Knowledge compilation from DNF to switch-list representations.

Ondřej Čepek[*] and **Radek Hušek**[†]
Charles University in Prague, Faculty of Mathematics and Physics,
Malostranské nám. 25, 118 00 Praha 1, Czech Republic

Abstract

In this short note we will present a less common way how to represent a Boolean function, namely a representation by switch-list. There are two problems connected to such representation: (1) a knowledge compilation problem, i.e. a problem of transforming a given representation of a Boolean function (e.g. a DNF, CNF, BDD ...) into a switch-listl representation, and (2) a knowledge compression problem, i.e. a problem of finding the most compact switch-list representation among those which represent the same function. We will summarize known results about these two problems and present a new one about the compilation problem.

Introduction

A Boolean function of n variables is a mapping from $\{0,1\}^n$ to $\{0,1\}$. This concept naturally appears in many areas of mathematics and computer science. There are many different ways in which a Boolean function may be represented. Common representations include truth tables (with 2^n rows where a function value is explicitly given for every input vector), various types of Boolean formulas (including CNF and DNF representations), binary decision diagrams (BDDs), ordered binary decision diagrams (OBDDs), and Boolean circuits.

In this paper we shall study a less common but quite interesting representations of Boolean functions, namely the representation by intervals and a closely related representation by switch-lists. 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 form 0 to $2^n - 1$. An interval representation is then an abbreviated truth table representation, where instead of writing out all the input vectors (binary numbers) with their function values, we write out only those binary numbers x for which $f(x) = 1$ (x is a truepoint of f) and simultaneously $f(x-1) = 0$ ($x-1$ is a falsepoint of f) and those binary numbers y for which $f(y) = 1$ (y is a truepoint of f) and simultaneously $f(y+1) = 0$ ($y+1$ is a falsepoint of f). Thus the function is represented by such pairs $[x, y]$ of integers, each pair specifying one interval of truepoints. Note that $x = y$ for those pairs which represent an interval with a single truepoint.

If the number of intervals is small, such a representation may be very concise ($O(n)$ for a constant number of intervals), which may be for some functions much shorter than any of the commonly used standard representations (truth table, Boolean formula, BDD, circuit). A task of transforming one of the standard representations into an interval representation or vice versa can be classified as a knowledge compilation problem (for a review paper on knowledge compilation see (Darwiche and Marquis 2002)).

Note here, however, that changing the order of variables may dramatically change the number of truepoint intervals - it is easy to construct functions with a single truepoint interval under one permutation of variables and $\Omega(2^n)$ truepoint intervals under another permutation. Hence the length of the interval representation may be $O(n)$ for one permutation of variables and $\Omega(n2^n)$ for another permutation. On the other hand, there exist Boolean functions (e.g. a parity function), where listing all truepoint intervals is asymptotically as space consuming as writing out the full truth table (i.e. $\Omega(n2^n)$) regardless of the chosen variable order.

The first knowledge compilation problem involving interval representations was studied in (Schieber, Geist, and Zaks 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 problem originated from the field of automatic generation of test patterns for hardware verification (Lewin et al. 1995;

[*]*Email address*: ondrej.cepek@mff.cuni.cz (*corresponding author*)
[†]*Email address*: husek@iuuk.mff.cuni.cz

Huang and Cheng 1999). In fact, the paper (Schieber, Geist, and Zaks 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, Geist, and Zaks 2005) combines a knowledge compilation problem (transforming an interval representation into a DNF representaton) with a knowledge compression problem (finding the shortest DNF representation).

In (Čepek, Kronus, and Kučera 2008) the reverse knowledge compilation problem was considered. Given a DNF, decide whether it can be represented by a single interval of truepoints with respect to some permutation of variables (and in the affirmative case output the permutation and the two n-bit numbers defining the interval). 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, Kronus, and Kučera 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, Kronus, and Kučera 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 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.

