Logarithmic Space Instances of Monotone Normal Form Equivalence Testing

Matthias Hagen *

Friedrich-Schiller-Universität Jena, Institut für Informatik, D–07737 Jena,
hagen@cs.uni-jena.de

Abstract. We examine the problem MONE—given a monotone DNF and a monotone CNF, decide whether they are equivalent. The exact complexity of MONE is a long standing open problem. On the one hand, MONE is probably not coNP-complete. But on the other hand, MONE is not known to be in P, although for numerous restrictions on the input formulas polynomial time algorithms exist. We improve the resource bound for several of these "easy" instances of MONE by showing that actually logarithmic space suffices for their decision. Among these instances are MONE-instances with a DNF that is regular, aligned, 2-monotonic, or that contains only very large monomials.

1 Introduction

Two Boolean formulas are *equivalent* if they have the same truth table. *Monotone formulas* are Boolean formulas with $\wedge$ and $\vee$ as only connectives. No negation signs are allowed. We consider the problem MONE (MOnotone Normal form Equivalence) of deciding equivalence of a monotone DNF and a monotone CNF. A monotone formula is in *DNF (disjunctive normal form)* if it is a disjunction of conjunctions of variables. Analogously, a monotone formula is in *CNF (conjunctive normal form)* if it is a conjunction of disjunctions of variables. A conjunction (disjunction) of variables is also called *monomial (clause)*. Monomials and clauses are also called *terms*. A monotone normal form ϱ is *irredundant* if there are no two terms in ϱ such that one is contained in the other. It is well-known that the irredundant DNF and CNF of a monotone formula are unique [Qui53]. And since monomials in φ (clauses in ψ) that contain other monomials (clauses) do not increase the complexity of MONE, we only concentrate on irredundant formulas as input. The problem MONE is defined as follows.

MONE:	*instance:*	irredundant, monotone formulas φ in DNF and ψ in CNF with variable set V
	question:	are φ and ψ equivalent?

The size n of a MONE-instance (φ, ψ) is the number of variable occurrences in φ and ψ. Besides their representation as formulas we often view φ and ψ as families of subsets of V. Thereby, a term t is represented by the subset of the

* Supported by a Landesgraduiertenstipendium Thüringen.

variables that occur in t. Assignments can also be viewed as sets of variables, with variable x being set to true if and only if x appears in the set. In this way, terms can be interpreted as assignments. Note that we do not have to take into account the complexity of testing whether the two inputs are well-formed formulas (correct parenthesis structure etc.), monotone, in normal form, irredundant, and have the same variable set, since these properties can be easily checked in space logarithmic in n (see Appendix A to E).

The problem MONE is equivalent to the well-known problems DUAL—given two monotone DNFs, decide whether they are mutually dual—and TRANSHYP—given two hypergraphs, decide whether one is the transversal hypergraph of the other. The currently best known algorithms for these problems are quasi-polynomial or use $O(\log^2 n)$ nondeterministic bits [FK96, EGM03, KS03]. Thus, on the one hand, MONE is probably not coNP-complete, but on the other hand a polynomial time algorithm is not yet known. Such a polynomial time algorithm solving MONE would also yield an output-polynomial algorithm for the generation of ψ out of φ [BI95]. Both variants of MONE—the generational as well as the decisional—have many applications in such different fields like artificial intelligence and logic [EG02, EG95], computational biology [Dam06], databases [MR92], data mining and machine learning [GKMT97], mobile communication systems [SS98], and distributed systems [GB85].

In this paper, we focus on known "easy" instances of MONE. These are restrictions of the input formulas for which polynomial time algorithms deciding the corresponding restricted versions of MONE are known. Such easy instances are an important branch of research concerning MONE, since they reveal information about the really hard parts of the problem. But how easy are these easy instances? The only known logspace instance of MONE is MONE where φ consists of monomials of constant size [GHM05]. We augment this first result by giving logarithmic space algorithms for several other easy instances of MONE improving the known polynomial time bounds.

