big\{\big((o_1, P_1, g_1), (o_2, P_2, g_2)\big) \in \mathcal{G}^2 \mid o_1 \in P_2\big\}\Big)$$

justifies calling $\mathcal{G}$ a graph as well, as the edges are already implicitly given by the gates in the Boolean circuit.

## 5.4 Visualization

We use the graph from the previous section as the input of an algorithm that exports to the dot format used by Graphviz [11], to render the graph as an image with the characteristic functions described by the result of the classification.

Dot files conceptually have the format '$digraph\{V, E\}$', where $V$ is a list of `vertex names` and $E$ is a list of `vertex name` -> `vertex name` representing the edges. If we assume the input to be given as an adjacency list, we can construct a valid dot-file by iterating over the gates, and for each gate over its connections. The default case for the text in a vertex, rendered by Graphviz, is given by the CNF set notation. If a gate can be successfully classified, we assume to know how the recognized function shall be rendered to text.

**Algorithm 3:** Boolean Circuit Visualizer

```
   Input: A formula F in CNF set notation
   Output: A string that is a valid dot file
1  Calculate G_R(F) as an adjacency list
2  Let v and e be empty strings
3  for g ∈ G_R(F) do
4  |  v* ← classification of g or otherwise set notation as string
5  |  Append v* to v
6  |  for each edge (g, h) do
7  |  |  w* ← classification of h or otherwise set notation as string
8  |  |  Append v* + " -> " + w* to e
9  return"digraph { " + v + e + " } "
```

We provide an implementation of a visualizer that uses hierarchical gate recognition and additionally renders input variables. As an example we render $LT^{n,k}_{SEQ}$ for small $n$ and $k$. The text in the first line of each gate is the function name assigned by the classifier.

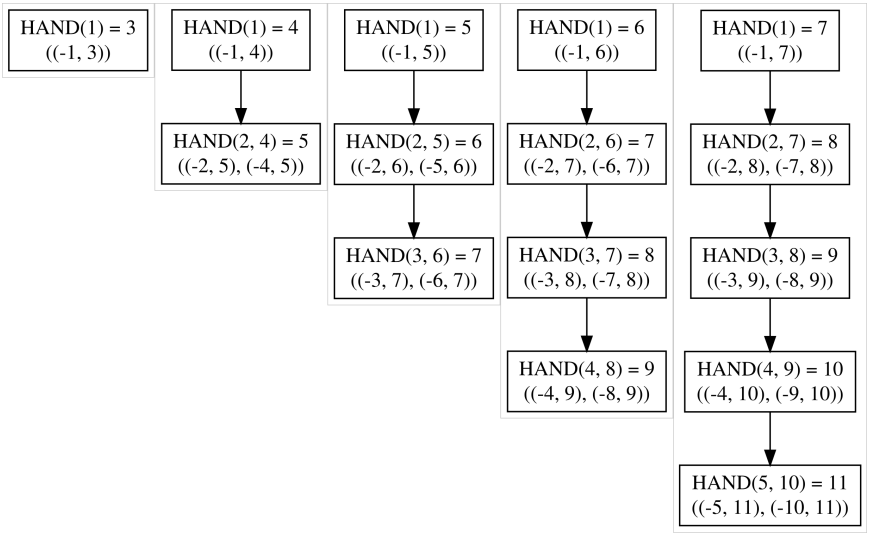



**Figure 5.1:** $LT^{2,1}_{SEQ}, LT^{3,1}_{SEQ}, LT^{4,1}_{SEQ}, LT^{5,1}_{SEQ}$ and $LT^{6,1}_{SEQ}$

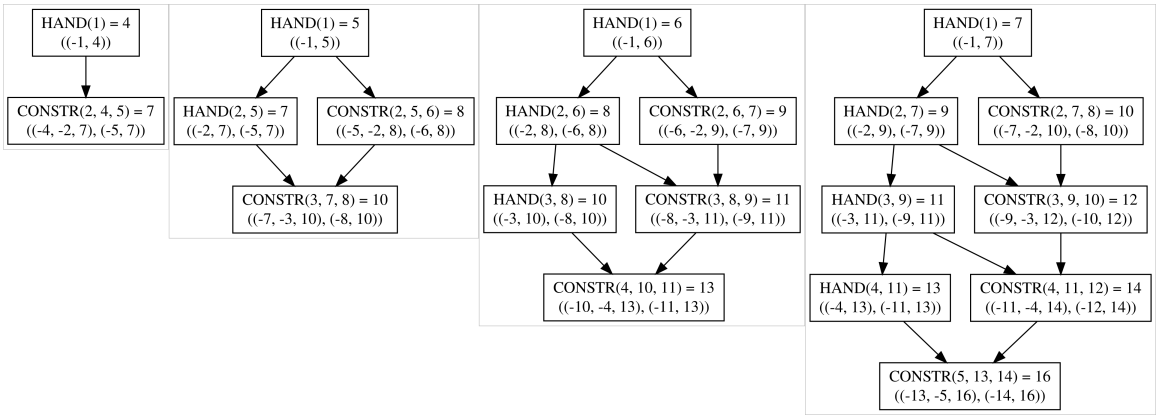



**Figure 5.2:** $LT^{3,2}_{SEQ}, LT^{4,2}_{SEQ}, LT^{5,2}_{SEQ}$ and $LT^{6,2}_{SEQ}$

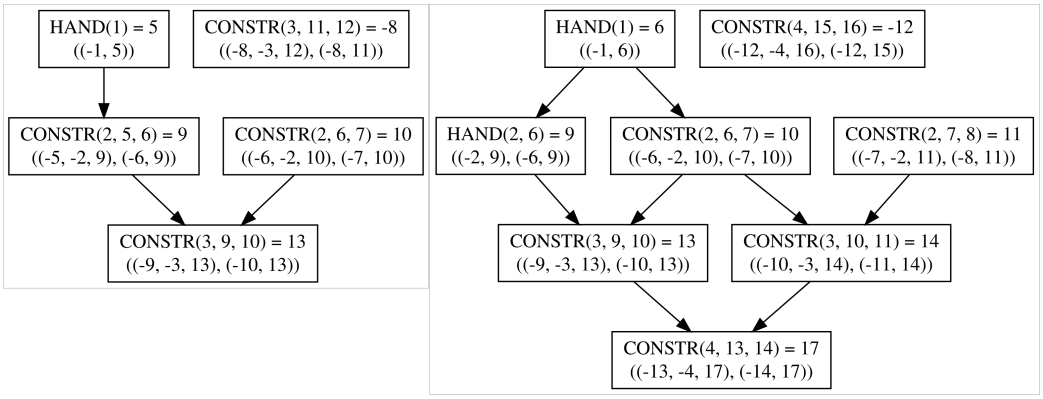



**Figure 5.3:** $LT^{4,3}_{SEQ}$ and $LT^{5,3}_{SEQ}$

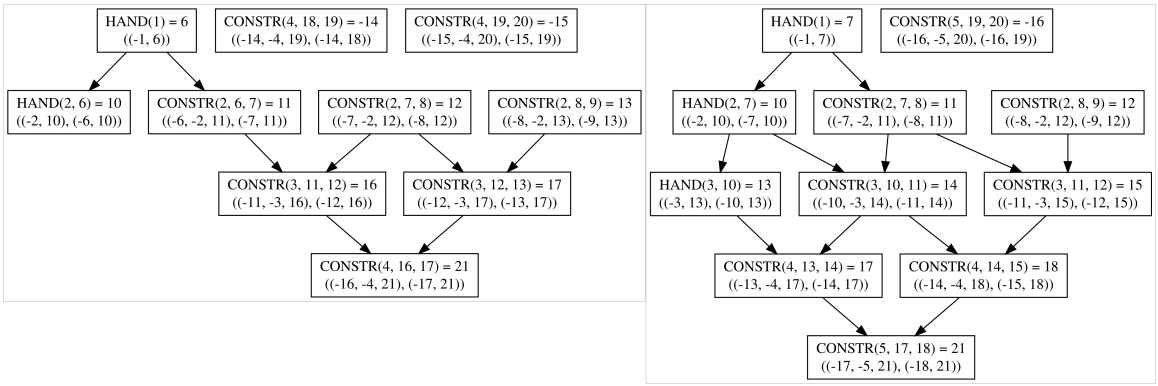



**Figure 5.4:** $LT^{5,4}_{SEQ}$ and $LT^{6,3}_{SEQ}$

# 6 Unification

In Example 14 on the non-confluence of hierarchical gate recognition we already observed that there is a single clause that never belongs to any gate. Visualizing the reconstructed circuit of the sequential counter constraint in the previous section further suggested the necessity to recognize not only a subcircuit, but additionally a set of clauses. To this end we introduce a unification algorithm. Later we will see the possibility to fix a major problem, also arising from the non-confluence.

