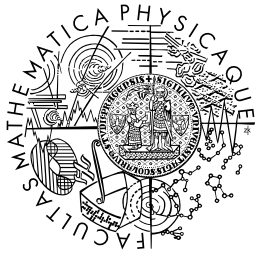


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Miloš Chromý

Boolean techniques in Knowledge representation

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the doctoral thesis:	Doc. RNDr. Ondřej Čepek, Ph.D.
Study programme:	Theoretical Computer Science and Artificial Intelligence
Study branch:	P4I1

Prague 2020

I declare that I carried out this doctoral thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to express my gratitude to my supervisor doc. RNDr. Ondřej Čepek, Ph.D. for a mentoring and guidance during preparation of this Thesis and also for his support through my whole doctoral studies. I would also like to thank RNDr. Petr Kučera, Ph.D. which provided a guidance through finalizing the results from my Diploma Thesis. Additionally i would like to thank my friends Tomáš Karella and Jan Tomášek for a proofread of this Thesis. Finally i want to thank my family, girlfriend and friends for their endorsement during my studies and in pursuing the research topics and also for a great emotional support.

Title: Boolean techniques in Knowledge representation

Author: Miloš Chromý

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Doc. RNDr. Ondřej Čepek, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: In this thesis we will investigate switch-list representations of Boolean function and we will explore the biclique satisfiable formulas.

Given a truth table representation of a Boolean function f the switch-list representation of f is a list of Boolean vectors from the truth table which have a different function value than the preceding Boolean vector in the truth table. We include this type of representation in the Knowledge Compilation Map [Darwiche and Marquis, 2002] and argue that switch-lists may in certain situations constitute a reasonable choice for a target language in knowledge compilation. First, we compare switch-list representations with a number of standard representations (such as CNF, DNF, and OBDD) with respect to their relative succinctness. As a by-product of this analysis we also give a short proof of a long standing open question from [Darwiche and Marquis, 2002], namely the incomparability of MODS (models) and PI (prime implicates) representations. Next, using the succinctness result between switch-lists and OBDDs, we develop a polynomial time compilation algorithm from switch-lists to OBDDs. To finalize the switch-list representation analysis, we describe which standard transformations and queries (those considered in [Darwiche and Marquis, 2002]) can be performed in polynomial time with respect to the size of the input if the input knowledge is represented by a switch-list. We show that this collection is very broad and the combination of polynomial time transformations and queries is quite unique. Some of the queries can be answered directly using the switch-list input, others require a compilation of the input to OBDD representations which are then used to answer the queries.

A biclique satisfiable formula is a CNF formula whose incidence graph admits a cover by distinct bounded bicliques. The class of biclique satisfiable formulas was introduced in [Szeider, 2005] where it was also shown that it is NP-complete to check if a formula is biclique satisfiable. In [Chromý, 2015], a heuristic for checking biclique satisfiability was introduced and it was experimentally checked that property of being biclique satisfiable exhibits a phase transition behaviour. The heuristic algorithm presented in [Chromý, 2015] runs in polynomial time, but it is incomplete. In this thesis, we propose a SAT based approach to checking biclique satisfiability which is complete, but not polynomial time. We compare both approaches experimentally.

Keywords: Boolean functions, Knowledge compilation, Switch-list representation, Interval representation, Biclique satisfiability, SAT encoding

Contents

1. Introduction

A Boolean function on n variables is a mapping from $\{0,1\}^n$ to $\{0,1\}$. We often associate 0 with *False* and 1 with *True*. This concept naturally appears and is extensively used in several areas of mathematics, computer science and many other fields. Let us start by describing different ways how a Boolean function can be represented together with the origins of these representations and some applications.

The most straightforward way to represent a Boolean function on n variables is a *Truth table (TT)* where we list all 2^n possible assignment and associate a function value with each assignment.

x_3	x_2	x_1	$f(x)$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Figure 1.1: A truth table of function f.

An obvious drawback of this representation is its size that is prohibitive for large values of n. For a function f, any assignment x such that $f(x) = 1$ is called a *model* of f and any assignment x such that $f(x) = 0$ is a *non model* of f. A *MODS representation* of f is an abbreviation of the truth table, where we list only the models of f.

x_3	x_2	x_1
0	0	0
0	0	1
0	1	1
1	0	0
1	0	1
1	1	0

Figure 1.2: A list of all models of f.

In Chapter 2 we will study two more representations each of which is also a compression of the truth table: interval representations and switch-list representations. Let f be a Boolean function and let us fix some order of its n variables. The input binary vectors can be now thought of as binary numbers (with bits in the prescribed order) ranging from 0 to $2^n - 1$. An *interval representation* is then an abbreviated MODS representation, where instead of writing out all the models, we write out only the first and the last model in every interval of consecutive models. Function f is then represented by an ordered list of such pairs $[x, y]$ of integers, each pair specifying one maximal interval of models.

x_3	x_2	x_1	
0	0	0	$[0,1]$
0	0	1	
0	1	1	$[3,6]$
1	1	0	

Figure 1.3: An interval representation of f.

A *switch-list representation* is a list of those vectors that have a different function value from their predecessor in the natural ordering of binary vectors. To prevent ambiguity the function value $f(0)$ at the all-zero vector is stored in the representation as well.

x_3	x_2	x_1
0	1	0
0	1	1
1	1	1

Figure 1.4: A list of switches that together with $f(0) = 1$ represents f.

Representations of Boolean functions arising from logic are *Boolean formulas*. Function f on n variables $x_1, \ldots, x_n$ can be written as a sequence of symbols from the following set: variables, the unary operator *negation* $\neg$, the binary operators *disjunction* ($\vee$, logical or) and *conjunction* ($\wedge$, logical and), and parentheses to disambiguate the priority of operations. A *literal* is a variable or its negation. We use a notation $\bigvee_I l_i$ ($\bigwedge_I l_i$) as a disjunction (conjunction) of literals l_i for $i \in I$. Note that we can also use other operators such as implication $\Rightarrow$, XOR $\oplus$, equivalence $\Leftrightarrow$, NAND, NOR and others. By using these operators we can get a shorter formula (see Figure 1.5), however it is possible to rewrite any Boolean formula into a logically equivalent Boolean formula using only negation, conjunction and disjunction.

$$f = \oplus_{i\in[n]} x_i = \bigvee_{I\subseteq[n],|I| isodd} (\bigwedge_{i\in I} x_i \wedge \bigwedge_{j\notin I} \neg x_j)$$

Figure 1.5: A Boolean formula of parity function.

Boolean formulas can be used to describe a logic system and to perform reasoning within such system. We can simply interpret formulas in this system in a natural language as in the example in Figure 1.6.

A conjunction (disjunction) is called *simple* if it contains every variable at most once. A *clause* is a simple disjunction of literals $C = \bigvee_I l_i$. A *term* is a simple conjunction of literals $T = \bigwedge_I l_i$. A *Conjunctive normal form (CNF)* of function f (or simply CNF of f) is a conjunction of clauses $\bigwedge_J(C_j) = \bigwedge_J(\bigvee_{I_j} l_i)$ representing f. Similarly, a *Disjunctive normal form (DNF)* of function f (or simply DNF of f) is a disjunction of terms of literals $\bigvee_J T_j = \bigvee_J(\bigwedge_{I_j} l_i)$ representing f.

An *implicant* of a Boolean function f is a term T such that $T \implies f$ holds. A *prime implicant* T is an implicant that cannot be *subsumed* by any other implicant

If the pressure is low then the cyclone will appear.	$p_{low} \Rightarrow c$	$\neg p_{low} \wedge c$
If the pressure is high then the anticyclone will appear.	$p_{hi} \Rightarrow ac$	$\neg p_{hi} \wedge ac$
If the cyclone appears then it will be cloudy.	$c \Rightarrow w_c$	$\neg c \wedge w_c$
If the anticyclone appears then it will be sunny.	$ac \Rightarrow w_s$	$\neg ac \wedge w_s$
If the arrow is up then the pressure is high.	$a_\uparrow \Rightarrow p_{hi}$	$\neg a_\uparrow \wedge p_{hi}$
If the arrow is down the pressure is low.	$a_\downarrow \Rightarrow p_{low}$	$\neg a_\downarrow \wedge p_{low}$
The arrow is up, what we can derive?	$a_\uparrow = 1$	
The arrow is up, hence the pressure is high, hence the anticyclone will appear and hence it will be sunny.	$a_\uparrow \Rightarrow p_{hi} \Rightarrow ac \Rightarrow w_s$	

Figure 1.6: A weather forecasting base.

of f (i.e. there exists no implicant T' of f such that $T' \subset T$). An *implicate* of f is a clause C such that $f \implies C$. An implicate C is a *prime implicate* of f if it cannot be subsumed by any other implicate of f (i.e. there exists no implicate C' of f such that $C' \subset C$). In Chapter 2 we will be studying representations of Boolean functions that consist of complete lists of all prime implicants or all prime implicates of a given function.

Apart from logic, another historical motivation for Boolean functions comes from electrical engineering. Other areas where Boolean functions are used include game theory, reliability theory and combinatorics [Crama and Hammer, 2011]. A logical gate is an abstract or physical device realizing a simple Boolean function of bounded arity such as conjunction or disjunction of arity two. By connecting outputs of some gates to inputs of other gates way may create a circuit. Such circuits are used to store data in an electronic memory (such as RAM or FLASH), and to compute mathematical functions in GPUs and CPUs. A *Boolean circuit* representing a Boolean function on n variables is an acyclic circuit with n inputs and a single output.

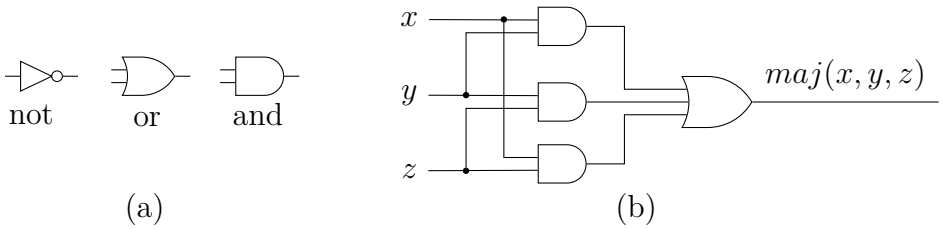


Figure 1.7: (a) Logical gates realizing functions and ($\wedge$), or ($\vee$), not ($\neg$). (b) Boolean circuit of a function majority(x,y,z).

A *Binary Decision Diagram (BDD)* is a rooted directed graph with two terminals labeled 0 and 1. Each non-terminal node is a decision node with exactly two outgoing edges. Each decision node corresponds to a propositional variable and the two outgoing edges correspond to the assignments of 0 and 1 to this variable. Each directed path from the root to a terminal thus corresponds to a (possibly partial) assignment of truth values to variables and the terminal specifies the function value for such an assignment.

Let $<$ be a total order on the set PS of all propositional variables (we assume this set to be denumerable). An *Ordered Binary Decision Diagram (OBDD)* with

respect to $<$ is a BDD such that on every path from the root to a terminal no two decision nodes correspond to the same variable and moreover every such path respects the prescribed order $<$. The second condition means that there does not exist a directed path p from the root to a terminal and two variables $x < y$, such that the decision node corresponding to y precedes the decision node corresponding to x on path p.

Binary decision diagrams and its subclasses are important both in theory (space complexity of a BDD representation and parallel complexity are closely related) and in applications (Verification of sequential networks, Electrical circuits) [Wegener, 2000].

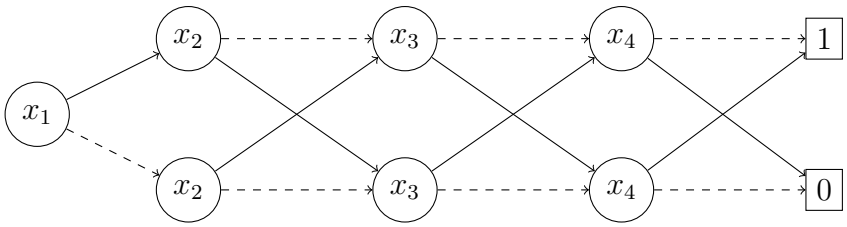


Figure 1.8: OBDD of a parity function.

The main body of the dissertation is divided into two parts. Chapter 2 deals with switch-list representations of Boolean functions and is based on [Chromý, 2019, Čepek and Chromý, 2018, 2020a,b] The main aim of Chapter 2 is threefold:

1. to compare switch-list representations with many other standard representations of Boolean functions with respect to relative succinctness,

2. to derive the complexity of standard queries such as consistency check, validity check, implicant check, or model counting where performed on a switch-list representation, and

3. to derive the complexity of standard transformations such as conjunction, disjunction, conditioning, or forgetting where all inputs are represented by switch-list representations.

Chapter 3 investigates Boolean functions representable by special subsets of CNFs defined by their graph properties, namely the classes of matched and biclique satisfiable formulas. Unlike Chapter 2 that is a theoretical work, the principal results in this chapter are heuristics that are verified experimentally. Chapter 3 builds on the master thesis Chromý [2015] but mainly presents a newer material from [Chromý and Kučera, 2016, Chromý and Kučera, 2019].

2. Switch-list representations

This chapter is based on a full version of a conference extended abstract [Čepek and Chromý, 2020a] and also includes the compilation algorithm form a switch-list representation to OBDD with a prescribed order first presented in [Čepek and Chromý, 2020b].

The task of transforming one of the representations of a given function f into another representation of f (e.g. transforming a DNF representation into an OBDD or a DNNF into a CNF) is called knowledge compilation. For a comprehensive review paper on knowledge compilation see [Darwiche and Marquis, 2002], where a Knowledge Compilation Map is introduced. Knowledge Compilation Map systematically investigates different representation languages with respect to

1. their relative succinctness
2. the complexity of common transformations, and
3. the complexity of common queries.

The *succinctness* of representations roughly speaking describes, how large the output representation in language B is with respect to the size of the input representation in language A when compiling from A to B. A precise definition of this notion will be given later in this chapter. Transformations include negation, conjunction, disjunction, conditioning, and forgetting. The complexity of such transformations may differ dramatically from trivial to NP-hard depending on the chosen representation language. The same is true for queries such as consistency check, validity check, clausal and sentential entailment, equivalence check, model counting, and model enumeration.

In [Le Berre et al., 2018] the authors included Pseudo-Boolean constraint (PBC) and Cardinality constraint (CARD) languages into Knowledge Compilation Map by showing succinctness relations among PBC, CARD, and languages already in the map, and by proving the complexity status of all queries and transformations introduced in [Darwiche and Marquis, 2002]. In this paper we aim at achieving exactly the same goal for switch-list representations.

Interval representations of Boolean functions were introduced in [Schieber et al., 2005], where the input was considered to be a function represented by a single interval (two n-bit numbers x, y) and the output was a DNF representing the same Boolean function f on n variables, i.e. a function which is true exactly on binary vectors (numbers) from the interval $[x, y]$. This knowledge compilation task originated from the field of automatic generation of test patterns for hardware verification [Lewin et al., 1995, Huang and Cheng, 1999]. In fact, the paper [Schieber et al., 2005] achieves more than just finding some DNF representation of the input 1-interval function — it finds in polynomial time the shortest such DNF, where "shortest" means a DNF with the least number of terms. Thus [Schieber et al., 2005] combines a knowledge compilation problem (transforming an interval representation into a DNF representation) with a knowledge compression problem (finding the shortest DNF representation).

In [Čepek et al., 2008] the reverse knowledge compilation problem was considered. Given a DNF, test if all models form a single interval under some permutation of variables, and in the affirmative case output the permutation and the two n-bit numbers defining the interval (note, that changing the order of variables may dramatically change the length of interval representations from $O(n)$ to $\Omega(2^n)$ — see Section 2.2 for examples). This problem can be easily shown to be co-NP hard in general (it contains tautology testing for DNFs as a subproblem), but was shown in [Čepek et al., 2008] to be polynomially solvable for tractable classes of DNFs (where tractable means that DNF falsifiability can be decided in polynomial time for the inputs from the given class). The algorithm presented in [Čepek et al., 2008] runs in $O(n\ell f(n,\ell))$ time, where n is the number of variables and ℓ the total number of literals in the input DNF, while $f(n,\ell)$ is the time complexity of falsifiability testing on a DNF on at most n variables with at most ℓ total literals. This algorithm serves as a recognition algorithm for 1-interval functions given by tractable DNFs. This result was later extended in [Kronus and Čepek, 2008] to monotone 2-interval functions, where an $O(\ell)$ recognition algorithm for the mentioned class was designed. Recently, these results were further extended to k-interval functions for arbitrary k (a function is k-interval if there exists a permutation of variables for which the interval representation consist of at most k intervals). Paper [Čepek and Hušek, 2017] presents a recognition algorithm which runs in polynomial time in the length of the input DNF for any constant k (the complexity is exponential in k).

Switch-list representations have an added advantage over the interval representations, namely that a function and its negation have the same switch-lists and the two representations differ only by the opposite values of $f(0,0,\ldots,0)$. The ease of negating for switch-list representations allows a trivial translation of any result relating switch-list representation and DNF to a result relating switch-list representation and CNF (and vice versa). This is due to the fact that given a DNF, its logical negation can be represented by a CNF of the same length (and vice versa), and the transformation is purely mechanical (replace disjunctions by conjunctions, conjunctions by disjunctions, and negate all literals). Negating is not as straightforward for interval representations, where the interval representations of a function and its negation may significantly differ, and even the number of intervals may be different (although the difference is at most one). For this reason, we shall use switch-list representations throughout this chapter. It is not a limiting assumption in any way: clearly, the list of intervals can be easily compiled in linear time from the list of switches and the function value $f(0,0,\ldots,0)$, and vice versa. In fact, we shall use this compilation between switch-list representations and interval representations later in this chapter, since some algorithms (e.g. the one for the forgetting transformation) are more naturally designed for interval representations than for switch-list representations.

The languages of switch-list representations and interval representations may be in some situations quite a good choice of a target compilation language. In the succinctness map these languages are placed strictly above TT and MODS, incomparable to prime implicates (PI) and prime implicants (IP), and strictly below CNF, DNF, and OBDD languages. However, compared to CNF, DNF, and OBDD (and even IP and PI) they have a wider set of supported queries and transformations.

Switch-list representations support all the queries from [Darwiche and Marquis, 2002] in polynomial time (see Table 2.1), which is of course better than CNFs and DNFs but also better than IP and PI which do not support model counting. It is also better than general OBDDs which do not support sentential entailment if the two input OBDDs respect different orders of variables. The only language considered in [Darwiche and Marquis, 2002] with the same set of supported queries is the language of OBDDs with a fixed order of variables. Hence, the advantage of switch-list representation is, that it does not require the same order of variables for all inputs to guarantee polynomial time bounds on all queries. Moreover, an added advantage of switch-list representations lies in a computational simplicity of answering most queries. Validity and consistency checks are trivial (constant time), clausal entailment, implicant check, and model counting take linear time (w.r.t. the input size), and model enumeration takes linear time w.r.t. the output size. The only queries that are time consuming are sentential entailment and equivalence check. We do not have direct algorithms that manipulate switch-list representations for these queries at the present moment, instead we compile both input switch-list representations into OBDDs (both respecting the same order of variables) and run the query algorithms for these OBDDs. The polynomial time compilation algorithm from a switch-list representation to an OBDD with different variable orders on the input and on the output is one of the main contributions of this chapter. Finding direct algorithms for sentential entailment and equivalence check which would avoid compilation into OBDDs may be a good research topic.