We only examine restrictions of the DNF φ, since they can be easily transformed to be restrictions of the CNF ψ by switching $\vee$ to $\wedge$ and vice versa. The discussed restrictions can be divided into two groups. The first group comprises restrictions on the size of φ in the broader sense, such as the number of monomials or the size of the monomials. The second group contains rather structural restrictions of the DNF φ, such as being regular, aligned or 2-monotonic.

The paper is organized as follows. In Section 2 we examine restrictions on the size of the DNF φ. MONE-instances with a DNF φ that contains only a constant number of monomials or that contains only monomials of large enough size are known to be polynomial time decidable [EG91, EG95]. We show that already logarithmic space is enough. Section 3 is dedicated to the more structural restrictions. MONE-instances with a DNF φ that is regular [BS87], aligned [Bor94] or 2-monotonic [BHIK97] are known to be polynomial time decidable. We improve the resource bound by showing that logarithmic space suffices. Some concluding remarks follow in Section 4.

2 Restrictions on the Size of the DNF φ

In this section we examine restrictions of the DNF φ of a MONE-instance (φ, ψ) that concern the size of φ in a broad sense. A first already examined limitation is to restrict the DNF φ to contain only monomials of size at most c for a constant c. The corresponding restricted version $\text{MONE}_{\text{cupp}}$ (MONE with a constant upper bound for the monomial size) is already known to be decidable in logarithmic space [GHM05]. We focus on two related size restrictions. The first is MONE_{cnm}, where the DNF φ contains only a constant number of monomials. The second is $\text{MONE}_{\text{clow}}$, where the DNF φ contains only monomials that have a size of at least $|V|-c$ for a constant c. Here the name comes from the somehow "constant" lower bound for the monomial size. Both variants, MONE_{cnm} as well as $\text{MONE}_{\text{clow}}$, are known to be decidable in polynomial time [EG91, EG95]. We show them to be already decidable in logarithmic space.

2.1 The DNF Contains Only a Constant Number of Monomials

MONE_{cnm}:	*instance:*	irredundant, monotone formulas φ in DNF and ψ in CNF with variable set V, where φ contains only c monomials for a constant c
	question:	are φ and ψ equivalent?

Theorem 2.1. MONE_{cnm} *is decidable in logarithmic space.*

Proof. Let n be the size of the MONE-instance (φ, ψ). We assume that the variables are $x_1, x_2, \ldots, x_{|V|}$. We describe the work of an appropriate machine.

Whether (φ, ψ) really is a MONE_{cnm}-instance, can be checked in logarithmic space. The reason is that counting the monomials of φ suffices and since the number of monomials of φ is bounded by n, logarithmic space is enough. Let c be the constant bounding the number of monomials of φ.

Having tested the constant number of monomials property, the machine has to perform the equivalence test of φ and ψ. The machine systematically generates candidates for clauses of a CNF equivalent to φ (the first candidate consists of the first variables from each monomial; the second candidate consists of the second variable from the last monomial and the first variables from all the other monomials; [...]; the last candidate consists of the last variables from all monomials). There are at most $|V|^c$ possible candidates and the machine counts the already tested ones. This counter is logarithmic in n. By counting the already tested candidates the machine knows which is the next candidate because of the systematic generation. Since a candidate consists of at most c variables and since an index of one variable has size $\log |V|$, the machine could write down the indices of the variables forming the current candidate in space $\leq c \log |V|$ which is clearly logarithmic in n.

The machine tests for each such candidate whether it is a maxterm of φ. A clause is a maxterm of φ if it is contained in the irredundant, monotone CNF of φ. The test can be performed in logarithmic space (see Appendix F). For

candidates that are maxterms, the machine has to ensure that they are included in ψ, since otherwise φ and ψ cannot be equivalent. Hence, the machine looks for a clause of ψ, that contains the same variables as the current candidate. This is done via comparing the indices of the variables. A counter that counts the already tested clauses ensures that the machine knows the next clause of ψ. This counter is logarithmic in n. If all candidates that are maxterms could be verified to be contained in ψ, the machine tests for each clause of ψ (systematically one after the other) if it is a maxterm of φ. Again, this can be done in logarithmic space (see Appendix F). If a clause is found that is not a maxterm of φ, then $(\varphi, \psi) \notin \text{MONE}_{\text{cnm}}$ with this clause as a counter-example. Otherwise, the machine can conclude $(\varphi, \psi) \in \text{MONE}_{\text{cnm}}$. Altogether, logarithmic space is enough. □

