How to apply SAT-solving for the equivalence test of monotone normal forms*

Martin Mundhenk and Robert Zeranski

Friedrich-Schiller-Universität Jena, Germany
{martin.mundhenk, robert.zeranski}@uni-jena.de

Abstract. The equivalence problem for monotone formulae in normal form Monet is in coNP, is probably not coNP-complete [10], and is solvable in quasi-polynomial time $n^{o(\log n)}$ [7].
We show that the straightforward reduction from Monet to UnSat yields instances, on which actual Sat-solvers (SAT4J) are slower than current implementations of Monet-algorithms [9]. We then improve these implementations of Monet-algorithms notably, and we investigate which techniques from Sat-solving are useful for Monet. Finally, we give an advanced reduction from Monet to UnSat that yields instances, on which the Sat-solvers reach running times, that seem to be magnitudes better than what is reachable with the current implementations of Monet-algorithms.

1 Introduction

The equivalence problem for Boolean formulae is one of the classical coNP-complete problems. It remains coNP-complete also if the formulae are given in normal form. For monotone formulae—i.e. formulae with conjunctions and disjunctions, but without negations—the equivalence problem is coNP-complete, too [20], but its complexity drops, if the formulae are in conjunctive or disjunctive normal form. The reason is that for every monotone formula, the minimal equivalent formula in the considered normal form is unique, and the minimal equivalent normal form formula is efficiently computable from the non-minimal normal form formula. Therefore, checking whether two monotone formulae in conjunctive normal form (resp. two monotone formulae in disjunctive normal form) are equivalent, can be done in polynomial time. The remaining case is the equivalence problem for monotone formulae, where one formula is in conjunctive normal form and the other is in disjunctive normal form. This is the problem Monet, i.e. Mo(notone) n(ormal form) e(quivalence) t(est). This problem is strongly related to dualization of monotone conjunctive normal forms and transversal hypergraph generation [5]. This means that an algorithm for Monet solves many fundamental problems in a wide range of fields, including artificial intelligence and logic, computational biology, database theory, data mining and machine learning, mobile communication systems, distributed systems, and graph theory (see [9] for an

* This work was supported by the DFG under grant MU 1226/8-1 (SPP 1307/2).

overview). The currently best MONET algorithms have quasi-polynomial running time $n^{o(\log n)}$, or polynomial time using $\mathcal{O}(\log^2 n)$ nondeterministic bits [6,7,10]. Thus, on the one hand, MONET is probably not coNP-complete, but on the other hand a polynomial time algorithm is not yet known. This situation turns MONET into one of the very few problems "between" P and NP- resp. coNP-hard. The exact complexity of the general problem MONET is a long standing famous open question [17].

As for evaluating the practical performance, there are several experimental studies on known algorithms for MONET or equivalent problems [1,4,12,11,15,24,26]. Over all, the algorithms by Fredman and Khachiyan [7], that are the MONET algorithms with the best worst-case upper-bounds, turn out to be strong practical performers [9].

In this paper, we mainly address the following two questions.

Can the performance of the algorithms by Fredman and Khachiyan be improved by using techniques from SAT*-solving?* The two algorithms by Fredman and Khachiyan basically use a technique similar to the DPLL-algorithm for SAT [3]. Both algorithms leave—more or less—open which variable to choose as the splitting variable. We added unit propagation and tried several strategies known from SAT-solving like MOMs [18], BOHM [2], and clause reduction heuristic [13]. In our experimental study we show that unit propagation and the mentioned strategies notably improve our implementations of the algorithm.

Are SAT*-solvers good for* MONET*?* Since MONET reduces to the complement of SAT, it is straightforward to use a reduction and a SAT-solver to solve MONET. Eventually, it turned out not to be that straightforward. We give a reduction from MONET to the complement of SAT that does not increase the size of the instances. Using this reduction function and a SAT-solvers reaches computation times that are very much better than what is currently possible with MONET solvers.

This paper is organized as follows. In Section 2 we introduce the basic notation and review the FK-algorithms [7]. Section 3 shows how unit propagation and strategies for choosing a splitting variable can be used in the FK-algorithms. Section 4 considers how MONET can be reduced UNSAT. Our experimental results are discussed in Section 5, and conclusions are drawn in Section 6.

2 Preliminaries

Monotone formulae and equivalence A Boolean formula φ is called *monotone* if φ has only $\wedge$ and $\vee$ as connectives–*no negations are allowed.* Let V_φ denote the set of variables of φ. An *assignment* is denoted as a set $\mathcal{A} \subseteq V_\varphi$ of variables, where x is assigned *true* iff $x \in \mathcal{A}$. Otherwise x is assigned *false*. A *term* is a set of variables that is either interpreted as a conjunction or as a disjunction of the variables. We call a term a *monomial* if it is a conjunction and we call it a *clause* if it is a disjunction. In this paper m always denotes a monomial and c always denotes a clause. A monotone Boolean formula is a DNF (*disjunctive normal*

form) if it is a disjunction of monomials, and it is a CNF (*conjunctive normal form*) if it is a conjunction of clauses. Throughout the whole paper we regard a CNF (resp. DNF) as a set of clauses (resp. monomials). A monotone DNF or CNF is called *irredundant* if it contains no two terms such that one contains the other. It is important, that every monotone Boolean formula has a unique irredundant monotone CNF and DNF [19]—and for a given monotone CNF (resp. DNF) the irredundant CNF (resp. DNF) can be obtained in quadratic time by deleting all supersets of terms. Two monotone Boolean formulae are *equivalent* if and only if they have the same irredundant monotone CNF. This paper deals with the equivalence test of monotone formulae in different normal forms.