L	CO	VA	CE	IM	EQ	SE	CT	ME
NNF	○	○	○	○	○	○	○	○
DNNF	X	○	X	○	○	○	○	X
d-NNF	○	○	○	○	○	○	○	○
BDD	○	○	○	○	○	○	○	○
FBDD	X	X	X	?	○	○	X	X
OBDD	X	X	X	X	X	○	X	X
OBDD$_<$	X	X	X	X	X	X	X	X
CNF	○	X	○	X	○	○	○	○
DNF	X	○	X	○	○	○	○	X
IP	X	X	X	X	X	X	○	X
MODS	X	X	X	X	X	X	X	X
CARD	○	X	○	X	○	○	○	○
PBC	○	X	○	X	○	○	○	○
SL	X	X	X	X	X	X	X	X
SL$_<$	X	X	X	X	X	X	X	X

Table 2.1: Polytime queries of languages introduced by [Darwiche and Marquis, 2002] and [Le Berre et al., 2018]. X means "satisfies" and ○ means "does not satisfy unless P=NP". Languages **SL** and **SL$_<$** are defined in Section 2.1

The biggest advantage of switch-list representations over the strictly more succinct representations such as CNFs, DNFs, and OBDDs rests in the collection of supported transformations (see Table 2.2). Switch-list representations support negation (in constant time) unlike CNFs, DNFs, and MODS, for which the

size of negation can grow exponentially. Switch-list representations also support conditioning (but all common representations do, so this is not an advantage) and more importantly forgetting (the general case, not just singleton forgetting) which distinguishes it from OBDDs that do not support forgetting. Switch-list representations also support unbounded conjunction and disjunction under the additional restriction that all conjuncts (disjuncts) are defined on the same set of variables and with the same order of variables, i.e. all switches are vectors of the same length with individual coordinates indexed by the same variables for all input switch-list representations (this is not shown in Table 2.2 where only the general forms of conjunction and disjunction is tabulated). It should be noted here that OBDDs and even OBDDs with prescribed variable order fail to support unbounded conjunction and disjunction even in this restricted case. Of course, DNFs do not support unbounded conjunction, and CNFs do not support unbounded disjunction. If we allow different conjuncts (disjuncts) to be defined on different sets of variables, the output switch-list representation may grow exponentially if we prescribe a variable order on the output. For non-prescribed order of variables (on the output) we have no results at all, the complexity of conjunction and disjunction is an open problem.

L	**CD**	**FO**	**SFO**	$\wedge$**C**	$\wedge$**BC**	$\vee$**C**	$\vee$**BC**	$\neg$**C**
DNNF	X	X	X	○	○	X	X	○
d-NNF	X	○	X	X	X	X	X	X
BDD	X	○	X	X	X	X	X	X
FBDD	X	•	○	•	○	•	○	X
OBDD	X	•	X	•	○	•	○	X
OBDD$_<$	X	•	X	•	X	•	X	X
CNF	X	○	X	X	X	•	X	•
DNF	X	X	X	•	X	X	X	•
IP	X	•	•	•	X	•	•	•
MODS	X	X	X	•	X	•	•	•
CARD	X	○	?	X	X	•	•	•
PBC	X	•	•	X	X	•	•	•
SL	X	X	X	?	?	?	?	X
SL$_<$	X	X	X	•	•	•	•	X

Table 2.2: Classes introduced by [Darwiche and Marquis, 2002] and [Le Berre et al., 2018] and their polytime transformations. X means "satisfies", • means "does not satisfy" and ○ means "does not satisfy unles P=NP". Languages **SL** and **SL**$_<$ are defined in Section 2.1

The collection of supported queries and transformations suggests that switch-list representations may be a good choice in cases where many queries (such as model counting) have to be answered under many different additional assumptions such as partial substitution of binary values to subsets of variables (i.e. conditioning) or existential quantification of subsets of variables (i.e. forgetting). None of the above mentioned more succinct representation would support such a scenario in polynomial time. An obvious problem for this approach is a lack of compilation algorithms with switch-list representation as the target representation. The only interesting example is the recognition algorithm from [Čepek and

Hušek, 2017], which may be in this context viewed as a compilation algorithm from tractable classes of DNFs into switch-list representations (and also from tractable CNFs by negating of the input, compiling into a switch-list and then negating on the output). Note that the algorithm has a parameter k on its input, and the compilation which runs in time exponential in k is successful if and only if there exists a target switch-list representation with at most k switches. Another representation which is also easy to compile into a switch-list representation is a binary decision tree with a fixed order of variables on all branches. By traversing the leaves of such a tree from left to right one can easily construct a switch-list representation of the given function. This procedure is in some sense just the reverse of the algorithm presented in Section 2.3.1.

In this chapter we establish the properties of switch-list representations and place the languages based on switch-list representations into Knowledge Compilation Map. We believe that the proven properties of switch-list representations will motivate an interest to find other classes of CNF, DNF, OBDD, or other representations, which can be efficiently compiled into switch-list representations.

As a final remark let us note that the combination of results from [Čepek et al., 2008] and [Schieber et al., 2005] gives a polynomial time minimization (optimal compression) algorithm for the class of 1-interval functions given by tractable DNFs, or in other words, for the 1-interval subclass of DNFs inside any tractable class of DNFs. DNF minimization (optimal compression) is a notoriously hard problem. It was shown to be Σ_2^p-complete [Umans, 2001] where there is no restriction on the input DNF (see also the review paper [Umans et al., 2006] for related results). It is also long known that this problem is NP-hard already for some tractable classes of DNFs — maybe the best known example is the class of Horn DNFs (a DNF is Horn if every term in it contains at most one negative literal) for which the NP-hardness was proved in [Ausiello et al., 1986, Hammer and Kogan, 1993] and the same result for cubic Horn DNFs in [Boros et al., 2013]. There exists a hierarchy of subclasses of Horn DNFs for which there are polynomial time minimization algorithms, namely acyclic and quasi-acyclic Horn DNFs [Hammer and Kogan, 1995], and CQ Horn DNFs [Boros et al., 2009]. Suppose we are given a Horn DNF. We can test in polynomial time using the algorithm from [Čepek et al., 2008] whether it represents a 1-interval function and then (in the affirmative case) use the algorithm from [Schieber et al., 2005] to construct a minimum DNF representing the same function as the input DNF. Thus we have a minimization algorithm for 1-interval Horn DNFs. It is an interesting research question in what relation (with respect to inclusion) is this class with respect to the already known hierarchy of polynomial time compressible subclasses of Horn DNFs (acyclic Horn, quasi-acyclic Horn, and CQ Horn DNFs).

The chapter is organized as follows. In Section 2.1 we shall introduce the necessary terminology and notation, and define the propositional languages studied in this chapter. Section 2.2 derives the succinctness relations for the defined languages. Section 2.3 provides a compilation algorithm from switch-list representations to OBDD representations respecting different prescribed orders of variables on the input and on the output. In Section 2.4 we will prove a lower bound on the size of the compiled OBDD. Next, in Section 2.5 we investigate the complexity of common transformations for switch-list representations, and finally

in Section 2.6 we do the same for common queries. We finish the chapter with few concluding remarks.

2.1 Definitions and Notation

In the Chapter 1 we introduced different representations of Boolean functions, which will appear in this thesis. In this chapter we describe relations among different propositional languages defined by representations. In such a class one representation of function f is a called a *sentence*.

By **MODS** we denote the propositional language of all MODS representations.

Definition 1. *Let $<$ be a total order on the set PS of all propositional variables, let X be a subset of PS of size n, and let f be a Boolean function on variables from X. Consider vector $x \in \{0,1\}^n$ where the bits of x correspond to the variables of X in the prescribed order $<$. Each such vector x can be in natural way identified with a binary number from $[0, 2^n - 1]$, so for every $x > 0$ the vector $x - 1$ is well defined. We call $x \in \{0,1\}^n$ a* switch *of f with respect to order $<$, if $f(x-1) \neq f(x)$. The list of all switches of f with respect to $<$ is called the* switch-list *of f with respect to $<$. The switch-list of f with respect to $<$ together with the function value $f(0)$ is called the* switch-list representation *of f with respect to $<$. The set of switch-list representations with respect to $<$ (of all functions) forms the propositional language $\mathbf{SL}_<$. Finally, the language* **SL** *is the union of $\mathbf{SL}_<$ languages over all total orders on the set PS.*

By **DNF** we shall denote the propositional language of all DNFs (of all functions), and similarly **CNF** shall denote the language of all CNFs.

The **IP** is a propositional language of all DNFs, which contains only prime implicants. Similary the **PI** is a propositional language of all CNFs, which contains only prime implicates.

In the Figure 2.1 are mentioned some other classes, which won't appear further in the chapter and their defintion can be found in [Darwiche and Marquis, 2002].

All OBDDs with respect to $<$ form the language $\mathbf{OBDD}_<$ and the language **OBDD** is defined as the union of $\mathbf{OBDD}_<$ languages over all total orders on the set PS.

Let us define the following two notions. Function f is called a *k-switch function* if there exists a switch-list representation of f with respect to some order $<$ of its variables that has at most k switches. Two sentences (possibly from two different propositional languages) are called *logically equivalent* if they represent the same function. Now we are ready to define the most important concept of the next section.

Definition 1. *A propositional language* $\mathbf{L}$ *is* at least as succinct *as a propositional language* $\mathbf{K}$*, denoted* $\mathbf{L} \leq \mathbf{K}$*, if and only if there exists a polynomial p such that for every sentence $\alpha \in \mathbf{K}$ there exists a logically equivalent sentence $\beta \in \mathbf{L}$ such that $|\beta| \leq p(|\alpha|)$ (where the size of a sentence is the number of bits necessary to encode it). If $\mathbf{L} \leq \mathbf{K}$ holds and $\mathbf{K} \leq \mathbf{L}$ does not (denoted $\mathbf{K} \not\leq \mathbf{L}$), we write $\mathbf{L} < \mathbf{K}$.*

The diagram in Figure 2.1, summarizing the succinctness relations of many commonly used propositional languages, appeared in [Darwiche and Marquis, 2002]. It is amended here by the results from [Le Berre et al., 2018] dealing with the **PBC** and **CARD** languages. The main aim of the next section is to add the languages $\mathbf{SL}_<$ and **SL** into the framed part of the diagram in Figure 2.1 by establishing the succinctness relations to the languages already presented there.

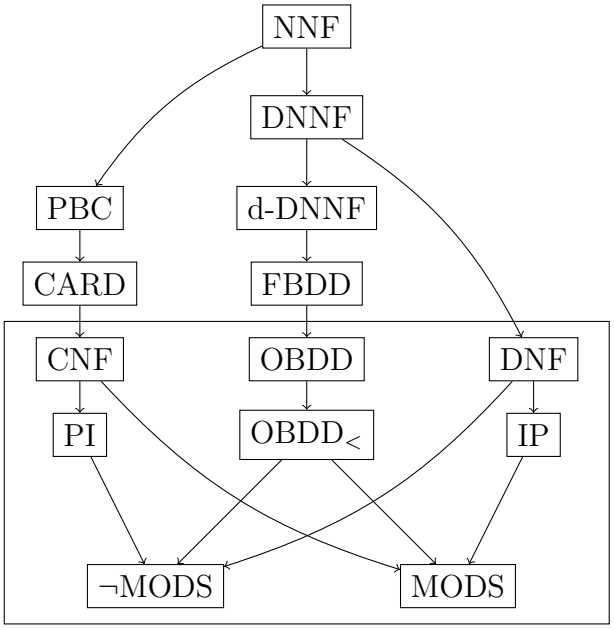


Figure 2.1: The diagram of succinctness relations from [Darwiche and Marquis, 2002] combined with the results from [Le Berre et al., 2018]. For symmetry reasons the language ¬**MODS** was added into the diagram. Each directed arc $A \longrightarrow B$ means that A is strictly more succinct than B, i.e. $A < B$. The rectangle highlights the area investigated in this chapter.

2.2 Succinctness of Switch-list Representations

In this section we prove the succinctness relations for $\mathbf{SL}_<$ and **SL** languages described in Figure 2.2, that is the highlighted part of Figure 2.1. We will use one subsection for each relation, the numbering of the subsections corresponds to the arrow numbers in Figure 2.2. Let us start with the most obvious of all the relations.

2.2.1 Relation between SL and $\mathbf{SL}_<$ languages

Proposition 1. $\mathbf{SL} < \mathbf{SL}_<$

Proof. The inequality $\mathbf{SL} \leq \mathbf{SL}_<$ follows from the fact that the language $\mathbf{SL}_<$ is a subset of the language **SL**. To show that $\mathbf{SL}_< \not\leq \mathbf{SL}$ let us consider function $f(x_1, \ldots, x_n, y) = y$ and two orders of its variables $<_1$ and $<_2$ where $y <_1 x_1 <_1 \ldots <_1 x_n$ and $x_1 <_2 \ldots <_2 x_n <_2 y$. Clearly, the switch-list representation of f with respect to $<_1$ has a single switch (all non-models precede all models) while the switch-list representation of f with respect to $<_2$ has $2^{n+1} - 1$ switches

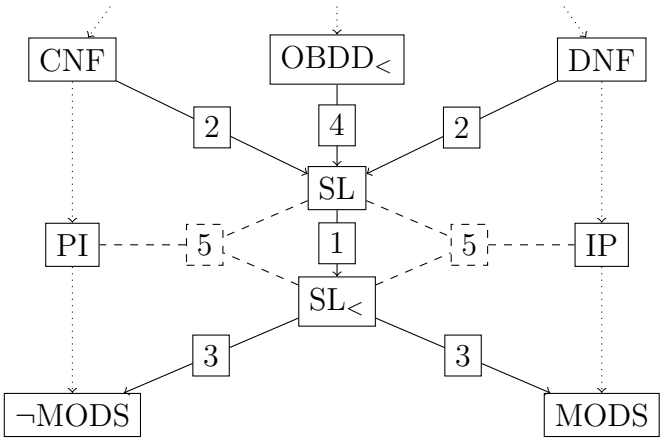


Figure 2.2: The highlighted area from Figure 2.1. Solid arrows correspond to strict succinctness results, dashed lines to incomparability results, and dotted lines to known strict succinctness relations from Figure 2.1 which do not follow from transitivity using the solid arrows.

(non-models and models alternate, so every vector except of the very first one is a switch). Thus the switch-list representation of f in the language $\mathbf{SL}_{<_2}$ is exponentially larger than the switch-list representation of f with respect to $<_1$ in the language **SL**. This simple example is not fully satisfactory, because f does not depend on variables $x_1, \ldots, x_n$ and so $f(x_1, \ldots, x_n, y) = f(y)$ is in fact a function in just one variable (and of course the exponential blowup disappears). This can be easily fixed by switching the last non-model of f into a model, i.e. by considering $f(x_1, \ldots, x_n, y) = y \vee (\bigwedge_{i=1}^{n} x_i)$. This function now depends on all $n + 1$ variables, still has a single switch with respect to $<_1$ (all non-models still precede all models), and still has exponential many switches with respect to $<_2$ (only the last two switches with respect to $<_2$ disappeared). □

2.2.2 Relations among SL, CNF, and DNF languages

Proposition 2. **CNF** $<$ **SL** *and* **DNF** $<$ **SL**

Proof. Let us start by proving **DNF** $\leq$ **SL**. It was shown in [Schieber et al., 2005] that any 1-interval function on n variables can be represented by a DNF with at most $2n - 4$ terms. Obviously, the output DNF has a size which is at most quadratic in n and hence also at most quadratic in the size of the input switch-list representation (two vectors of length n). Now assume we have switch-list representation of function f with k switches on the input, which means that f has $\lfloor k/2 \rfloor$ or $\lfloor k/2 \rfloor + 1$ intervals of models (depending on the parity of k and the function value f(0)). Let us construct a DNF representation for each interval using the algorithm from [Schieber et al., 2005]. Obviously, the disjunction of these DNFs represents f and the size of this aggregated DNF is $O(kn^2)$ while the size of the input is $O(kn)$.

Now let us show **CNF** $\leq$ **SL**. Starting with the switch-list representation of a function f with k switches we can turn it in a constant time into the switch-list representation of $\neg f$ by negating the value of $f(0)$ and keeping the switch-list unchanged. Both switch-list representations have size $O(kn)$. Using the construction from the previous paragraph, we get a DNF of $\neg f$ of size $O(kn^2)$.

This DNF can be switched in $O(kn^2)$ time into a CNF of the same size which represents f (by mechanically applying de Morgan rules to propagate the negation on the outside of the DNF formula towards the literals, which changes the DNF into a CNF).

In order to prove $\mathbf{SL} \not\leq \mathbf{DNF}$ we shall assume by contradiction that $\mathbf{SL} \leq \mathbf{DNF}$. Note, that by the transitivity of succinctness relations and the above proved relation $\mathbf{CNF} \leq \mathbf{SL}$ we could conclude $\mathbf{CNF} \leq \mathbf{DNF}$, which is known to be false (see e.g. [Darwiche and Marquis, 2002] for a counterexample). The proof of $\mathbf{SL} \not\leq \mathbf{CNF}$ proceeds in a completely similar way, only the roles of languages **CNF** and **DNF** are exchanged. □

2.2.3 Relations among $\mathbf{SL}_<$, MODS, and ¬MODS languages

Proposition 3. $\mathbf{SL}_< < \mathbf{MODS}$ *and* $\mathbf{SL}_< < \neg\mathbf{MODS}$

Proof. Consider the function $f(x_1, \ldots, x_n, y) = y \vee (\bigwedge_{i=1}^{n} x_i)$ from Proposition 1. It depends on all $n+1$ variables, has a single switch with respect to order $y < x_1 < \ldots < x_n$ of the variables, and has $2^n + 1$ models and $2^n - 1$ non-models. This proves $\mathbf{MODS} \not\leq \mathbf{SL}_<$ and $\neg\mathbf{MODS} \not\leq \mathbf{SL}_<$.

The relations $\mathbf{SL}_< \leq \mathbf{MODS}$ and $\mathbf{SL}_< \leq \neg\mathbf{MODS}$ are more or less obvious. It is easy to see that there can be at most twice as many switches as models and symmetrically also at most twice as many switches as non-models. Indeed, every switch x can be identified with a pair of consecutive vectors x and $x-1$ with opposite function values and every model (and non-model) is identified in this way with at most two switches. □

2.2.4 Relation between SL and $\mathbf{OBDD}_<$ languages

Let us start this subsection with several definitions and technical lemmas.

Definition 2. *Let $f(x_1, \ldots, x_n)$ be a Boolean function on n variables. A Boolean vector $(y_1, \ldots, y_l) \in \{0,1\}^l$ where $l < n$ is called a* relevant vector for f of length l *if there exist two vectors $(x_{l+1}, \ldots, x_n) \in \{0,1\}^{n-l}$ and $(x'_{l+1}, \ldots, x'_n) \in \{0,1\}^{n-l}$ such that*

$$f(y_1, \ldots, y_l, x_{l+1}, \ldots, x_n) \neq f(y_1, \ldots, y_l, x'_{l+1}, \ldots, x'_n)$$

Relevant vectors are prefix vectors (for the given order of variables) which are not sufficient for determining the value of f. The motivation behind the above definition is the easily verifiable fact that a k-switch function (i.e. function which has k switches with respect to the prescribed order of variables) has at most k relevant vectors of any length.