2.2 The DNF Contains Only Very Large Monomials

$\text{MONE}_{\text{clow}}$:	*instance:*	irredundant, monotone formulas φ in DNF and ψ in CNF with variable set V, where φ contains only monomials of size at least $\lvert V\rvert - c$ for a constant c
	question:	are φ and ψ equivalent?

Before proving that $\text{MONE}_{\text{clow}}$ is decidable in logarithmic space we need some technical definitions and an important fact due to Eiter and Gottlob [EG95]. The complement $\overline{V'}$ of a subset V' of V and the complement $\overline{\mathcal{F}}$ of a family $\mathcal{F}$ of subsets of V are defined as $\overline{V'} = V \setminus V'$ and $\overline{\mathcal{F}} = \{\overline{F} : F \in \mathcal{F}\}$. For an irredundant, monotone DNF φ with the set M_φ of monomials we define the operator τ as $\tau(M_\varphi) = \{m \setminus \{x\} : m \in M_\varphi, x \in m\}$.

Proposition 2.2 ([EG95]). *Let φ be an irredundant, monotone DNF with the set M_φ of monomials. Every clause of the irredundant, monotone CNF ψ equivalent to φ is contained in $\overline{\tau(M_\varphi)}$.*

With Proposition 2.2 at hand, we can give an algorithm deciding $\text{MONE}_{\text{clow}}$ in logarithmic space.

Theorem 2.3. $\text{MONE}_{\text{clow}}$ *is decidable in logarithmic space.*

Proof. Let n be the size of the $\text{MONE}_{\text{clow}}$-instance (φ, ψ) and M_φ the set of monomials of φ. Whether (φ, ψ) is a $\text{MONE}_{\text{clow}}$-instance can be tested in logarithmic space. Counting the variables in each monomial of φ is enough and the counter clearly stays logarithmic in n. Let the lower bound for the monomial size be $|V| - c$ for constant c.

It remains to check the equivalence of φ and ψ. From Proposition 2.2 it follows that the only candidates for clauses of a CNF equivalent to φ are contained in $\overline{\tau(M_\varphi)}$. The machine performs a candidate generation and check procedure very analogous to the one from the proof of Theorem 2.1. Each candidate arises from a monomial and includes all variables that are not included in the monomial plus one variable from the monomial. Hence, the candidate size is bounded by $c + 1$. For such a candidate the machine can check in logarithmic space whether it is a

maxterm (see Appendix F). If a candidate is a mexterm, the machine searches it in ψ like the machine from the proof of Theorem 2.1 does. To know the current candidate the machine may write it down because of the constant size. To know the next candidate the machine systematically generates them and counts the number of already generated candidates. It starts by generating all candidates from the first monomial m_1 of φ. The first candidate is the set of all variables not contained in m_1 and the first variable from m_1. The second candidate is the set of all variables not contained in m_1 and the second variable from m_1, etc. After finishing the generation of all candidates from the first monomial, the machine generates all candidates from the second monomial in the same way. After that, all candidates from the third monomial, etc. Altogether, there are at most $(c+1) \cdot |M_\varphi|$ many candidates. Hence, the counter stays logarithmic in n. If a candidat that is a maxterm is not found in ψ, the machine rejects. After finishing the candidate generation, the machine has to test all clauses of ψ whether they are maxterms of φ like the machine in the proof of Theorem 2.1 does. As we have seen, logarithmic space suffices to decide $\text{MONE}_{\text{clow}}$. □

3 Structural Restrictions on the DNF φ

Having examined size restrictions, we now adress more structural restrictions. We focus on MONE-instances with a DNF φ that is regular, aligned, or 2-monotonic. All three instances allow for polynomial time algorithms [BS87, Bor94, BHIK97]. We improve the resource bounds by giving logarithmic space algorithms.