For clauses of length two and three, as we observe in the case of sequential counter encodings, it is practical to implement the clause-search by a simple case analysis. Under the perspective of further generalizations it is of interest to provide an algorithm for any formula and clause length. Further this algorithm provides a more generic alternative to the classification algorithm we presented in the respective section. Ultimately, this purely syntactical approach on clauses provides a promising path to fixing the problems arising from the non-confluence of gate recognition procedures and a general view on the problem of subcircuit recognition.

## 6.1 Formal Definition of Syntactical Unification

Before we give a formal definition, we adapt and extend previous definitions to suit our needs in this chapter.

For the purpose of improving the performance of unification it will prove useful to impose an ordering on the clauses and literals in a CNF formula. However, not any arbitrary ordering is of interest here, which is why we impose a fixed ordering in the following definition. This will prove to be substantial to the algorithms developed later.

**Definition 6.1.1.** *(CNF Tuple Notation). Let $F$ be a formula given in CNF set notation. By ordering the clauses in the sets above as follows we obtain a tuple notation:*

*(i) Pick all disjunctions only containing negative literals and order them increasing in the number of literals they contain.*

*(ii) Then append clauses with negative and positive literals, only this time sort them*

*a) decreasing in the number of literals they contain, then within clauses of fixed length,*

*b) sort the clauses decreasing in the number of negative literals they contain, then sort each individual clause by,*

*c) first listing all negative clauses (in any order), secondly all positive clauses (in any order).*

*(iii) Lastly, append all disjunctions only containing positive literals and, again, order them increasing in size.*

*The resulting formula has the following structure*

$$N_1, \ldots, N_n, M_2, \ldots, M_m, P_1, \ldots, P_p = F.$$

*where $N_i$ represents disjunctions of $i$-many negative literals and $P_j$ represents disjunctions of $j$-many positive literals. For $k$ from 2 to the maximum length $m$ of mixed clauses $M_k = M'_1, \ldots, M'_{m_k}$ where $M'_j$ represents a disjunction of $j > 0$ negative literals and $k - j > 0$ positive literals. Any $M'_j$ may be empty.*
*Again, for the sake of simplicity, we choose to write the equality.*

The empty set and the empty tuple both represent the formula that is always satisfied. In comparison the set $\{\emptyset\}$, and the tuple $(\emptyset)$ represent the formula that is always unsatisfied.

We give an illustrating example of the different kinds of representations:

**Example 6.1.2.** *(Set and Tuple Notation). The formula*

$$(x_3 \vee \neg x_4 \vee \neg x_6) \wedge (1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1) \wedge (\neg x_3) \wedge (x_2 \vee x_4 \vee \neg x_5) \wedge (x_8 \vee \neg x_5 \vee \neg x_7 \vee \neg x_9)$$

*can be represented by the set*

$$\big\{\{x_3, \neg x_4, \neg x_6\}, \{1\}, \{x_1, x_2, x_3\}, \{x_1\}, \{\neg x_3\}, \{x_2, x_4, \neg x_5\}, \{x_8, \neg x_5, \neg x_7, \neg x_9\}\big\},$$

*or alternatively is given in tuple notation by*

$$\big((\neg x_3), (\neg x_5, \neg x_7, \neg x_9, x_8), (\neg x_4, \neg x_6, x_3), (\neg x_5, x_4, x_2), (x_1), (1), (x_1, x_2, x_3)\big).$$

*All three terms represent the same Boolean function.*

**Definition 6.1.3.** *(Variable and Literal Tuples). Let there be a formula in tuple notation $T_F$, we define the following*

- *$lits(T_F)$ is defined as the duplicate-free tuple of literals appearing in the same order as in $T_F$,*
- *$vars(T_F)$ is defined as the duplicate-free tuple of variables appearing in the same order as their corresponding literals appear in $T_F$.*

Analogously we define the tuples $lits(c)$ and $vars(c)$ for a clause $c$.

**Definition 6.1.4.** *($\eta$-Equivalence). Let $F, G \in \Phi$ be two formulas. We call these $\eta$-equivalent, iff there is a bijection $\eta : vars(F) \to vars(G)$, and with the renamed variables in $F$ we obtain $\tilde{F}$, under which it holds that $\widetilde{F} = G$; we denote this by $F \overset{\eta}{=} G$. Thus $\eta$-equivalent formulas only differ in variable-naming, not in semantics.*

**Lemma 6.1.5.** *$\eta$-equivalence is an equivalence relation.*

Equality, as well as the existence of a bijection, are equivalence relations. Thus the same must hold for $\eta$-equivalence. ◯

As usual, the corresponding equivalence class to a formula $F$ is $[F]_\eta = \{G \mid F \stackrel{\eta}{=} G\}$. Informally, this is the set of all formulas, which can be transformed to $F$ by consistently renaming the occurring variables.

Firstly, this is of interest because it suggests that there is a canonical variable naming of the variables. The DIMACS format [3] uses seamless numbering starting from $1$ to represent the variables, posing a possible definition of this canonical naming. Secondly, this provides a simple language to formulate the problem of syntactical unification.

**Definition 6.1.6.** *(Unification and Unificator). Let $F$ and $G$ be two formulas. We call deciding whether $F \in [G]_\eta$ or $F \notin [G]_\eta$ unification. If $F$ and $G$ unify, any bijection $\eta : vars(F) \to vars(G)$, under which the formulas are equivalent, shall be called a unificator.*

## 6.2 Algorithms

Continuing from the formal definition of the problem we now develop a unification algorithm.

The naïve approach, to test every possible bijection between variables and then check whether the formulas are equal, has time complexity $\Omega(n!)$ where $n$ is the size of the intersection of the variables under which the formulas shall be compared. We give an algorithm that improves the expected time complexity of the naïve approach drastically for many cases.

As a starting point we assume a given set $\kappa$ of predefined connectives in CNF tuple notation. The input function $F$, however, is given by a sequence of clauses. We remark that the set definition of CNFs implies the absence of duplicates.

Our first algorithm calculates a possible CNF tuple notation for $F$.

**Algorithm 4:** CNF Tuple Notation

`Input`: A formula ${}^{CNF}F$

$N' \leftarrow \{c = \{l_1, \ldots, l_n\} \in F \mid l_i \text{ is a negative literal}\}$
$M' \leftarrow \{c = \{l_1, \ldots, l_n\} \in F \mid \exists l_i \text{ positive and } \exists l_j \text{ negative}\}$
$P' \leftarrow \{c = \{l_1, \ldots, l_n\} \in F \mid l_i \text{ is a positive literal}\}$

$N \leftarrow$ sort clauses in $N'$ increasing in size
$\{M_i\} \leftarrow \{M \in M' \mid M \text{ is a disjunctions of } i \text{ literals }\}$
$\{M'_{i,k}\} \leftarrow \{M_i \mid \text{there are } k \text{ negative literals in } M_i\}$
$\{M_{i,k}\} \leftarrow$ move negative literals in each $M'_{i,k}$ to the front
$M \leftarrow$ sort $M_{i,k}$ lexicographically first by $i$, then by $k$
$P \leftarrow$ sort clauses in $P'$ increasing in size

`return` $(N, M, P)$

This algorithm can also be employed to construct the set $\kappa$.

**Example 6.2.1.** *(Tuple Notations of Common Connectives). We give the tuple notation of commonly used propositional connectives using the algorithm above*

$$\underbrace{((a),(b))}_{=a\wedge b},\ \underbrace{((a,b))}_{=a\vee b},\ \underbrace{((\neg a,b))}_{=a\implies b},\ \underbrace{((\neg a,b),(\neg b,a))}_{=a\iff b},\ \underbrace{((\neg a,\neg b),(a,b))}_{=a\oplus b}.$$

It is notable that each of these can be uniquely identified by the number of positive, negative, and mixed clauses. Extending this idea to incorporate the number of literals in the clauses in $N$, $M$, and $P$, and to the number of negative literals in clauses in $M$ we arrive at the signature of a formula.

