Compiling SL representations of Boolean functions into OBDDs

Ondřej Čepek, Miloš Chromý*
Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics, Charles University
Prague, Czech Republic
{cepek,chromy}@ktiml.mff.cuni.cz

Abstract

Given a truth table representation of a Boolean function f the switch-list (SL) representation of f consists of the function value $f(0)$ at the all-zero vector and the list of all Boolean vectors from the truth table of f which have a different function value then the preceding vector. The main result of this paper is a polynomial time compilation algorithm from a SL representation of a given function f to an Ordered Binary Decision Diagram (OBDD) representation of f where the output OBDD respects some prescribed order of variables possibly different than the input order used by the SL representation of f. Furthermore we provide a lower bound construction which shows that the presented compilation algorithm yields an asymptotically optimal size OBDD of f respecting the prescribed output order of variables.

1 Introduction

A Boolean function on n variables is a mapping from $\{0,1\}^n$ to $\{0,1\}$. There are many different ways in which a Boolean function may be represented. Common representations include truth tables (TT – with 2^n rows where a function value is explicitly given for every binary vector), list of models (MODS – list of binary vectors on which the function evaluates to 1), various types of Boolean formulas (including CNF and DNF representations), various types of binary decision diagrams (BDDs, FBDDs, OBDDs), and Boolean circuits.

The task of transforming one of the representations of a given function f into another representation of f (e.g. transforming a MODS representation into an OBDD or a circuit into a DNF) is called knowledge compilation. For a comprehensive review paper on knowledge compilation see (Darwiche and Marquis 2002), where a Knowledge Compilation Map is introduced. This map systematically investigates different representation languages with respect to the complexity of common transformations (negation, conjunction, disjunction, conditioning, forgetting) and common queries (consistency check, validity check, clausal and sentential entailment, model counting, model enumeration). The time complexity of particular transformations and queries of course differs dramatically from trivial to NP-hard depending on the chosen representation language.

In this paper we shall study less common representations of Boolean functions, namely the representation by intervals of truepoints and the 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 TT or MODS 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 an ordered list of 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.

Interval representation of Boolean functions was introduced 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 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, 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 representation) 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,

*Contact author

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 solvable in polynomial time 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 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 of course exponential in k).

In fact, (Čepek and Hušek 2017) departs from interval representations and introduces switch-list representations which we shall use in this paper. A switch point is a vector (binary number) x such that $f(x-1) \neq f(x)$. A switch-list is an ordered list of all switches of a given function. A switch-list of f together with the function value $f(0,0,\ldots,0)$ forms a switch-list representation of f.

Switch-list representations have an added advantage over the truepoint intervals representations. 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). Clearly, both the function and its negation have the same switch-lists and the representations differ only by opposite values of $f(0,0,\ldots,0)$. Thus the results relating switch-list representations to DNFs can be easily rewritten into results relating them to CNFs. This is not true for interval representations because an interval of truepoints turns into an interval of falsepoints by negation, and hence the truepoint intervals representations of a function and its negation significantly differ, and even the number of intervals may be different (although the difference is at most one). For this reason, we shall use the switch-list representations throughout this paper. 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.

Switch-list representations directly support many queries from (Darwiche and Marquis 2002) in polynomial time. 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 (Čepek and Chromý 2019). On the other hand, switch-list representations do not directly support sentential entailment and equivalence check. It is a well known fact (Darwiche and Marquis 2002) that OBDDs with a fixed order of variables (i.e. input OBDDS respect the same order) do support both of these queries in polynomial time. Thus, to guarantee the same for switch-list representations, it suffices to design a polynomial time compilation algorithm from switch-lists to OBDDs. Such an algorithm is the main result of this paper. In fact the presented polynomial time algorithm compiles a switch list to an OBDD which respects an arbitrary given order of variables different form the order on the input. This of course implies that sentential entailment and equivalence check can be decided in polynomial time even for switch-list representations defined on two distinct variable orders.

As a final remark let us note that the combination of results from (Čepek, Kronus, and Kučera 2008) and (Schieber, Geist, and Zaks 2005) gives a polynomial time minimization algorithm for the 1-interval subclass of functions inside any tractable class of functions given by DNFs (and similarly for CNFs but we shall recall results in the DNF context). DNF minimization is a notoriously hard problem (Σ_2^p-complete (Umans 2001)) when there is no restriction on the input. It is also long known that this problem is NP-hard for some tractable classes of DNFs such as Horn DNFs (Ausiello, D'Atri, and Sacca 1986; Hammer and Kogan 1993) and even cubic Horn DNFs in (Boros, Čepek, and Kučera 2013). On the other hand, 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, Kronus, and Kučera 2008) 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).