3.1 The DNF is Regular

Definition 3.1 (regular). *A formula ϱ with the set $V = \{x_1, \ldots, x_{|V|}\}$ of variables is regular, if for every pair of variable indices $i < j$ and every assignment $\mathcal{A}$ with $x_i \notin \mathcal{A}$ and $x_j \in \mathcal{A}$ it holds that $\mathcal{A}(\varrho) \leq \mathcal{A}'(\varrho)$, where $\mathcal{A}' = (\mathcal{A} \setminus \{x_j\}) \cup \{x_i\}$.*

As an example consider the regular DNF

$$\varphi = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4 \wedge x_5) \vee (x_2 \wedge x_3 \wedge x_4).$$

We examine the following special case of MONE.

MONE_{reg}:	*instance:*	irredundant, monotone formulas φ in DNF and ψ in CNF with variable set V, where φ is regular
	question:	are φ and ψ equivalent?

It is known that MONE_{reg} is decidable in polynomial time [BS87]. We show that already logarithmic space suffices.

Theorem 3.2. MONE_{reg} *is decidable in logarithmic space.*

Proof. Let n be the size of the MONE-instance (φ, ψ) and $V = \{x_1, \ldots, x_{|V|}\}$ be the set of variables. Regularity testing of φ can be managed in logarithmic space (see Appendix G).

As for the equivalence test, we slightly adapt the `Procedure RSC` of Bertolazzi and Sassano [BS87]. `RSC` computes all the clauses of the irredundant, monotone CNF of a given regular, irredundant, monotone DNF φ. Bertolazzi and Sassano have found the following coherence between a regular DNF and its maxterms. For each monomial m of φ and every variable $x_j \in m$ whose index is larger than l the set $F_j(m) \cup \{x_j\}$ is a maxterm of φ. Thereby, l is the smallest index of a variable contained in monomial m of φ but not in the lexicographic predecessor of m and $F_j(m) = \{x_k \notin m : k < j\}$. The lexicographic ordering means that the monomials are ordered lexicographically by their characteristic vectors. The characteristic vector of monomial m is $|V|$-dimensional and contains a 1 at position k if $x_k \in m$; otherwise the entry is 0. Monomial m_i is lexicographic larger than monomial m_j, $m_i >_{lex} m_j$, if and only if the characteristic vector of m_i is larger than the characteristic vector of m_j.

The above property for the maxterms of φ is used in lines `05` and `06` of our equivalence test (listing below). It computes the maxterms of φ, one after the other, and checks whether they are contained in ψ. Afterwards we have to check whether each clause of the given CNF ψ really is a maxterm of φ.

`RSC` precomputes a lexicographic ordering of the monomials and an $|M|$-dimensional vector γ containing the smallest variable indices that distinguish lexicographic adjacent monomials. Our algorithm does not have enough space to store such precomputations. Instead, it processes the monomials in the ordering they are given and computes the index (in the given ordering) of the predecessor in a lexicographic ordering (function `pred` in the listing below) every time it is needed. Analogously, the smallest variable index that distinguishes the current monomial from its predecessor (function `least_diff` in the listing below) is computed every time it is needed.

---

input: MONE$_{\text{reg}}$-instance (φ, ψ) with the set M_φ of monomials and
the set C_ψ of clauses

```
01   for i = 1 to |M_φ| do
02      p = pred(m_i, M_φ)
03      l = least_diff(m_i, m_p)
04      for j = 1 to |V| do
05         if (x_j ∈ m_i) ∧ (j > l) then
06            if c = F_j(m_i) ∪ {x_j} ∉ C_ψ then reject
07         endif
08      endfor
09   endfor
10   for i = 1 to |C_ψ| do
11      if c_i is not a maxterm of φ then reject
12   endfor
13   accept
```

The listings of `pred` and `least_diff` may be found in Appendix J and Appendix K. Since they are correct, the correctness proof of the above algorithm

is straightforward. It follows from the the correctness of `Procedure RSC` [BS87], which is very similar to our algorithm.