Instead of directly defining the signature, we say that $\mathcal{S}$ is the signature of a sequence of clauses $F$, iff the following algorithm returns $\mathcal{S}$ given the input $F$.

**Algorithm 5:** Calculate Formula Signature

`Input`: A formula ${}^{CNF}F$

1 $((N_1, \ldots, N_n), M, (P_1, \ldots, P_p)) \leftarrow$ tuple notation of $F$

2 $N^* \leftarrow (|N_1|, \ldots, |N_n|)$

3 $M^* \leftarrow$ iterate over lexicographically ordered entries $M_{i,k}$ in $M$ and append $(i, k)$

4 $P^* \leftarrow (|P_1|, \ldots, |P_p|)$

5 `return` $(N^* \mid M^* \mid P^*)$ // Here $\mid$ is an arbitrary separator

**Definition 6.2.2.** *(Signature of a Clause). The signature of a clause $c$ is the signature of the formula $(c)$.*

**Example 6.2.3.** *The signatures of the propositional connectives from above are*

$$\underbrace{((\,) \mid (\,) \mid (1), (1))}_{=a\wedge b},\ \underbrace{((\,) \mid (\,) \mid (1,1))}_{=a\vee b},\ \underbrace{((\,) \mid (1,2) \mid (\,))}_{=a\implies b},\ \underbrace{((1,2) \mid (1,2) \mid (\,)}_{=a\iff b},\ \underbrace{((2) \mid (\,) \mid (2))}_{=a\oplus b}.$$

*Here we clearly see that different signatures of the given connectives correspond to different formulas.*

This example leads to the following insight.

**Lemma 6.2.4.** *Let $F$ and $G$ be formulas with $F \in [G]_\eta$. Then $F$ and $G$ have the same signature.*

The contraposition, different signatures for given formulas, implies that they are different, will be used in the following algorithms to cut the search space further.

## 6.2.1 Syntactical Unification

Before we give the actual unification algorithm we need an auxiliary algorithm that returns a data-structure that implicitly represents all possible CNF tuple notations. We note that clauses need to be of the same size and need to contain the same number of negative and positive literals. Thus, only clauses of the same signature in $N, M, P$, respectively, need to be compared and possibly permuted.
We give an algorithm that returns a list of tuples $(start, finish)$, such that all clauses in $start, \ldots, finish$ have the same signature and neither $start$ nor $finish$ can be changed to include more clauses in the sequence. Clauses are identified by their position in the CNF tuple notation.

Lastly, we give an algorithm that decides $F \overset{?}{\in} [G]_\eta$ for given formulas $F$ and $G$ in tuple notation. An important prerequisite for this algorithm is another algorithm that allows

**Algorithm 6:** Permutation List

Input: A formula $F$ in CNF tuple notation

```
1  L a tuple of elements in ℕ₀²
2  start ← 0
3  finish ← 0
4  S_start ← signature of F[start]
5  S_finish ← S_start
6  while finish < |F| do
7      while S_start = S_finish and finish + 1 < |F| do
8          finish ← finish + 1
9          S_finish ← signature of F[finish]
10     Append (start, finish) to L
11     start ← finish + 1
12     finish ← start
13     S_start ← signature of F[start]
14     S_finish ← S_start
15 return L
```

iterating over all permutations of a given tuple. For example, Heap's algorithm [12] can be employed.

The bijection $\eta$ is constructed during the execution of the algorithm, to test whether $\eta$-equivalence can be established. For practical purposes it is useful to not map variables to variables, as in the definition of $\eta$, but instead to define it over a sortable set. Inspired by the DIMACS format [3] for CNFs, we define $\eta_{\mathbb{Z}}$ over the integers, thus

$$\eta_{\mathbb{Z}} : lits(F) \to D,$$

where $D := \{\delta(l) \mid l \in lits(F)\}$ and if $x_i$ is a variable, then $\delta(x_i) = i$ and $\delta(\neg x_i) = -i$. For $G$ we expect to already know a map to $D$; i.e. the formula tuple notation is not given in terms of literals but as integers representing the literals.

Furthermore, it is of value to introduce placeholders; these are literals in a formula that can be freely mapped by $\eta$, after considering their polarity.

**Definition 6.2.5.** *(Fixed Literals and Placeholders). Let $F$ be a formula, $l \in lits(F)$ a variable, and $\eta_{\mathbb{Z}} : lits(F) \to D$ a partially defined bijection. We call $l$ a fixed literal, if $\eta_{\mathbb{Z}}(l)$ is defined, otherwise we call $l$ a placeholder if it is not assigned to any integer (yet). We annotate placeholder literals $l$, we write $l^*$.*

The bijection $\eta_{\mathbb{Z}}$ serves the same purpose as $\eta$. Because it is possible to freely translate between the DIMACS format and any other variable naming, we do not formally distin-

guish between the two functions and continue with the notation used so far to avoid inflating the complexity of the notation in the following algorithms.

**Algorithm 7:** Unify Literal

```
Input: Two literals l_1, l_2 ∈ D, a partially defined bijection η
Output: True iff the literals unify, η is possibly updated
if η(l_1) is defined then
    return η(l_1) ≟ l_2
else
    if l_1 has the same polarity as l_2 then
        η : l_1 ↦ l_2
        return true
    else
        return false
```

**Pseudocode Notation.** (Updating Maps). If an algorithm defines a new unique key-value-pair $(x, y)$ in a map $\eta$, we denote this by writing $\eta : x \mapsto y$.

**Algorithm 8:** Unify Clauses

```
Input: A clause c_1 and a clause c_2 ∈ D^n both in CNF tuple notation, a partially
       defined bijection η
Output: True iff the literals unify, η is possibly updated
Return false if c_1 and c_2 have different length or a different number of negative literals
Sort c_1 by first listing literals defined under η in order, then by listing the placeholders
Sort c_2 by listing literals the order under η
for i = 0, ..., n − 1 do
    if c_1[i] does not unify with c_2[i] then return false
return true
```

**Definition 6.2.6.** *(Distinct Clause). Given two formulas $F$ and $G$, and a clause $c \in F$. We call $c$ distinct over $G$ iff there is exactly one clause in $d \in G$ that unifies with $c$. Similarly, we call $(c, d)$ a pair of distinct clauses over $F$ and $G$.*

**Lemma 6.2.7.** *(Distinct Clauses and Signatures). A pair $(c, d)$ of clauses is distinct, iff they have the same signature and there is exactly one clause with this signature in $G$.*

*Proof.* Let $(c, d)$ be a pair of distinct clauses. The clause $c$ can be unified with $d$ by mapping the $i$-th literal $lits(c)$ to the $i$-th literal in $lits(d)$. Conversely, if there is exactly one clause

$d$ with a given signature then the unificator to $c$ can be constructed as above, making $(c, d)$ a pair of distinct clauses. □

Distinct clauses can be used to update a unificator early in the unification process, since they restrict the possible bijections.

**Algorithm 9:** Unify Formulas

`Input:` Two formulas $F, G$ in CNF tuple notation, a partially defined bijection $\eta$, that is defined for all literals in $G$

`Output:` True, iff the formulas unify, and in the case of successful unification a completely defined $\eta$, otherwise an update of this map

```
1  Calculate the signatures S_F and S_G of F and G respectively
2  Return false if S_F ≠ S_G
3  D ← collect all pairs of distinct clauses in F over G
4  for (c, d) ∈ D do
5  |  Unify c with d or return false
6  |  Remove c from F and d from G
7  P ← calculate permutation list of F
8  for (s, f) ∈ P do
9  |  for each permutation of the clauses in F[s, f] do
10 |  |  η' ← η
11 |  |  for c_i ∈ F[s, f] = (c_1, ..., c_{f−s+1}) do
12 |  |  |  η'' ← η'
13 |  |  |  if c_i does not unify with G[i] then
14 |  |  |  |  η' ← η''
15 |  |  |  else
16 |  |  |  |  if i = f − s + 1 then return true
17 |  |  η ← η'
18 return false
```

The performance of Algorithm **??** is largely improved over the naïve approach by (a) terminating immediately for formulas with different signatures, (b) exploiting the tuple notation, (c) cutting the search space by only permuting necessary clauses and literals based on permutation lists, and (d) sorting the literals on demand.