It is interesting to note that the combination of results from (Čepek, Kronus, and Kučera 2008) and (Schieber, Geist, and Zaks 2005) gives a polynomial time minimization (optimal compression) algorithm for the class of 1-interval functons given by tractable DNFs, or in other words, for the 1-interval subclass of functions inside any tractable class of functions. DNF minimization (optimal compression) is a notoriously hard problem. It was shown to be Σ_2^p-complete (Umans 2001) when there is no restriction on the input DNF (see also the review paper (Umans, Villa, and Sangiovanni-Vincentelli 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, D'Atri, and Sacca 1986; Hammer and Kogan 1993) and the same result for cubic Horn DNFs in (Boros, Čepek, and Kučera 2013). There exists a hierarchy of subclasses of Horn CNFs for which there are polynomial time minimization algorithms, namely acyclic and quasi-acyclic Horn CNFs (Hammer and Kogan 1995), and CQ Horn CNFs (Boros et al. 2009). There are also few heuristic minimization algorithms for Horn CNFs (Boros, Čepek, and Kogan 1998). Suppose we are given a Horn DNF. We can test in polynomial time using the algorithm from (Čepek, Kronus, and Kučera 2008) (or the current paper) whether it represents a 1-interval function and then (in the affirmative case) use the algorithm from (Schieber, Geist, and Zaks 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).

In the present paper we generalize the knowledge compilation part of (Čepek, Kronus, and Kučera 2008) and (Kronus and Čepek 2008). Given a DNF from a tractable class of DNFs we show how to list all intervals of truepoints with respect to a fixed permutation of variables (i.e. compile a DNF into an interval representation) in polynomial time with respect to the size of the input DNF and the number of output intervals.

In fact, in the present paper we shall not consider intervals of truepoints of the given function f. Instead, we shall consider switches, i.e. those vectors x such that $f(x-1) \neq f(x)$. This is of course an equivalent problem because the list of intervals can be easily obtained from the list of switches (and the function values $f(0,0,\ldots,0)$ and $f(1,1,\ldots,1)$), and vice versa.

Preliminaries

A *Boolean function*, or a *function* in short, is a mapping $f : \{0,1\}^n \mapsto \{0,1\}$, where $x \in \{0,1\}^n$ is called a *Boolean vector* (a *vector* in short). When the order of bits in vector x is fixed, we shall also interpret x as the corresponding binary number. Propositional variables $x_1,\ldots,x_n$ and their negations $\overline{x}_1,\ldots,\overline{x}_n$ are called *literals* (*positive* and *negative literals* respectively). An elementary conjunction of literals

$$t = \bigwedge_{i\in I} x_i \wedge \bigwedge_{j\in J} \overline{x}_j \tag{1}$$

is called a *term*, if every propositional variable appears in it at most once, i.e. if $I \cap J = \emptyset$. A *disjunctive normal form* (or DNF) is a disjunction of terms. It is a well known fact, that every Boolean function can be represented by a DNF (typically by many different ones). Two DNFs $\mathcal{F}$ and $\mathcal{F}'$ are called *logically equivalent* (which is denoted by $\mathcal{F} \equiv \mathcal{F}'$) if they represent the same function.

For a DNF $\mathcal{F}$ and a term t we denote by $t \in \mathcal{F}$ the fact, that t is contained in $\mathcal{F}$. Similarly, for a term t and a

literal x we denote by $x \in t$ the fact, that x is contained in t. Thus we will treat DNFs both as disjunctions of terms and as sets of terms, and terms both as conjunctions of literals and as sets of literals, depending on the context. In the subsequent text the "$\wedge$" sign in elementary conjunctions (terms) will be frequently omitted (we shall write xyz instead of $x \wedge y \wedge z$). The set of variables appearing in a DNF $\mathcal{F}$ will be denoted by $\mathrm{Var}(\mathcal{F})$. For a function f represented by a DNF $\mathcal{F}$, variable x and value $a \in \{0,1\}$ we will denote by $f[x := a]$ the subfunction of f obtained by substituting the value a for variable x in f, and by $\mathcal{F}[x := a]$ the DNF obtained by substituting the value a for variable x in $\mathcal{F}$ (of course $\mathcal{F}[x := a]$ is a DNF representation of $f[x := a]$).

The DNF version of the *satisfiability problem* (usually called the *falsifiability problem*) is defined as follows: given a DNF $\mathcal{F}$, does there exist an assignment of truth values to the variables which makes $\mathcal{F}$ evaluate to 0?