MONET: Instance: irredundant, monotone DNF D and CNF C
Question: are D and C equivalent?

In this paper D always denotes a monotone DNF and C always denotes a monotone CNF. The length of a term is the number of variables in this term, and the length of a normal form D (resp. C) is the number of terms in it.

The algorithms of Fredman and Khachyian

The algorithms with the best known worst-case upper bound for solving MONET are by Fredman and Khachyian [7]. Both these algorithms search for a witness of non-equivalence by a depth-first search in the tree of all assignments—we call such a witness *conflict assignment*. Note that this technique is very similar to the DPLL-algorithm for SAT [3]. The FK-algorithms work as follows. In the first step the input formulae are modified to irredundant normal forms.[1] Unless the formulae are small enough to check them by brute force, the algorithm chooses a variable, sets the value of this variable to *false* and modifies the formulae due to this assignment—we call the variable chosen the *splitting variable*. Next, the equivalence of the new formulae will be tested recursively. If the recursive call does not yield a conflict assignment, the splitting variable is set to *true* and the accordingly modified formulae are tested recursively. If this second recursive call does not yield a conflict assignment, then the formulae must be equivalent.

The modifications of the formulae are as follows. If a variable is set to true in a DNF, then this variable can be deleted in each monomial. And if it is set to false, then all terms which contain this variable can be deleted. For the CNF it is dual. Let ϕ be a DNF or CNF, and let x be a splitting variable. Then ϕ_0^x denotes the formula that consists of terms of ϕ from which x is removed. Analogously, ϕ_1^x denotes the formula that consists of all terms of ϕ that do not contain x.

$$\phi_0^x = \{t - \{x\} : t \in \phi \text{ and } x \in t\} \qquad \phi_1^x = \{t : t \in \phi \text{ and } x \notin t\}$$

Thus, if x is set to true in D and C, we obtain $D_0^x \vee D_1^x$ and C_1^x. If it is set to false, we obtain D_1^x and $C_0^x \wedge C_1^x$.

Fredman and Khachyian [7] provide necessary conditions for equivalence, that are also checked during the depth-first search, as follows.

[1] This step is necessary because the formulae will be modified in further steps and the algorithms are recursive.

(1) $m \cap c \neq \emptyset$ for every monomial $m \in D$ and every clause $c \in C$.
(2) D and C must contain exactly the same variables, i.e. $V_D = V_C$.
(3) $\max\{|m| : m \in D\} \leq |C|$ and $\max\{|c| : c \in C\} \leq |D|$.

FK-algorithm A [7] The central question is about the choice of the splitting variable. Fredman and Khachiyan provide an additional necessary condition for the FK-algorithm A which ensures the existence of a frequent variable.

$$\sum_{m \in D} 2^{|V_D|-|m|} + \sum_{c \in C} 2^{|V_C|-|c|} \geq 2^{|V_D|}. \tag{4}$$

If this condition is violated the formulae are not equivalent and a conflict assignment can be computed in linear time. As splitting variable FK-algorithm A chooses a variable with frequency $\geq 1/\log(|D| + |C|)$ in either D or C.

Theorem 1. [7] *The FK-algorithm A has running time $n^{\mathcal{O}(\log^2 n)}$ on input (D, C), where $n = |D| + |C|$.*

Algorithm 1 shows a pseudo-code listing of FK-algorithm A. It is shown in an experimental study in [12] that FK-algorithm A performs well in practice. There is also a version presented in [23] which works in space polynomial in $|D|$.

FK-algorithm B [7] What happens if the first recursive call FK-A$(D_1^x, C_0^x \wedge C_1^x)$ of FK-algorithm A does not yield a witness for non-equivalence? In this case we gain the information that D_1^x is equivalent to $C_0^x \wedge C_1^x$. FK-algorithm A does not use the fact that the second recursive call is performed only if the first recursive call does not yield a witness for non-equivalence. But FK-algorithm B makes use of this. The main conclusion is a restriction for the search tree when the value of the splitting variable is set to true. It then suffices to find a conflict assignment $\mathcal{A}$ for the formulae D_0^x and C_1^x with the restriction that $\mathcal{A}(C_0^x) = 0$ (cf. [7,9]). Hence, one has to check all maximal assignments not satisfying C_0^x only. Note that there are exactly $|C_0^x|$ assignments, one for every clause $c \in C_0^x$ (the resp. assignment is $V_{C_0^x} - c$).

Thus, if the first recursive call does not yield a conflict assignment it suffices to perform a recursive call for every clause $c \in C_0^x$ on the pair $(D_0^{c,x}, C_1^{c,x})$, where $D_0^{c,x}$ and $C_1^{c,x}$ denote the formulae we obtain if we set all variables in c to false. We receive a similar result if we swap the chronological order of the first and the second recursive call, cf. [7,9]. If the recursive call on the pair $(D_0^x \vee D_1^x, C_1^x)$ does not yield a witness for non-equivalence it suffices to perform a recursive call for every monomial $m \in D_0^x$ on the pair $(D_1^{m,x}, C_0^{m,x})$, where $D_1^{m,x}$ and $C_0^{m,x}$ denote the formulae we obtain if we set all variables in m to true (if we found a conflict, the resp. assignment is m).