The time complexity of Algorithm **??** can be determined as follows. Each of the signatures can be calculated in linear time. They can be compared in linear time as well. The permutation list $\mathcal{P}$ can be calculated in linear time. The set $D$ can be calculated in a single iteration over $\mathcal{P}$, and thus in linear time. Let there be $n$ many elements in $\mathcal{P}$. For every

pair $(s_i, f_i)$ there are at most $(f_i - s_i)!$ many permutations to be checked. The unification of a single clause can be done in linear time (assuming a linear time sorting algorithm) or otherwise in time $n \cdot \log n$. Combining these results we see that the time complexity of the complete algorithm is in $\Theta\big(\prod_{i=1}^{n}(f_i - s_i)! + max(|F|, |G|)\big)$.

Although the time complexity is still daunting, the algorithm is expected to perform reasonably well for many desirable cases. For example, if $\eta$ is fully defined, the search for clauses is expected to be similarly complex as a simple search.

### 6.2.2 Syntactical Classification

The approach taken in chapter 5.2 to classify the gates is based on the knowledge that the clauses form a blocked set and additional insights on the properties of characteristic functions. The presented unification algorithm allows for a different approach that does not rely on specific knowledge and instead only requires a canonical form of the function to be recognized. This approach is expected to be substantially slower, but more generic and easier to extend.

For a fixed set of connectives $\kappa$ in tuple notation we define the following algorithm.

**Algorithm 10:** Gate Classification

`Input:` A set of clauses $E$

1 $T_E \leftarrow$ tuple notation of $E$
2 $T_{\overline{E}} \leftarrow$ tuple notation of $\overline{E}$
3 `return` first $K \in \kappa$ that unifies with $T_E$ or $T_{\overline{E}}$, or otherwise $\bot$.

The power of this algorithm is determined by the number of predefined Boolean functions in $\kappa$. Adding more, and especially longer, formulas to $\kappa$ can drastically worsen the run time of this algorithm. If $|T_i| = n$ is the length of the longest tuple in any $K \in \kappa$, and $K$ is defined by $k$ many tuples, then the time complexity of the classification is in $\Omega(|\kappa| \cdot k! \cdot n!)$. For many practical usages only a handful of connectives are of interest, which in turn are of manageable, i.e. they are small and of constant size.

**Example 6.2.8.** *(Tuple Notation of Propositional Connectives). As a starting point we suggest to include the following, binary connectives in* $\kappa$

$$
\begin{aligned}
AND_{full} &= ((\neg x_1, \neg x_2, o), (\neg o, x_1), (\neg o, x_2)),\\
AND_{half} &= ((\neg o, x_1), (\neg o, x_2)),\\
OR_{full} &= ((\neg o, x_1, x_2), (\neg x_1, o), (\neg x_2, o)),\\
OR_{half} &= ((\neg o, x_1, x_2)),\\
XOR &= ((\neg o, \neg x_1, \neg x_2), (\neg o, x_1, x_2), (\neg x_1, x_2, o), (\neg x_2, x_1, o)),\\
NAND &= ((\neg o, \neg x_1, \neg x_2), (x_1, o), (x_2, o)),\\
NOR &= ((\neg o, \neg x_1), (\neg o, \neg x_2), (x_1, x_2, o)).
\end{aligned}
$$

Encodings can possibly appear minimized, which is why different forms of the same Boolean function can be of interest.

# 7 Sequential Counter Recognition

After finding constraints in the formula, it is our intent to extract these and represent them in an abstract manner. We thus wish to extract maximal structural encodings of the constraint. However, this criterion is not sufficient for a sensible reconstruction. We begin this chapter by formalizing an additional property a recognition algorithm should satisfy. Furthermore, we present a data structure and auxiliary algorithms and conclude the section with the general Boolean circuit we intend to find.

## 7.1 Preliminaries

### 7.1.1 Soundness of Recognition Algorithms

We explain the soundness of recognition algorithms under the following definition.

**Definition 7.1.1.** *(Soundness of At-Most Extractions). Let $F$ be the starting formula for an extraction of a subset $E \subseteq F$. Furthermore, we require knowledge of the variables constrained, i.e. $\{x_1, \ldots, x_n\}$, and the number $k$ of variables that can be true at most. We call an extraction $(F \setminus E) \cup \leq k(x_1, \ldots, x_n) =: R$ sound, iff $R \equiv F$.*

Based on Definition 3 an algorithm that returns a subset $X' \subset \{x_1, \ldots, x_n\}$ is sound. However, soundness is violated, if a $k' < k$ is returned.

In summary, soundness is a necessary criterion on our recognition algorithm, maximality is a desirable but not necessary criterion on our algorithm.

### 7.1.2 Data Structures and Auxiliary Algorithms

The data structure used to hold the Boolean circuit is based on an adjacency list. Without modification the following auxiliary algorithms would each take linear time to traverse the gates. If we however introduce a second adjacency list that represents the edges in reversed order, we can implement both auxiliary algorithms to only take constant time.

The first auxiliary algorithm shall be called `hop over` and is best explained in conjunction with figure 7.1.

Given the leftmost gate we want to reach the rightmost gate at the top. Our chosen graph representation allows us to find the out-edge of the starting gate and the in-edge of the hop

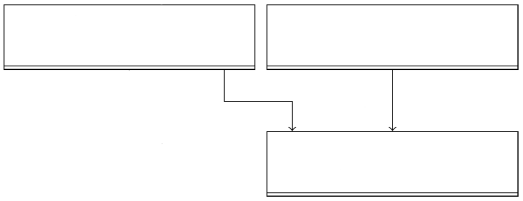


**Figure 7.1:** Part of the *rectangle*

gate in constant time. Additionally we want two properties to be incorporated as well. The first is to ensure we hop in the right direction. That is, when hopping from $a$ to $b$ we do not want to hop back to $a$, given the input $b$ for the next hop. After considering the first condition we want to ensure there is exactly one possible hop. In any other case the result shall be $\perp$.

The second algorithm shall be called `hop under`. After switching the directions of the edges in Figure 7.1, we require essentially the same behaviour as `hop over`.

Both of these auxiliary algorithms are used to traverse the *rectangle* in 7.2.

Lastly, we assume the gates to be classified as described in chapter 5.2 with the result being saved in an array and the result being obtainable by a function $c$ that maps a gate to its classification.

### 7.1.3 The General Subcircuit

Lastly, we give the general, reconstructed Boolean circuit for $LT^{n,k}_{SEQ}$ for sufficiently large $n$ and $k$.

For the sake of ease we name several parts of the graph in Figure 7.2. The gates with an extra line at the bottom form a $(n-4) \times (k-1)$ grid of gates. We refer to this part as the *rectangle*. Both hop-algorithms are for the traversal of this part. The column of $n-1$ many gates to the left of the *rectangle* shall be called the $k1$-column, as it corresponds to the reconstructed circuits where $k = 1$. The remaining part of the connected part of the Boolean circuit consists of the three gates at the rightmost bottom of the *rectangle*; we refer to those gates as the *closing* part. The remaining gates, detached from the other parts, shall be called the *floating* gates. The following algorithms correspond to these four parts.