Lemma 4. *Let $f(x_1, \ldots, x_n)$ be a k-switch function and $1 \leq l \leq n$ an arbitrary number. Then there are at most k relevant vectors for f of length l.*

Proof. Let $y^i = (y^i_1, \ldots, y^i_n)$, $1 \leq i \leq k$ be the switch vectors for f and let us assume these vectors are lexicographically ordered. That is $\forall i \in \{1, \ldots, k-1\}$: $y^i < y^{i+1}$ if we identify switch vectors with binary numbers. Consider the vectors $p^i = (y^i_1, \ldots, y^i_l)$, $1 \leq i \leq k$, that is the prefixes of the switch vectors of length

l. There are at most k distinct vectors in this set as some pairs of prefixes may coincide. We claim that no other vector (different from $p^1, \ldots, p^k$) is a relevant vector for f of length l. Let $z = (z_1, \ldots, z_l)$ be any such vector. Since z differs from all p^i's, there is no switch vector among vectors $(z_1, \ldots, z_l, x_{l+1}, \ldots, x_n)$ for $(x_{l+1}, \ldots, x_n) \in \{0,1\}^{n-l}$ and thus

$$f(z_1, \ldots, z_l, x_{l+1}, \ldots, x_n) = f(z_1, \ldots, z_l, x'_{l+1}, \ldots, x'_n)$$

for any two vectors $(x_{l+1}, \ldots, x_n) \in \{0,1\}^{n-l}$ and $(x'_{l+1}, \ldots, x'_n) \in \{0,1\}^{n-l}$, which proves that z is not relevant for f. □

Remark. Note that in the above proof vectors $p^1, \ldots, p^k$ are the only candidates for relevant vectors of length l, however not all of them have to be relevant for f, since the "decision" determining the value of f may be taken at an earlier index than l. A trivial example is a 1-switch function where $f(0, x_2, \ldots, x_n) = 0$ and $f(1, x_2, \ldots, x_n) = 1$ for all vectors $(x_2, \ldots, x_n)$. Here no (non-empty) prefix of the single switch vector $(1, 0, 0, \ldots, 0)$ is a relevant vector for f.

Definition 3. *Let $f(x_1, \ldots, x_n)$ be a Boolean function on n variables. A relevant vector $(y_1, \ldots, y_l)$ for f of length l is called* maximal relevant for f *if neither $(y_1, \ldots, y_l, 0)$ nor $(y_1, \ldots, y_l, 1)$ are relevant vectors for f of length $l+1$.*

Lemma 5. *Let f be a k-switch function. Then there are at most k maximal relevant vectors for f.*

Proof. As we have seen in the proof of Lemma 4, only prefixes of the switch vectors are candidates to relevant vectors. Moreover for each switch vector at most one length of prefix can be a maximal relevant length, which proves the claim. (Note that for some switch vectors no prefix is relevant so there is also no maximal relevant prefix - see Remark 2.2.4.) □

Lemma 6. *Let $f(x_1, \ldots, x_n)$ be a Boolean function and $I \subseteq \{x_1, \ldots, x_n\}$ be a subset of variables of size $|I| = i$. Let x_m be the variable with the smallest index in I and let us assume that f has at most k maximal relevant vectors for f of length at least m. Then there are at most $(ik+1)$ different Boolean functions that originate from f by fixing the values of variables in I.*

Proof. If $k = 0$, that is there is no relevant vector for f of length m or larger, then f does not depend on variables from I. That means that all substitutions for the variables of I lead to the same function of $(n-i)$ variables and the claim holds $((ik+1) = (i \cdot 0 + 1) = 1)$.

If $k \geq 1$ we shall proceed by induction on $|I| = i$.

Base case $(i = 1)$*:*

By fixing the only variable in $I \subseteq \{x_1, \ldots, x_n\}$ to 0 or 1 we get at most two different functions of $n-1$ variables, namely $f(x_1, \ldots, x_{m-1}, 0, x_{m+1}, \ldots, x_n)$ and $f(x_1, \ldots, x_{m-1}, 1, x_{m+1}, \ldots, x_n)$. Since $i = 1$ and $k \geq 1$ we get $ik + 1 \geq 2$ and the base case is verified.

Induction step $(i - 1 \Rightarrow i)$*:*

Let us assume that $i > 1$ and the statement of the lemma is true for $1, 2, \ldots, i-1$. Let

$$V = \{p^i | 1 \leq i \leq l\}$$

be the set of all maximal relevant vectors for f of length at least m (by assumption $l \leq k$). Consider the partition of V depending on the value of x_m into

$$V_0 = \{p^i | p^i_m = 0\}$$

$$V_1 = \{p^i | p^i_m = 1\}$$

and denote $|V_0| = l_0$ and $|V_1| = l_1$. Clearly $l_0 + l_1 = l$.

Denote $I' = I \setminus \{x_m\}$ and let $x_{m'}$ be the variable with the smallest index in I' (of course $m' > m$). Consider functions of $(n-1)$ variables

$$f_0(x_1, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n) = f_{|x_m=0}$$

$$f_1(x_1, \ldots, x_{m-1}, x_{m+1}, \ldots, x_n) = f_{|x_m=1}$$

There are at most l_0 maximal relevant vectors for f_0 of length at least m' (and similarly for f_1). Indeed, such maximal relevant vectors can originate from vectors in V_0 (or V_1 respectively) by deleting p^i_m, if those vectors are long enough (have length at least m'). Thus we may use the induction hypothesis for f_0, f_1 and I' of size $|I'| = i - 1$. We get that

1. there are at most $(i-1)l_0 + 1$ different Boolean functions that originate from f_0 by fixing the values of variables in I'
2. there are at most $(i-1)l_1 + 1$ different Boolean functions that originate from f_1 by fixing the values of variables in I'

This altogether implies that there are at most

$$(i-1)l_0 + 1 + (i-1)l_1 + 1 = (i-1)(l_0 + l_1) + 2 =$$
$$= (i-1)l + 2 \leq (i-1)k + 2 = ik - k + 2 \leq ik + 1$$

different Boolean functions that originate from f by fixing the values of variables in I, since $k \geq 1$, which finishes the proof. □

Corollary. Let $f(x_1, \ldots, x_n)$ be a k-switch Boolean function and $I \subseteq \{x_1, \ldots, x_n\}$ a subset of its variables of size $|I| = i$. Then there are at most $(ik + 1)$ different functions that originate from f by fixing variables in I.

Proof. This claim is direct consequence of Lemma 5 and Lemma 6 □

Let us consider a k-switch function $f(x_1, \ldots, x_n)$, and let us consider an arbitrary reordering of the variables given by some linear order $<$. If we start branching on variables in the order given by $<$, then Corollary 2.2.4 states that after branching on the first i variables, we get at most $(ik + 1)$ different Boolean functions of the remaining variables (as opposed to at most 2^i for general functions). This is sufficient for a bound on the size of a minimal OBDD representation of f due to the following theorem.

Theorem 7. [Wegener, 2000, Theorem 3.2.2]. *Let f be a function on variables $X = \{x_1, \ldots, x_n\}$ and let $<$ be a linear order on X. Then the minimal-size OBDD representation of f respecting order $<$ contains as many x_i-nodes as there are different subfunctions $|f_{|\{x_j | x_j < x_i\}}|$.*

Now Theorem 7 together with Corollary 2.2.4 imply the desired result.

Proposition 8. $\mathbf{OBDD}_< < \mathbf{SL}$

Proof. Let $f(x_1, \ldots, x_n)$ be a k-switch function and $<$ some linear order of the variables. By Theorem 7 and Corollary 2.2.4 a minimum size OBDD respecting $<$ contains at most $(ik + 1)$ nodes on branching level $i + 1$ for $0 \leq i \leq n - 1$. Therefore such an OBDD has at most $\sum_{i=0}^{n-1}(ik + 1) = \frac{1}{2}kn(n - 1) + n$ nodes which is polynomial in the size of the input switch-list for f of size kn. This proves $\mathbf{OBDD}_< \leq \mathbf{SL}$.

On the other hand, it is easy to see that $\mathbf{SL} \not\leq \mathbf{OBDD}_<$. A good example is the parity function on n variables which is symmetric, and thus changing the order imposed on the set of propositional variables changes neither the minimum size OBDD nor the minimum size of switch-list representation. The parity function is well known to have an OBDD of linear size in n, while the number of switches in any switch-list representation is exponential in n. The last fact follows e.g. from an easy observation that every second vector in the truth table that corresponds to an even number changes its parity where the last bit is flipped from 0 to 1 to get the next odd number. Thus every vector which corresponds to an odd number is a switch, and therefore the parity function has $\Omega(2^n)$ switches. □

2.2.5 Relations among SL (or $\mathbf{SL}_<$), PI, and IP languages

Proposition 9. SL *is incomparable with both* **PI** *and* **IP**

Proof. First we prove $\mathbf{SL} \not\leq \mathbf{IP}$ and $\mathbf{SL} \not\leq \mathbf{PI}$. Let us proceed by contradiction assuming $\mathbf{SL} \leq \mathbf{IP}$ ($\mathbf{SL} \leq \mathbf{PI}$ respectively). This assumption together with the relation $\mathbf{OBDD} \leq \mathbf{SL}$ proved in Proposition 8 would imply $\mathbf{OBDD} \leq \mathbf{IP}$ ($\mathbf{OBDD} \leq \mathbf{PI}$ respectively). However, both these relations are known to be false, see e.g. [Darwiche and Marquis, 2002] for counterexamples.

Now we shall show $\mathbf{PI} \not\leq \mathbf{SL}$. Let us consider a function f on $2n$ variables $x_1, ..., x_n, y_1, ..., y_n$ defined as follows. The models of f are the vectors $\{v_i | i = 1, ..., n\}$ where v_i assigns only variables x_i and y_i to 1, and all other variables to 0. Thus f has exactly n models, and the size of its switch-list representation is $O(n^2)$ ($2n$ switches each of size $2n$). Now for an arbitrary subset of indices $S \subseteq 1, ..., n$, let us define a clause $C_S = (\bigvee_{i \in S} x_i \vee \bigvee_{i \notin S} y_i)$. We shall show that for every S, C_S is a prime implicate of f, and therefore f has at least 2^n prime implicates, showing the claim.

Take an arbitrary model v_i of f which by definition satisfies both x_i and y_i. No matter how S was selected, either x_i or y_i appears in C_S satisfying it. Thus C_S is an implicate of f. Now take an arbitrary proper subclause C of C_S. By the definition of C there is an index i such that neither x_i nor y_i appear in C. That means that the model v_i of f falsifies C, which implies that C is not an implicate of f. Thus C_S is prime.

It remains to show $\mathbf{IP} \not\leq \mathbf{SL}$. This relation is more or less a consequence of $\mathbf{PI} \not\leq \mathbf{SL}$ due to the duality between CNFs and prime implicates on one hand and DNFs and prime implicants on the other hand. Consider the negation of the function from the previous proof. Obviously, the switch-list representation of $\neg f$ has exactly the same size as the switch-list representation of f. Vectors $\{v_i | i = 1, ..., n\}$ are now the only non-models of $\neg f$ and for every $S \subseteq 1, ..., n$, we

can show that the term $T_S = (\bigwedge_{i \in S} \neg x_i \wedge \bigwedge_{i \notin S} \neg y_i)$ is a prime implicant of $\neg f$. So $\neg f$ has at least 2^n prime implicants, showing the claim.

Take an arbitrary non-model v_i of $\neg f$ which by definition falsifies both $\neg x_i$ and $\neg y_i$. No matter how S was selected, either $\neg x_i$ or $\neg y_i$ appears in T_S falsifying it. Thus T_S is an implicant of $\neg f$. Now take an arbitrary proper subterm T of T_S. By the definition of T there is an index i such that neither $\neg x_i$ nor $\neg y_i$ appear in T. That means that the non-model v_i of $\neg f$ satisfies T, which implies that T is not an implicant of $\neg f$. Thus T_S is prime. □

Since renumbering variables has no effect on the number of prime implicates or prime implicants, we get the same result as above also for the language $\mathbf{SL}_<$.

Corollary. $\mathbf{SL}_<$ is incomparable with both **PI** and **IP**

Note, that function f from the proof of Proposition 9 has an exponential number of prime implicates with respect to the number of models. This, together with an obvious fact that $\mathbf{MODS} \not\leq \mathbf{PI}$, gives a short proof of the long standing open problem from [Darwiche and Marquis, 2002, stated as a question mark in Table 3 on page 237].

Corollary. $\mathbf{PI} \not\leq \mathbf{MODS}$ and hence **PI** is incomparable with **MODS**

2.3 Compilation from SL to OBDD

In this section we shall show that the existential proof of Proposition 8 can be extended into a compilation algorithm, that for an input switch-list representation of size kn outputs an OBDD of size $O(kn^2)$. The compilation algorithm works in two steps. First it compiles the input switch-list representation of function f into a binary decision tree (BDT) that respects the same order of variables as the input switch-list representation. In the second step it uses the constructed BDT to create an OBDD of f that respects a given prescribed order of variables that may differ from the order used by the input switch-list representation. Note, that if the input order and the prescribed output order are the same, then the first step suffices, and the constructed linear size BDT gives the output OBDD simply by unifying all zero terminals into a single zero terminal and the same for one terminals.

Our algorithm is similar to the approach taken in [Wegener, 2000, the proof of Theorem 5.7.10]. However, our implementation uses techniques that allow us to speed up the minimization process at each level of the constructed OBDD in which we merge nodes representing the same subfunction. To achieve this, we construct "prefix contracted" binary decision trees, that are then used to cache different subfunctions on a given level. This improves the worst case time $O(|H||G| \log |H|)$ proved by Savický and Wegener [1997] and matches the average case time of $O(|H||G|)$ [Tani and Imai, 1994]. Our technique accomplishes a worst case time complexity $O(|H||G|)$, where $|G|$ is the size of the input switch-list representation (which is the same as the size of the BDT constructed in the first step of our algorithm) and $|H|$ is the size of the target OBDD representation.

2.3.1 Compilation from SL to BDT

Let us consider a switch-list representation of a k-switch function f and construct an equivalent decision tree representation with respect to the same order of variables from the switch-list representation of f. The idea behind the construction is quite simple. Let the order of variables in the input switch-list representation be $x_1, x_2, \ldots, x_n$. Consider the complete binary decision tree of f which branches in the prescribed order, i.e. a tree with n levels of decision nodes and 2^n function values on level $n+1$. Then start a bottom-up process of eliminating redundant decision nodes. In this process, every decision node with both outgoing edges leading to the same function value t is deleted and replaced by an edge from its parent node to function value t. This process obviously stops with a binary decision tree that represents f. Note that this output tree is unique, it depends only on function f, and does not depend on the order in which nodes are contracted. What is the size of this unique contracted decision tree? Consider the leaf nodes, that is decision nodes with both outgoing edges going to terminals (function values). Obviously, for every leaf node these two edges necessarily go to different function values (otherwise the node would have been eliminated) and so the path from the root to any leaf node encodes a prefix of some switch of f. Moreover, by the definition of a leaf node, no two leaf nodes can encode a prefix of the same switch, so the number of leaf nodes is upper bounded by the number of switches. It follows that the number of decision nodes in the constructed decision tree is at most n times the number of leaf nodes, that is at most n times the number of switches, which is exactly the size of the input switch-list representation (each switch is a vector of length n). Therefore the constructed decision tree has a linear size with respect to the size of the input switch-list representation.

The above considerations suffice for a proof of an existence of a linear size decision tree equivalent to the input switch-list representation. However, if we want to obtain also a polynomial time compilation procedure that constructs the output decision tree, we have to avoid building the exponentially large initial decision tree. This can be easily avoided by building the output decision tree from top to bottom rather than bottom-up. We start by creating the root node, assigning the interval $[0, 2^n - 1]$ and variable x_1 to the root node, and inserting the root node into a queue. Then we start processing the nodes from the queue in the following manner. Extract the first node v with an assigned interval $[a, b]$ and a variable x_k from the queue. Scan the input switch-list until one of the following two situations occurs:

1. (Non-constant interval) Switch x in the switch-list is found, such that, if interpreted as a binary number, $a < x \leq b$ holds. In this case construct two children nodes v_L, v_R of v, assign variable x_{k+1} to both v_L and v_R, and assign interval $[a, (a+b+1)/2-1]$ (the left half of $[a, b]$) to v_L and interval $[(a+b+1)/2, b]$ (the right half of $[a, b]$) to v_R (a is always even, b is always odd, and the length of $[a, b]$ is always a power of 2, so there are no issues with rounding). Insert v_L and v_R to the end of the queue.

2. (Constant interval) Two consecutive switches x, y in the switch-list are found, such that, if interpreted as binary numbers, $x \leq a$ and $b < y$ hold. This means that there is no switch in the interval $[a+1, b]$ and hence all vectors in the interval $[a, b]$ share the function value of x. So node v can be

deleted from the tree of decision nodes and replaced by an edge from the parent node of v to a terminal with function value $f(x)$.

The procedure stops when the queue is empty and constructs exactly the same unique decision tree as the bottom-up procedure described above. The work per decision node is linear in the size of the input switch-list (we scan the switch-list once per node), so the overall complexity of the compilation procedure is at most quadratic in the size of the input switch-list[1].

2.3.2 Compilation from BDT to OBDD

Now we shall show how to compile a BDT of f which respects the identical order of variables $x_1, \ldots, x_n$ into a polynomial size OBDD of f which respects some other prescribed order of variables $y_1, \ldots, y_n$, where $y_j = x_{\sigma(i)}$ for a given permutation σ. Let us start by defining a special type of binary decision tree.

Definition 4. *Let T be a binary decision tree on variables $x_1, \ldots, x_n$ which branches on the variables (on every branch) in this prescribed order. Then T is called a **prefix** BDT if every branch in T of length l contains the first l decision variables $x_1, \ldots, x_l$, and T is called a **contracted** BDT if for every decision node x the subtree of T rooted at x contains both terminals 0 and 1.*

Note that the BDT constructed from the input switch-list representation of function f as described in Section 2.3.1 is a contracted prefix BDT representing f. It is also an easy observation that given a function h on variables $z_1, \ldots, z_k$ it has a one-to-one correspondence with its contracted prefix BDT which respects the given order of variables. We have already observed in Section 2.3.1 that given h and a fixed order of variables, the resulting contracted prefix BDT is unique. The reverse direction is trivial, given a contracted prefix BDT it clearly represents exactly one function h.

The principal idea behind our algorithm is to build a minimum size OBDD of f (which respects the order of variables $y_1, \ldots, y_n$) level by level in a BFS manner, where each node u on a level i of the constructed OBDD will be associated with a contracted prefix BDT representing the corresponding subfunction f_u of f in variables $y_i, \ldots, y_n$ defined by $f_u(y_i, \ldots, y_n) = f(u_1, \ldots, u_{i-1}, y_i, \ldots, y_n)$, where the vector $(u_1, \ldots, u_{i-1}) \in \{0,1\}^{i-1}$ represents the path from the root of the OBDD to the node u. The uniqueness of the contracted prefix BDT representations will allow us to efficiently detect whether two subfunctions on the same level of the OBDD are logically equivalent, which is necessary to build a minimum size OBDD of f.