One of the main differences to FK-algorithm A is the choice of the variable and the advanced branching. The choice of the splitting variable does not matter in [7] for the theoretical upper bound, because FK-algorithm B chooses

Algorithm 1 The FK-algorithm A (FK-A)

Input: irredundant, monotone DNF D and CNF C
Output: $\emptyset$ in case of equivalence; otherwise, assignment $\mathcal{A}$ with $\mathcal{A}(D) \neq \mathcal{A}(C)$
1: make D and C irredundant
2: **if** one of conditions (1)–(4) is violated **then**
3: **return** conflict assignment
4: **if** $|D| \cdot |C| \leq 1$ **then**
5: **return** appropriate assignment found by a trivial check
6: **else**
7: choose a splitting variable x with frequency $\geq 1/\log(|D|+|C|)$ in D or C
8: $\mathcal{A} \leftarrow$ FK-A$(D_1^x, C_0^x \wedge C_1^x)$ // recursive call for x set to *false*
9: **if** $\mathcal{A} = \emptyset$ **then**
10: $\mathcal{A} \leftarrow$ FK-A$(D_0^x \vee D_1^x, C_1^x)$ // recursive call for x set to *true*
11: **if** $\mathcal{A} \neq \emptyset$ **then return** $\mathcal{A} \cup \{x\}$
12: **return** $\mathcal{A}$

Algorithm 2 The FK-algorithm B (FK-B)

Input: irredundant, monotone DNF D and CNF C
Output: $\emptyset$ in case of equivalence; otherwise, assignment $\mathcal{A}$ with $\mathcal{A}(D) \neq \mathcal{A}(C)$
1: make D and C irredundant
2: **if** one of conditions (1)–(3) is violated **then return** conflict assignment
3: **if** $\min\{|D|, |C|\} \leq 2$ **then**
4: **return** appropriate assignment found by a trivial check
5: **else**
6: choose a splitting variable x from the formulae
7: **if** x is at most μ-frequent in D **then**
8: $\mathcal{A} \leftarrow$ FK-B$(D_1^x, C_0^x \wedge C_1^x)$ // recursive call for x set to *false*
9: **if** $\mathcal{A} \neq \emptyset$ **then return** $\mathcal{A}$
10: **for all** clauses $c \in C_0^x$ **do**
11: $\mathcal{A} \leftarrow$ FK-B$(D_0^{c,x}, C_1^{c,x})$ // see ⟨1⟩
12: **if** $\mathcal{A} \neq \emptyset$ **then return** $\mathcal{A} \cup \{x\}$
13: **else if** x is at most μ-frequent in C **then**
14: $\mathcal{A} \leftarrow$ FK-B$(D_0^x \vee D_1^x, C_1^x)$ // recursive call for x set to *true*
15: **if** $\mathcal{A} \neq \emptyset$ **then return** $\mathcal{A} \cup \{x\}$
16: **for all** monomials $m \in D_0^x$ **do**
17: $\mathcal{A} \leftarrow$ FK-B$(D_1^{m,x}, C_0^{m,x})$ // see ⟨2⟩
18: **if** $\mathcal{A} \neq \emptyset$ **then return** $\mathcal{A} \cup m$
19: **else**
20: $\mathcal{A} \leftarrow$ FK-B$(D_1^x, C_0^x \wedge C_1^x)$ // recursive call for x set to *false*
21: **if** $\mathcal{A} = \emptyset$ **then**
22: $\mathcal{A} \leftarrow$ FK-B$(D_0^x \vee D_1^x, C_1^x)$ // recursive call for x set to *true*
23: **if** $\mathcal{A} \neq \emptyset$ **then return** $\mathcal{A} \cup \{x\}$
24: **return** $\mathcal{A}$

⟨1⟩: $D_1^x \equiv C_0^x \wedge C_1^x$: recursive call for all maximal non-satisfying assignments of C_0^x for x set to true
⟨2⟩: $D_0^x \vee D_1^x \equiv C_1^x$: recursive call for all minimal satisfying assignments of D_0^x for x set to false

an appropriate branching with respect to the frequency of the splitting variable. Therefore, the algorithm uses a frequeny-threshold $\mu(n)$ with the property $\mu(n)^{\mu(n)} = n$. Note, that $\mu(n) \sim \log n / \log\log n$. Thus, $\mu(n) \in o(\log n)$. A pseudo-code listing of FK-algorithm B is given in Algorithm 2. There, a variable x is called *at most μ-frequent in D* (resp. C) if its frequency is at most $1/\mu(|D| \cdot |C|)$, i.e. $|\{m \in D : x \in m\}|/|D| \leq 1/\mu(|D| \cdot |C|)$.

Theorem 2. [7] *FK-algorithm B has running time $n^{o(\log n)}$ on input (D, C), where $n = |D| + |C|$.*

3 Unit propagation and decision strategies

As mentioned before, the choice of the splitting variable is free in FK-algorithm B, and it is almost free in FK-algorithm A. In the old implementations in [9], the first variable in the formula is taken as splitting variable for FK-algorithm B, and the first variable that satisfies the frequency condition is taken as splitting variable in FK-algorithm A. In our new implementations, we replaced this by running unit propagation and choosing a splitting variable according to a somewhat more involved strategy.