Given Boolean functions f and g on the same set of variables, we denote by $f \leq g$ the fact that g is satisfied for any assignment of values to the variables for which f is satisfied. Hence, for example, if a term t consists of a subset of literals which constitute term t' then $t' \leq t$ (and in such a case we say that term t *absorbs* term t'). For every term t which constitutes a term in a DNF $\mathcal{F}$ it holds that $t \leq \mathcal{F}$ since when $t = 1$ for some evaluation of variables then for the same evaluation $\mathcal{F} = 1$ holds. We call a term t an *implicant* of a DNF $\mathcal{F}$, if $t \leq \mathcal{F}$. Hence every term $t \in \mathcal{F}$ is an implicant of $\mathcal{F}$. We call t a *prime implicant*, if t is an implicant of $\mathcal{F}$ and there is no implicant $t' \neq t$ of $\mathcal{F}$, for which $t \leq t' \leq \mathcal{F}$. We call DNF $\mathcal{F}$ prime, if it consists of only prime implicants. A prime implicant of $\mathcal{F}$ is called *essential* if it appears in every prime DNF logically equivalent to $\mathcal{F}$. A DNF $\mathcal{F}$ is called *essential* if it contains all its essential implicants.

It is a well known fact, that if $\mathcal{F}$ belongs to some class of DNFs, for which we can solve the falsifiability problem in polynomial time and which is closed under partial assignment (we shall call such classes *tractable*), then we can test in polynomial time for a term t and a DNF $\mathcal{F}$, whether t is an implicant of $\mathcal{F}$. To see this, observe that given a term $t = x_1 \dots x_{l_p} \overline{y}_1 \dots \overline{y}_{l_n}$, t is an implicant of f if and only if $\mathcal{F}[x_1 := 1, \dots, x_{l_p} := 1, y_1 := 0, \dots, y_{l_n} := 0]$ is not falsifiable (there is no assignment to the remaining variables which makes the DNF evaluate to 0). This simple property suffices for any DNF from a tractable class to be modified into a logically equivalent prime DNF (and hence also essential DNF) in polynomial time (by checking whether subterms of the current terms are implicants of the given DNF). See (Čepek, Kučera, and Savický 2012) for details on how this procedure works. In the subsequent text we shall denote by $p(n, \ell)$ the time needed to transform a DNF with at most n variables of total length at most ℓ into a logically equivalent essential DNF. The above discussion implies that $p(n, \ell)$ is polynomial in n and ℓ for tractable classes of DNFs.

We say, that two terms t_1 and t_2 *conflict in a variable* x, if t_1 contains literal x and t_2 contains literal $\overline{x}$. Two terms t_1 and t_2 have a *consensus*, if they conflict in exactly one variable. If $t_1 = Ax$ and $t_2 = B\overline{x}$, where A, B are two sets of literals and x is the only variable, in which t_1 and t_2 have conflict, we call a term $t = AB$ a *consensus of* terms t_1 and t_2. It is a well known fact, that a consensus of two implicates of a DNF $\mathcal{F}$ (or of a function f) is again an implicate of $\mathcal{F}$ (or f).

Compiling a DNF into an interval representation

In this section we present an algorithm that lists all switches (in increasing order) of a given DNF under a given permutation of variables. This is a knowledge compilation task for a fixed permutation of variables. There is an obvious way how to change a list of switches into a list of truepoint intervals in linear time (with respect to the length of the input list) so listing all switches achieves the announced goal of compiling into an interval representation.

Definition 1. *Fix a Boolean function f. We say that variable $x \in \mathrm{Var}(f)$ is* **simple**[1] *if either $f[x := 0]$ or $f[x := 1]$ is a constant function. We denote $\mathrm{Simp}(f)$ set of all simple variables of f.*

The notion of a simple variable is important in the switch listing algorithm because branching on a simple variable saves time that the algorithm would otherwise spend on transforming the DNFs in both branches into an essential form. To see how this works we need two simple lemmas. The first one gives us a way how to quickly recognize simple variables in an essential DNF, and the second one states that a DNF of subfunction obtained by assigning a value to a simple variable in an essential DNF is again essential (and thus no transformation is needed).