We have to examine the space requirement of the algorithm. All three `for`-loops could manage counters that contain the number of the already tested monomials in the original ordering, the index of the current variable, or the number of already tested clauses to know which are the current monomial, variable or clause. Such counters stay logarithmic in n. Both, p and l, store indices that remain logarithmic in n.

The "$c \notin C_\psi$"-test in line `06` is answered by an oracle. Therefor, C_ψ is written on the oracle tape. Afterwards, each variable $x_1, \ldots, x_{j-1}$ (logspace counter till $j-1$) is tested whether it is contained in m_i (using a logspace counter for searching through m_i systematically). If the variable is not contained in m_i, then it belongs to $F_j(m_i)$ and is written on the oracle tape. Finally, x_j is written on the oracle tape and it is asked whether the clause $F_j(m_i) \cup \{x_j\}$ as a set is contained in C_ψ as a set of subsets of variables. The oracle machine is a logspace machine (see Appendix I) and since $\mathsf{L}^{\mathsf{L}} = \mathsf{L}$, the oracle does not increase the resource requirements.

The maxterm-test in line `11` is implemented as an oracle query as well. Therefor, c_i and φ are written on the oracle tape. The oracle is a logspace oracle (see Appendix F). Hence, it does not increase the space requirement.

Since `pred` (see Appendix J) and `least_diff` (see Appendix K) run in logarithmic space, our above algorithm decides MONE_{reg} in logarithmic space. □

3.2 The DNF is Aligned

A monomial m is a prime implicant of a monotone formula ϱ, if it is contained in the irredundant, monotone DNF of ϱ.

Definition 3.3 (aligned). *A monotone formula ϱ with the variable set $V = \{x_1, \ldots, x_{|V|}\}$ is aligned, if for all prime implicants m of ϱ and all variables $x_i \notin m$ with $i \leq max_m = \max\{j : x_j \in m\}$ the assignment $m' = (m \setminus \{x_{max_m}\}) \cup \{x_i\}$ also satisfies ϱ.*

Every regular formula is aligned (compare Proposition G.1 and the last definition). But the converse does not hold, as can be seen by the following example.

$$\varphi = (x_1) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4) \vee (x_2 \wedge x_5) \vee (x_3 \wedge x_4) \vee (x_3 \wedge x_5 \wedge x_6) \vee (x_4 \wedge x_5 \wedge x_6 \wedge x_7) \vee (x_5 \wedge x_6 \wedge x_7 \wedge x_8).$$

The DNF φ is aligned but it is not regular, since $\{x_5, x_6, x_7, x_8\} - \{x_7\} \cup \{x_4\}(\varphi) = 0$. Hence, aligned formulas are a generalization of regular ones. In this section, we consider the following special case of MONE.

MONE_{ali}:	*instance:*	irredundant, monotone formulas φ in DNF and ψ in CNF with variable set V, where φ is aligned
	question:	are φ and ψ equivalent?

It is known that MONE_{ali} is decidable in polynomial time [Bor94]. We show that already logarithmic space suffices.

Theorem 3.4. MONE_{ali} *is decidable in logarithmic space.*

Proof. Let n be the size of the MONE-instance (φ, ψ) with variable set $V = \{x_1, \ldots, x_{|V|}\}$. Testing whether φ is aligned can be managed in logarithmic space (see Appendix L). As for the equivalence test, we use an approach of Boros [Bor94] but modify the algorithm to show that it works in logarithmic space.

For an assignment $\mathcal{A}$ let $max_{\mathcal{A}}$ denote the largest variable index that is included in $\mathcal{A}$. An assignment $\mathcal{A}$ satisfying a monotone formula ϱ is called leftmost, if $\mathcal{A} \setminus \{x_{max_{\mathcal{A}}}\}(\varrho) = 0$. Boros has shown that the irredundant, monotone DNF of an aligned formula ϱ exactly comprises of all leftmost assignments of φ [Bor94]. Furthermore, he has shown that a special type of binary decision tree (BDT) representation of aligned formulas is polynomial space bounded in its size. He uses this BDT to give a polynomial time algorithm for MONE_{ali}. We will modify this algorithm to achieve a logarithmic space bound.