2 Definitions and notation

A *Boolean function* (*function* in short) in n propositional variables is a mapping $f : \{0,1\}^n \to \{0,1\}$. Here $x \in \{0,1\}^n$ is a *Boolean vector* (*vector* in short).

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 of propositional variables. An *Ordered Bi-*

nary 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. If the underlying graph of BDD is a tree, we say it is a *Binary decision Tree (BDT)*.

Finally, let us define switch-list representations. Again, let $<$ be a total order on the set X of n propositional variables, and let f be a 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 $<$.

3 Compilation from SL to OBDD

Our compilation algorithm works in two steps. First it compiles the input SL representation of function f into a binary decision tree (BDT) which respects the same order of variables as the SL representation. In the second step it uses that BDT to create an OBDD of f which respects a given prescribed order of variables which may differ from the order used by the input SL 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 u single zero terminal and the same for one terminals.

3.1 Compilation from SL to BDT

Let us consider a SL representation of a k-switch function f and construct from it an equivalent decision tree representation with respect to the same order of variables. The idea behind the construction is quite simple. Fix the order of variables of the given function f, say $x_1, x_2, \ldots, x_n$. 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 not depend on the order in which nodes are contracted. What is the size of this unique contracted decision tree? Consider the leaf decision nodes, that is decision nodes with both outgoing edges going to terminals (function values). Obviously, for every leaf decision 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 decision 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, which 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 SL representation.

The above considerations suffice for a proof of an existence of a linear size decision tree equivalent to the input SL 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].

[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 which 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 the factor n.

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 1. *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 SL representation of function f as described in Section 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 in Section 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 of course 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 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 2. *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 SL representation as described in Section 3.1. During the processing of nodes on level i (of the OBDD that is being built), the algorithm will keep 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 which processes 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. 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 $O(kn^2)$ (proof in (Čepek and Chromý 2019)), and its construction takes $O(k^2n^3)$ time. This complexity bound follows from the fact that for each of the $O(kn^2)$ nodes in the OBDD the associated contracted prefix BDT has size $O(kn)$, and everything the above described procedure does when processing a given node of the OBDD is linear in the size of the associated BDT.

4 Lower bound for construction

In this section we shall show that the quadratic blowup in the number of variables in the compilation algorithm from Section 3 which produces an output OBDD of size $O(kn^2)$ is unavoidable. We shall consider a situation with $k = 1$ (the input switch-list representation contains a single switch point) and construct a 1-switch function for which any OBDD w.r.t. a certain prescribed order of variables has size $\Omega(n^2)$.

We will accomplish this by considering a function f with on n variables where n is even and the only switch point of f with respect to the natural ordering $\pi : x_1 < x_2 < \ldots < x_n$ is $s = (010101\ldots01)$ (see Figure 1 for an example with $n = 8$). For an 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). 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 is trivial, there is a single node on the first level of any OBDD. For the induction step let us assume that we have i nodes 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 $x_{n+2-2i}, x_{n-2i}, \ldots, x_2, x_1, x_3, \ldots, x_{n-3}, x_{n-1}$. 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 will of course imply 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 form Figure 1 that when branching on an even variable in the natural order π the value of f is either decided earlier or the latest when setting this variable (in our case $x_{n+2-2i} = 0$). In other words, in this situation f does not depend on any variable x_k for $k > n+2-2i$ and thus in particular on variables $x_{n-2i}, \ldots, x_{n-2}, x_n$, i.e. on those variables on which the OBDD branches on the first $i - 1$ levels in the order σ. Therefore substituting $x_{n+2-2i} = 0$ into $f_0, \ldots, f_{i-1}$ produces a single function g_i. This moreover implies, that every pair of vectors X^j, X^k for $j \neq k$ that guarantees $f_j \neq f_k$ (note that X^j, X^k are defined only on variables $x_{n-2i}, \ldots, x_{n-2}, x_n$) must have $X^j_{n+2-2i} = X^k_{n+2-2i} = 1$. Thus substituting $x_{n+2-2i} = 1$ into $f_0, \ldots, f_{i-1}$ produces i pairwise distinct functions $g_0, \ldots, g_{i-1}$. Now it remains to show that g_i is distinct from every function in this set. So let $0 \leq j \leq i - 1$ be arbitrary and consider a vector $X = (0, 1, 0, 1, \ldots 0, p, 1)$ where p is in position $n + 2 - 2i$. Notice, that all variables that on which OBDD branched on levels $1, \ldots, i-1$ are outside of the scope of indices used by X. Now $g_i(X)$ originates from $f_j(X)$ by setting $p = 0$ and so clearly $g_i(X) = 0$ while $g_j(X)$ originates from $f_j(X)$ by setting $p = 1$ and so clearly $g_j(X) = 1$ (this is easy to see from Figure 1). Thus g_i is distinct from g_j which finishes the proof of the main claim.