Lemma 2 (About essential DNFs)**.** *Let $\mathcal{F}$ be a DNF and $x \in \mathrm{Var}(\mathcal{F})$. Then:*

- $\mathcal{F}[x := 0] \equiv 0 \Leftrightarrow (\forall t \in \mathcal{F})(x \in t)$
- $\mathcal{F}[x := 1] \equiv 0 \Leftrightarrow (\forall t \in \mathcal{F})(\overline{x} \in t)$

Moreover if $\mathcal{F}$ is non-constant and essential:

- $\mathcal{F}[x := 0] \equiv 1 \Leftrightarrow \{\overline{x}\} \in \mathcal{F}$
- $\mathcal{F}[x := 1] \equiv 1 \Leftrightarrow \{x\} \in \mathcal{F}$

Proof. The first two items hold trivially for every DNF and so do the implications $\{x\} \in \mathcal{F} \Rightarrow \mathcal{F}[x := 1] \equiv 1$ and $\{\overline{x}\} \in \mathcal{F} \Rightarrow \mathcal{F}[x := 0] \equiv 1$. Thus the only interesting part of the proof is the implication $\mathcal{F}[x := 1] \equiv 1 \Rightarrow \{x\} \in \mathcal{F}$ and its analogy for $x := 0$ for non-constant and essential $\mathcal{F}$. To prove this implication

[1] Or that f is simple in x.

it suffices to show that if linear term t is an implicant of a non-constant Boolean function f then t is its essential implicant.

Without a loss of generality let us assume $t = \{x\}$ (the case $t = \{\overline{x}\}$ is similar). Since f is not a constant function, empty term is not its implicant so t is a prime implicant of f. Moreover no other prime implicant t' contains x (because then t absorbs t') or $\overline{x}$ (because then the consensus of t and t' which is an implicant of $\mathcal{F}$ absorbs t'). Now assume that t is not an essential implicant. Then there is a prime DNF $\mathcal{F}'$ representing f which doesn't contain the variable x. It means that f is independent of x which together with $f[x := 1] \equiv 1$ contradicts the assumed non-constantness of f. □

Lemma 3 (About an assignment of a simple variable)**.** *Let $\mathcal{F}$ be an essential DNF which is simple in variable x. Then both $\mathcal{F}[x := 0]$ and $\mathcal{F}[x := 1]$ are essential DNFs of the corresponding subfunctions.*

Proof. Let $a \in \{0, 1\}$. If $\mathcal{F}[x := a]$ is trivial, it is also essential. Let $\mathcal{F}[x := a]$ be a nontrivial function. Due to Lemma 2 it suffices to distinguish two cases: either x (or $\overline{x}$) is a universal literal in $\mathcal{F}$ or x (or $\overline{x}$) is linear term in $\mathcal{F}$.

The universal literal case: without a loss of generality we may assume that x (and not $\overline{x}$) is a universal literal which means that $a = 1$ in this case. We can transform any prime DNF representing $\mathcal{F}$ into prime DNF representing $\mathcal{F}[x := 1]$ by removing x from all the terms and vice versa.[2] Therefore essential implicants of $\mathcal{F}[x := 1]$ are exactly essential implicants of $\mathcal{F}$ after removing literal x from them. Hence $\mathcal{F}[x := 1]$ is essential.

The linear term case: without a loss of generality we may assume that the linear term is x which means that $a = 0$. We know that x is a prime implicant of $\mathcal{F}$ and no other prime implicant contains variable x. Therefore we can transform prime DNF representing $\mathcal{F}$ into primary DNF representing $\mathcal{F}[x := 0]$ by removing term x (and vice versa). It means that essential implicants of $\mathcal{F}[x := 0]$ are all essential implicants of $\mathcal{F}$ except x. Hence $\mathcal{F}[x := 0]$ is essential. □

Now we are ready to present the switch-listing algorithm. The algorithm works recursively. First it transforms $\mathcal{F}$ into an essential form if it is not essential yet.[3] Then it checks whether $\mathcal{F}$ is constant. If $\mathcal{F}$ is non-constant, the algorithm selects the first variable x in the current permutation π and considers the subfunctions $\mathcal{F}[x := 0]$ and $\mathcal{F}[x := 1]$ under the permutation σ of the remaining variables, which is obtained from π by deleting x. First the algorithm checks for a switch in the middle (between the largest input vector of $\mathcal{F}[x := 0]$ and the smallest input vector of $\mathcal{F}[x := 1]$), then it recurses on the left half (by calling $\texttt{SwitchSet}(\mathcal{F}[x := 0], \sigma)$) and on the right half (by calling $\texttt{SwitchSet}(\mathcal{F}[x := 1], \sigma)$), and finally it glues all three returned values together (of course it has to shift all switches returned from the right half by the size of the left half).