We can encode a contracted prefix BDT T representing function h into a string on an alphabet $\Sigma = \{0, 1, \ell, r, b\}$ using a DFS traversal of T which writes ℓ when traversing from a parent to its left descendant (we assume that DFS first branches left, i.e. on value 0, at every decision node of T), writes r when

[1] In fact, since the tree is built in a BFS manner level by level, the procedure can be modified to restart the scan of the switch-list from the beginning only once per level, improving the complexity upper bound to n times the size of the input switch-list. Using a smarter data structures that for each decision node define not only the relevant interval of binary numbers but also the relevant interval in the switch-list, the overall complexity can be brought further down to linear time complexity by eliminating by the factor n.

traversing from a parent to its right descendant, writes b when backtracking from a decision node, and writes 0 or 1 when traversing from a terminal with that function value. This procedure yields a string of length $O(|T|)$ which is of course also unique for h. Therefore we can check the equivalence of two functions h_1 and h_2 (defined on the same set of variables) simply by comparing the encodings of their contracted prefix BDTs which both respect the same prescribed order of variables. A reasonable data structure to support such string comparisons is a trie. Recall, that given a trie which represents a set R of strings and a string s of length t, one can test in $O(t)$ time whether $s \in R$, in the positive case output the node of the trie that represents s, and in the negative case update the trie by inserting s into R. This gives us the following observation.

Observation. Let R be a set of encodings of contracted prefix BDTs stored in a trie and let T be a contracted prefix BDT. Then encoding T into a string s and checking if s is in the trie storing R can be done in $O(|T|)$ time. Moreover, if s is not in the trie storing R, we can add it to the trie in time $O(|T|)$.

Our construction of the output OBDD will start with a root node (the only node on level 1) and the associated contracted prefix BDT in variables $y_1, \ldots, y_n$ which respects the order $x_1, \ldots, x_n$, constructed from the input switch-list representation as described in Section 2.3.1. During the processing of nodes on level i (of the OBDD that is being built), the algorithm keeps a trie $\mathcal{S}$ containing encodings of contracted prefix BDTs associated with nodes on level $i+1$ (starting with an empty $\mathcal{S}$ before the first node on level i is processed).

In a step processing node u on level i with an associated contracted prefix BDT T_u, let $\mathcal{V}$ denote the (possibly empty) set of all already created nodes on level $i+1$, let $\mathcal{T}$ denote the set of all contracted prefix BDTs associated which nodes in $\mathcal{V}$, and let $\mathcal{S}$ denote the trie which contains encodings of all BDTs from $\mathcal{T}$. Now consider the assignment $y_i = 0$. It is easy to modify T_u representing $f(u_1, \ldots, u_{i-1}, y_i, \ldots, y_n)$ into BDT T_u^0 representing $f(u_1, \ldots, u_{i-1}, 0, y_{i+1} \ldots, y_n)$. Note that T_u respects the original variable order given by $x_1, \ldots, x_n$. If T_u branches on y_i in its root, then T_u^0 is simply the root subtree of T_u corresponding to $y_i = 0$, otherwise T_u^0 originates from T_u by connecting the parent of every node u that branches on y_i directly to the child of u which corresponds $y_i = 0$ (and deleting u). Note that T_u^0 is a prefix BDT, on the other hand it is not necessarily contracted. However, we can transform T_u^0 into a contracted prefix tree by a single DFS pass over T_u^0. When DFS gets to a node u for which both descendants are terminals with the same value c, it connects a parent of u to a terminal c.

After building contracted prefix BDT T_u^0 we can check if T_u^0 is equivalent to some $T_v \in \mathcal{T}$ using Observation 2.3.2. If we find such T_v associated with a node v, we connect the branch corresponding to $y_i = 0$ at node u to node v. If such T_v does not exist we create a new node w with an associated contracted prefix BDT $T_w = T_u^0$ and connect the branch corresponding to $y_i = 0$ at a node u to a node w. Moreover, we add w into $\mathcal{V}$, add T_w into $\mathcal{T}$, insert the encoding of T_w into $\mathcal{S}$, and associate the corresponding node in the trie $\mathcal{S}$ with w (so that we have a constant time access to w next time the encoding of T_w is found in $\mathcal{S}$). The procedure for $y_i = 1$ is symmetric.

Assuming that the input k-switch function f is given by a switch-list of size kn, the constructed OBDD has size $O(kn^2)$, and its construction takes $O(k^2n^3)$ time. The complexity is bound by the fact that for each of the $O(kn^2)$ nodes

in the OBDD the associated contracted prefix BDT has size $O(kn)$, and all the above described procedure does when processing a given node of the OBDD is linear in the size of the associated BDT.

Remark. The fact that the existential proof of Proposition 8 can be extended into a compilation algorithm can be shown also in another way. In Section 2.5 we show that conditioning (for any variable) can be done in linear time on an input switch-list representation, and so the output OBDD can be efficiently built level by level using subsequent conditioning on variables in the order prescribed for the output. The complexity bottleneck is the same as for the approach described above in this section, namely checking whether a node corresponding to a given subfunction already exists on the given level of the constructed OBDD (in that case it suffices to add an arc to such a node) or not (in that case a new node must be created). If the switch-list representations for nodes on the current level (the one being built) are cached in an intelligent way to allow such equivalence checks, the overall complexity of the compilation algorithm can be also bounded by $O(k^2n^3)$. However, it seems to us that the two step algorithm described above is easier to implement.

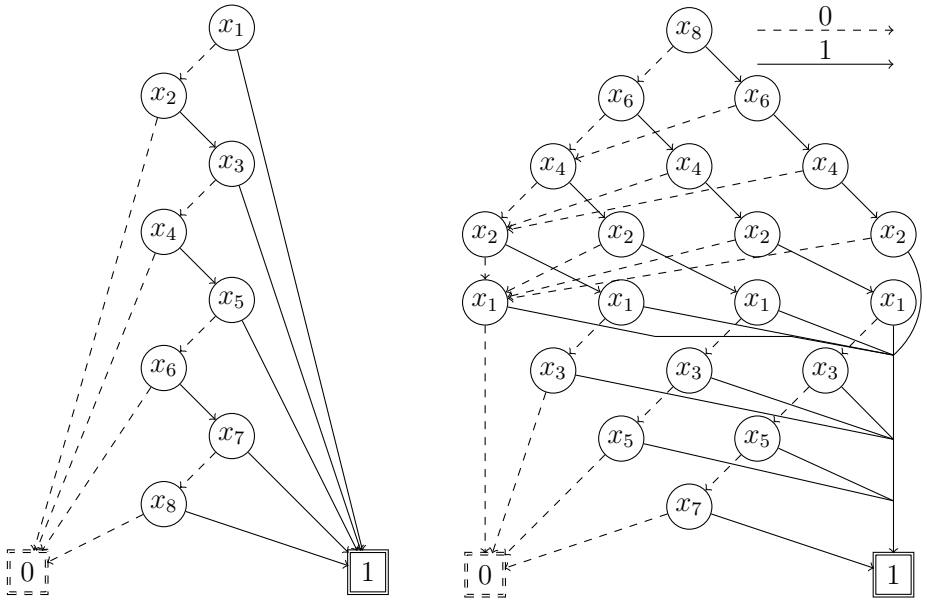
2.4 Lower Bound for the Size of Target OBDD

In this section we shall show that the quadratic blowup in the number of variables in the compilation algorithm from Section 2.3 which produces an output OBDD of size $O(kn^2)$ is unavoidable. We shall consider a case with $k = 1$ when the input switch-list representation contains a single switch and thus has size $O(n)$, and construct a 1-switch function f for which any OBDD w.r.t. a certain prescribed order of variables has size $\Omega(n^2)$.

Let f be a function on n variables where n is even and the only switch of f with respect to the natural ordering $\pi : x_1 < x_2 < \ldots < x_n$ is $s = (0, 1, 0, 1, 0, 1, \ldots, 0, 1)$. See Figure 2.3a for an OBDD of f (respecting π) with $n = 8$, which can be produced from the input switch-list representation by the first step of the compilation algorithm from Section 2.3 (which produces a BDT) and an unification of both terminals. For the output OBDD representation of f we will prescribe the ordering $\sigma : x_n < x_{n-2} < x_{n-4} < \ldots < x_2 < x_1 < x_3 < \ldots < x_{n-3} < x_{n-1}$. See Figure 2.3b for the minimal output OBDD of f (respecting σ) with $n = 8$. We shall prove that any OBDD representation of f with respect to the ordering σ must have at least i distinct nodes on every level $i \leq n/2$. Thus the total number of nodes on the first $n/2$ levels is at least $\sum_{i=1}^{n/2} i$ which is $\Omega(n^2)$ proving the claim.

Let us proceed by induction on i. Starting the induction for $i = 1$ is trivial, there is a single node on the first level of any OBDD. For $i = 2$ it suffices to verify that $f(x_n = 0) \neq f(x_n = 1)$ which is easy to see from Figure 2.3a as $f(0, 1, 0, 1, 0, 1, \ldots, 0, 0) = 0$ while $f(0, 1, 0, 1, 0, 1, \ldots, 0, 1) = 1$. Thus, there must be two nodes on the second level of the output OBDD.

For the general induction step let us assume that we have i nodes (where $i \geq 2$) denoted $n_0, \ldots, n_{i-1}$ on the i-th level of an OBDD representing f (this level branches on variable x_{n+2-2i}) and these nodes correspond to pairwise distinct subfunctions $f_0, \ldots, f_{i-1}$ in variables $x_{n+2-2i}, x_{n-2i}, \ldots, x_2, x_1, x_3, \ldots, x_{n-3}, x_{n-1}$.



(a) Original BDT. (b) Compiled OBDD.

Figure 2.3: OBDD with two different orderings of a function f with variables $x_1, \ldots, x_8$ and one switch $(0, 1, 0, 1, 0, 1, 0, 1)$.

We shall show that there exist $i+1$ pairwise distinct subfunctions $g_0, \ldots, g_i$ that originate from $f_0, \ldots, f_{i-1}$ by fixing a value of x_{n+2-2i}. This of course implies that there must be at least $i+1$ distinct nodes $m_0, \ldots m_i$ on level $i+1$ in any OBDD representing f and respecting the order σ.

First let us consider the case $x_{n+2-2i} = 0$. It is clear from Figure 2.3a that for any vector $x \in \{0,1\}^n$ with $x_{n+2-2i} = 0$ (note that $n+2-2i$ is even) where we fix the values of all variables in order π, then the value of $f(x)$ is either decided when setting $x_{n+2-2i} = 0$ or earlier. Hence, $f(x_{n+2-2i} = 0)$ does not depend on any variable x_k for $k > n+2-2i$ and thus in particular on variables $x_n, x_{n-2}, \ldots, x_{n+4-2i}$, i.e. on those variables on which the output OBDD respecting order σ branches on the first $i-1$ levels. Therefore fixating the values of variables $x_n, x_{n-2}, \ldots, x_{n+4-2i}$ in all possible ways, which is how $f_0, \ldots, f_{i-1}$ originate from f, together with fixing $x_{n+2-2i} = 0$ always produces the same function which we denote by g_i. In other words, substituting $x_{n+2-2i} = 0$ into functions $f_0, \ldots, f_{i-1}$ produces a single function g_i.

The above observation moreover implies, that every pair of vectors x^j, $x^k \in \{0,1\}^{n-(i-1)}$ for $0 \leq j < k \leq i-1$ that guarantees $f_j \neq f_k$ (note that f_j and f_k are functions in $n-(i-1)$ variables with values of $x_{n+4-2i}, \ldots, x_{n-2}, x_n$ already fixed) must satisfy $x^j_{n+2-2i} = x^k_{n+2-2i} = 1$. Thus by substituting $x_{n+2-2i} = 1$ into $f_0, \ldots, f_{i-1}$ we produce i pairwise distinct functions denoted $g_0, \ldots, g_{i-1}$.

It remains to show that the function g_i is distinct from every function in the set $\{g_0, \ldots, g_{i-1}\}$. So let $0 \leq j \leq i-1$ be arbitrary but fixed, and consider vector $x = (0, 1, 0, 1, \ldots 0, p, 1) \in \{0,1\}^{n+3-2i}$, i.e. we are considering the first $n+3-2i$ variables in order π and p is in position $n+2-2i$. Notice, that all variables on which the output OBDD respecting order σ branches on levels $1, \ldots, i-1$ are outside of the scope of indices used by x. Now g_i originates from f_j by setting $p = 0$. Vector x specifies only a partial assignment on variables of g_i, but this

	$\neg\mathbf{C}$	$\wedge\mathbf{C}^*$	$\wedge\mathbf{BC}$	$\wedge\mathbf{C}$	$\vee\mathbf{C}^*$	$\vee\mathbf{BC}$	$\vee\mathbf{C}$	**CD**	**SFO**	**FO**
$\mathbf{SL}_<$	X	X	×	×	X	×	×	X	X	X
SL	X	?	?	?	?	?	?	X	X	X

Table 2.3: Transformations for the **SL** and $\mathbf{SL}_<$ languages, where X means the existence of polytime algorithm and × means that such an algorithm does not exist. Here $\wedge\mathbf{C}^*$ and $\vee\mathbf{C}^*$ assume that all input switch-list representations are defined on the same set of variables.

partial assignment is sufficient to enforce $g_i(x) = 0$ as can be easily seen from Figure 2.3a. On the other hand, g_j originates from f_j by setting $p = 1$ and the partial assignment x in this case implies $g_j(x) = 1$ which is again easy to see from Figure 2.3a). Thus g_i is distinct from g_j which finishes the proof of the claim.

2.5 Transformations

In this section we investigate which of the common transformations can be implemented to run in polynomial time for languages $\mathbf{SL}_<$ and **SL**. Precise definitions of the studied transformations can be found in [Darwiche and Marquis, 2002], we give only a short descriptions here:

$\neg\mathbf{C}$ *Negation* of a sentence.

$\wedge\mathbf{BC}$ *Bounded conjunction* of two sentences.

$\wedge\mathbf{C}^*$ *Conjunction* of any finite number of sentences on *the same set of variables*.

$\wedge\mathbf{C}$ *Conjunction* of any finite number of sentences.

$\vee\mathbf{BC}$ *Bounded disjunction* of two sentences.

$\vee\mathbf{C}^*$ *Disjunction* of any finite number of sentences on *the same set of variables*.

$\vee\mathbf{C}$ *Disjunction* of any finite number of sentences.

CD *Conditioning* of sentence f by term α, i.e. a partial assignment of values forced by satisfying α into f.

SFO *Singleton forgetting*, i.e. a transformation of f into $\exists x f$ for a variable x.

FO *Forgetting*. i.e. a transformation of f into $\exists X f$ for is a subset X of variables.

2.5.1 Negation ($\neg$C)

The negation of switch-list representation L_f representing function f can be produced in a constant time by flipping the function value of f at the vector $(0, \ldots, 0)$ while all switches remain the same. This is of course true for both **SL** and $\mathbf{SL}_<$ languages.

2.5.2 Conjunctions ($\wedge$BC, $\wedge$C* and $\wedge$C)

Let us consider the conjunction of two switch-list representations L_f and L_g representing functions f and g. We shall distinguish three cases.

1. *L_f and L_g respect the same order of variables, and moreover f and g are defined on the same set of variables* (i.e. the switches in L_f and L_g are vectors of the same length and their coordinates are indexed by variables in the same order). Observe, that if x is neither a switch in L_f nor a switch in L_g, it cannot be a switch in the switch-list representation of $f \wedge g$ which respects the same order of variables. Indeed, if $f(x) = f(x-1) = 0$ or $g(x) = g(x-1) = 0$ then $(f \wedge g)(x) = (f \wedge g)(x-1) = 0$, if $f(x) = f(x-1) = g(x) = g(x-1) = 1$ then $(f \wedge g)(x) = (f \wedge g)(x-1) = 1$. Hence the switch-list of $f \wedge g$ is a subset of the union of the two input switch-lists. Since both input switch-lists are ordered, they can be easily merged into an ordered list and during the merge each switch can be checked whether it is a switch of $f \wedge g$ or not. This can be done in linear time in the size of the input, and moreover this idea can be easily extended to any number of conjuncts and hence to an unbounded conjunction. Note, that to be able to delete duplicate switches and compute the values of $f \wedge g$ efficiently (where checking whether a given x is a switch of $f \wedge g$), it is essential that the input switch-list representations are ordered. This is one of the reasons why we maintain switch-lists, not just switch sets.

   This special case unfortunately covers only a subset of the $\mathbf{SL}_<$ language, however, it is a subset which occurs frequently in practical applications (which deal with several different Boolean functions on the same set of variables). It is also important to note, that contrary to switch-list representations, OBDDs do not support unbounded conjunction even in this restricted case (see Table 7 on page 242 and more details in [Darwiche and Marquis, 2002, pp. 259-261]).

2. *L_f and L_g respect the same order of variables but do not use the same set of variables.* In this case the switch-list representation of $f \wedge g$ that respects the same order of variables as L_f and L_g may be exponentially large with respect to the size of L_f and L_g. This may be true even if one of the sets of variables is a strict subset of the other. Consider $f(x_1, \ldots, x_n) = \bigvee_{i=1}^{n} \neg x_i$ (the all-one vector is the only false point of f) and $g(x_n) = x_n$. Clearly both L_f and L_g have just one switch with respect to the prescribed order of variables, however, function $f \wedge g$ in variables $x_1, \ldots, x_n$, has a switch-list of size $2^n - 1$, because every assignment $x_1, \ldots, x_n$, associated with an odd number except of the all-one vector is a model and every assignment associated with an even number is a non-model of $f \wedge g$. Thus the $\mathbf{SL}_<$ language does not in this case support bounded conjunction in polynomial time.

3. *L_f and L_g do not respect the same order of variables.* This case is open. We conjecture that the **SL** language does not support bounded conjunction in polynomial time nevertheless constructing examples where the conjunction is exponentially large with respect to all permutations of variables seems to be difficult.

2.5.3 Disjunctions ($\vee$BC, $\vee$C* and $\vee$C)

The complexity status for disjunctions is the same as for conjuctions thanks to the constant time negation.

2.5.4 Conditioning (CD)

Let f be a function on variables $x_1, \ldots, x_n$ and let x_i be an arbitrary variable. We interpret an assignment of variables $x_1, \ldots, x_{i-1}$ as a binary number ℓ, $0 \leq \ell \leq 2^{i-1} - 1$, and denote the corresponding block of consecutive vectors sharing the same prefix ℓ in the truth table of f as B_ℓ. Furthermore, we split B_ℓ into B^0_ℓ and B^1_ℓ depending on the value of x_i. We shall show how the switch-list representation of $f_1 = f_{|x_i=1}$ can be obtained from the switch-list representation of f by a single pass through the input switch-list (the process for $f_0 = f_{|x_i=0}$ is similar). We will write the vectors from the truth table of f as triples $(\ell, *, q)$ where $* \in \{0, 1\}$ represents the value of x_i and $\mathbf{0} \leq q \leq 2^{n-i} - 1 = \mathbf{1}$ is a binary number representing $x_{i+1}, \ldots, x_n$ (note that in a vector notation $\mathbf{0}$ is a shorthand for an all zero vector of length $n - i$ and $\mathbf{1}$ is a shorthand for an all one vector of length $n - i$). Similarly, we will write the vectors from the truth table of f_1 as pairs (ℓ, q).

When processing a switch of the type $(\ell, 0, q)$ we just count the parity p of the number of switches having the same prefix $(\ell, 0)$, i.e. the parity of the number of switches in the block B^0_ℓ. After we pass the last switch in B^0_ℓ, let us inspect the next switch in the list. If it differs from $s = (\ell, 1, \mathbf{0})$ (the first vector in B^1_ℓ), p is odd, and $\ell > 0$, we output $s' = (\ell, \mathbf{0})$. In this case s' which originates from s by removing the x_i coordinate becomes a switch of f_1, replacing the odd number of switches in B^0_ℓ. This is because $f_1(s') = f(s)$ differs from $f_1(\ell - 1, \mathbf{1}) = f(\ell - 1, 1, \mathbf{1})$. Note that $(\ell - 1, 1, \mathbf{1})$ is the last vector in $B^1_{\ell-1}$ and so $(\ell - 1, \mathbf{1})$ is a predecessor of s' in the truth table of f_1. If p is even or $\ell = 0$ then the switches in B^0_ℓ "disappear" without creating any switch for f_1.

