A Direct Equivalence-Testing Algorithm for SLRs

Ondřej Čepek and James Weigle
Department of Theoretical Computer Science and Mathematical Logic, Charles University
Malostranské náměstí 25, 118 00 Praha 1, Czech Republic
ondrej.cepek@mff.cuni.cz, weigle@posteo.net

Abstract

In this paper we study the recently introduced switch-list representations (SLRs) of Boolean functions. An SLR is a compressed truth table representation of a Boolean function: we only store the function value of the first row, and a list of switches (Boolean vectors whose function value differs from the value of the preceding vector). The paper (Čepek and Chromý 2020) systematically studies the properties of SLRs and among other results gives polynomial time algorithms for all standard queries investigated in the Knowledge Compilation Map (Darwiche and Marquis 2002). In particular, the equivalence query (EQ) is implemented in (Čepek and Chromý 2020) by first compiling both input SLRs into OBDDs and then running the EQ query on the constructed OBDDs. In this short note we present an algorithm that answers the EQ query directly by manipulating the input SLRs (hence eliminating the compilation step into OBDD) which improves the time complexity of the procedure.

Introduction

Switch-list representations of Boolean functions were introduced in (Čepek and Hušek 2017) as a variant to interval representations introduced in (Schieber, Geist, and Zaks 2005). Given a fixed order of variables of function f defined on variables $\{x_1, \ldots, x_n\}$, a *switch* of f is a vector (binary number) x with bits in the prescribed order 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 (SLR)* of f. It is important to note that switch-lists are ordered by the natural order on binary numbers (as opposed to just maintaining unordered sets of switches) as this is required by the algorithms presented in this paper. Given a SLR F of function f we shall denote $F = (\mathrm{FV}(F), \mathrm{SL}(F))$ where $\mathrm{FV}(F)$ is the value of $f(0,0,\ldots,0)$ and $\mathrm{SL}(F)$ is the ordered switch-list.

SLRs form an interesting representation language with a reasonable trade-off between succinctness on one hand and the complexity of queries and transformations on the other. A SLR may be exponentially smaller than the full truth table or the list of models, yet the language of SLRs supports answering all standard queries (considered in (Darwiche and Marquis 2002)) in polynomial time. These queries are: checks for consistency (CO), validity (VA), clausal entailment (CE), implicant (IM), equivalence (EQ), and sentential entailment (SE), and furthermore model counting (CT), and model enumeration (ME). We refer the reader to (Darwiche and Marquis 2002) for the definitions of these queries.

In this paper we consider the equivalence query (EQ) which takes two SLRs as an input and decides whether they represent the same Boolean function. We shall without loss of generality assume that both SLRs are defined on the same set of variables. If both input SLRs moreover respect the same order of variables then the query is trivial: it suffices to compare whether both input SLRs are identical (note that once the order of variables is fixed, a function has a unique SLR representation). The interesting case is when the two input SLRs respect orders given by two different permutations of variables.

The paper (Čepek and Chromý 2020) (and also the preceding conference version (Čepek and Chromý 2020)) provide an indirect algorithm for the equivalence query. Both SLRs are first compiled into OBDDs where both OBDDs respect the same order of variables and then the (EQ) query is answered using an algorithm from (Wegener 2000). Hence the main difficulty of this compilation step is the fact that for one of the input SLRs the order of variables must be changed during the compilation procedure. This makes the compilation step computationally expensive and the time complexity of (EQ) is $\mathrm{O}(k^2n^3)$ where n is the number of variables and k is the total number of switches in both input SLRs.

In this paper we present a direct equivalence testing algorithm that manipulates the input SLRs without compiling them into another representation language. The direct algorithm we introduce here has time complexity $\mathrm{O}(k^2n^2)$ which beats the complexity of the previous algorithm by a factor of n. It heavily uses the conditioning transformation for SLRs (also introduced in (Čepek and Chromý 2020)) so we shall start by describing this procedure in the next section.

Conditioning for SLRs

This section is based on the description of conditioning for SLRs in (Čepek and Chromý 2020), but provides more detail and describes this transformation in a more algorithmic style, which should be helpful for a future computer implementation.