Unit propagation for MONET Unit propagation (UP), or also called one-literal rule, is a technique for simplifying a set of clauses in an automated theorem proving system. This technique is also used for the DPLL-algorithm [3]. A clause (resp. monomial) is a *unit clause* (resp. *unit monomial*) if it consists of one variable only. How can the FK-algorithms gain from considering unit clauses or unit monomials? There are two cases.
Case (a): *There is a unit clause in C.* Let $\{x\} \in C$, and let D and C satisfy condition (1). Then C_0^x–i.e. the set of clauses obtained from CNF C by setting x to false–is unsatisfiable, and D_1^x–i.e. the set of monomials obtained from DNF D by setting x to false–is unsatisfiable, too, because x is contained in all monomials of D (cf. condition (1)). Thus, the recursive call for setting the splitting variable x to false will yield no conflict assignment and can be left out.

Lemma 1. *Let $\{x\} \in C$. Then $D \equiv C$ if and only if $D_1^x = \emptyset$ and $D_0^x \equiv C_1^x$.*

According to Lemma 1, if C contains a unit clause $\{x\}$ then it suffices to check condition (1) on (D_0^x, C_1^x) and to do the recursive call FK-A(D_0^x, C_1^x).

Case (b): *There is a unit monomial in D.* This case is dual to case (a).

Lemma 2. *Let $\{x\} \in D$. Then $D \equiv C$ if and only if $C_1^x = \emptyset$ and $D_1^x \equiv C_0^x$.*

Note, that for formulae D and C satisfying condition (1) it is impossible that D and C contain the same unit term (excepted $D = C = \{\{x\}\}$). Thus, one can search for all unit clauses and unit monomials in the formulae and setting the variables to the resp. values. Because both formulae are irredundant we only have to delete the unit terms in the resp. formula. If there is no unit term left in

D and C we can choose a splitting variable. It is also possible to avoid checking condition (1). Thus, if we find a unit term $\{x\}$ in D (resp. C) we have to check that x is contained in all terms of C (resp D).

Decision strategies for the splitting variable A main difference between FK-algorithm A and FK-algorithm B is the choice of the splitting variable. FK-algorithm A chooses a variable that is at least $\log(|D| + |C|)$-frequent in either D or C—one can also simply choose the most frequent variable. (If D is equivalent to C, then there exists a $\log(|D|+|C|)$-frequent variable [7]). However, FK-algorithm B is free to choose any variable as splitting variable. In general, a random choice is not a good strategy (see experiments with FKB(rMin) in Section 5). Thus, it is interesting to investigate whether strategies for choosing the splitting variable improve FK-algorithm B. We tried the following strategies.

(i) Choose the first free variable [9]—it is related to a random choice.
(ii) Choose the most frequent variable.
(iii) Choose the most frequent variable in the smallest terms (MOMs [18]).
(iv) Choose a variable randomly in the smallest terms.
(v) Choose the variable with maximal occurence in small terms (BOHM [2]).
(vi) Clause reduction heuristic (CRH [13]).

4 Solving Monet using SAT-solvers

Since Monet is in coNP, it is polynomial-time reducible to UnSat. Therefore, a straightforward approach to solve Monet is to use this reduction and a common SAT-solver. Clearly, (D, C) is in Monet if and only if $\neg(D \to C) \notin$ Sat and $\neg(C \to D) \notin$ Sat.

Since D is a DNF and C is a CNF, the formula $\neg(C \to D)$ is represented by the CNF $C \cup D^{\neg}$, where $D^{\neg}$ is the set of monomials in D, in which all appearances of variables are negated.[2] Similarly, the other part $\neg(D \to C)$ can be seen as a conjunction of two DNFs and can be brought into conjunctive normal form using the standard translation by Tseitin [25] that results in an equisatisfiable formula in CNF. Even though this translation enlarges the formula only linearly, our experiments with SAT-solvers on the translated formulae yielded computation times that were worse than that of the FK-algorithms (see Section 5, Table 2).

Let us consider the formula $\neg(D \to C)$ more precisely. It is satisfied by assignments that satisfy a monomial in the DNF D and falsify all clauses in the CNF C. This happens if and only if there is a monomial $m \in D$ and a clause $c \in C$ such that $m \cap c = \emptyset$. This condition can be checked in time $\mathcal{O}(|D| \cdot |C| \cdot n)$, where n denotes the number of variables. Therefore, this test can be used in the polynomial time function f that reduces Monet to UnSat as follows.

$$f(D, C) = \begin{cases} C \cup D^{\neg}, & \text{if } \ m \cap c \neq \emptyset \text{ for all } m \in D \text{ and all } c \in C \\ \mathit{true}, & \text{otherwise} \end{cases}$$

[2] Remind that $C \cup D^{\neg}$ represents $C \wedge D^{\neg}$.

where *true* denotes a formula that is satisfied by every assignment.

Lemma 3. *The above function f is a polynomial-time function that reduces* MONET *to* UNSAT*.*

This reduction function can be seen as a generalization of reduction used in [5,8]. Note that the property of non-empty intersection of every clause and monomial is condition (1) from the necessary conditions for equivalence, and this is also checked in the FK-algorithms. Nevertheless, this check is not necessary for the correctness of the algorithms, but needed in the proof of the upper bound for the running time [7]. In our implementations we avoid to check condition (1), because in our experiments it seems to waste time only.

5 Experiments

We experimentally compare the following implementations of algorithms for MONET in Java. All experiments were conducted on an Intel i7-860, 2.8 GHz, 8 GB RAM running Ubuntu 10.04.

(1) The old implementations used by [9] for the FK-algorithms A and B. We call these implementations FKA(HHM) and FKB(HHM). The strategy of FKA(HHM) for choosing the splitting variable is to choose the firstly found log-frequent variable. The strategy of FKB(HHM) is to take the first variable.