When processing a switch of the type $s = (\ell, 1, q)$ where $q > 0$ we simply output $s' = (\ell, q)$ (all such switches of course "survive" the conditioning $x_i = 1$). If $s = (\ell, 1, \mathbf{0})$ (switch s is the first vector in B^1_ℓ) we output $s' = (\ell, \mathbf{0})$ only if p obtained from B^0_ℓ is even (this includes the case if there are no switches in B^0_ℓ) and $\ell > 0$. Clearly, also in this case s' is indeed a switch of f_1 because its function value differs from the last vector in $B^1_{\ell-1}$ which becomes its predecessor in the truth table of f_1. On the other hand, if p is odd or $\ell = 0$ then s "disappears" without creating a switch for f_1.

If the input switch-list representation has k switches, the above described process of conditioning on x_i takes $O(n)$ time per switch (and each switch is processed exactly once), and therefore can be implemented to run in $O(kn)$ time. Since the output switch-list representation has at most as many switches as the input switch-list representation, we can repeat the process $|S|$ times to achieve conditioning on any set S of variables in $O(kn^2)$ time. However, the $O(kn)$ time complexity can be maintained even in this case. If we divide the truth table of f into blocks with respect to the least significant variable in S (the rightmost one in the truth table), then instead of the alternating pattern of disappearing and surviving blocks for $|S| = 1$ (as described above) we get a pattern of possibly many disappearing blocks followed by a single surviving block. However, the

idea of conditioning can remain the same. We count the parity of the number of switches in between two surviving blocks, treat the first vector in the next surviving block accordingly, then output the remaining switches in the surviving block.

2.5.5 Forgetting (SFO and FO)

Let f be a function on variables $x_1, \ldots, x_n$ and let x_i be an arbitrary variable. We shall show how the switch-list representation of $f_i = \exists x_i f$ can be obtained from the switch-list representation of f in polynomial time. The procedure can be implemented directly on switch-list representations, but we find it more understandable if explained on interval representations which actually motivated the introduction of switch-list representations. We first compile the switch-list representation of f into an interval representation of f, then transform this into an interval representation of f_i, and finally compile back into an switch-list representation of f_i. The first and third steps take linear time, so it remains to describe the second step. We again (as for **CD**) consider the block structure $B_\ell = B^0_\ell \cup B^1_\ell$ for $0 \leq \ell \leq 2^{i-1} - 1$ of the truth table of f and also use the $(\ell, *, q)$ notation for the vectors from the truth table of f and (ℓ, q) for the vectors from the truth table of f_i.

When passing through the ordered list of intervals in interval representation of f, an interval $[a, b]$ is processed depending on whether $a \in B^0_\ell$ or $a \in B^1_\ell$ (for some ℓ) as follows. For $a = (\ell, 0, q)$ if

(a) $b = (\ell, 0, r)$ output $[(\ell, q), (\ell, r)]$,

(b) $b = (\ell, 1, r)$ output $[(\ell, \mathbf{0}), (\ell, r)]$ and $[(\ell, q), (\ell, \mathbf{1})]$,

(c) $b = (k, 0, r)$ for $k > \ell$ output $[(\ell, \mathbf{0}), (k, r)]$,

(d) $b = (k, 1, r)$ for $k > \ell$ output $[(\ell, \mathbf{0}), (k, \mathbf{1})]$.

On the other hand, for $a = (\ell, 1, q)$ if

(e) $b = (\ell, 1, r)$ output $[(\ell, q), (\ell, r)]$,

(f) $b = (k, 0, r)$ for $k > \ell$ output $[(\ell, q), (k, r)]$,

(g) $b = (k, 1, r)$ for $k > \ell$ output $[(\ell, q), (k, \mathbf{1})]$.

It is easy to check in each of the above seven cases that the models of f in the interval $[a, b]$ really translate to models of f_i in the specified output intervals. However, the output intervals may of course overlap (or even be identical, e.g. a pair of intervals obtained from (a) and (e) may be identical) so another "consolidation" pass through the output is necessary, which replaces any set of overlapping intervals with their union.

The above considerations imply that forgetting a single variable **SFO** can be performed in polynomial time. To see that the same is true for **FO**, we must analyze more carefully case (b), which is the only one when a single interval $[a, b]$ of f may produce two intervals of f_i. If it does, we will call $[a, b]$ a *splitting* interval. Note that if $q \leq r$, the two output intervals merge in the consolidation

pass, so $[a, b] = [(\ell, 0, q), (\ell, 1, r)]$ is splitting if and only if $q > r$. Hence $q \neq \mathbf{0}$ and $r \neq \mathbf{1}$ are necessary conditions for $[a, b] = [(\ell, 0, q), (\ell, 1, r)]$ to be a splitting interval.

Observe also, that for $a = (\ell, *, \mathbf{0})$ the interval $[a, b]$ produces only such intervals $[a', b']$ where $a' = (\ell, \mathbf{0})$, and for $b = (k, *, \mathbf{1})$ the interval $[a, b]$ produces only such intervals $[a', b']$ where $b' = (k, \mathbf{1})$.

Combining the facts from the previous two paragraphs implies, that if we forget the variables in the decreasing order of significance (most significant variables first), neither of the two intervals generated by a splitting interval $[a, b]$ can become a splitting interval when forgetting subsequent variables. Indeed, either the suffix of the left margin of the generated interval is all zeros (and stays all zeros from then on in subsequent forgetting), or the suffix of the right margin of the generated interval is all ones (and stays all ones). Thus, forgetting any subset of variables may altogether at most double the number of intervals on the output, which implies that **FO** can be done in polytime by repeating **SFO**.

2.6 Queries

In this section we investigate which of the common queries can be answered in polynomial time for languages $\mathbf{SL}_<$ and **SL**. As for transformations, precise definitions of the studied queries can be found in [Darwiche and Marquis, 2002], we again give only a short descriptions here:

CO *Consistency.* Test whether a sentence has a model.

VA *Validity.* Test whether a sentence S is a tautology (all 2^n vectors are models of S).

IM *Implicant Check.* For a sentence S and a given term T test if $T \models S$.

CE *Clausal Entailment.* For a sentence S a given clause C test if $S \models C$.

SE *Sentential Entailment.* For two given sentences S, S' test if $S \models S'$.

EQ *Equivalence.* For two given sentences S, S' test if $S \equiv S'$.

CT *Model Counting.* Output the number of models of a sentence.

ME *Model Enumeration.* Output all models of a sentence.

Since all of the above queries can be answered in polynomial time time for the $\mathbf{OBDD}_<$ language (see Table 2.1 or [Darwiche and Marquis, 2002, Table 5]), the same is true for languages $\mathbf{SL}_<$ and **SL** which can be both compiled into $\mathbf{OBDD}_<$ in polynomial time. It is clear, however, that for most queries direct algorithms using switch-list representations are much more efficient than indirect algorithms that first compile switch-list representation into an OBDD (which takes $O(k^2n^3)$ time for switch-list representation with k switches on n variables) and only then answer the query. Obviously, some queries are completely trivial for switch-list representations (consistency, validity) and some can be easily implemented using polynomial time conditioning (clausal entailment, implicant check). It should be

	CO	VA	IM	CE	SE*	SE	EQ	CT	ME
$\mathbf{SL}_<$	$O(1)$	$O(1)$	$O(kn)$	$O(kn)$	$O(kn)$	$O(k^2n^2)$	$O(kn)$	$O(kn)$	$O(mn)$
SL	$O(1)$	$O(1)$	$O(kn)$	$O(kn)$	$O(k^2n^3)$	$O(k^2n^3)$	$O(k^2n^3)$	$O(kn)$	$O(mn)$

Table 2.4: Time complexity of queries for the $\mathbf{SL}_<$ and **SL** languages where n is the number of variables, k is the number of switches in the input switch-list representation, and m is the number of models. SE* additionally assumes that both input switch-list representations are defined on the same set of variables.

also noted that switch-list representations are very well suited for model counting — a linear number of arithmetic operations (subtractions) on n-bit numbers suffices, and for model enumeration. However, for (the general case of) sentential entailment and equivalence check we currently have no direct algorithms, and so the indirect approach using a compilation to OBDD is the only one we can use now.

In the rest of this section we will describe a polynomial time algorithm for each of the considered queries and determine its time complexity (see Table 2.4 for a summary). Let us assume that the input is a switch-list representation consisting of k switches representing function f on n variables (i.e. the input switch-list representation has size $O(nk)$).

2.6.1 Consistency (CO) and Validity (VA)

To test that f is valid (all assignments are models) it suffices to check $k = 0$ and $f(0, \ldots, 0) = 1$. This takes $O(1)$ time. To test that f is consistent (has a model) it suffices to check $k > 0$ or $f(0, \ldots, 0) = 1$. This also takes $O(1)$ time. Of course, checking consistency of a function is the same as checking non-validity of its negation (recall from Section 2.5 that a negation can be constructed in $O(1)$ time).

2.6.2 Implicant Check (IM) and Clausal Entailment (CE)

Let T be a term (simple conjunction of literals). We have shown in Section 2.5 that conditioning on a subset of literals takes $O(kn)$ time. Therefore it suffices to set each literal in T to true and then to check the validity of the switch-list representation resulting from this conditioning in $O(1)$ time to determine whether T is an implicant of f or not.

Checking that clause C is an implicate of f is equivalent to checking whether a term $\neg C$ is an implicant of $\neg f$. Thus, due to the fact that switch-list representation of $\neg f$ can be constructed from the switch-list representation of f in $O(1)$ time, also clausal entailment can be checked in $O(kn)$ time.

2.6.3 Sentential Entailment (SE)

Let f and g be two Boolean functions and L_f, L_g their switch-list representations with k_f and k_g switches respectively. Let us denote $k = \max\{k_f, k_g\}$. **SE** amounts to checking whether f implies g. This is the same as checking that $\neg f \vee g$ is valid. Let us distinguish three cases in a similar fashion as we did for conjunctions and disjunctions in Section 2.5.

1. L_f, L_g have the same order of variables and the same set of variables. In this case we can check **SE** in $O(kn)$ time using negation, disjunction, and validity check.

2. L_f, L_g have the same order of variables and different sets of variables. In this case we cannot use the same approach as above because the switch-list representation of the disjunction may be exponentially large with respect to the input. Hence we compile L_f, L_g into OBDDs G_f, G_g both respecting the same fixed order of variables as L_f, L_g. This can be done in $O(k^2n^2)$ time and the output is of size $O(kn)$ since the first part of the compilation from Section 2.3.1 suffices to produce G_f, G_g (no need to change the order of variables). Now we can take a negation of G_f in $O(1)$ time, disjoin it with G_g in $O(|G_f| \times |G_g|) = O(k^2n^2)$ time, reduce the result to a canonic OBDD in $O(k^2n^2)$ time, and check validity (see [Wegener, 2000, Theorem 3.3.6. pp. 56] for the complexity of these operations with OBDDs, more details can be found in [Bryant, 1986, Meinel and Theobald, 1998]).

3. L_f, L_g have different orders of variables. In this case we again compile into OBDDs but this time we have to change the order of variables for one of the input representations. So let us assume that we produce G_f of size $O(kn)$ in $O(k^2n^2)$ time as above (we keep the order of variables) and G_g of size $O(kn^2)$ in $O(k^2n^3)$ time by the compilation algorithm from Section 2.3.2 (here we change the order of variables to the one in G_f). Now we proceed as above negating, disjunction and checking validity. In this case we need $O(|G_f| \times |G_g|) = O(k^2n^3)$ time.

2.6.4 Equivalence Check (EQ)

For **EQ** the the situation is simpler than for **SE** since when testing equivalence we may assume that f and g are defined on the same set of variables. If L_f, L_g have the same order of variables then **EQ** can be obviously tested in $O(kn)$ time by performing two **SE** checks. In fact, in this case if $f \equiv g$ then L_f and L_g must be identical, as for a given function and a given order of variables its switch-list representation is uniquely defined. So instead of two **SE** checks we may just test in $O(kn)$ whether L_f and L_g are identical, which is easy to do as the switch-lists are ordered.

If the order of variables differs, we proceed similarly as in **SE** by compiling L_f of size $O(kn)$ into G_f of size $O(kn)$ in $O(k^2n^2)$ time and L_g of size $O(kn)$ into G_g of size $O(kn^2)$ in $O(k^2n^3)$ time. Now we can save some time by checking the equivalence of f and g by testing the isomorphism of the reduced forms of G_f and G_g in $O(kn^2)$ time [Wegener, 2000, Theorem 3.3.1. pp. 51], but asymptotically this does not help as the complexity of the compilation step dominates.

2.6.5 Model Counting (CT), Model Enumeration (ME)

Let $p_1, \ldots, p_k$ be the switches of f understood as n-bit binary numbers. If $f(1, \ldots, 1) = 1$ then set $p_{k+1} = 2^n$. Now there are two cases. If $f(0, \ldots, 0) = 1$ then set $p_0 = 0$ and the number of models equals to $\sum_{i \in I}(p_i - p_{i-1})$, where I is set of odd indices, if $f(0, \ldots, 0) = 0$ then the number of models equals to

$\sum_{i \in I}(p_i - p_{i-1})$, where I is set of even indices. In either case, the computation consists of $O(k)$ addition and subtraction operations on n-bit numbers which takes $O(kn)$ time.

Model enumeration can be performed in a similar manner as above by outputting models between pairs of switches p_{i-1} and p_i for $i \in I$ depending on the values of $f(0, \ldots, 0)$ and $f(1, \ldots, 1)$ as in model counting. The time complexity is of course linear in the size of the output and takes $O(mn)$ time, where m is the number of models.

2.7 Conclusions

The main aim of this chapter is to include the languages $\mathbf{SL}_<$ and **SL** into the Knowledge Compilation Map [Darwiche and Marquis, 2002] and to argue that they may in some situations constitute reasonable target languages for knowledge compilation. This aim is is justified by completing three subtasks:

1. derive the relative succinctness of $\mathbf{SL}_<$ and **SL** compared to the languages already considered in [Darwiche and Marquis, 2002],

2. establish the complexity status of common transformations for $\mathbf{SL}_<$ and **SL**, and

3. do the same for common queries.

This goal is achieved with few open problems remaining, namely the complexity of conjunctions and disjunctions for **SL**.

The results in this chapter are dependent on the fact that vectors in the truth table are assumed to be ordered by the natural lexicographical order. i.e. by standard inequalities when vectors are interpreted as binary numbers. There are other orders that are quite natural and can be generated effectively. For instance, one can order vectors based on the number of ones (and complement this by some natural order on the sets of vectors with the same number of ones). Such an order has quite different properties, note that e.g. the parity function which has an exponentially large switch-list representation with respect to the standard order of vectors has a linear size switch-list representation with respect to this less standard one. Examining the properties of switch-list representations with respect to non-standard orders of vectors may be the subject of a future study.

3. Biclique Satisfiability

The results presented in this chapter were published in [Chromý and Kučera, 2019] and they extend the results from the master thesis of the applicant [Chromý, 2015]. In particular, we introduce a SAT based approach to checking biclique satisfiability and experimental comparison with the heuristic approach which was described in thesis [Chromý, 2015]. Most of the work on the encoding and evaluation was performed by the applicant of this thesis. Petr Kučera was the expert supervisior of the work and the text and was a source of many important advices and comments.

Let us start with introducing the notion of incidence graph which serves as a basis for the definition of biclique satisfiable formulas. Given a formula f in *conjunctive normal form* (CNF) we consider its *incidence graph* $I(f)$ defined as follows. $I(f)$ is a bipartite graph with one part consisting of the variables of f and the other part consisting of the clauses of f. An edge $\{x, C\}$ for a variable x and a clause C is in $I(f)$ if x or $\neg x$ appears in C. It was observed by the Aharoni and Linial [1986] and Tovey [1984] that if $I(f)$ admits a matching of size m (where m is the number of clauses in f), then f is satisfiable. Later the formulas satisfying this condition were called *matched formulas* by Franco and Van Gelder [2003]. Since a matching of maximum size in a bipartite graph can be found in polynomial time (see e.g. Hopcroft and Karp [1971], Lovász and Plummer [1986]), one can check efficiently whether a given formula is matched.

It is clear that if f is a formula on n variables and m clauses then f can be matched only if $m \leq n$. Franco and Van Gelder [2003] asked an interesting question: What is the probability that a formula f is matched depending on the ratio m/n? We can also ask if the property "being matched" exhibits a phase transition.

A phase transition was studied in context of satisfiability [Cheeseman et al., 1991, Gent and Walsh, 1994, Mitchell et al., 1992, Dubois et al., 2000, Connamacher and Molloy, 2012]. A k-CNF formula is a formula with exactly k literals in each clause. The so-called satisfiability threshold for a given k is a value r_k satisfying the following property: A random formula f in k-CNF on n variables and m clauses is almost surely satisfiable if $m/n < r_k$ and it is almost surely unsatisfiable if $m/n > r_k$. For instance the value r_3 is approximately 4.3 [Dubois et al., 2000, Connamacher and Molloy, 2012].

In the same sense we can study threshold for property "being matched". It was shown by Franco and Van Gelder [2003] that a 3-CNF f on n variables and m clauses is almost surely matched if $m/n < 0.64$. This is merely a theoretical lower bound and the experiments we have performed as part of the master thesis [Chromý, 2015] suggest that the threshold is in fact much higher. Moreover, we observed that the property "being matched" has a sharp threshold or phase transition as a function of ratio m/n. We present the results in Section 3.2 to have the presentation complete, later these results are compared with the results on biclique satisfiable formulas.

Matched formulas have an interesting property: Consider a matched formula f and modify it by switching polarity of any occurrence of a selected literal (i.e. change a positive literal x into a negative literal $\neg x$ or vice versa). The formula

produced by this operation is matched and thus satisfiable as well. This is because the definition of incidence graph completely ignores the polarities of variables. The formulas with this property were called *var-satisfiable* by Szeider [2005] and they form a much bigger class than matched formulas. Unfortunately, it was shown by Szeider [2005] that the problem of checking whether a given formula f is var-satisfiable is complete for the second level of polynomial hierarchy.

Szeider [2005] defined a subclass of var-satisfiable formulas called *biclique satisfiable formulas* which extends matched formulas. It was shown by Szeider [2005] that checking if f is biclique satisfiable is an NP-complete problem. In this chapter we describe a heuristic algorithm to test whether a formula is biclique satisfiable. Our heuristic algorithm is based on a heuristic for covering a bipartite graph with bicliques described by Heydari et al. [2007]. We test our heuristic algorithm experimentally on random formulas. Our heuristic algorithm is incomplete meaning that whenever it finds that a formula is biclique satisfiable, then it is so, but it may happen that a formula is biclique satisfiable even though our algorithm is unable to detect it. In order to check the quality of our heuristic, we propose a SAT based approach to checking biclique satisfiability of a formula. We compare both approaches on random formulas.