Let f be a Boolean function on variables $x_1, \ldots, x_n$ and let x_i be an arbitrary variable. Consider the truth table of f that respects the identical permutation of variables. 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 half-blocks B^0_ℓ and B^1_ℓ depending on the value of x_i. We will write the vectors from the truth table of f that belong to B_ℓ as triples (ℓ, b, q) where $b \in \{0, 1\}$ represents the value of x_i and $0 \leq q \leq 2^{n-i} - 1$ is a binary number representing $x_{i+1}, \ldots, x_n$. We will use $\mathbf{0}$ and $\mathbf{1}$ as shorthands for the all-zero and all-one vectors of length $n - i$, so $\mathbf{0} \leq q \leq \mathbf{1}$ holds if we treat q as a vector rather than a number.

Our aim is to generate SLRs of

$$f_0 = f_{|x_i=0} \text{ and } f_1 = f_{|x_i=1}$$

which originate from the SLR of f by conditioning (CD) on the value of variable x_i. We will denote the vectors from the truth tables of f_0 and f_1 as pairs (ℓ, q) (consistently with the notation introduced above).

For a switch-list representation F of function f, let $\mathrm{FV}(F)$ denote the function value of the first row of the truth table (i.e., $f(0, \ldots, 0)$), and let $\mathrm{SL}(F)$ denote the ordered list of switches. We can also just write $\mathrm{FV}(f)$ and $\mathrm{SL}(f)$ to denote the same things, when the SLR in question is obvious by context. Let us now consider a fixed input SLR F and let $\mathrm{SL}(F) = \{s_1, \ldots, s_k\}$. Moreover, for $1 \leq j \leq k$ let

$$s_j = (\ell_j, b_j, q_j) \in \{0, 1\}^n$$

be the split of each switch in triples as described above.

We will treat the construction of SLRs for f_0 and f_1 separately. First we present the procedure for constructing the SLR F_0 of f_0 by a single pass through $\mathrm{SL}(F)$.

```
CD(F, x_i = 0)
 1  FV(F_0) ← FV(F)
 2  p ← 0 // parity initialization
 3  j ← 1 // index of the currently processed switch s_j
 4  while (j ≤ k) do
 5    cp ← ℓ_j // current prefix for this iteration
 6    if (b_j = 0) and (q_j = 0)
 7      then {if (p = 0) then output (ℓ_j, 0); j++}
 8      else {if (p = 1) then output (ℓ_j, 0)}
 9    p ← 0 // parity reset
10    while (ℓ_j = cp) and (b_j = 0) do
11      {output (ℓ_j, q_j); j++}
12    while (ℓ_j = cp) and (b_j = 1) do
13      {p ← (1 − p) ; j++}
14    if (p = 1) and (cp ≠ 1) and (cp + 1 ≤ ℓ_j)
15      then {output (cp + 1, 0); p ← 0}
```

Let us explain in words what the code does. The first line sets the initial value $\mathrm{FV}(F_0)$, which in this case is the same as $\mathrm{FV}(F)$: the all-zero vector from the truth table of f "survives" conditioning and becomes the all-zero vector in the truth table of f_0. Each iteration of the main while-loop on line 4 processes a subsequence of switches from $\mathrm{SL}(F)$ that belong to the same block B_{ℓ_j}—which is to say that it processes those switches that share the same prefix ℓ_j stored in the variable cp. The body of the main while-loop (lines 5–15) has three parts where switches of $\mathrm{SL}(F_0)$ are generated (by the command **output**).

The first part (lines 6–8) deals with the first vector in $B_{\ell_j} = B_{\mathtt{cp}}$, namely the vector $v = (\ell_j, 0, \mathbf{0})$, which requires a special treatment:

- If v is the current switch (i.e. $v = s_j$) which happens when $(b_j = 0)$ and $(q_j = \mathbf{0})$, then $(\ell_j, \mathbf{0})$ becomes a switch in $\mathrm{SL}(F_0)$ in two cases. One possibility is that $j = 1$ (v is the first switch in $\mathrm{SL}(F)$) in which case $p = 0$ thanks to its initialization. Note that in this case necessarily $\ell_j \neq \mathbf{0}$ since the all-zero vector is never a switch. The other possibility is that $j > 1$ and the immediately preceding half-block $B^1_{\mathtt{cp}-1}$ contains even number of switches, in which case $p = 0$ also holds, since otherwise (if $B^1_{\mathtt{cp}-1}$ contained odd number of switches) $p = 1$ would be set in the previous iteration of the main while-loop (see the next explanation for details). If $p = 1$ was set in this way then the current switch s_j is skipped (nothing is generated but j is incremented).
- If v is not a switch and the immediately preceding half-block $B^1_{\mathtt{cp}-1}$ contains odd number of switches then $(\ell_j, \mathbf{0})$ becomes a "new" switch in $\mathrm{SL}(f_0)$ generated by the disappearing switches from $B^1_{\mathtt{cp}-1}$. In this case $p = 1$ holds because it was set on lines 12–13 of the previous iteration of the main while-loop in which $\mathtt{cp} = \ell_j - 1$, and p could not be reset to zero on lines 14–15 because $\mathtt{cp{+}1} = \ell_j$ prevents such a reset.