## 7.2 Algorithms

### 7.2.1 Graph-Based Recognition

We develop a recognition algorithm that recognizes constraints in formulas $F = LT^{n,k}_{SEQ}$. This is done by simultaneously traversing the reconstructed graph and building a map be-

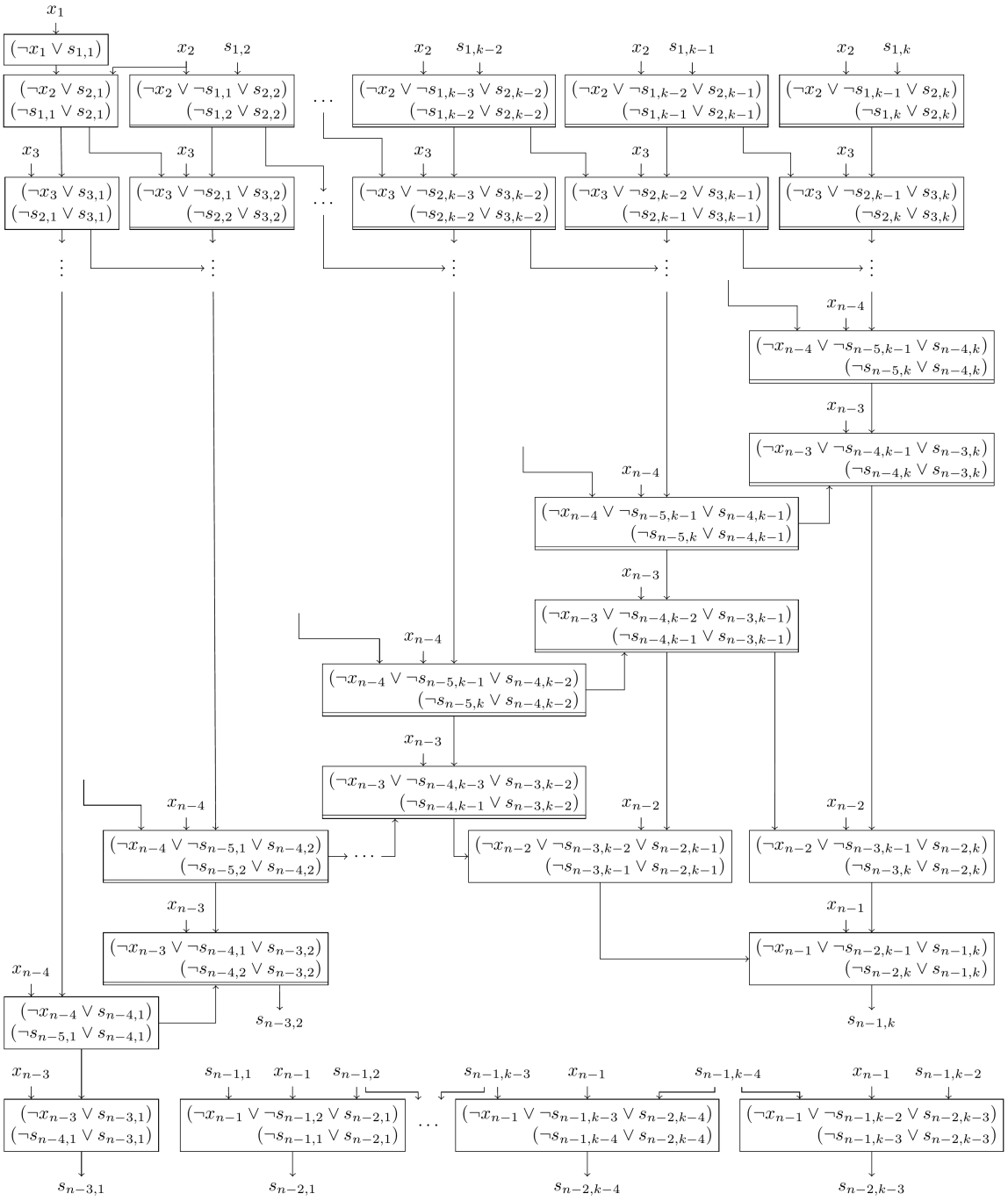



**Figure 7.2:** The result of hierarchical gate recognition applied to $LT^{n,k}_{SEQ}$

tween variables in the gates and variables in the encoding. This map can then be used to find clauses not distributed over the gates.

The recognition algorithm consists of four building blocks corresponding to the parts of the graph, namely the $k1$-column, the *rectangle*, the *closing*-part and the *floating* gates. At the end of this chapter we consolidate these parts into a single algorithm, as edge cases must be handled and the recognition of multiple constraints in an arbitrary formula is the end goal.

## Part I

The first part searches for the base case ($n = 2$, $k = 1$) and extends the set of constrained variables as far as possible at this stage. This corresponds to the recognition of the $k1$-column above.

**Algorithm 11:** Recognize $LT_{SEQ}^{n,1}$ ($k1$-column part)

`Input`: CNF formula $F$, a corresponding Boolean Circuit $\mathcal{G}(F)$, a base gate $g$
`Output`: A possible structural encoding $(E, \Gamma)$ of $LT_{SEQ}^{n,1}$, and a map $\eta$

```
1  Initialize an empty map η
2  v_b ← find a gate that unifies with (¬x*_1 ∨ s*_{1,1})
3  η : x_1 ↦ var(v_b[0]) and η : s_{1,1} ↦ var(v_b[1])
4  Γ ← {v_b}
5  v_c ← v_b
6  v_s ← ⊥
7  i ← 2
8  do
9      v_s ← ⊥
10     for edge (v_c, w) where c(w) is an implication do
11         if w unifies with (¬s_{i−1,1} ∨ s*_{i,1})(¬x*_i ∨ s_{i,1}) then
12             v_c ← v_s
13             v_s ← w
14             η : s_{i,1} ↦ var(v_s[0]) and η : x_i ↦ var(v_s[2])
15             Γ ← Γ ∪ {v_s}
16             i ← i + 1
17             break for-loop
18 while v_s ≠ ⊥
19 E ← find non-gate clauses in F or return ⊥
20 return (E, Γ, η)
```

Not all clauses of $LT_{SEQ}^{n,1}$ are distributed over the gates found by Algorithm 11. Nonethe-

less, the map $\eta$ allows to construct the missing clauses, which in turn can be used to efficiently find them in $F$ to complete the encoding. This is done in line 17 with help of the following lemma.

**Lemma 7.2.1.** *(Non-Gate Clauses of $LT^{n,1}_{SEQ}$). Let $\Gamma$ be the gates found in Algorithm 11, and $\eta$ the bijection defined after the completion of the while-loop. Now*

$$\big(\{\neg\eta(x_2) \vee \neg\eta(s_{1,1})\} \cup \{\neg\eta(x_{i+1}) \vee \neg\eta(s_{i,1}) \mid i \in \{2,\ldots,n-1\}\}, \Gamma\big)$$

*is a (not necessarily maximal) structural CNF encoding of $\leq 1(x_1,\ldots,x_n)$.*

The missing clauses can be determined by comparing the recursive forms of $LT^{n,k}_{SEQ}$ with the clauses in the gates found by Algorithm 11. ◯

We remark, however, that this algorithm alone is not sound, as it does not consider the possibility of $k > 1$.

### Part II and Part III

The next step is to extend the recognition to extend the found structural encodings to find possible constraints with $k \geq 2$. Informally, the *rectangle* in Figure 7.2 increases in width as $k$ grows. The following algorithm traverses the rows of this rectangle and verifies the existence of the necessary clauses in each gate.

During this part we can determine the possible $k$ of the constraint. The width of the *rectangle* plus one is the desired value.

**Algorithm 12:** Recognize $LT_{SEQ}^{n,k}$ (*rectangle* part)

`Input:` Boolean Circuit $\mathcal{G} = \mathcal{G}(F)$, the result $(E, \Gamma, \eta)$ of Algorithm 11
`Output:` A possible partial encoding of $LT_{SEQ}^{n,k}$, a partially defined map $\eta$

1 $\Gamma' \leftarrow \emptyset$
2 $i \leftarrow 1$
3 `while` $i < |\Gamma|$ `do`
4 $\quad v_s \leftarrow \Gamma[i]$
5 $\quad j \leftarrow 2$
6 $\quad$ `do`
7 $\quad\quad E_n \leftarrow (\neg\eta(x_i) \vee \neg\eta(s_{i,j-1}) \vee s^*_{i,j})(\neg s^*_{i-1,j} \vee s_{i,j})$
8 $\quad\quad v_n \leftarrow$ hop $v_s$ and verify the result unifies with $E_n$
9 $\quad\quad$ `if` $v_n = \perp$ `then return` $(E, \Gamma, \eta)$
10 $\quad\quad$ `if` $v_n[0]$ does not unify with $E_n[0]$ `then` swap clauses in $v_n$
11 $\quad\quad \eta : s_{i,j} \mapsto var(v_n[2])$ and $\eta : s_{i-1,j} \mapsto var(v_n[3])$
12 $\quad\quad \Gamma' \leftarrow \Gamma' \cup v_n$
13 $\quad\quad F \leftarrow F \setminus \{(\neg x_i \vee s_{i-1,1} \vee s_{i,k})(s_{i-1,k} \vee s_{i,k})\}$
14 $\quad\quad j \leftarrow j + 1$
15 $\quad$ `while` $\eta$ *is defined for* $s_{i,j}, s_{i-1,j}$ *and* $v_n \neq \perp$
16 $\quad i \leftarrow i + 1$
17 $E' \leftarrow$ find non-gate clauses in $F$ or `return` $\perp$
18 `return` $(E, E', \Gamma, \Gamma', \eta)$

Again, not all clauses are distributed over the gates, in line 17 we search for those. The missing clauses corresponding to the *rectangle* are $\neg\eta(s_{1,j})$ for $j \in \{1, \ldots, k\}$.