In Section 3.1 we recall some basic definitions and related results used in the rest of this chapter. In Section 3.2 we give the results of experiments on matched formulas which were performed in the master thesis [Chromý, 2015]. In Section 3.3 we describe our heuristic algorithm for determining if a formula is biclique satisfiable and we give the results of its experimental evaluation which was presented at master thesis [Chromý, 2015]. The new results [Chromý and Kučera, 2019] are described in Section 3.4, where we describe a SAT based approach to checking biclique satisfiability and compare it experimentally with the heuristic approach. We close this chapter with concluding remarks in Section 3.5.

3.1 Definitions and Related Results

In this section we shall introduce necessary notions and results used in this chapter.

3.1.1 Graph Theory

We use the standard graph terminology (see e.g. Bollobás [1998]). A *bipartite graph* $G = (V_v, V_c, E)$ is a triple with vertices split into two parts V_v and V_c and the set of edges E satisfying that $E \subseteq V_v \times V_c$. Given a bipartite graph G we shall also use the notation $V_v(G)$ and $V_c(G)$ to denote the vertices in the first and in the second part respectively. For two natural numbers n, m we denote by $K_{n,m}$ the *complete bipartite graph* (or a *biclique*) that is the graph $K_{n,m} = (V_v, V_c, E)$ with $|V_v| = n$, $|V_c| = m$ and $E = V_v \times V_c$.

Given a bipartite graph $G = (V_v, V_c, E)$ the *degree* of a vertex $v \in V_v \cup V_c$ is the number of incident edges. A subset of edges $M \subseteq E$ is called a *matching* of G if every vertex in G is incident to at most one edge in M. A vertex v is *matched* by matching M if v is incident to some edge from M. M is a *maximum matching* if for every other matching M' of G we have that $|M| \geq |M'|$. There is a polynomial algorithm for finding a maximum matching of a bipartite graph $G = (V_v, V_c, E)$

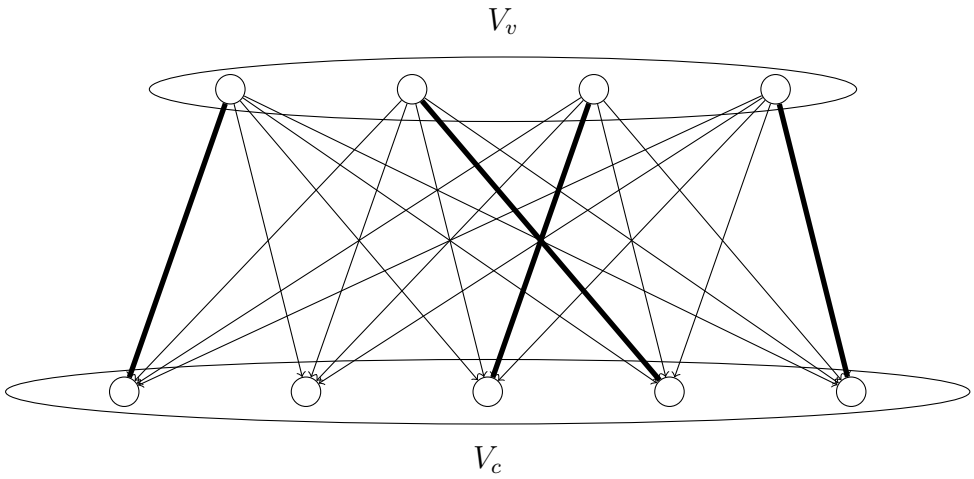


Figure 3.1: A complete bipartite graph $K_{4,5}$. Thick edges represent one of maximum matchings of $K_{4,5}$.

which runs in $O(|E|\sqrt{|V_v| + |V_c|})$ [Hopcroft and Karp, 1971, Lovász and Plummer, 1986].

3.1.2 Boolean Formulas

Let us now recall the definition of probability space $\mathcal{M}^k_{m,n}$ introduced by Franco and Van Gelder [2003].

Definition 2 (Franco and Van Gelder [2003])**.** *Let $V_n = \{v_1, \ldots, v_n\}$ be a set of Boolean variables and let $L_n = \{v_1, \neg v_1, \ldots, v_n, \neg v_n\}$ be the set of literals over variables in V_n. Let $\mathcal{C}^k_n$ be the set of all clauses with exactly k variable-distinct literals from L_n. A random formula in probability space $\mathcal{M}^k_{m,n}$ is a sequence of m clauses from $\mathcal{C}^k_n$ selected uniformly, independently, and with replacement.*

3.1.3 Matched Formulas

Let $f = C_1 \wedge \cdots \wedge C_m$ be a CNF formula on n variables $X = \{x_1, \ldots, x_n\}$. We associate a bipartite graph $I(f) = (X, f, E)$ with f (also called the *incidence graph* of f), where the vertices correspond to the variables in X and to the clauses in f. A variable x_i is connected to a clause C_j (i.e. $\{x_i, C_j\} \in E$) if C_j contains x_i or $\neg x_i$. A CNF formula f is *matched* if $I(f)$ admits a matching of size m, i.e. if there is a matching which pairs each clause with a unique variable. It was observed by Aharoni and Linial [1986] and Tovey [1984] that a matched CNF is always satisfiable since each clause can be satisfied by the variable matched to the given clause. A variable which is matched to some clause in a matching M is called *matched* in M.

We can see that checking if a formula is matched amounts to checking if the size of a maximum matching of $I(f)$ is $|V_C|$ (size of partition which is asociated with clauses). This can be done in time $O(\ell\sqrt{m+n})$ where m denotes the number of clauses in f, n denotes the number of variables in f, and ℓ denotes the total length of formula f that is the sum of the widths of the clauses in f.

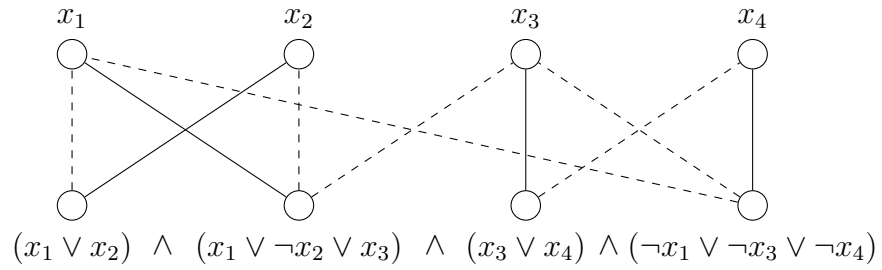


Figure 3.2: An incidence graph of a matched formula with matching highlighted by solid lines.

The following result on density of matched formulas in the probability space $\mathcal{M}^k_{m,n}$ was shown by Franco and Van Gelder [2003].

Theorem 10 ([Franco and Van Gelder, 2003])**.** *Under $\mathcal{M}^k_{m,n}$ $k \geq 3$, the probability that a random formula f is matched tends to 1 if $r \equiv m/n < 0.64$ as $n \to \infty$.*

3.1.4 Biclique Satisfiable Formulas

One of the biggest limitations of matched formulas is that if f is a matched formula on n variables and m clauses, then $m \leq n$. To overcome this limitation while keeping many nice properties of matched formulas, Szeider [2005] introduced biclique satisfiable formulas.

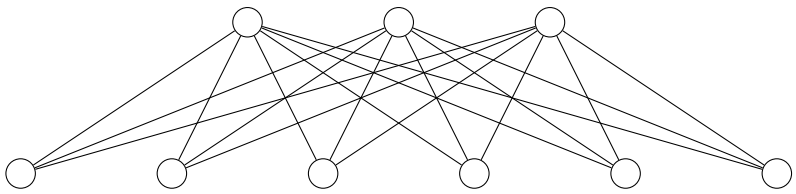


Figure 3.3: A graph of a bounded biclique $K_{3,6}$. Note that maximum biclique with 4 vertices in the first part can have at most 7 vertices in the second part.

We say that a biclique $K_{n,m}$ is *bounded* if $m < 2^n$. Let f be a CNF on n variables and m clauses and let us assume that $I(f) = K_{n,m}$ where $m < 2^n$. Then f is satisfiable [Szeider, 2005]. This is because we have $m < 2^n$ clauses each of which contains all n variables. Each of these clauses determines exactly one unsatisfying assignment of f, but there is 2^n assignments in total. Thus one of these must be satisfying.

Based on this observation we can define biclique satisfiable formulas [Szeider, 2005]. We say, that a bipartite graph $G = (V_v, V_c, E)$ has a *bounded biclique cover* if there exists a set of bounded bicliques $\mathcal{B} = \{B_1, \ldots, B_k\}$ satisfying the following conditions.

- every $B_i, i = 1, \ldots, k$ is a subgraph of G,
- for any pair of indices $1 \leq i < j \leq k$ we have that $V_v(B_i) \cap V_v(B_j) = \emptyset$, and
- for every $v \in V_c(G)$ there is a biclique $B_i, i = 1, \ldots, k$ such that $v \in V_c(B_i)$.

If every biclique $B_i \in \mathcal{B}$ in the cover satisfies that $|V_v(B_i)| \leq k$, then we say the graph G has a bounded k-biclique cover. A formula f is *(k-)biclique satisfiable* if its incidence graph $I(f)$ has a bounded (k-)biclique cover.

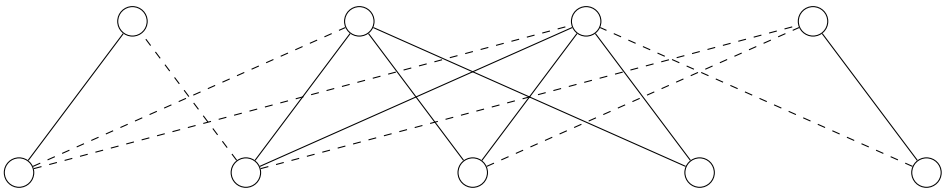


Figure 3.4: The incidence graph of a formula with a bounded biclique cover. Solid lines are the biclique cover of the incidence graph.

It can be easily shown that any biclique satisfiable formula is indeed satisfiable but it is an NP-complete problem to decide if a formula is biclique satisfiable even if we only restrict to 2-biclique satisfiable formulas. For proofs of both results see [Szeider, 2005]. On the other hand it is immediate that 1-satisfiable formulas are matched formulas, because a single edge is a bounded biclique.

3.1.5 Generating experimental data

Whether a formula f in CNF is matched or not depends only on its incidence graph $I(f)$. Instead of random formulas from the probabilistic space $\mathcal{M}^k_{m,n}$ we thus consider random bipartite graphs $G = (V_v, V_c, E)$ from probabilistic space $\mathcal{G}^k_{m,n}$.

Definition 3. *Probability space $\mathcal{G}^k_{m,n}$ is defined as follows. A random bipartite graph $G \in \mathcal{G}^k_{m,n}$ is a bipartite graph with parts V_v, V_c where $|V_v| = n$, $|V_c| = m$. Each vertex $v \in V_c$ has k randomly uniformly selected neighbours from V_v.*

In our experiments we generated bipartite graphs $G \in \mathcal{G}^k_{m,n}$. Since we consider choosing clauses in formula $f \in \mathcal{M}^k_{m,n}$ with replacement, we can have several copies of the same clause in f. It follows that given a bipartite graph $G \in \mathcal{G}^k_{m,n}$, we have exactly $2^k m$ formulas $f \in \mathcal{M}^k_{m,n}$ which have $I(f) = G$ — each vertex $c \in V_c$ can be replaced with 2^k different clauses with setting polarities to variables $x \in V_v$ adjacent to v in G. In particular, the probability that a random formula $f \in \mathcal{M}^k_{m,n}$ is matched is the same as the probability that a random bipartite graph $G \in \mathcal{G}^k_{m,n}$ admits a matching of size m. The same holds for the biclique satisfiability.

3.2 Phase Transition on Matched Formulas

In this section we shall describe the results of experiments performed on matched formulas which was presented in master thesis by Chromý [2015]. In particular we were interested in phase transition of k-CNF formulas with respect to the property "being matched" depending on the ratio of the number of clauses to the number of variables. We will also compare the results with the theoretical bound proved by Franco and Van Gelder [2003] (see Theorem 10).

Note that the graphs in $\mathcal{G}^k_{m,n}$ correspond to incidence graphs of k-CNFs on n variables and m clauses. In particular, the probability that a random formula

$f \in \mathcal{M}^k_{m,n}$ is matched is the same as the probability that a random bipartite graph $G \in \mathcal{G}^k_{m,n}$ admits a matching of size m. In the experiments we were working with random bipartite graphs and we identified them with random formulas. The difference between a random formula f and a random bipartite graph G is in polarities of variables which have no influence on whether the formula is matched or not.

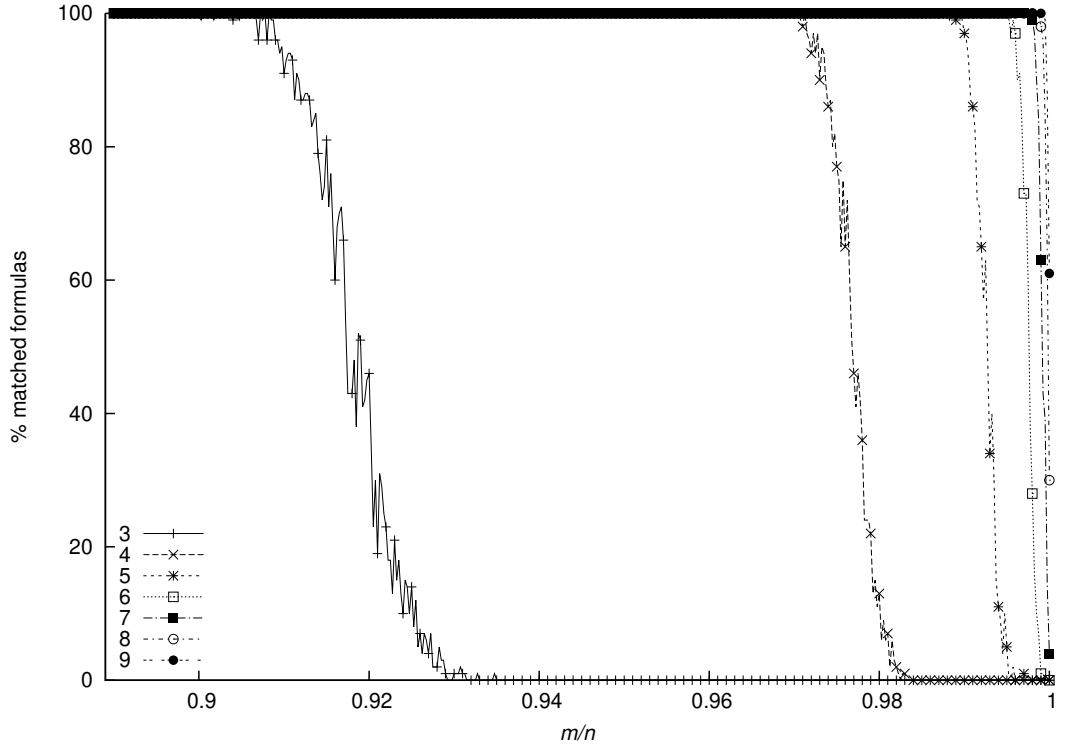


Figure 3.5: Results of experiments on random graph $\mathcal{G}^k_{m,n}$ with $n = 4000$ and $k = 3, \ldots, 9$. The horizontal axis represents the ratio m/n. The vertical axis represents the percentage of graphs which admit matching of size m. For each k and m was generated a 1000 random graphs from $\mathcal{G}^k_{m,n}$.

	3-CNF		4-CNF		5-CNF		6-CNF		7-CNF	8-CNF	9-CNF	10-CNF
n	low	high	low	high	low	high	low	high	low	low	low	low
100	0.85	0.98	0.95	1	0.97	1	0.98	1	1	1	1	1
200	0.88	0.96	0.96	0.99	0.98	1	0.99	1	1	1	1	1
500	0.89	0.95	0.96	0.99	0.99	1	1	1	1	1	1	1
1000	0.895	0.939	0.97	0.989	0.986	0.999	0.99	0.995	0.997	0.998	0.999	0.999
2000	0.903	0.9325	0.97	0.985	0.988	0.9965	0.995	0.9995	0.998	0.9985	0.9995	0.9995
4000	0.909	0.929	0.9715	0.982	0.99	0.995	0.995	0.992	0.998	0.999	0.9995	0.9995

Table 3.1: Phase transition intervals of matched formulas as two values *high* and *low*. We provide only *low* value for $k \geq 7$, because the *high* value was 1 in this case for all configurations.

In our experiments we considered values of number of variables $n = 100$, 200, 500, 1000, 2000, 4000 and $k = 3, 4, \ldots, 10$. For each such pair n, k we have generated 1000 random graphs $G \in \mathcal{G}^k_{m,n}$ for ratio $m/n = 0.64, 0.65, \ldots, 1$. Figure 3.5 shows the graph with the results of experiments for value $n = 4000$. The graph contains a different line for each value of $k = 3, \ldots, 9$ which shows the percentage of graphs which admit matching of size m among the generated random graphs

depending on the ratio $m/n = 0.64, 0.65, \ldots, 1$. The complete results of the experiments are shown in Table 3.1. For each value of k we distinguish two values *high* and *low* where only 1% of the graphs generated in $\mathcal{G}^k_{m,n}$ with $m/n \geq high$ admit matching of size m, and on the other hand 99% of the graphs generated in $\mathcal{G}^k_{m,n}$ with $m/n < low$ admit matching of size m.

We can see that for higher values of n the interval $[low, high]$ gets narrower and we can thus claim that the property "being matched" indeed exhibits a phase transition phenomenon. Moreover, we can say that the average of values *low* and *high* limits to the threshold of this phase transition. We can see that the threshold ratio for $k = 3$ is around 0.92 which is much higher than the theoretical bound 0.64 proven by Franco and Van Gelder [2003] (see Theorem 10). In all configurations with $k \geq 7$ the *high* value was 1 while the *low* value was close to 1 as well. Thus in the experiments we made with $k \geq 7$ even in the case $m = n$ almost all of the randomly generated graphs admitted matching of size m.

3.3 Bounded Biclique Cover Heuristic

The class of biclique satisfiable formulas form a natural extension to the class of matched formulas. This class was introduced by Szeider [2005], where the author showed that it is NP-complete to decide whether a given formula f is biclique satisfiable. Recall that this decision is equivalent to checking if the incidence graph $I(f)$ has a bounded biclique cover. In this section is described a heuristic algorithm for finding a bounded biclique cover introduced in master thesis by Chromý [2015]. The algorithm we introduce is incomplete, which means that it does not necessarily find a bounded biclique cover if it exists, on the other hand the algorithm runs in polynomial time.

3.3.1 Description of Heuristic Algorithm

Our heuristic approach is described in Algorithm 1. It is based on a heuristic algorithm for finding a smallest biclique cover of a bipartite graph described by Heydari et al. [2007]. The Algorithm 1 expects three parameters. The first two parameters are a bipartite graph G and an integer t which restricts the size of the first part of bounded bicliques used in the cover, in other words only bicliques S satisfying that $|V_v(S)| \leq t$ are included in the cover which is output by the Algorithm 1. The last parameter used in Algorithm 1 is the strategy for selecting a seed.

Let G be a bipartite graph $G = (V_v, V_c, E)$. A *seed* in G is a biclique S which is a nonempty subgraph of G with $|V_v(S)| = 2$ and $V_c(S) \neq \emptyset$. We say that S is a *maximal seed* if there is no seed S' so that $V_v(S) = V_v(S')$ and $V_c(S) \subsetneq V_c(S')$.