Function $\texttt{SwitchSet}(\mathcal{F}, \pi)$
  Input: DNF $\mathcal{F}$ from a fixed tractable class, π permutation of $\mathrm{Var}(\mathcal{F})$
  Output: S set of switches of $\mathcal{F}$ under permutation π
  If needed, transform $\mathcal{F}$ into an essential DNF
  if $\mathcal{F} \equiv 0 \vee \mathcal{F} \equiv 1$ **then return** $\emptyset$
  $M \leftarrow \emptyset$
  $x \leftarrow \pi[1]$
  $\sigma \leftarrow \pi[2..]$
  if $\mathcal{F}[x := 0](1, \ldots, 1) \neq \mathcal{F}[x := 1](0, \ldots, 0)$ **then** $M \leftarrow \{2^{|\sigma|}\}$
  $L \leftarrow \texttt{SwitchSet}(\mathcal{F}[x := 0], \sigma)$
  $R \leftarrow \texttt{SwitchSet}(\mathcal{F}[x := 1], \sigma)$
  return $L \cup M \cup (R + 2^{|\sigma|})$
end

Algorithm 1: Switch-listing algorithm

Because we are primarily interested in the polynomiality of the running time of the algorithm, we present only a simplified complexity analysis here which proves $\mathcal{O}(|S|\,(n^2 + n\ell + p(n, \ell)))$ running time. A more detailed and much more technical analysis which improves the time complexity to $\mathcal{O}(|S|\,(n + \ell + p(n, \ell)))$ may be found in diploma thesis (Hušek 2014).

Theorem 4 (About the switch-listing algorithm)**.** *Algorithm 1 correctly outputs all switches of the input DNF $\mathcal{F}$ under permutation π in*

$$\mathcal{O}(|S|\,(n^2 + n\ell + p(n, \ell)))$$

time, where $n = |\mathrm{Var}(\mathcal{F})|$ is the number of variables, ℓ is the total number of literals (sum of term lengths) in $\mathcal{F}$, $p(n, \ell)$ is the time needed to transform a DNF with at most n variables of total length at most ℓ into an essential form (which is polynomial in n and ℓ for tractable classes of DNFs), and S is the output (the list of all switches of $\mathcal{F}$ under permutation π).

Proof. First of all the algorithm terminates because each recursive call decreases the number of variables by one and – in the worst case – every function on zero

[2] We know that x is universal literal in any DNF representing $\mathcal{F}$ because of Lemma 2.

[3] The algorithm always does the transformation when called by user (before the recursion is invoked) and during the recursive calls it passes the information about the need of the transformation in a hidden parameter.

variables is constant. Correctness is easily shown by induction on the number of variables. It is trivially true for constant functions. For the induction step it suffices to realize that the algorithm correctly detects a switch in the middle and all switches in both subfunctions are detected correctly by the induction hypothesis.

No let us analyze the time complexity. One invocation of `SwitchSet` without recursion and modification of elements of R takes time $\mathcal{O}(n + \ell + p(n, \ell))$ if we perform the transformation into essential form, and $\mathcal{O}(n + \ell)$ if we do not. Every switch can be modified at most n times (because n is the depth of recursion) and each modification can be done in time $\mathcal{O}(n)$.[4] So all modifications of switches take $\mathcal{O}(n^2 |S|)$ through the whole run of the algorithm.

The next step is to determine the number of invocations of function `SwitchSet`. The tree of recursion is binary and every node whose both children are leaves outputs a switch. This is because such a node did perform recursion so its input was not a constant function, but both of its children did not recurse so their inputs were constant functions. So there is at most $|S|$ of such nodes with two leaves as children. Let us denote the set of such internal nodes by T. We want to count the number of invocations of `SwitchSet`, i.e. the number of all internal nodes. However, since each internal node has two children, each internal node must have at least one node in T below it (as a descendant). Thus, if we trace up the paths from nodes in T upwards to the root of the tree, the union of these paths must contain all internal nodes. The length of each such path is at most n (the depth of recursion), so there are at most $n |T| \leq n |S|$ internal nodes in the recursion tree.