A binary decision tree (BDT) T is a directed binary tree. The nodes of the tree have either two or no outgoing edges. The nodes reachable from node v are the succesors of v and together with v they form the subtree $T(v)$. The nodes $w \neq v$ for which $v \in T(w)$ are the predecessors of v. There is only one node without predecessors, the root r. The nodes with two outgoing edges are the inner nodes of T. The nodes with no outgoing edges are the leaves of T. The leaves of T have labels 0 (false leaves) or 1 (true leaves) such that there is no inner node v for which $T(v)$ only contains leaves with the same label. Let L_0 (L_1) be the set of all false (true) leaves of T. The set of predecessors of node v form a directed path $D(v) = \{v_1 = r, v_2, \ldots, v_{d(v)} = v\}$, where $d(v)$ denotes the depth of v (the distance from the root). Each inner node gets a variable as label. In our case, the label of node v is $x_{d(v)}$. Let u be an inner node with the outgoing edges (u, v) and (u, w). We say that v and w are the sons of u and u is their father. One son is the true-son $ts(u)$ and the other the false-son $fs(u)$. For the path $D(v)$ we define the sets $true(v) = \{x_{d(v_k)} : v_{k+1} = ts(v_k), k = 1, \ldots, d(v)\}$ and $false(v) = \{x_{d(v_k)} : v_{k+1} = fs(v_k), k = 1, \ldots, d(v)\}$ of nodes appearing as true-sons respectively false-sons. Each such BDT T represents a DNF of a Boolean formula ϱ and its dual in the following way, $\varrho = \bigvee_{v \in L_1} \bigwedge_{x_i \in true(v)} x_i \bigwedge_{x_i \in false(v)} \neg x_i$ and $\varrho^d = \bigvee_{v \in L_0} \bigwedge_{x_i \in false(v)} x_i \bigwedge_{x_i \in true(v)} \neg x_i$. For the dual ϱ^d of a formula ϱ it holds that $\mathcal{A}(\varrho) = \neg\mathcal{A}(\neg\varrho^d)$. Note that the irredundant CNF of an irredundant, monotone DNF φ can be produced by switching the roles of $\wedge$ and $\vee$ in the irredundant DNF of φ^d. This will be the key for our algorithm. Namely, Boros has proven that for each monotone formula ϱ there exists an unique BDT T_ϱ whose true-leaves (false-leaves) correspond one-to-one to the leftmost assignments of ϱ (ϱ^d) [Bor94]. We have $\varrho = \bigvee_{v \in L_1} \bigwedge_{x_i \in true(v)} x_i$ and $\varrho^d = \bigvee_{v \in L_0} \bigwedge_{x_i \in false(v)} x_i$. Remember that our input is an irredundant, monotone DNF φ that is aligned. Boros has shown that for aligned formulas the BDT is only polynomial in size and constructs an algorithm that computes the BDT and from it the CNF of φ [Bor94].

Our algorithm cannot compute the whole BDT since it would require polynomial space to store it. But remember that our input is the DNF φ and it is

aligned. Hence, from the results of Boros it follows that the monomials of φ are in a one-to-one relation with the true-leaves of the BDT for φ. We only have to search all false-leaves v and check whether $false(v)$ is contained in ψ since no other maxterms exist. But how do we search through all false-leaves? It can be easily proven that they are sons of nodes lying on the path to a true-leave (see Appendix M). Our algorithm will test each node in the BDT described by the monomials of φ as a potential father of a false-leave. Therefor, it checks each branch on the way described by a monomial whether the false-son is a false-leave and if so whether the corresponding maxterm is contained in ψ. In a second step our algorithm checks whether all clauses of ψ are maxterms of φ. We give the listing in the following. Let $m^j = m \cap \{x_1, \ldots, x_j\}$ for a monomial m and $\overline{s} = V \setminus s$ for a subset s of V.