The second part of the main while-loop contains two nested while-loops. The first one (lines 10–11) generates switches (ℓ_j, q_j) with $q_j \neq \mathbf{0}$ in the half-block $B^0_{\mathtt{cp}}$ which of course all survive the conditioning. The second one (lines 12–13) then just counts the parity of the number of switches in the half-block $B^1_{\mathtt{cp}}$.

Finally, the last part of the main while-loop (lines 14–15) generates a switch ($\mathtt{cp{+}1}$,$\mathbf{0}$) in the beginning of the next block $B^0_{\mathtt{cp}+1}$ if (i) there is an odd number of disappearing switches in $B^1_{\mathtt{cp}}$ (so $p = 1$ was set), (ii) the current block is not the last one in the truth table ($\mathtt{cp} \neq \mathbf{1}$ holds and thus the next block exists), and (iii) the next block contains no switches (hence $\mathtt{cp{+}1} < \ell_j$) in which case the next block will be "skipped" by the main while-loop.

Let us remark, that it is better to treat each ℓ_j and q_j as a bit string rather than a number. In such a case the pseudocode works also for the cases $i = 1$ (conditioning on the leftmost variable x_1 in the truth table) and $i = n$ (conditioning on the rightmost variable x_n in the truth table). In the former case ℓ_j is an empty string for every j (we have only one block $B_\emptyset$ and the pseudocode makes a single pass through the main while loop) and in the latter case q_j is an empty string for every j (every block then contains only two vectors and every half-block is a single vector). When working with bit strings rather than numbers, we interpret inequalities as comparing the bit strings with respect to the lexicographic order. Similarly, we interpret $\mathtt{cp{+}1}$ as the bit string that follows after cp in the lexicographic order. Finally, to make the pseudocode terminate properly, we define ℓ_{k+1} to

be a string lexicographically greater than any bit string (even in the case when we consider only empty bit strings).

To construct the SLR F_1 for f_1 by a single pass through $\mathrm{SL}(F) = \{s_1, \ldots, s_k\}$, the algorithm works differently.

```
CD(F, x_i = 1)
 1  p ← 0 // parity initialization
 2  j ← 1 // index of the currently processed switch s_j
 3  while (ℓ_j = 0) and (b_j = 0) do
 4     {p ← (1 − p); j++}
 5  if (ℓ_j = 0) and (b_j = 1) and (q_j = 0)
 6     then {p ← (1 − p); j++}
 7  if (p = 0)
 8     then FV(F_1) ← FV(F)
 9     else FV(F_1) ← ¬FV(F)
10  while (ℓ_j = 0) and (b_j = 1) do
11     {output (0, q_j); j++}
12  while (j ≤ k) do
13     cp ← ℓ_j // current prefix for this iteration
14     p ← 0 // parity reset
15     while (ℓ_j = cp) and (b_j = 0) do
16        {p ← (1 − p); j++}
17     if (ℓ_j = cp) and (b_j = 1) and (q_j = 0)
18        then {if (p = 0) then output (ℓ_j, 0); j++}
19        else {if (p = 1) then output (ℓ_j, 0)}
20     while (ℓ_j = cp) and (b_j = 1) do
21        {output (ℓ_j, q_j); j++}
```

In this case it is more complicated to compute $\mathrm{FV}(F_1)$ and therefore the block B_0 is processed separately before the main while-loop starts. First the parity p of the number of switches in the half-block B_0^0 is counted on lines 3–4 and if $(\mathbf{0}, 1, \mathbf{0})$ is a switch then it is added to the computation of p on lines 5–6. Then the value of $\mathrm{FV}(f_1)$ stays the same as $\mathrm{FV}(f)$ if p is even and it takes the opposite value if p is odd. Note also that $(\mathbf{0}, 1, \mathbf{0})$ never generates a switch as it becomes the first (all-zero) vector in the truth table of f_1. Next we generate on lines 10-11 all remaining switches in the half-block B_0^1 (which of course survive conditioning). This part of the code (lines 3–11) covers also the case $i = 1$ (conditioning on the leftmost variable x_1) by identifying an empty string with an all-zero bit string with no bits.