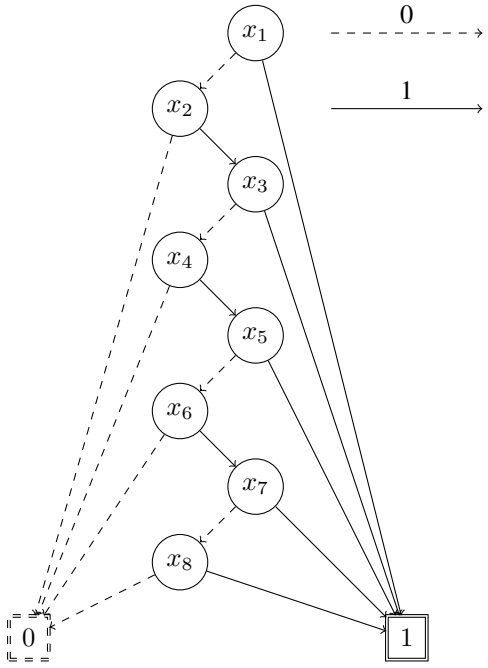

Figure 1: Original BDT.

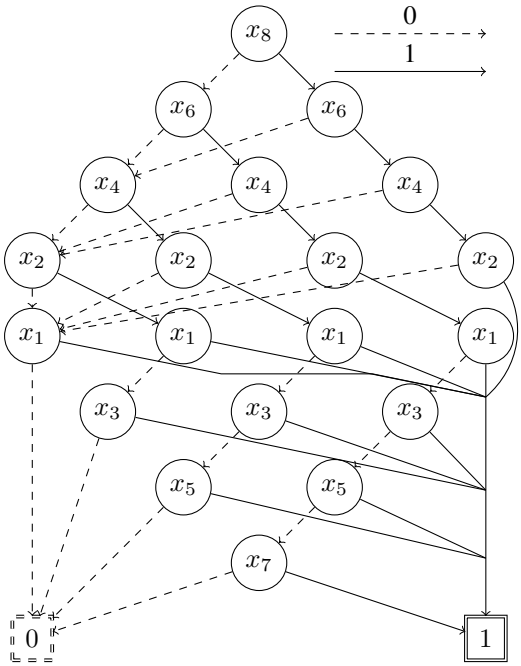

Figure 2: Compiled OBDD.

5 Conclusion

Section 3 describes an algorithm which compiles a SL representation of a k-switch function f on n variables which respects some input order π to an OBDD representation of f which respects some different output order σ of variables. The algorithm works in $O(k^2n^3)$ time and produces an output OBDD of size $O(kn^2)$. Section 4 introduces a family of 1-switch functions for which any output OBDD has size $\Omega(n^2)$ with respect to a prescribed variable ordering. This shows that our compilation algorithm creates asymptotically smallest OBDDs with respect to the number of variables. It is an interesting open research question whether the construction from Section 4 can be generalized to a family of k-switch functions for any $k > 1$, guaranteeing an output OBDD of size $\Omega(kn^2)$.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge a support by Czech Science Foundation (Grant 19-19463S). This research was partially supported by SVV project number 260 453.

References

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

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

Čepek, O., and Chromý, M. 2019. Switch-list representations in a knowledge compilation map. *Unpublished manuscript available from the authors.*

Čepek, O., and Hušek, R. 2017. Recognition of tractable dnfs representable by a constant number of intervals. *Discrete Optimization* 23:1–19.

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

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

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

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, 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*.

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, D.; Fournier, L.; Levinger, M.; Roytman, E.; and Shurek, G. 1995. Constraint satisfaction for test program generation. In *IEEE 14th Phoenix Conference on Computers and Communications*, 45–48.

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