(2) Our new implementations of the FK-algorithms A and B. They distinguish in the strategy how the splitting variable is chosen.
   - FKA(mf) and FKB(mf) are the FK-algorithm A and B with the strategy of choosing the most frequent variable.
   - FKA(th) is FK-algorithm A that chooses the firstly found log-frequent (threshold) variable. This is a new implementation of FKA(HHM).
   - FKB(BOHM) (resp. CRH and MOMs) denotes FK-algorithm B with BOHM (resp. CRH and MOMs) heuristic.
   - FKB(rMin) is FK-algorithm B that chooses randomly a variable in a term of minimal length.

(3) The implementation that uses our reduction to the complement of SAT and Sat4j [14] as SAT-solver. For simplicity, we call this *reduction to* SAT .

We run tests on test data that are equivalent formulae, and on those that are not equivalent. For equivalent formulae, the runtimes strongly depend on the structure of the test data. For non-equivalent formulae, this is not the case. Therefore we consider both these cases separately. We use test data M(k), TH(k), and SDTH(k) that was used also in previous studies [12,9] and that are articifially produced MONET instances. Additionally we have test data from the UC Irvine Machine Learning Repository [22,4] and from the Frequent Itemset Mining Implementations Repository (FIMI) [21]. Notice that an instance $(A, B) \in$ MONET if and only if $(B, A) \in$ MONET, where on the left hand side the set of terms A is read as a DNF and on the right hand side it is read as a CNF (B similarly).

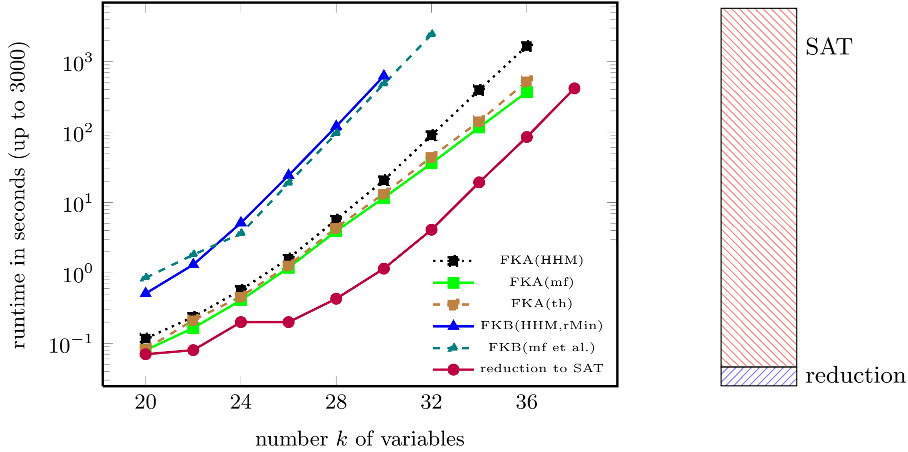

Fig. 1. Runtimes and reduction ratio on $\mathrm{M}(k)$

Matching M(k). The formula M_k of k variables (k even) consists of the terms $\{\{x_i, x_{i+1}\} : 1 \leq i < k, i \text{ is even}\}$. Thus, $|\mathrm{M}_k| = k/2$. The formula $\widetilde{\mathrm{M}}_k$ equivalent to M_k consists of the $2^{k/2}$ terms obtained by choosing one variable from every term of M_k. The instance $\mathrm{M}(k)$ is the pair $(\mathrm{M}_k, \widetilde{\mathrm{M}}_k)$. Notice that the size of $\mathrm{M}(k)$ is exponential in k. Simply said, the matching instances are pairs of equivalent formulae, where one formula is exponentially larger than the other.

Figure 1 shows the runtimes for the matching instances. It shows that the FKB-implementations are the slowest, the FKA-implementations are intermediate and the reduction to SAT is the fastest. For the reduction to SAT, the *reduction ratio* shows how much of the runtime was used by the reduction function and by the SAT-solver. One can see that the new FKA-implementation is better than the old one.

Threshold TH(k). The formula T_k of k variables is the set of terms $\{\{x_i, x_j\} : 1 \leq i < j \leq k, j \text{ is even}\}$. Thus, $|\mathrm{T}_k| = k^2/4$. The formula $\widetilde{\mathrm{T}}_k$ equivalent to T_k is $\widetilde{\mathrm{T}}_k = \{\{1, \ldots, 2t-1\} \cup \{2t+2, 2t+4, \ldots, k\} : 1 \leq t \leq k/2\} \cup \{\{2, 4, \ldots, k\}\}$. This yields $|\widetilde{\mathrm{T}}_k| = k/2+1$. The instance $\mathrm{TH}(k)$ is $(\mathrm{T}_k, \widetilde{\mathrm{T}}_k)$ and has size in $\mathcal{O}(k^2)$. Simply said, the threshold instances are pairs of equivalent formulae, where one formula is quadratic in the size of the other.

Figure 2 shows the runtimes for the threshold instances. It shows that the old FKA-implementation is the slowest, and the old FKB-implementation and the new FKB(rMin) are the fastest among the FK-implementations. The reason is that the choice of the variable does not really matter on these instances, and using a strategy wastes time. Again, the reduction to SAT is the fastest and seems to have the slowest slope.