Then the main while-loop starts on line 12. In this case it simpler than when constructing F_0. The first nested while-loop (lines 15–16) counts the parity p of the number of switches in the half-block $B_{\mathtt{cp}}^0$. The conditional statement on lines 17–19 treats the vector $v = (\ell_j, 1, \mathbf{0})$. If v is a switch then it survives the conditioning if p is even. If v is not a switch then it generates a switch $(\ell_j, \mathbf{0})$ in $\mathrm{SL}(F_1)$ if p is odd. The last nested while cycle on lines 20–21 generates switches (ℓ_j, q_j) with $q_j \neq \mathbf{0}$ in the half-block $B_{\mathtt{cp}}^1$ which of course all survive the conditioning.

If the input SLR F has k switches then both of the above-described pseudocodes for conditioning on x_i take $\mathrm{O}(n)$ time per switch (which is the work between two consecutive j++ commands), and therefore can be implemented to run in $\mathrm{O}(kn)$ time. Since each of the two output SLRs F_0 and F_1 have at most as many switches as the input SLR F, we can repeat the process $|S|$ times to achieve conditioning on any set S of variables in $\mathrm{O}\big(kn^2\big)$ time.

Generating switches for a permuted SLR

In this section we shall introduce an algorithm NEXT-SWITCH that takes as an input a SLR F of function f which respects a (non-identical) permutation $\pi = (x_{\pi(1)}, \ldots, x_{\pi(n)})$ of variables and a Boolean vector $b = (b_1, \ldots, b_n)$, and outputs the first (lexicographically smallest) switch s in the truth table of f with respect to the identical permutation $\phi = (x_1, \ldots, x_n)$ of variables that follows after b (i.e. s is lexicographically strictly greater than b). NEXT-SWITCH uses the conditioning algorithm from the previous section as a subroutine, and uses a stack of SLRs which originate from F by subsequent conditioning on the variables in the order given by ϕ. NEXT-SWITCH(f, b) starts with a full stack of depth $n + 1$ which has $F_0 = F$ on the bottom of the stack and each subsequent F_i is obtained from F_{i-1} by conditioning on variable x_i when its value is set to b_i:

$$F_i = \mathrm{CD}(F_{i-1}, x_i = b_i), 1 \leq i \leq n$$

We can think of the work of NEXT-SWITCH(f, b) as a walk on the complete binary tree which branches on the variables in the order given by the identical permutartion ϕ. The walk starts at the leaf of the tree indexed by b and backtracks from this leaf as far as necessary, which is followed by a forward walk towards the next switch s. Note that if $f(b) = v$ then s is the lexicographically smallest vector greater than b for which $f(s) = \neg v$. If we backtrack over an edge where $x_i = 1$ was set, we do not have to check anything: indeed going forward to locate s now makes no sense as we would have to go back to the vertex from which we just backtracked. On the other hand, if we backtrack over an edge where $x_i = 0$ was set, we must check whether s is in the subtree accessible by the edge $x_i = 1$. This can be checked by conditioning $x_i = 1$ and looking whether the resulting F_i contains no Boolean vector with value $\neg v$. If yes, we still have to backtrack, if no, we start the forward walk. The following subroutine NO-SWITCH (which returns yes/no) performs such a check.

```
NO-SWITCH(i)
1  POP(F_i)
2  F_i ← CD(F_{i−1}, x_i = 1)
3  PUSH(F_i)
4  return (FV(F_i) = v) and (SL(F_i) = ∅)
```

Here POP/PUSH are standard stack operations which remove/add an SLR from/to the top of the stack. NO-SWITCH removes the SLR F_i that we got by conditioning $x_i = 0$ and replaces it by SLR obtained by setting $x_i = 1$. Then it tests whether this SLR represents a constant v function (i.e. there is no switch in the subtree) which happens if and only if the first value is v and the switch-list is empty.

Now we are ready to present the pseudocode of NEXT-SWITCH. Here we present an iterative version of the algorithm, where we maintain a stack of SLRs explicitly. We find this version easier to explain and analyze. However, the algorithm also admits of a good functional implementation, where the stack is handled implicitly in recursive calls.