After initializing an empty cover $\mathcal{C}$, Algorithm 1 starts with a pruning step (`unitGPropagation`) which is used also in the main loop. In this step a simple reduction rule is repeatedly applied to the graph G: If a vertex $C \in V_c$ is present in a single edge $\{v, C\}$, then this edge has to be added into the cover $\mathcal{C}$ as a biclique in order to cover C. In this case vertices v and C with all edges incident to v are removed from the graph G. If a vertex $C \in V_c$ which is not incident to any edge in E is encountered during this process, Algorithm 1 fails and returns an empty cover.

Algorithm 1 continues with generating a list $\mathcal{S}$ of all maximal seeds induced by all pairs $\{v_i, v_j\} \subseteq V_v, i < j$. The input graph is modified during Algorithm 1 by removing edges and vertices. In the following description $G = (V_v, V_c, E)$ always denotes the current version of the graph.

The main loop of Algorithm 1 repeats while there are some seeds available and G does not admit a matching of size $|V_c|$. This is checked by calling function `testMatched` which also adds the matching to $\mathcal{C}$ if it is found.

The body of the main loop starts with selecting one seed S by function `chooseSeed`. This choice is based on a given strategy. We consider three strategies for selecting a seed: Strategy S_{min} chooses a seed with the smallest second part. Strategy S_{max} chooses a seed with the largest second part. And strategy S_{rand} chooses a random seed. Seed S is then expanded by repeatedly calling `expandSeed`. This function selects a vertex $v \in V_v \setminus V_v(S)$ which maximizes the size of the second part of the biclique induced in G with left part being $V_v(S) \cup \{v\}$ (the second part is induced to be all the vertices incident to all vertices in $V_v(S) \cup \{v\}$). The expansion process continues while the size of the first part $V_v(S)$ satisfies the restriction imposed by the parameter t and while S is not a bounded biclique (that is while $2^{|V_v(S)|} \leq |V_c(S)|$).

If the expansion process ends due to the restriction on the size $|V_v(S)|$ given by t, S is not necessarily a bounded biclique. In this case we use a function `restrictSeed` which simply removes randomly choosen vertices from $V_c(S)$ so that S becomes a bounded biclique.

Once a bounded biclique S is found, it is removed from the graph and it is added to the cover $\mathcal{C}$. This is realized by a function `removeBiclique` which simply sets $V_v \leftarrow V_v \setminus V_v(S)$, $V_c \leftarrow V_c \setminus V_c(S)$, and $E \leftarrow E \cap (V_v \times V_c)$. Then we call `unitGPropagation` to prune the graph. After that the function `removeInvalidSeeds` removes from $\mathcal{S}$ all seeds S' with $V_v(S') \cap V_v(S) \neq \emptyset$. For remaining seeds $S' \in \mathcal{S}$ the function sets $V_c(S') \leftarrow V_c(S') \cap V_c$.

After the loop finishes, the current cover $\mathcal{C}$ is returned.

Let us estimate the running time of Algorithm 1. Let us denote $n = |V_v|$, $m = |V_c|$, and $\ell = |E|$ (also corresponds to the length of a formula). Generating all seeds requires time $O(n\ell)$. The main loop will repeat at most n times, because we cannot have more bounded bicliques than the number of vertices in V_v. In case that the second part is bigger then the first one, graph cannot be an incidence graph of a matched formula, so checking if a graph admits a matching of size $|V_c|$ has constant time complexity if $m > n$. In case that $m \leq n$ function `testMatched` will run in $\mathcal{O}(\ell\sqrt{n})$ [Hopcroft and Karp, 1971, Lovász and Plummer, 1986]. All other steps within the main loop (including the pruning step) can be performed in time $O(n\ell)$ and thus the complexity of our heuristic is $\mathcal{O}(n^2\ell)$.

If a nonempty set of bicliques $\mathcal{C}$ is returned by Algorithm 1, then it is a bounded biclique cover of G. It should be noted that the opposite implication does not necessarily hold; if the seeds are chosen badly then Algorithm 1 may fail even if there is some bounded biclique cover in G. In the next section we aim to evaluate Algorithm 1 experimentally.

Data: Bipartite graph $G(V_v, V_c, E)$, $t \in \{2, \ldots, |V_v|\}$ — maximal size of $|V_v(S)|$ for a biclique S which we put into the cover and a seeds selection strategy $\texttt{st} \in \{S_{min}, S_{rand}, S_{max}\}$.
Result: biclique cover $\mathcal{C}$ of graph G if a heuristic found one, $\emptyset$ otherwise

```
C ← ∅
if unitGPropagation(G, C) fails then return ∅                    // O(nm)
S ← generateSeeds(G)                                             // O(nℓ)
while |S| > 0 and not testMatched(G, C) do                       // O(ℓ√n)
    S ← chooseSeed(S, st)                                        // O(n²)
    while |V_v(S)| < t ∧ 2^|V_v(S)| ≤ |V_c(S)| do
        S ← expandSeed(S)                                        // O(ℓ + m)
    end
    if 2^|V_v(S)| ≤ |V_c(S)| then S ← restrictSeed(S)            // O(|V_c(S)|)
    G ← removeBiclique(G, S)                                     // O(ℓ)
    C ← C ∪ {S}
    if unitGPropagation(G, C) fails then return ∅                // O(nm)
    S ← removeInvalidSeeds(S)                                    // O(nℓ)
end
return C
```

Algorithm 1: An heuristic for checking if there is a bounded biclique cover of a bipartite graph $G = (V_v, V_c, E)$. The complexity of each step is noted in comments where we consider $n = |V_v|$, $m = |V_c|$, and $\ell = |E|$.

3.3.2 Experimental Evaluation of Heuristic Algorithm

In this section we shall describe the experiments performed with Algorithm 1 described in Section 3.3.1.

Algorithm 1 works with bipartite graphs. We have tested proposed heuristic on bipartite graphs G from the probabilistic space $G \in \mathcal{G}^k_{m,n}$ with $n = 100, 200$ and with the degrees of vertices in the second part being $k = 3, \ldots, 100$. This corresponds to formulas in k-CNF for these values. We have considered different sizes of the second part given by ratios $m/n = 1, 1.01, \ldots, 1.5$. The upper bound 1.5 was chosen because we were mainly interested in bounded 2-biclique cover. For graphs with $m/n > 1.5$ there is no bounded 2-biclique . For comparison, we have also performed experiments with unrestricted sizes of bounded bicliques and we have tried the three strategies S_{min}, S_{rand} and S_{max} for selecting a seed. In the experiments we checked whether Algorithm 1 found a bounded biclique cover of a given random graph generated according to the above mentioned parameters.

Due to time complexity of Algorithm 1 we have only generated a hundred random graphs in $\mathcal{G}^k_{m,n}$ for each configuration (given by a strategy, bound t on the size of $V_v(S)$ of each biclique, and ratio m/n).

Table 3.2 summarizes the results of our experiments. Each row corresponds to a combination of a strategy for selecting a seed and a bound imposed on the size of biclique (superscript $2, 3$ for bounded 2-biclique cover, ∞ for general bounded biclique cover). Each column corresponds to a ratio m/n, we have included only ratios 1, 1.1, 1.2, 1.3, 1.4, and 1.5 in the table. For each configuration we have two bounds *low* and *high* on degree k of vertices in the second part V_c of graph

	1		1.1		1.2		1.3		1.4		1.5	
	low	high	low	high	low	high	low	high	low	high	low	high
$S^{2,3}_{min}$	4	5	5	6	7	8	9	15	13	24	33	47
$S^{2,3}_{rand}$	4	5	5	6	7	8	9	15	13	24	33	47
$S^{2,3}_{max}$	4	5	5	6	7	8	9	15	14	24	41	87
S^{∞}_{min}	4	5	5	39	7	40	9	40	39	42	36	44
S^{∞}_{rand}	4	5	5	39	7	40	9	40	39	42	36	44
S^{∞}_{max}	4	5	5	6	7	17	15	20	18	22	20	23

Table 3.2: Results of experiments with Algorithm 1 on graphs with size of second part$|V_v| = 100$. Each pair of columns *low* and *high* represents a phase transition interval. Each row corresponds to one strategy. A more detailed explanation can be found in the main text.

G. Algorithm 1 succeeded only on 1% of graphs with degree $k \leq low$ and on the other hand it succeeded on 99% of graphs with degree $k \geq high$.

We can see that for a bounded 2-biclique cover $S^{2,3}$, the strategies $S^{2,3}_{min}$ and $S^{2,3}_{rand}$ are never worse than $S^{2,3}_{max}$ and that they even get better for higher ratios. This makes S_{rand} the best strategy for seed size restriction $S^{2,3}$ — it is easiest to implement and randomness means that repeated calls of Algorithm 1 may eventually lead to finding a biclique cover. As we can expect, heuristic performs quite well on lower values of ratio m/n and it gets worse on higher values of this ratio. For general bounded biclique cover the heuristics S^{∞}_{rand} and S^{∞}_{min} behave very similarly while S^{∞}_{max} is better in most cases.

We can observe a phase transition behaviour in the results of experiments on both strategies $S^{2,3}$ and S^{∞}. As we can see on Figure 3.6 and Figure 3.7 there is a phase transition $r_{m/n}$ for a fixed ratio m/n. Most of random graphs $G \in \mathcal{G}^k_{m,n}$ with $k \geq r_{m/n}$ have a biclique cover and Algorithm 1 will find it. However, since our heuristic is incomplete, it is not clear how many random graphs $G \in \mathcal{G}^k_{m,n}$ with $k \leq r_{m/n}$ have biclique cover.

In case of strategies with $S^{2,3}$ the most interesting case is when $m/n \leq 1.4$. As the ratio m/n gets close to 1.5 we can expect smaller percentage of graphs $G \in \mathcal{G}^k_{m,n}$ having a bounded 2-biclique cover, hence Algorithm 1 fails to find one in most cases.

Strategy S^{∞} behaves very similarly to $S^{2,3}$ but it doesn't have an upper limit to phase transition. As we can see, there is an interesting phenomenon on Figure 3.7 between 1.15 and 1.25. The strange shift is caused by using bigger bicliques by Algorithm 1.

As we can see from Table 3.2, for ratios m/n smaller than 1.4 it is better to use Algorithm 1 with a heuristic for finding a bounded 2-biclique cover. For bigger ratios m/n it is better to use a heuristic for general bounded biclique cover. It would be also interesting to perform more experiments with bounded 3-biclique covers and observe if a similar phenomenon will occur on strategy S^{∞}.

Average runtime of experiments on our heuristic can be seen in Table 3.3. For 3-CNF it has the same runtime for both strategies and all ratios of $m/n = 1, \ldots, 1.5$. This is because quite often an isolated vertex $v \in V_c$ was created during the work of Algorithm 1. Which means Algorithm 1 failed quickly in many cases. Runtime of strategy $S^{2,3}$ which uses bounded bicliques in cover is much worse

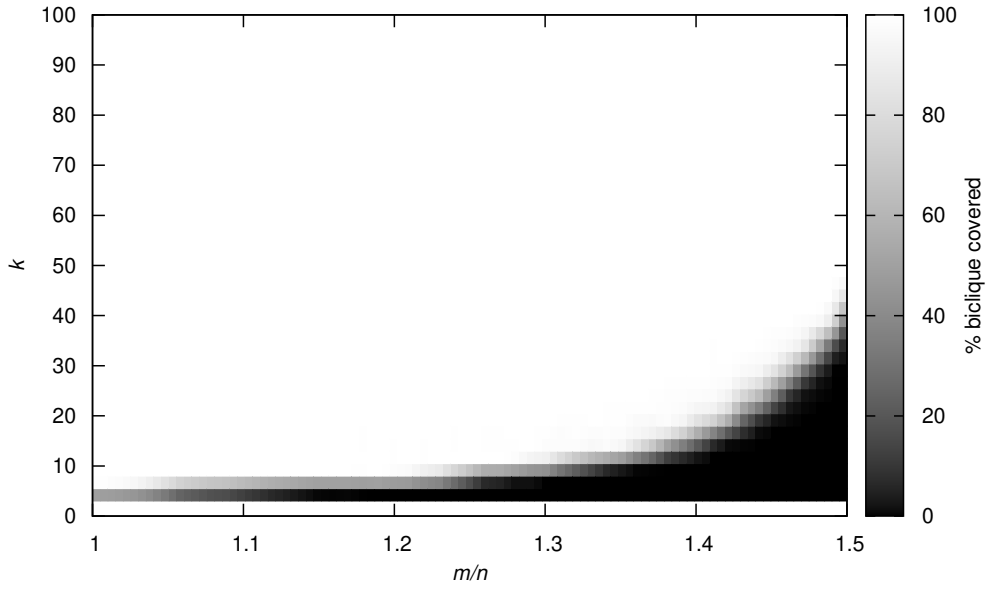


Figure 3.6: Results of experiments with Algorithm 1 with strategy $S^{2,3}_{rand}$ and $|V_v| = 100$. The horizontal axis represents the ratio m/n. The vertical axis represents the degree of vertices $v \in V_c(G)$. The more white pixel is, the more random graphs were covered by a bounded 2-biclique cover by Algorithm 1.

	1			1.1			1.2			1.3			1.4			1.5		
	3	10	20	3	10	20	3	10	20	3	10	20	3	10	20	3	10	20
$S^{2,3}_{rand}$	3.01	4.66	1.18	3.01	6.03	34.25	3.01	7.83	51.26	3.01	9.09	61.68	3	10.34	69.61	3	11.8	77.25
S^{∞}_{max}	3.01	4.78	1.18	3	6.47	19.35	3	7.95	30.44	3.01	8.99	38.88	3	9.96	44.9	3	11	49.58

Table 3.3: Average running time (in μs) of experiments with Algorithm 1. Each column represents k for k-CNF and each group of three columns corresponds to a ratio m/n.

than unbounded strategy S^{∞}. For k-CNF as k grow, the difference gets bigger. Its because with unbounded strategy S^{∞} we admit bigger bicliques in cover and hence our heuristic Algorithm 1 will run fewer iterations of the main loop and succeeds or fails faster than $S^{2,3}$.

3.4 Bounded Biclique SAT Encoding

First we describe the encoding of the problem of checking if a bipartite graph has a bounded biclique cover into SAT, then we will describe and evaluate the experiments we have performed to compare this approach with Algorithm 1. We also describe the environment we have used to run the experiments. This encoding with its experimental evaluation was presented in [Chromý and Kučera, 2019].

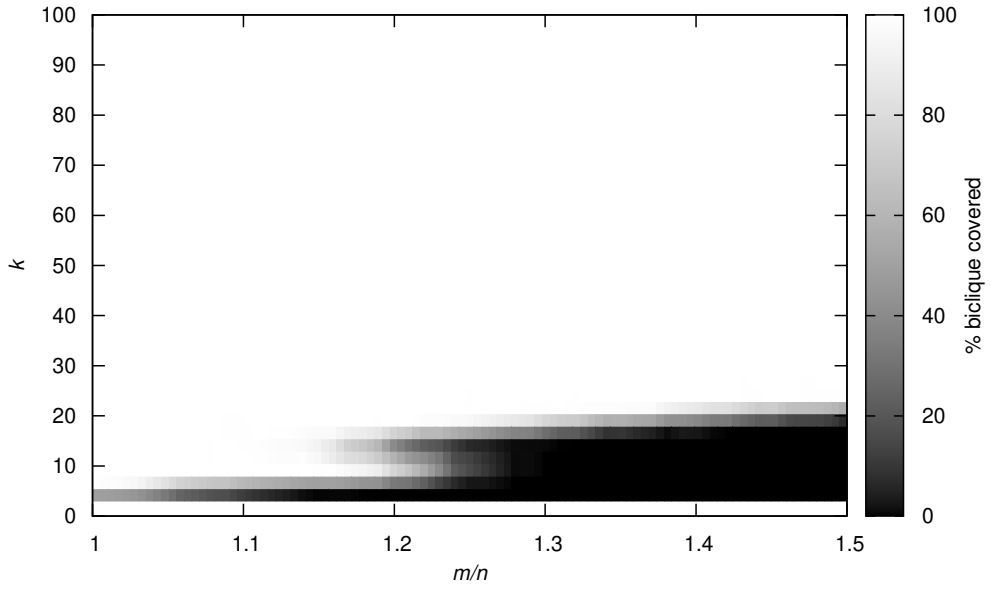


Figure 3.7: Results of experiments with Algorithm 1 with strategy S^{∞}_{max} and $|V_v| = 100$. The horizontal axis represents the ratio m/n. The vertical axis represents the degree of vertices $v \in V_c(G)$. The more white pixel is, the more random graphs were covered by a general bounded biclique cover by Algorithm 1.

3.4.1 Description of SAT Encoding

A valid biclique B of a bipartite graph G is a complete bipartite subgraph B of the bipartite graph G which follows the restriction on the size of the second partition. In particular, we require $|V_c(B)| < 2^{|V_v(B)|}$. Let us consider a bipartite graph G and $k \geq 1$, let us define $\mathcal{B}_k = \{B \mid B \text{ is a valid biclique of } G \text{ with } |V_v(B)| \leq k\}$. We also denote $\mathcal{B}_\infty$ the set of all bounded bicliques within the bipartite graph G without restriction on the size of V_v. We would use set of bicliques $\mathcal{B}_k$ to check existence of a bounded k-biclique cover and B_∞ to check existence of a general bounded biclique cover. We will encode problem of bounded (k-)biclique cover on a bipartite graph $G = (V_v, V_c, E)$. Let us fix $\mathcal{B}_k$ where k is either a natural number, or ∞ and let us describe formula g for a given graph G. With each biclique $B \in \mathcal{B}_k$ we associate a new variable x_B. Every assignment of Boolean values to variables x_B, $B \in \mathcal{B}_k$ then specifies a set of bicliques. We want to encode the fact that the satisfying assignments of g exactly correspond to bounded biclique covers of G. To this end we use the following constraints:

- For each vertex $v \in V_v$ we add to g an at-most-one constraint on variables x_B, $v \in V_v(B)$. This encodes the fact that the first partitions of bicliques in the cover have to be pairwise disjoint. We use a straightforward representation of the at-most-one constraint with a quadratic number of negative clauses of size 2.

- For each clause $C \in V_c$ we add to g a clause representing an at-least-one

constraint on variables x_b, $C \in V_c(B)$. This encodes the fact that each vertex of second partition belongs to a biclique in the cover.

3.4.2 Experimental Evaluation of Heuristic Algorithm

The number of variables in our encoding is equal to the number of all valid bicliques within the bipartite graph G. If we consider bicliques in $\mathcal{B}_k$ for a fixed k, then the number of valid bicliques $|\mathcal{B}_k|$ is polynomial in the size of f but it can be exponential in i (for $\mathcal{B}_\infty$ the number can be exponential in the size of f as well). For this reason we tested the encoding only with bicliques in $\mathcal{B}_2$, thus checking bounded 2-biclique cover. For bigger bicliques the running times of experiments increased so much that we would not be able to repeat the tests enough times for a reasonable number of variables. We used the encoding described in Section 3.4.1 to check the success rate of Algorithm 1 on random bipartite graphs and to check the phase transition for an existence of bounded biclique cover. We ran the experiments on 100 random bipartite graphs $G \in \mathcal{G}^k_{m,n}$ with $n = 40$ for combinations of $k = 1, \ldots, 8$ and the size of the second part $m = rn$ for $r = 1.00, \ldots, 1.25$ with step 0.05. For each k we tested random graphs with the ratios around the expected phase transition as observed in the Table 3.2.