In the case $k > 1$, i.e. $\Gamma' \neq \emptyset$ in Algorithm 12, we additionally need to verify the existence of the *closing* part and the *floating* gates that belong to a constraint. The search for the *closing* part is a verification that the gates in this subgraph are connected as presented in 7.2 and contain the correct clauses. This can be implemented as a simple, but rather lengthy, case analysis in conjunction with a unification. This part binds two of the last three variables in the constraint, i.e. $\eta$ maps $x_{n-2}$ and $x_{n-1}$ to variables in $F$. We assume that the resulting gates are saved in $\Gamma''$. Previously unmapped auxiliary variables are now also known.

### Part IV

The recognition of the *floating* gates remains subject to the last part.

**Algorithm 13:** Recognize $LT_{SEQ}^{n,k}$ (*floating* part)

```
Input: Boolean Circuit 𝒢 = 𝒢(F), the result (E, E', Γ, Γ', Γ'', η) of the previous
       algorithms
Output: An encoding (E, Γ) of LT_SEQ^{n,k≥2} or LT_SEQ^{n,1}
1  Γ_F ← {G = (o, P, g) ∈ 𝒢(F) : η(x_{n−1}) ∈ P, and c(g) = constraint,
                                  and deg⁺(G) = 0 = deg⁻(G)}
2  return (E, Γ) if |Γ| < k − 2
3  Create a hash table h of size 2 · 3 · (k − 2) and initialize entries in h with (0, ∅)
4  for G = (o, P, g) ∈ Γ_F do
5      for p ∈ P do
6          (count, gates) ← h(p)
7          count ← count + 1
8          gates = gates ∪ G
9  Γ''' ← ∅
10 for (count, gates) ∈ h do
11     if count = 1 then
12         G_f ← ⊥
13         G ← first and only element in gates
14         Find ¬s_{n−1,2} in G[0]
15         for i = 3, . . . , k − 4 do
16             G' ← gate with input s_{n−1,i}
17             Break loop if count ≠ 2 in result of h(s_{n−1,i})
18             Γ''' ← Γ''' ∪ {G'}
19             if i = k − 1 then G_f ← G'
20         if G_f ≠ ⊥ then
21             Update η by iterating over Γ'''
22             Return (E, Γ, Γ', Γ'', Γ''', η) if h(s_{n−2,k−3}) = 1
23         else
24             Γ''' ← ∅
25 return (E, Γ)
```

We remark to line 13 and 15 that the desired input is the negative literal not equal to $\neg\eta(x_{n-1})$.

This algorithm first finds a superset of potential gates of interest by checking the requirements. If there are enough gates, the algorithm continues beyond line 2, otherwise we

return the gates and clauses belonging to the $k = 1$ case.

After this sieve we allocate a hash table to save $count$, which is the number of times $p$ is an input variable, and $gates$ which is the set of gates that have $p$ as an input.

The last loop finds potential start inputs, and tries to find the necessary amount ($k - 2$ many) gates that have the structure of the *floating* gates. After the sieve in line 1 a search for the desired input variable in each gate is sufficient.

### Putting it Together

If all algorithms complete we find the a clause that unifies with $(\neg x_n^* \vee \neg\eta(s_{n-1,k}))$ and update $\eta$. Now $\eta$ is a unificator

$$\{x_1, \ldots, x_n\} \cup \{s_{i,j}\}_{1 \leq i < n, 1 \leq j \leq k} \to E$$

such that $E \subseteq F$ is an encoding of the constraint $\leq k(x_1, \ldots, x_n)$ with auxiliary variables $s_{i,j}$. To obtain this map from the previous four algorithms we execute them in order, passing the output of the previous to the input of the next algorithm. However, if any part fails, i.e. it returns $\perp$, we cannot conclude the absence of a constraint. This is due to the fact that the structure given in Figure 7.2 only takes a consistent form for $n \geq 7$ and $k \neq 2$.

The cases with $n < 7$ can be handled by case analysis based on the examples in 9.

The case $k = 2$ is handle as a special case or, for larger $n$, as follows. For $k = 2$ we recognize the $k1$-column with the corresponding algorithm. Then it remains to check whether the base gate has an outgoing edge to the first column of the *rectangle*, but with the output of the last gate being the input of the last gate in the $k1$-column.

### Time Complexity

Let $m$ be the number of gates in $\mathcal{G}(F)$.

**(Part I)**: Finding a start gate (line 2) requires searching all gates. Accessing the classification result takes constant time. The while-loop iterates at most once over all gates. The number and length of clauses in gates is bound by small constants. The map $\eta$ is accessed (find and insert) for each gate, each access taking logarithmic time, and at most all variables in $F$ are mapped. Altogether, the first part is bound by $\mathcal{O}(m \cdot \log(|vars(F)|)$.

**(Part II)**: We iterate at most once over the complete *rectangle*. As in Part I, $\eta$ is accessed for each gate. Altogether, we see the second part is bound by $\mathcal{O}(m \cdot \log(|vars(F)|)$ as well.

**(Part III)**: The case analysis of this part takes constant time. And there are three insertions to $\eta$. Altogether, the third part is bound by $\log(|vars(F)|$.

**(Part IV)**: The size of the set found in line 1 takes one iteration over all gates and its size is bound by $m$. The for-loop in line 4 thus also iterates over at most $m - (k - 4)$ many gates. The number of inputs is bound by half the size of the hash table, which is

$3 \cdot (k-2)$. The for-loop in line 10 takes at most $3 \cdot (k-2)$ many iterations as well, each access to the hash table taking constant time. Finding the desired variables takes constant time. Accessing $\eta$ for each of the at most $m$ gates takes logarithmic time, similarly as in the previous parts. Altogether, the fourth part is also bound by $\mathcal{O}(m \cdot k \cdot \log(|vars(F)|)$.

**(Case Analysis)**: The case analysis described to recognize smaller instances of the constraint can be covered in a constant number of cases.

### Soundness and Expectations

Our graph based algorithm returns $\perp$ if the exact $k$ cannot be determined and is thus sound. It is also perceivable that not all constrained variables can be recognized but a valid $k$ can be determined. This, again, is due to the non-confluence of the gate recognition procedure used. In this case gates corresponding to $n > k$ are found, but gates corresponding to other constrained variables are separated from the $k1$-column.

### 7.2.2 Syntactical Recognition

As we've observed in previous examples, the constraint recognition procedure described in possibly fails in cases where ambiguity is possible. More specifically, the gates in the $k1$-column may not appear in a column of this structure at all. If other gates contain the constrained variables and the Boolean circuit is this not reconstructed as expected, then the soundness criterion leads to a termination of our graph based approach, as no complete constraint can be found. However, the auxiliary variables introduced in the encoding $LT_{SEQ}^{n,k}$ only appear in the constraint itself, limiting the possible reconstructed graphs, even if the gates of the $k1$-column are spread elsewhere.

Alternatively, to working on a reconstructed graph, Algorithm 8 allows to build an algorithm that consecutively unifies clauses, to find a subformula in a CNF that matches that of a constraint. This recognition algorithm only implicitly exploits the gate structure by constructing the next, expected clauses to unify, on demand. Motivated by this we sketch the pseudocode for the case $k = 1$, expecting to be able to fix the mentioned problem of the splitting of the $k1$-column, at least in theory.

Once more we base our algorithm on the recursive forms in 3. The first unification based algorithm finds a clause corresponding to the base case $LT_{SEQ}^{2,1}$.

**Algorithm 14:** Find Base Clause

`Input`: A formula $F$ in CNF Tuple Notation

1 `return` $(c_1, c_2) \in F^2$ for which holds that $c_1$ unifies with $(\neg x_1^* \vee s_{1,1}^*)$ and $c_2$ unifies with $(\neg x_2^* \vee \neg s_{1,1})$

We restrict ourselves to the $k = 1$ case. Given the result of Algorithm 14 and the case $LT_{SEQ}^{n+1,k}$ for $k = 1$ we can construct a two part recognition of sequential counter encodings.