Self Dual Threshold SDTH(k). The formula ST_k of k variables (k even) is the set of terms $\{\{x_{k-1}, x_k\}\} \cup \{\{x_k\} \cup m : m \in \mathrm{T}_{k-2}\} \cup \{\{x_{k-1}\} \cup m : m \in \widetilde{\mathrm{T}}_{k-2}\}$.

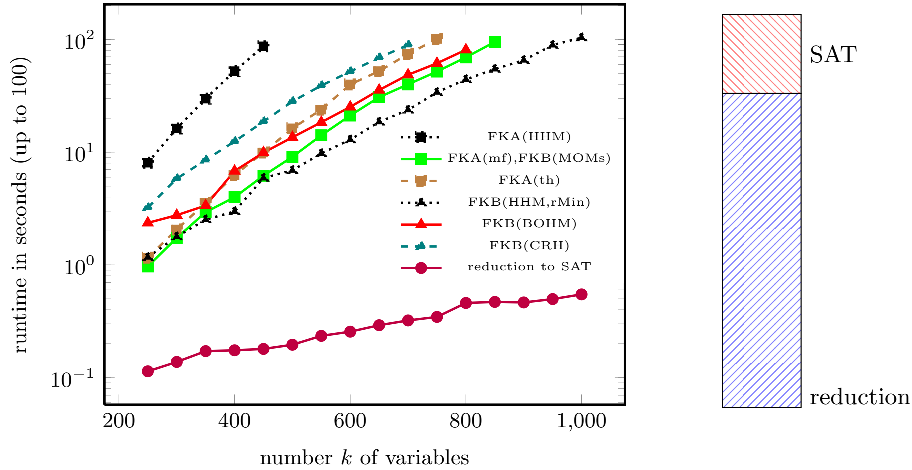

Fig. 2. Runtimes and reduction ratio on TH(k)

Note that ST_k read as DNF is equivalent to ST_k read as CNF. The instance SDTH(k) is $(\mathrm{ST}_k, \mathrm{ST}_k)$ and has size $\mathcal{O}(k^2)$. Simply said, the self dual threshold instances are pairs of equivalent formulae of the same size.

Figure 3 shows the runtimes for the self-dual-threshold instances. It shows that the old FKA-implementation is the slowest, but the new FKB(rMin)—that was quite fast on the threshold instances—is very slow, too. The other new FKB-implementations are the fastest among the FK-implementations. Again, the reduction to SAT is the fastest.

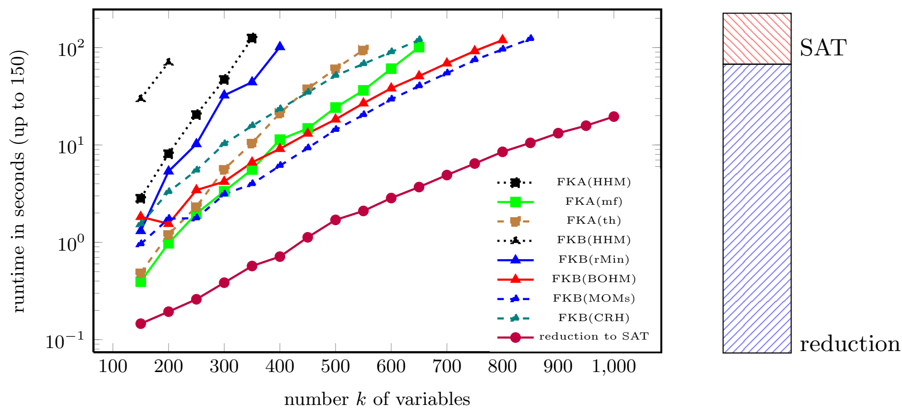

Fig. 3. Runtimes and reduction ratio on SDTH(k)

Connect-4 L(r) and W(r). "Connect-4" is a board game. Each row of the dataset corresponds to a minimal winning (W) or losing (L) stage of the first player, and is represented as a term. A term of an equivalent formula of a set of winning stages (represented as a formula in DNF or CNF) is a minimal way to disturb winning/losing moves of the first player. To form a dataset, we take the first r rows of the minimal winning stage (called W_r) and the first r rows of the minimal losing stage (called L_r) [4,16]. To compute the equivalent formula $\widetilde{\mathrm{L}}_r$ and $\widetilde{\mathrm{W}}_r$ we used the DL-algorithm [4]. Thus, we have $\mathrm{L}(r) = (\mathrm{L}_r, \widetilde{\mathrm{L}}_r)$ and $\mathrm{W}(r) = (\mathrm{W}_r, \widetilde{\mathrm{W}}_r)$ as instances. The set of testdata are from the UC Irvine Machine Learning Repository [22]. It is used to compare algorithms that compute equivalent normal forms. The smallest formula L(100) consists of 2,441 terms with 77 variables, and L(1600) is the largest and has 214,361 terms with 81 variables. W(100) has a size of 387 terms with 76 variables, and W(3200) has 462,702 terms with 82 variables. Figure 4 shows the runtimes for the Connect-4