	1	1.05	1.1	1.15	1.2	1.25
3	18/ 18/100	0/ 0/100				
4	95/ 95/100	50/ 56/100	7/ 10/100	0/ 1/100	0/ 0/100	
5	100/100/100	100/100/100	90/ 98/ 98	52/ 87/ 87	10/ 34/ 34	1/ 1/ 1
6		100/100/100	100/100/100	100/100/100	85/ 99/ 99	42/ 53/ 53
7				100/100/100	99/100/100	89/ 90/ 90
8						99/100/100

Table 3.4: Number of bipartite graphs with bounded biclique cover (found by Algorithm 1/ SAT finished with true/ SAT finished within time limit). See the description within the text for more details.

	1	1.05	1.1	1.15	1.2	1.25
3	0.005/0.014	0.005/ 0.01				
4	0.005/1.7	0.006/25	0.006/21	0.005/ 13	0.006/ 7	
5	0.005/0.2	0.005/ 0.98	0.006/11	0.006/216	0.006/993	0.005/4589
6		0.006/ 0.4	0.006/ 2.4	0.006/ 50	0.006/838	0.006/3646
7				0.006/ 73	0.006/166	0.007/1666
8						0.006/ 272

Table 3.5: Average runtime of Algorithm 1/average runtime of SAT on encoding in seconds.

The results of experiments are contained in Tables 3.4 to 3.6. All these tables have a similar structure. Each cell represents a single configuration (row corresponding by a value of k and column corresponding to the ratio m/n where m denotes the number of clauses and $n = 40$ denotes the number of variables). In Table 3.4 each cell contains three numbers separated with slashes. The first is

	1	1.05	1.1	1.15	1.2	1.25
3	0.52/2.16	0.58/2.39				
4	0.13/1.05	0.02/0.40	0.002/0.06	0.001/0.005	10^{-3}/0.009	
5	0.25/1.13	0.06/1.16	0.007/0.12	0.002/0.096	$8 \cdot 10^{-5}$/0.0008	$6 \cdot 10^{-7}/6 \cdot 10^{-7}$
6		0.12/0.88	0.023/0.30	0.003/0.095	$2 \cdot 10^{-4}$/0.007	$8 \cdot 10^{-5}$/0.0007
7				0.006/0.086	10^{-3}/0.039	0.00018/0.0015
8						0.00026/0.0028

Table 3.6: Average/maximum ratio between running time of Algorithm 1 and running time of SAT on encoding in seconds.

the number of instances (out of 100) on which Algorithm 1 successfully found a bounded biclique cover. The second is the number of instances on which the SAT solver successfully solved the encoding and answered positively. The third is the number of instances on which the SAT solver finished within time limit which was set to 4 hours for each instance. In some cells the values are missing, for these configurations we did not run any experiments, because they are far from the observed phase transition (see Table 3.2). In case of black colored cells we assume that the results would be 100/100/100, in case of white colored cells we assume that the results would be 0/0/100. The gray colored cells mark the borders of observed phase transition intervals of existence of bounded biclique cover, light gray corresponds to the results given by the SAT solver, dark gray to the results given by the heuristic which form an upper bound on the correct values. We can see that in most cases the number of positive answers given by Algorithm 1 is close to the number of positive answers given by SAT solver. However, there are some cases where one of the approaches was more successful — namely in cases of $k = 5, m/n = 1.15, k = 5$ and $k = 5, m/n = 1.2$. In the first case the SAT solver answered on 87 instances positively while Algorithm 1 answered positively only on 52 instances and in the second one SAT solver answered on 34 positively and Algorithm 1 answered positively only on 10 instances. However, in the second case we can see that SAT solver run over time limit (4 hours) in $\frac{2}{3}$ cases.

We also compared runtime of Algorithm 1 and the SAT solver. As we can see in the Table 3.5 Algorithm 1 is much faster in average case. Standard deviation of runtime of Algorithm 1 is around 10^{-2} and the standard deviation of runtime of SAT solver is up to 10^4 (where we have evaluated the average value and the standard deviation only on instances in which the SAT solver finished within the time limit 4 hours). These values are quite high compared to the running times. One of the reasons is perhaps the fact that the experiments were not run on a single computer, but on several comparable computers (see Section 3.4.3 for more details). Although in all cases on a single instance, the SAT solver and Algorithm 1 were run on a single computer and it makes thus sense to look at the ratio between the running times of these two. These are contained in Table 3.6. We can see that our heuristic is in most cases faster than the SAT solver using the encoding described in Section 3.4.1. Only for $k = 3$ we can see that the maximum ratio is bigger than one. It means that in some cases SAT solver was faster than Algorithm 1, although not on average.

We can see from the results that on random k-CNF formulas Algorithm 1 has a success rate close to the one of the SAT based approach. A big advantage of Algorithm 1 is that it is much faster.

3.4.3 Experiments environment

For our experiments we used Glucose parallel SAT solver [Audemard and Simon, 2009, Eén and Sörensson, 2004]. Our experiments were executed on grid computing service [Cesnet]. All experiments were run on a single processor machine (Intel Xenon, AMD Opteron) with 4 cores and frequency 2.20GHz-3.30GHz. On each random bipartite graph $G \in \mathcal{G}^k_{m,n}$, Algorithm 1 and the SAT solver were always run on the same computer. However, for the same configuration and different formulas, the experiments may have run on different computers. As we have noted in Section 3.4.2, this could be a reason of significantly high values of standard deviation of runtimes. The fact that the computer speed varied while the time limit for the SAT solver was still the same (4 hours) could have led to situations where the SAT solver would not finished, because it was run on a slower computer, and could potentially finish had it been run on a faster computer. We can see in Table 3.7 that most of the total 2000 instances finished within an hour, then only 26 finished between an hour and 2 hours, only 15 finished between 2 hours and 3 hours and only 9 finished between 3 hours and 4 hours. We can thus expect that the number of the border cases is similarly small. We can conclude that the variance in computer speeds had only minor influence on the number of SAT calls which finished within the time limit.

<1h	1h-2h	2h-3h	3h-4h	>4h	total
1912	26	15	9	238	2200

Table 3.7: Number of test cases finished in given interval.

3.5 Conclusion

Algorithm 1 presented in master thesis [Chromý, 2015] exhibits similar behaviour as matched formulas. It suggests the property of being "biclique satisfiable" has the phase transition. However Algorithm 1 is not complete; in particular, it can happen that a formula is biclique satisfiable, but Algorithm 1 is unable to detect it. It means that we can only trust a positive answer of Algorithm 1. To verify the suggested phase transition of biclique satisfiability, we have proposed a SAT encoding of this problem and run a SAT solver on the encoded formula. We used that approach to see how much biclique satisfiable formulas are not recognized by Algorithm 1. We can see in Table 3.4 that formulas on which Algorithm 1 fails to answer correctly, are concentrated around the observed phase transition, and that the Algorithm 1 answers correctly in most cases for other configurations. We can say that the success rate of Algorithm 1 is not far from the complete SAT based method. Moreover, as we can see in tables 3.5 and 3.6, our heuristic is significantly faster than a SAT solver on the encoding we have described.

It would be interesting to improve the SAT encoding of biclique satisfiability to run bigger experiments to see exploit more the phase transition behaviour. Another posible way of research would be to generalize Algorithm 1 and the SAT encoding to var-satisfiable formulas and study the behaviour of var-satisfiable formulas.

Bibliography

Ron Aharoni and Nathan Linial. Minimal non-two-colorable hypergraphs and minimal unsatisfiable formulas. *Journal of Combinatorial Theory, Series A*, 43(2):196 – 204, 1986. ISSN 0097-3165. doi: http://dx.doi.org/10.1016/0097-3165(86)90060-9. URL `http://www.sciencedirect.com/science/article/pii/0097316586900609`.

Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, page 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc. URL `http://www.labri.fr/perso/lsimon/glucose/`. Accessed: June 2017.

Giorgio Ausiello, Alessandro D'Atri, and Domenico Sacca. Minimal representation of directed hypergraphs. *SIAM Journal on Computing*, pages 418–431, 1986.

Béla Bollobás. *Modern Graph Theory*, volume 184 of *Graduate Texts in Mathematics*. Springer, 1998.

Endre Boros, Ondřej Čepek, Alexander Kogan, and Petr Kučera. A subclass of Horn CNFs optimally compressible in polynomial time. *Annals of Mathematics and Artificial Intelligence*, 57:249–291, 2009.

Endre Boros, Ondřej Čepek, and Petr Kučera. A decomposition method for CNF minimality proofs. *Theoretical Computer Science*, 510:111–126, 2013.

Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986. ISSN 0018-9340. doi: 10.1109/TC.1986.1676819. URL `http://dx.doi.org/10.1109/TC.1986.1676819`.

Ondřej Čepek, David Kronus, and Petr Kučera. Recognition of interval Boolean functions. *Annals of Mathematics and Artificial Intelligence*, 52(1):1–24, 2008.

Ondřej Čepek and Miloš Chromý. Succinctness of Switch-List Representations of Boolean Functions. In *The 21st CZECH-JAPAN SEMINAR on Data Analysis and Decision Making. Kamakura, November 23rd - 26th*, 2018. URL `http://www.ise.aoyama.ac.jp/~opt_out/cjseminar/proceedings/proceedings_online.pdf`.

Ondřej Čepek and Miloš Chromý. Switch-List Representations in a Knowledge Compilation Map. In *The 29th International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on Artificial Intelligence, IJCAI-PRICAI 2020. Yokohama, Japan, July 11-17h*, 2020a. ACCEPTED PAPER.

Ondřej Čepek and Miloš Chromý. Compiling SL representations of Boolean functions into OBDDs. In *International Symposium on Artificial Intelligence*

and Mathematics, ISAIM 2020, Fort Lauderdale, Florida, USA, January 6-8, 2020b. URL http://isaim2020.cs.ou.edu/papers/ISAIM2020_Boolean_Chromy_Cepek.pdf.

Ondřej Čepek and Radek Hušek. Recognition of tractable DNFs representable by a constant number of intervals. *Discrete Optimization*, 23:1–19, 2017. doi: 10.1016/j.disopt.2016.11.002. URL https://doi.org/10.1016/j.disopt.2016.11.002.

Cesnet. Metacentrum grid computing. URL https://metavo.metacentrum.cz/en/. Accessed: June 2017.

Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, pages 331–337, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. ISBN 1-55860-160-0. URL http://dl.acm.org/citation.cfm?id=1631171.1631221.

Miloš Chromý. Rozšíření matched formulí. Master's thesis, Charles University, Faculty of Mathematics and Physics, 2015.

Miloš Chromý. POSTER "Relative Succinctness of OBDDs and Switch-List Representations" in the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 2, Prague, Czech Republic, February 19-21, 2019, 2019.

Miloš Chromý and Petr Kučera. POSTER "Phase Transition in Matched Formulas and a Heuristic for Biclique Satisfiability" in Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS 2016, Telč, Czech Republic, 21st-23rd October, 2016.

Miloš Chromý and Petr Kučera. Phase Transition in Matched Formulas and a Heuristic for Biclique Satisfiability. In *SOFSEM 2019: Theory and Practice of Computer Science - 45th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 27-30, 2019, Proceedings*, pages 108–121, 2019. doi: 10.1007/978-3-030-10801-4_10. URL https://doi.org/10.1007/978-3-030-10801-4_10.

Harold S. Connamacher and Michael Molloy. The Satisfiability Threshold for a Seemingly Intractable Random Constraint Satisfaction Problem. *CoRR*, abs/1202.0042, 2012. URL http://arxiv.org/abs/1202.0042.

Yves Crama and Peter L. Hammer. *Boolean Functions: Theory, Algorithms, and Applications*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2011. ISBN 9781139498630.

Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *Journal Of Artificial Intelligence Research*, 17:229–264, 2002.

Olivier Dubois, Yacine Boufkhad, and Jacques Mandler. Typical Random 3-SAT Formulae and the Satisfiability Threshold. In *Proceedings of the Eleventh*

Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '00, pages 126–127, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics. ISBN 0-89871-453-2. URL `http://dl.acm.org/citation.cfm?id=338219.338243`.

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. ISBN 978-3-540-24605-3. URL `http://minisat.se/`. Accessed: June 2017.

John Franco and Allen Van Gelder. A perspective on certain polynomial-time solvable classes of satisfiability. *Discrete Applied Mathematics*, 125 (2–3):177 – 214, 2003. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/S0166-218X(01)00358-4. URL `http://www.sciencedirect.com/science/article/pii/S0166218X01003584`.

Ian P. Gent and Toby Walsh. The SAT Phase Transition. In *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 105–109. John Wiley & Sons, 1994.

Peter L. Hammer and Alexander Kogan. Optimal compression of propositional Horn knowledge bases: Complexity and approximation. *Artificial Intelligence*, 64:131–145, 1993.

Peter L. Hammer and Alexander Kogan. Quasi-acyclic propositional Horn knowledge bases: Optimal compression. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):751–762, 1995.

Mohammad Heydari, Linda Morales, Charles Shields, and Ivan Sudborough. Computing Cross Associations for Attack Graphs and Other Applications. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, page 270, 01 2007. doi: 10.1109/HICSS.2007.141.

John E. Hopcroft and Richard M. Karp. A $n^{\frac{5}{2}}$ Algorithm for Maximum Matchings in Bipartite. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (Swat 1971)*, SWAT '71, pages 122–125, Washington, DC, USA, 1971. IEEE Computer Society. doi: 10.1109/SWAT.1971.1. URL `http://dx.doi.org/10.1109/SWAT.1971.1`.

Chung-Yang Huang and Kwang-Ting Cheng. Solving constraint satisfiability problem for automatic generation of design verification vectors. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, 1999.

David Kronus and Ondřej Čepek. Recognition of Positive 2-Interval Boolean Functions. In *Proceedings of 11th Czech-Japan Seminar on Data Analysis and Decision Making under Uncertainty*, pages 115–122, 2008.

Daniel Le Berre, Pierre Marquis, Stefan Mengel, and Romain Wallon. Pseudo-Boolean Constraints from a Knowledge Representation Perspective. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 1891–1897. AAAI Press, 2018. ISBN 9780999241127.

Daniel Lewin, Laurent Fournier, Moshe Levinger, Evgeny Roytman, and Gil Shurek. Constraint Satisfaction for Test Program Generation. In *IEEE 14th Phoenix Conference on Computers and Communications*, pages 45–48, 1995.

László Lovász and Michael D. Plummer. *Matching Theory*. North-Holland, 1986. ISBN 0444879161.

Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1998. ISBN 3540644865.

David Mitchell, Bart Selman, and Hector Levesque. Hard and Easy Distributions of SAT Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, San Jose, CA, USA, 1992.

Petr Savický and Ingo Wegener. Efficient Algorithms for the Transformation Between Different Types of Binary Decision Diagrams. *Acta Inf.*, 34(4):245–256, 1997. doi: 10.1007/s002360050083. URL `https://doi.org/10.1007/s002360050083`.

Baruch Schieber, Daniel Geist, and Ayal Zaks. Computing the minimum DNF representation of Boolean functions defined by intervals. *Discrete Applied Mathematics*, 149:154–173, 2005. ISSN 0166-218X. doi: 10.1016/j.dam.2004.08.009.

Stefan Szeider. Generalizations of matched CNF formulas. *Annals of Mathematics and Artificial Intelligence*, 43(1):223–238, 2005. ISSN 1573-7470. doi: 10.1007/s10472-005-0432-6. URL `http://dx.doi.org/10.1007/s10472-005-0432-6`.

Seiichiro Tani and Hiroshi Imai. A Reordering Operation for an Ordered Binary Decision Diagram and an Extended Framework for Combinatorics of Graphs. In *Proceedings of the 5th International Symposium on Algorithms and Computation*, volume 834, pages 575–583, 08 1994. doi: 10.1007/3-540-58325-4-225.

Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8(1):85 – 89, 1984. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/0166-218X(84)90081-7. URL `http://www.sciencedirect.com/science/article/pii/0166218X84900817`.

Christopher Umans. The minimum equivalent DNF problem and shortest implicants. *J. Comput. Syst. Sci.*, 63(4):597–611, 2001.

Christopher Umans, Tiziano Villa, and Alberto L. Sangiovanni-Vincentelli. Complexity of two-level logic minimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(7):1230–1246, 2006.

Ingo Wegener. *Branching Programs and Binary Decision Diagrams: Theory and Applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. ISBN 0-89871-458-3.

List of Figures

List of Tables

List of Abbreviations

BDD	Binary Diagram
BDT	Binary Decision Tree
BFS	Breadth First Search
CARD	Cardinality Constraint
CD	Conditioning
CE	Clausal Entailment
CO	Consistency
CNF	Conjunctive Normal Form
CT	Model Counting
d-NNF	decision Negation Normal Form
DFS	Depth First Search
DNF	Disjunctive Normal Form
DNNF	Deterministic Negation Normal Form
EQ	Equivalence
FBDD	Free Binary Decision Diagram
FO	Forgetting
IM	Implicant Check
IP	Prime Implicants
ME	Model Enumeration
MODS	Models
NAND	Negated And
NNF	Negation Normal Form
NOR	Negated Or
NP	Nondeterministic Polynomial time
OBDD	Ordered Binary Decision Diagram
PBC	Pseudo-Boolean Constraint
PI	Prime Implicates
SAT	Satisfiablity
SE	Sentential Entailment
SFO	Singleton Forgetting
SL	Switch-List
TT	Truth Table
VA	Validity
XOR	Exclusive Or
$\wedge$ C	Conjunction
$\wedge$ BC	Bounded Conjunction
$\wedge$ C*	Conjunction on the same set of variables
$\vee$ C	Disjunction
$\vee$ BC	Bounded Disjunction
$\vee$ C*	Disjunction on the same set of variables
$\neg$ C	Negation

List of publications

- Ondřej Čepek and Miloš Chromý. Succinctness of Switch-List representations of Boolean functions. In The 21st CZECH-JAPAN SEMINAR on Data Analysis and Decision Making. Kamakura, November 23rd-26th, 2018.URL

- Ondřej Čepek and Miloš Chromý. Compiling SL representations of Boolean functions into OBDDs. In International Symposium on Artificial Intelligence and Mathematics, ISAIM 2020, Fort Lauderdale, Florida, USA, January 6-8, 2020. URL `http://isaim2020.cs.ou.edu/papers/ISAIM2020_Boolean_Chromy_Cepek.pdf`

- Ondřej Čepek and Miloš Chromý. Switch-List representations in a knowledge compilation map. In The 29th International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on ArtificialIntel ligence, IJCAI-PRICAI 2020. Yokohama, Japan, July 11-17h, 2020. ACCEPTED PAPER.

- Miloš Chromý. POSTER "Relative Succinctness of OBDDs and Switch-List Representations" in the 11th International Conference on Agents and Artificial Intelligence, ICAART 2019, Volume 2, Prague, Czech Republic, February 19-21, 2019

- Miloš Chromý and Petr Kučera. POSTER "Phase Transition in Matched Formulas and a Heuristic for Biclique Satisfiability" in Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS 2016, Telč, Czech Republic, 21st-23rd October.

- Miloš Chromý and Petr Kučera. Phase transition in matched formulas and a heuristic for biclique satisfiability. In SOFSEM 2019: Theory and Practice of Computer Science - 45th International Conference on Current Trends in Theory and Practice of Computer Science, Nový Smokovec, Slovakia, January 27-30, 2019, Proceedings, pages 108–121, 2019. URL `https://doi.org/10.1007/978-3-030-10801-4_10`