We know that the transformation into an essential form is needed only after assigning for a non-simple variable (because of Lemma 3) and at the very beginning of the algorithm. However, it is easy to see, that if in a given node of the tree of recursion an assignment for a non-simple variable was performed, then both subtrees induced by its children must output at least one switch each. So when we denote by q the number of nodes that assign for non-simple variable then the algorithm outputs at least $q + 1$ switches. Hence $q < |S|$ and the time complexity of the algorithm is bounded by $q\mathcal{O}(n + \ell + p(n, \ell)) + n |S| \mathcal{O}(n + \ell) + \mathcal{O}(n^2 |S|) = \mathcal{O}(|S| (n^2 + n\ell + p(n, \ell)))$. □

Acknowledgements

The authors gratefully acknowledge a support by the Czech Science Foundation (grant P202/15-15511S).

[4] Actually in can be done in $\mathcal{O}(1)$ even on a Pointer Machine with appropriate representation but we do not need this improved bound here.

References

[Ausiello, D'Atri, and Sacca 1986] Ausiello, G.; D'Atri, A.; and Sacca, D. 1986. Minimal representation of directed hypergraphs. *SIAM Journal on Computing* 418–431.

[Boros et al. 2009] Boros, E.; Čepek, O.; Kogan, A.; and Kučera, P. 2009. A subclass of horn CNFs optimally compressible in polynomial time. *Annals of Mathematics and Artificial Intelligence* 57:249–291.

[Boros, Čepek, and Kučera 2013] Boros, E.; Čepek, O.; and Kučera, P. 2013. A decomposition method for CNF minimality proofs. *Theoretical Computer Science* 510:111–126.

[Boros, Čepek, and Kogan 1998] Boros, E.; Čepek, O.; and Kogan, A. 1998. Horn minimization by iterative decomposition. *Annals of Mathematics and Artificial Intelligence* 23:321–343.

[Čepek, Kronus, and Kučera 2008] Čepek, O.; Kronus, D.; and Kučera, P. 2008. Recognition of interval Boolean functions. *Annals of Mathematics and Artificial Intelligence* 52(1):1–24.

[Čepek, Kučera, and Savický 2012] Čepek, O.; Kučera, P.; and Savický, P. 2012. Boolean functions with a simple certificate for cnf complexity. *Discrete Applied Mathematics* 160(4):365–382.

[Darwiche and Marquis 2002] Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *Journal Of Artificial Intelligence Research* 17:229–264.

[Hammer and Kogan 1993] Hammer, P. L., and Kogan, A. 1993. Optimal compression of propositional horn knowledge bases: Complexity and approximation. *Artificial Intelligence* 64:131–145.

[Hammer and Kogan 1995] Hammer, P. L., and Kogan, A. 1995. Quasi-acyclic propositional horn knowledge bases: Optimal compression. *IEEE Transactions on Knowledge and Data Engineering* 7(5):751–762.

[Huang and Cheng 1999] Huang, C., and Cheng, K. 1999. Solving constraint satisfiability problem for automatic generation of design verification vectors. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop*.

[Hušek 2014] Hušek, R. 2014. Properties of interval boolean functions. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics. [in Czech].

[Kronus and Čepek 2008] Kronus, D., and Čepek, O. 2008. Recognition of positive 2-interval Boolean functions. In *Proceedings of 11th Czech-Japan Seminar on Data Analysis and Decision Making under Uncertainty*, 115–122.

[Lewin et al. 1995] Lewin, D.; Fournier, L.; Levinger, L.; Roytman, E.; and Shurek, G. 1995. Constraint

satisfaction for test program generation. In *Computers and Communications, 1995., Conference Proceedings of the 1995 IEEE Fourteenth Annual International Phoenix Conference on*, 45–48.

[Schieber, Geist, and Zaks 2005] Schieber, B.; Geist, D.; and Zaks, A. 2005. Computing the minimum DNF representation of boolean functions defined by intervals. *Discrete Applied Mathematics* 149:154–173.

[Umans, Villa, and Sangiovanni-Vincentelli 2006] Umans, C.; Villa, T.; and Sangiovanni-Vincentelli, A. L. 2006. Complexity of two-level logic minimization. *IEEE Trans. on CAD of Integrated Circuits and Systems* 25(7):1230–1246.

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