NEXT-SWITCH(f, b)

```
1  v ← FV(F_n)
2  i ← n // depth down the tree
3  while (i ≥ 1) and [(b_i = 1) or (NO-SWITCH(i))] do
4    {POP(F_i); i−−}
5  if (i = 0) // we reached the root; no next switch exists
6    then {output (0, ..., 0); terminate}
7  // we know the first i bits
8  for j from i + 1 to n do
9    if (FV(F_{j−1}) = ¬v)
10     then // all bits of the switch starting at j are zero
11       for r from j to n do
12         c_r ← 0 // we set the rth bit
13         F_r ← CD(F_{r−1}, x_r = 0)
14         PUSH(F_r)
15       exit the main for-loop over j // i.e. go to line 26
16     else
17       F_j ← CD(F_{j−1}, x_j = 0)
18       if SL(F_j) ≠ ∅
19         then
20           c_j ← 0 // we found the jth bit
21           PUSH(F_j)
22         else
23           c_j ← 1 // we found the jth bit
24           F_j ← CD(F_{j−1}, x_j = 1)
25           PUSH(F_j)
26 output (b_1, ..., b_{i−1}, 1, c_{i+1}, ..., c_n)
```

NEXT-SWITCH starts by setting $v = f(b)$. Note that F_n actually just represents the constant value $f(b)$ as the values of all variables were set to b: this constant is stored in $\mathrm{FV}(F_n)$, while $\mathrm{SL}(F_n)$ is empty. The while-loop on lines 3–4 does the backward part of the walk through the tree. It unconditionally backtracks over edges where $x_i = 1$ was set (i.e. $b_i = 1$ holds), and if this is not the case (the backtrack occurs over an edge where $x_i = 0$ was set) it runs NO-SWITCH(i) and backtracks only if the result is TRUE. When the loop ends there are two possibilities:

- The stack is empty except for F_0, which means that we backtracked to the root of the tree. Hence, the truth table of f with respect to ϕ is constant v from vector b onwards, so there is no switch to output. We treat this "failure" case on lines 5–6 by outputting the all-zero vector $\mathbf{0}$. No confusion is possible here, since $\mathbf{0}$ cannot be a switch.
- Some F_i for which $x_i = 1$ was set sits at the top of the stack and the subtree rooted at this node of the tree is guaranteed to contain some vector with value $\neg v$. Note that we are seeking a vector that respects the ordering ϕ, but the SLRs on our stack still respect π. Nevertheless, after reordering the columns of the truth table of this partial function with respect to ϕ, the desired vector (the first with value $\neg v$) is still going to be in the subtree rooted at the current node. Also note that this scenario means we know the first i bits of the next switch: the first $i - 1$ are given by $b_1, \ldots, b_{i-1}$, and the ith bit must be 1.

In the latter case the main for-loop on line 8 starts the forward walk one step per iteration. In each iteration we first check on line 9 whether the current subfunction has value $\neg v$ at its all-zero vector (which is of course the first vector with respect to both π and ϕ) and if this is the case we blindly follow the all-zero branch of the tree setting the c variables to zero and pushing the corresponding SLRs to the stack. Then we exit the main for-loop as we have already reached the leaf of the tree which represents the switch we were looking for.

If the all-zero vector of the current subfunction has value v we enter the else branch of the conditional command on line 16. We first try to follow the $x_j = 0$ branch and generate F_j for this branch. If $F_j \neq \emptyset$ then the first switch is in this subtree and we iterate, in the other case this subtree is constant v and the first switch then must be in the $x_j = 1$ branch so we generate F_j for this branch. In both cases we record the direction of the walk in the tree in the c variable and push the corresponding SLR to the stack.

Finally, when we exit the main for-loop, we output on line 26 the switch which corresponds to the leaf of the tree where the forward walk terminated.

To analyze the complexity of NEXT-SWITCH(f, b), suppose that the input SLR F of f has k switches (which are vectors of length n similarly as b). The backward walk in NEXT-SWITCH calls CD (conditioning on a variable) at most once per step (when NO-SWITCH is called), and the forward walk calls CD at most twice per step. Hence the total number of CD calls is $\mathrm{O}(n)$ and each such call takes $\mathrm{O}(kn)$ time as discussed in the previous section. The complexity of the rest of the NEXT-SWITCH algorithm (in particular of the PUSH and POP calls) is dominated by the CD calls, and hence the overall complexity is $\mathrm{O}(kn^2)$.