This algorithm is potentially very slow. However, we expect the tuple notation and the underlying gate structure to restrict the search space dramatically in each step. A fixed literal appears in each clause to unify. Additionally, the auxiliary variables only appear in the constraint and thus in a small amount of clauses. This raises the question whether this algorithm performs acceptably given real-world instances.

The main interest in this approach is the possibility to fix the aforementioned problems with the $k1$-column. We approach this as follows.

A blockedness and right-uniqueness check shows that the same gates as previously seen in the $k = 1$ case can be extracted in a single iteration. This ensures that we have a predictable reconstruction of a partial formula. Further we can now initialize $\Gamma$ in Algorithm 2 with the gates found, to reconstruct a Boolean circuit out of the complete formula $F$, all while ensuring the problematic clauses are distributed in the expected manner. To complete the recognition we can execute the graph-based recognition algorithm on the formula, given the already known (partial) constraint to ensure the soundness of this approach.

**Algorithm 15:** Find $LT_{SEQ}^{n,1}$ Clauses (Base)

`Input`: A formula $F$ in CNF Tuple Notation, and the base clauses $(c_1, c_2)$
`Input`: A unificator $\eta$ to an encoded constraint

$\eta : x_2 \mapsto var(c_2[0])$
$\eta : s_{1,1} \mapsto var(c_1[1])$
`for` *clause* $c_{i-1} \in F$ *that unifies with* $(\neg s_{1,1} \vee s_{2,1}^*)$ `do`
  $\eta : s_{2,1} \mapsto c_{i-1}[1]$
  `if` *there is a clause* $c = (\neg\eta(x_2) \vee \eta(s_{2,1})) \in F$ `then`
    Remove $c$ from $F$
  `else`
    Continue for-loop
  `if` *there is a clause* $c \in F$ *that unifies with* $(\neg x_3^* \vee \neg s_{2,1})$ `then`
    Call the recursive case with $F$, $\eta$ and $i = 3$ and return the result // $n > 2$ case
  `else`
    `return` $\eta$ // $n = 2$ case
`return` $\eta$

**Algorithm 16:** Find $LT_{SEQ}^{n,1}$ Clauses (Recursive)

`Input`: A formula $F$ in CNF Tuple Notation, the map $\eta$ from the base algorithm, the number $i$ of variables constrained

`for` $c_1 \in F$ *that unifies with* $(\neg s_{n-1,1} \vee s_{n,1})^*$ `do`
  $\eta : s_{n,1} \mapsto var(c_1[1])$
  `if` *there is a clause* $c_2 \in F$ *that unifies with* $(\neg x_n^* \vee s_{n,1})$ `then`
    $\eta : x_i \mapsto$ `if` *there is a clause* $c_3 \in F$ *that unifies with* $(\neg x_{i+1}^* \vee s_{i,1})$ `then`
      $\eta : x_i \mapsto var(c_3[0])$
      `return` the result of this function called with $F$, $\eta$ and $i + 1$
  Remove mappings added in this call of the algorithm
  `return` $\eta$ // $n = i$ case
`return` $\eta$

# 8 Implementation and Evaluation

### Implementation

A part of the algorithms presented in this thesis have been implemented in a branch of the modular SAT-solver Candy [1]. All algorithms that employ hierarchical gate recognition use the respective implementation in Candy.

The implementation includes a visualizer with additional styling and the option to render input variables. The implemented classifier uses the criteria described in chapter 5.2 and maps the result to a fixed set of functions saved in an enumeration. Lastly, a recognition algorithm has been implemented. This implementation however only finds the described subgraph, no full unificator is returned and neither is the formula searched for the non-gate clauses.

## 8.1 Experimental Setup and Problem Instances

### Experimental Setup

All tests were executed on a computer running Ubuntu 18.04.5 LTS on an Intel Core i5-5300U CPU @ 2.30GHz (2 cores / 4 threads) with 8GB of DDR3 RAM clocked at 1600 MHz.

### Problem Classes

We devise three classes of problems against which we test our recognition algorithm. The first class is the set of formulas consisting of only the sequential counter encoding, $\mathcal{T}_1 = \{LT_{SEQ}^{n,k}\}$ for small $n$ and $k$. The class $\mathcal{T}_1$ is used to verify that the algorithm works and is also used for a performance evaluation.

The second class models a controlled, practical test. We choose a real-world instance and randomly select variables to constrain from the formula. The set of test-cases is $\mathcal{T}_2 = \{F\} \cup \{LT_{SEQ}^{n,k}(x_1, \ldots, x_n)\}$ for a formula $F$ without any encoded constraints.

The third class $\mathcal{T}_3$ consists of a set of real-world instances for which we know that they contain sequential counter encodings [6]. However, we do not know which variables in the formula are constrained.

| | | |
|---|---|---|
| (01) | 02223564bd2f5c20768e63cf28c785e3 | mp1-squ_ali_s10x10_c39_abio_SAT |
| (02) | 166e1e5a9f63fcf94ddae8533fa2a090 | mp1-squ_any_s09x07_c27_abix_UNS |
| (03) | 1bda6f076bbed75ed4250946919446a6 | mp1-squ_any_s09x07_c27_sinx_UNS |
| (04) | 22fdbcb094a1c1b513b8fd90e9b50f65 | mp1-squ_any_s09x07_c27_sinz_UNS |
| (05) | 37fde221cfa265eb7ec2c221e5272686 | squ_any_s09x07_c27_abio_UNS-sc2017 |
| (06) | 44f9463b7fb3a23027aa4520f3bafe61 | mp1-squ_ali_s10x10_c39_sinx_SAT |
| (07) | 4c9b7b964c5b2b540ee5e09b3a5f3c8e | mp1-tri_ali_s11_c35_sinz_UNS |
| (08) | 5a0120a44efe84c0cb4e58cebc3c4982 | mp1-squ_any_s09x07_c27_bail_UNS |
| (09) | 6587d5077bc6962cf8939636eeb57c35 | mp1-tri_ali_s11_c35_abio_UNS |
| (10) | 67158b829ad61fbc3ae12f9fafe2de3c | squ_ali_s10x10_c39_abix_SAT-sc2017 |
| (11) | 6965d5369d3ab26daaf074303c3d1739 | mp1-squ_ali_s10x10_c39_abix_SAT |
| (12) | 703ca000646a0789dfb0939aae936a6b | mp1-squ_any_s09x07_c27_abio_UNS |
| (13) | b023ea0eb9adbd9182c01d7e2d443289 | mp1-tri_ali_s11_c35_bail_UNS |
| (14) | b09e9e8cf2f55863d9e27dad49e1a3b7 | mp1-tri_ali_s11_c35_sinx_UNS |
| (15) | b1c44904cf06682f973b60c8282f45e8 | mp1-squ_ali_s10x10_c39_bail_SAT |
| (16) | f293dc11e842227dd1a49fe1571bcb8b | mp1-tri_ali_s11_c35_abix_UNS |
| (17) | fc5605315fa468603aff94a24fc41272 | mp1-squ_ali_s10x10_c39_sinz_SAT |

**Table 8.1:** Problem Instances

The instances have been obtained from the Global Benchmark Database [2] through the query 'author like %wynn%', the obtained instances are identified below by their gbd-hash and their given name in Table 8.1.

## Evaluation

**(Correctness and Performance Test)**. As expected the algorithm recognizes all constraints in $\mathcal{T}_1$ for $2 \leq n \leq 40$ and $1 \leq k < n$. Figure 8.1 plots the number of gates against the time needed to initialize the recognizer and execute the recognition itself.

Quite clearly the time needed for the recognition is linear in the number of gates.

**(Random Variable Constraining Test)**. For this test we chose the problem instance mp1-squ_ali_s10x10_c39_abio_SAT and removed all recognized constraints. The result is the formula $F$ used in $\mathcal{T}_2$. For each $k \in \{1, 2, 4, 8\}$ and for $2 \leq n \leq 40$ we randomly choose variables in $vars(F)$ and added the constraint to the formula. As we know exactly which constraint has been added, we can assess the proportion of constraints recognized.

The result of this test shows that no constraint could be found.

**(Real-World Instance Search)**. The problem instances given in Table 8.1 were searched for constraints. We obtained the following result:

- In (10), (13) and (15) we found nothing.