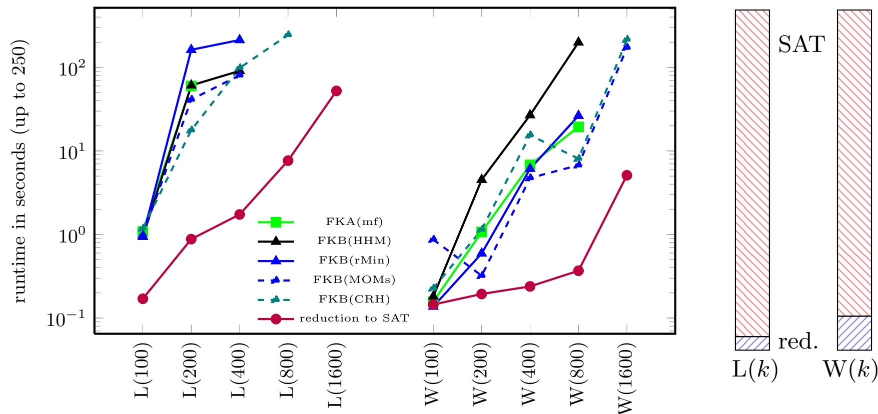

Fig. 4. Runtimes and reduction ratio on L(r) and W(r)

instances. Only few instances were solvable within the given time bound. All FK-implementations behave similar, and for sake of clarity we left some of them out in Figure 4. The new FKB(CRH) is the fastest among the FK-implementations. As before, the reduction to SAT is the fastest.

BMS-WebView-2 BMS(s) and accidents AC(s). This testdata is generated by enumerating all maximal frequent sets from datasets "BMS-WebView-2" and "accidents". For a dataset and a support threshold s, an itemset is called *frequent* if it is included in at least s members, and *infrequent* otherwise. A frequent itemset included in no other frequent itemset is called a *maximal frequent itemset*, and an infrequent pattern including no other infrequent itemset is called a *minimal infrequent itemset*. A minimal infrequent itemset is included in no maximal frequent itemset, and any subset of it is included in at least one maximal frequent itemset. Thus, the dual of the set of the complements of

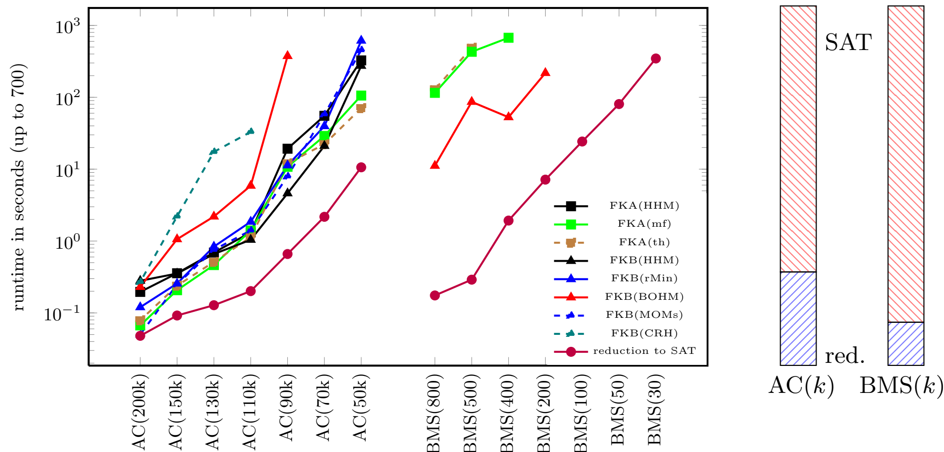

Fig. 5. Runtimes and reduction ratio on AC(s) and BMS(s)

maximal frequent itemsets is the set of minimal infrequent itemsets [16]. Note, if we want to check the correctness of enumerating all maximal frequent sets we can use MONET, because it is equivalent to this problem [7]. The problem instances are generated by enumerating all maximal frequent sets from datasets BMS-WebView-2 BMS(s) and accidents AC(s) with threshold s, taken from Frequent Itemset Mining Implementations Repository (FIMI) [21]. The smallest formula AC($150k$) has a size of 1,486 terms with 64 variables, and the largest is AC($30k$) with 320,657 terms and 442 variables. BMS(500) has a size of 17,143 terms with 3340 variables, and BMS(30) has 2,314,875 terms with 3340 variables. Figure 5 shows the runtimes for the AC and BMS instances. The BMS instances show impressively, how good the reduction to SAT works.

The experiments described up to now used instances that consist of equivalent formulae. To produce non-equivalent instances, we randomly delete variables and terms in the above formulae. If we delete few variables or terms, we obtain few conflict assignments. We compare some experiments on 236 non-equivalent instances with thresholds of 60 and 360 seconds, where mf(¬UP) denotes the mf-strategy without UP (see Table 1). The reduction to SAT solves all instances within a time limit of 360 seconds, whereas our best implementation only solves 223 of 236 instances with this time. Furthermore, the experiments show that unit propagation (UP) helps to solve more non-equivalence instances, since without unit propagation less instances are solved. The runtimes do not depend on the classes of test data introduced above.

seconds	FKA			FKB						reduction to SAT
	mf(¬UP)	mf	HHM	mf(¬UP)	mf	MOMs	BOHM	CRH	HHM	
60	194	201	166	162	181	182	148	143	161	221
360	209	223	196	201	213	216	186	188	189	236

Table 1. Non-equivalent instances (of 236) solved within 60 and 360 seconds

Finally, we show that in order to solve MONET using reduction to SAT with a SAT-solver, it is much better to use the reduction function f (see Section 4) than the usual Tseitin-translation [25]. Table 2 shows that using the Tseitin-translation the runtimes are worse than using the FK-algorithms.

reduction	M(k)		TH(k)			SDTH(k)	
	22	24	250	500	700	250	400
using f	0.1	0.2	0.1	0.2	0.3	0.3	0.7
using Tseitin-translation	94	453	44	854	3604	539	4974
max. of FK-algorithms	1.9	5.1	8	136	563	20.5	211
	FKB(HHM)		FKA(HHM)[3]				

Table 2. Comparison of runtimes of different reductions to SAT in seconds

6 Conclusion

The main finding is that a good reduction function and a SAT-solver provides a more effective way for MONET than any current implementation of the FK-algorithms. It is a little surprising that it does not help to use the Tseitin translation only. Essentially, our reduction solves one direction of the equivalence test, and the SAT-solver solves the other direction. Eventually, it is not that surprising that the SAT-solvers are better than the implementations of the FK-algorithms.