Equivalence testing

Let f and g be two Boolean functions on the same set $\{x_1, \ldots, x_n\}$ of variables represented by SLRs F and G where F respects an arbitrary permutation $\pi = (x_{\pi(1)}, \ldots, x_{\pi(n)})$ and G respects the identical permutation ϕ (we may assume this without loss of generality: if ϕ is not an identical permutation we can renumber the variables accordingly). Let us furthermore assume that SL(F) consists of k_f switches and SL(G) consists of k_g switches where SL(G) = $(s_1, \ldots, s_{k_g})$.

To test whether f and g are logically equivalent one can either compile F and G into some other representation language (e.g. into OBDDs as in (Čepek and Chromý 2020)) or try to answer the equivalence query directly by manipulating F and G. It should be obvious that constructing a SLR of f that respects ϕ (or a SLR of g that respects π) first and only then comparing the two SLRs of f and g which both respect the same permutation of variables will not work, since the output representation after the first step may be exponentially large with respect to the size of the input. However, we do not have to construct the entire SLR of f that respects ϕ to test the equivalence. It suffices to generate its switches one by one and compare them with the switch-list SL(G) until either a mismatch is found or an identical switch-list is generated (note that two SLRs which respect the same permutation of variables represent the same function if and only if they are identical). This idea is described in the following

algorithm EQ-TEST.

```
EQ-TEST(F, G)
 1  if (FV(F) ≠ FV(G))
 2     then {return FALSE}
 3  F_0 ← F
 4  PUSH(F_0)
 5  for i from 1 to n do
 6     F_i ← CD(F_{i-1}, x_i = 0)
 7     PUSH(F_i)
 8  p ← 1 // running index through SL(G)
 9  b ← NEXT-SWITCH(f, (0, ..., 0))
10  while (p ≤ k_g) and (b = s_p) do
11     {b ← NEXT-SWITCH(f, b); p++}
12  if (p ≤ k_g)
13     then {return FALSE}
14     else {return (b = (0, ..., 0))}
```

The algorithm first tests whether the values at the all-zero vector are the same and if not trivially outputs (on line 2) that f and g are not equivalent. Then it fills the stack on lines 3–7 with SLRs that correspond to the all-zero vector (this is necessary before running NEXT-SWITCH for the first time) and computes the first switch of f with respect to the identical permutation of variables on line 9. The main while-loop on lines 10–11 iteratively checks that the generated switch of f matches the corresponding switch in SL(G) and generates the next switch of f. If the loop ends before going through the entire SL(G) which is tested on line 12 then a mismatch was found and non-equivalence is reported on line 13. If the loop goes through SL(g) entirely, then f and g are equivalent if and only if the last call of NEXT-SWITCH(f, b) found no further switch of f and hence returned the all-zero vector.

EQ-TEST(F,G) first fills the stack which takes $0(k_f n^2)$ and then at most $k_g + 1$ times calls NEXT-SWITCH(f,b). Each such call takes $0(k_f n^2)$ time (as discussed in the previous section) and so the overall complexity of EQ-TEST(F,G) is $0(k_f k_g n^2)$ which beats the $0(k_f k_g n^3)$ time complexity of indirect equivalence testing presented in (Čepek and Chromý 2020) (which uses compilation into OBDDs) by a factor of n.

Conclusions

In this paper we have presented an equivalence testing algorithm that manipulates the two input SLRs to answer the query directly without a compilation into another representation language. Its time complexity is $0(k^2 n^2)$ where k is the total number of switches in the two input switch-list representations and n is the number of variables. The presented algorithm beats the $0(k^2 n^3)$ time complexity of indirect equivalence testing from (Čepek and Chromý 2020) (which uses compilation into OBDDs) by a factor of n. This leaves the sentential entailment query the only one from the standard set of queries introduced in the Knowledge Compilation Map (Darwiche and Marquis 2002) that has no direct algorithm for the switch list representations. Constructing such an algorithm (perhaps using the switch generation technique presented in this paper) will be a subject of our future research.

Acknowledgments

The authors gratefully acknowledge a support by Czech Science Foundation (Grant 19-19463S). This research was also partially supported by TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215.

References

Čepek, O.; and Chromý, M. 2020. Properties of Switch-List Representations of Boolean Functions. *Journal Of Artificial Intelligence Research* 69: 451–479.

Čepek, O.; and Chromý, M. 2020. Switch-List Representations in a Knowledge Compilation Map. In Bessiere, C., ed., *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 1651–1657. ijcai.org. doi:10.24963/ijcai.2020/229. URL https://doi.org/10.24963/ijcai.2020/229.

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

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

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. ISSN 0166-218X. doi:10.1016/j.dam.2004.08.009.

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