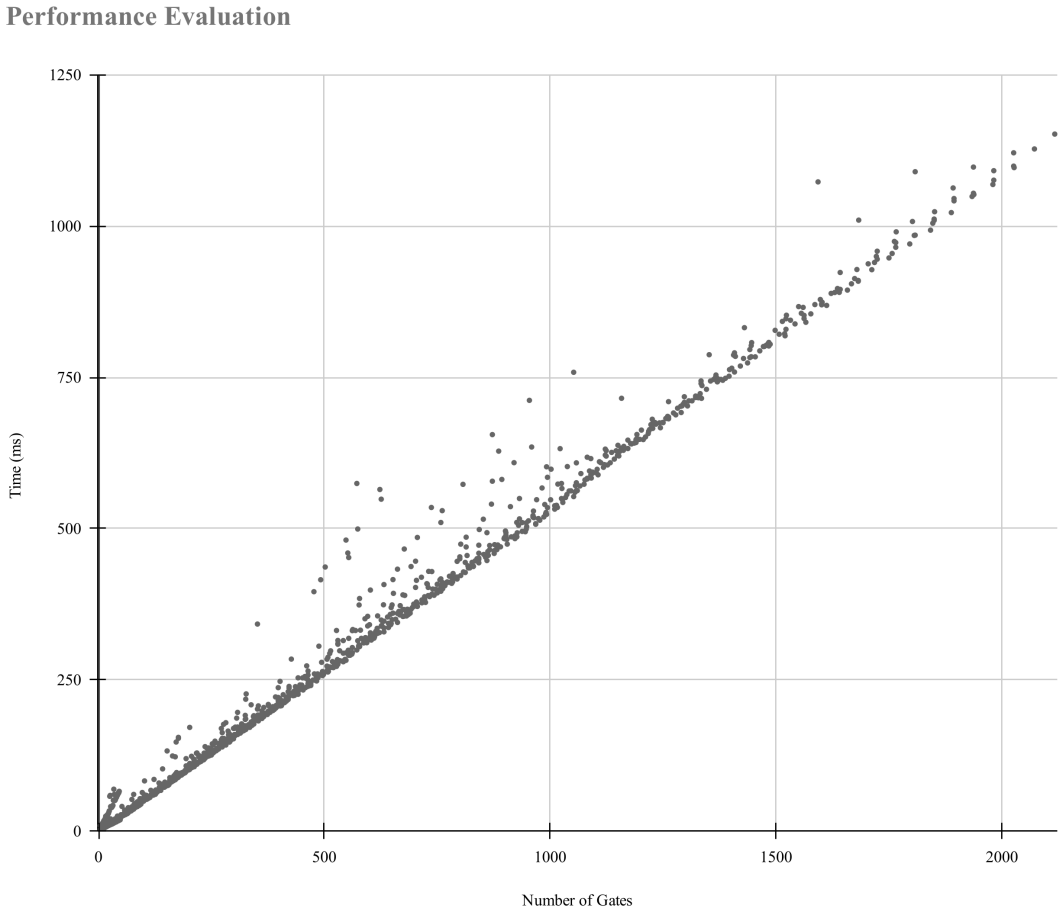



**Figure 8.1:** Constraint Recognizer Performance

- In (01), (05) and (11) we found a single constraint with one gate.
- In (02), (08) and (12) we found two constraints with a single gate.
- In (09) and (16) we found 3 constraints each with one gate
- In (07) we found a single constraint with 31 gates.
- In (04) we found a single constraint with 36 gates.
- In (17) we found a single constraint with 61 gates.
- In (14) we found 30 constraints each with one gate and single constraint with 31 gates.
- In (03) we found 35 constraints each with one gate and single constraint with 36 gates.
- In (06) we found 60 constraints with a single gate and one constraint with 61 gates.

## Summary

The first test was to verify that the implemented algorithm works and at least finds the constraints when they are given in their plain form. The remaining two tests are interesting, especially under comparison. Randomly choosing variables and constraining them leads to the worst ratio of gates recovered. Apparently, the clauses of every constraint added were distributed over the reconstructed Boolean circuit in a manner that made recognition unlikely. However, the recognition of constraints with our algorithm is not impossible, as the real-world instances show. Even though most constraints found were of small size, some large constraints were recognized in the formulas.

# 9 Discussion, Future Work and Conclusion

## 9.1 Discussion and Future Work

Throughout this thesis we encountered the problem of non-confluence. Any (constraint) recognition algorithm that relies on a specific structure of the Boolean circuit resulting from a non-confluent gate recognition procedure likely breaks. As the number of possible reconstructions is not known in the instance of hierarchical gate recognition, the likelihood of any such reconstruction being unsuitable to a specific recognition procedure is subject to experimentation. This further raises the question of how much performance improvement can be expected from the exploitation of additional knowledge under the considerable effort necessary to extract this information.

Our constraint recognition algorithm requires little, additional time to recognize a constraint. However, the proportion of constraints recognized versus the constraint known to be encoded is unsatisfying. To counter this we suggested an additional, purely syntactic algorithm based on unification to extract the clauses corresponding to gates that possibly would not appear and suggested adding them to the set of known gates in the hierarchical gate recognition algorithm. It is not obvious from our standpoint, whether a purely syntactic approach delivers satisfying performance in real-world instances. We suggest an implementation of this approach and its evaluation as subject to future work.

If such an algorithm proves performant enough for practical applications several more questions arise. Firstly, we may ask whether it is then practical to recognize not only partial, but complete constraints with this method. This would motivate an algorithm that only requires an encoder for the recognition and no other special construction by a developer. Vastly improving the time necessary to develop a recognition algorithm.

Secondly, if the recognition of a complete constraint is not sensible, a general procedure to determine a minimal set of gates which arise from encoded constraints that are subject to possible ambiguity in a reconstruction. It is perceivable that all clauses corresponding to the gates of a constraint are in close relationship with other clauses, making a suitable reconstruction intractable.

In the concrete case of sequential counter constraints we observed that not all clauses of an encoding are distributed over gates. Hence, if algorithms unifying partial formulas appear to be unsuitable for practical purposes, it is to be expected that recognition procedures

on other constraints are unsuitable for real-world usage as well.

As non-confluence is a central problem for all graph-based recognition approaches it would be desirable to develop a confluent gate recognition procedure. However, it is not clear whether such a recognition procedure can be powerful enough to allow the efficient recognition of a constraint, or further of a subcircuit. We briefly discussed trivial gate recognition to illuminate this problem.

## 9.2 Conclusion

With the general interest in mind to be able to visualize the result of gate recognition procedures, we implemented a visualizer that employs a common tool to render graphs to images. With the intent of experimentation we wrote a sequential counter constraint encoder and rendered instances with our visualizer. Based on these visualizations and a recursive form of the encoding we derived desirable properties of constraint recognition algorithms. Which we took to construct a general recognition procedure, which we completed by covering a number of non-consistent reconstructions by case analysis. This approach required the combination of graph and syntactical algorithms. Nonetheless, the necessity for even further work on the recognition became apparent, as the problem of non-confluence of the used gate recognition procedure poses a problem.

# Bibliography


[1] Candy - a modular sat-solver.

[2] Global benchmark database.

[3] Satisfiability suggested format, 1993.

[4] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J. Stuckey. To encode or to propagate? the best choice for each constraint in sat. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, pages 97–106, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[5] Olivier Bailleux and Yacine Boufkhad. Efficient cnf encoding of boolean cardinality constraints. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, pages 108–122, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[6] Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, and Carsten Sinz, editors. *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, Finland, 2012.

[7] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[8] Alan Mishchenko Baruch Sterin Robert Brayton. Structural reverse engineering of arithmetic circuits.

[9] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[10] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[11] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003.

[12] B. R. Heap. Permutations by Interchanges. *The Computer Journal*, 6(3):293–298, 11 1963.

[13] Markus Iser. Recognition and exploitation of gate structure in sat solving. 2020.

[14] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Saïs. Recovering and exploiting structural knowledge from cnf formulas. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 185–199, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[15] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293 – 304, 1986.

[16] Jarrod Roy, Igor Markov, and Valeria Bertacco. Restoring circuit structure from sat instances. 05 2004.

[17] Carsten Sinz. Towards an optimal cnf encoding of boolean cardinality constraints. pages 827–831, 10 2005.

[18] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[19] Joost P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63 – 69, 1998.

[20] Ed Wynn. A comparison of encodings for cardinality constraints in a sat solver. 2018.