On the other hand, we could improve the old implementations of the FK-algorithms [9] by using better data structures and unit propagation. Among the strategies for finding a splitting variable, it seems that MOMs is a good choice. This is not that surprising because MOMs is similar to choosing the most frequent variable, and the latter is a straightforward strategy intended in the formulation of FK-algorithm A [7].

Our next steps will be to figure out which strategies of SAT-solvers are responsible for the fast solution of reduced MONET instances and to see whether they can be integrated into the FK-algorithms. For example, clause learning seems to be useless for the FK-algorithms. Does the SAT-solver use it however for solving MONET instances? Moreover, the clauses obtained from the reduction function are easy in the sense that they are not needed for the NP-hardness of SAT—otherwise MONET would be coNP-complete. Therefore, one can assume that the "full power" of SAT-solvers is not necessary in order to solve reduced MONET instances fast. Another question is whether it makes the equivalence test easier if one checks both implication directions separately. The combination of reduction and SAT-solver works this way, whereas the FK-algorithms recursively make equivalence tests on decreasing formulae.

Acknowledgements The authors thank Sebastian Kuhs for some implementations, and Markus Chimani and Stephan Kottler for helpful comments.

[3] Note that FKB(HHM) does not finish on SDTH(400).

References

1. J. Bailey, T. Manoukian, and K. Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *Proc. of the 3rd IEEE Intl. Conference on Data Mining (ICDM 2003)*, pages 485–488, 2003.
2. M. Buro and H. K. Büning. Report on a SAT competition, 1992.
3. M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM 5*, 5(7):394–397, 1962.
4. G. Dong and J. Li. Mining border descriptions of emerging patterns from dataset pairs. *Knowledge and Information Systems*, 8(2):178–202, 2005.
5. T. Eiter and G. Gottlob. Hypergraph transversal computation and related problems in logic and AI. In *Proc. JELIA 2002*, volume 2424 of *LNCS*, pages 549–564. Springer, 2002.
6. T. Eiter, G. Gottlob, and K. Makino. New results on monotone dualization and generating hypergraph transversals. *SIAM J. on Computing*, 32(2):514–537, 2003.
7. M. L. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, 1996.
8. N. Galesi and O. Kullmann. Polynomial time SAT decision, hypergraph transversals and the hermitian rank. In *Proc. SAT 2004*, 2004.
9. M. Hagen, P. Horatschek, and M. Mundhenk. Experimental comparison of the two Fredman-Khachiyan-algorithms. In *Proc. ALENEX*, pages 154–161, 2009.
10. D. J. Kavvadias and E. C. Stavropoulos. Checking monotone Boolean duality with limited nondeterminism. Technical Report TR2003/07/02, Univ. of Patras, July 2003.
11. D. J. Kavvadias and E. C. Stavropoulos. An efficient algorithm for the transversal hypergraph generation. *J. of Graph Algorithms and Applications*, 9(2):239–264, 2005.
12. L. Khachiyan, E. Boros, K. M. Elbassioni, and V. Gurvich. An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals. *Discrete Applied Mathematics*, 154(16):2350–2372, 2006.
13. O. Kullmann. Investigating the behaviour of a sat solver on random formulas. Technical Report CSR 23-2002, University of Wales, October 2002.
14. D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.
15. L. Lin and Y. Jiang. The computation of hitting sets: Review and new algorithms. *Information Processing Letters*, 86(4):177–184, 2003.
16. K. Murakami. Personal communication, October 2010.
17. C. H. Papadimitriou. NP-completeness: A retrospective. In *Proc. ICALP*, pages 2–6, 1997.
18. D. Pretolani. Efficiency and stability of hypergraph sat algorithms. In *Proc. DIMACS Challenge II Workshop*, 1993.
19. W. Quine. Two theorems about truth functions. *Boletin de la Sociedad Matemática Mexicana*, 10:64–70, 1953.
20. S. Reith. On the complexity of some equivalence problems for propositional calculi. In *Proc. MFCS*, pages 632–641, 2003.
21. F. I. M. I. Repository. http://fimi.ua.ac.be/, February 2010.
22. U. C. I. Repository. http://archive.ics.uci.edu/ml/, February 2010.
23. H. Tamaki. Space-efficient enumeration of minimal transversals of a hypergraph. In *Proc. SIGAL*, pages 29–36, 2000.

24. V. I. Torvik and E. Triantaphyllou. Minimizing the average query complexity of learning monotone Boolean functions. *INFORMS Journal on Computing*, 14(2):144–174, 2002.
25. G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. Slisenko, editor, *Studies in constructive mathematics and mathematical logics, Part II*, pages 115–125. 1968.
26. T. Uno and K. Satoh. Detailed description of an algorithm for enumeration of maximal frequent sets with irredundant dualization. In *Proc. FIMI*, 2003.