---

input: MONE$_{\text{ali}}$-instance (φ, ψ) with the set M_φ of monomials and the set C_ψ of clauses

```
01   for i = 1 to |M_φ| do
02      for each variable x_j ∈ m_i do
03         if m_i^{j-1} ∪ {x_{j+1}} is not a subimplicant of φ then begin
04            if ¬∃c_k ∈ C_ψ : c_k ⊆ overline(m_i^{j-1}) ∩ {x_1,...,x_j} then reject
05         endif
06      endfor
07   endfor
08   for i = 1 to |C_ψ| do
09      if c_i is not a maxterm of φ then reject
10   endfor
11   accept
```

The correctness proof of the above algorithm is straightforward, since it just implements the techniques described above. A monomial m is subimplicant of φ if it is subset of a monomial contained in M_φ. In the lines 01 to 07 the algorithm tests for all false-branchs, that are not subimplicants of φ, on the path to any true-leave whether they are covered by any clause of ψ. Thereafter, it is tested whether each clause of ψ is a maxterm of φ.

We analyse the space requirement. All three `for`-loops can know the current monomial, variable or clause by using logarithmically space bounded counters. The `if`-tests in lines 03, 04 and 09 are answered by three oracles. In line 03 the algorithm writes $\overline{m_i^{j-1}} \cup \{x_{j+1}\}$ together with φ on the oracle tape. In line 04 the algorithm writes $\overline{m_i^{j-1}} \cap \{x_1, \ldots, x_j\}$ and ψ on the oracle tape. And in line 09 it writes c_i and φ on the oracle tape. All three oracles have logarithmically space bounded decision algorithms (see Appendices N, O and F). Since $\mathsf{L}^{\mathsf{L}} = \mathsf{L}$ the oracles do not increase the space requirement. Altogether, logarithmic space suffices to decide MONE$_{\text{ali}}$. □

3.3 The DNF is 2-monotonic

Another generalization of regular formulas are 2-monotonic formulas.

Definition 3.5 (2-monotonic). *A monotone formula ϱ is 2-monotonic if there exists a permutation π of the variables such that $\pi(\varrho)$ is regular.*

We consider the following version of MONE.

MONE_{2m}:	*instance:*	irredundant, monotone formulas φ in DNF and ψ in CNF with variable set V, where φ is 2-monotonic
	question:	are φ and ψ equivalent?

It is known that MONE_{2m} is decidable in polynomial time [BHIK97]. We show that already logarithmic space suffices. Therefor, we use the following result about 2-monotonic formulas.

Proposition 3.6 ([Win62]). *Let ϱ be a 2-monotonic formula with the set $V = \{x_1, \ldots, x_{|V|}\}$ of variables. For every $x_j \in V$ let position k of a $|V|$-dimensional vector $\alpha^{(j)}$ be*

$$\alpha_k^{(j)} = |\{m \text{ is a prime implicant of } \varrho : x_j \in m, |m| = k\}|.$$

Let $\alpha^{(j_1)} \geq_{lex} \alpha^{(j_2)} \geq_{lex} \ldots \geq_{lex} \alpha^{(j_{|V|})}$, where $\geq_{lex}$ denotes the lexicographic order between $|V|$-dimensional vectors, and let π be a permutation of variables such that $\pi(x_{j_i}) = x_i$ for all i. Then $\pi(\varrho)$ is regular.

We refer to permutation π from Proposition 3.6 as the w-permutation.

Theorem 3.7. MONE_{2m} *is decidable in logarithmic space.*

Proof. Let n be the size of the MONE-instance (φ, ψ) with variable set $V = \{x_1, \ldots, x_{|V|}\}$.

Note that in order to test whether the DNF φ of a MONE-instance is 2-monotonic, we could test whether $\pi(\varphi)$ is regular with the w-permutation π from Proposition 3.6. Since $\pi(\varphi)$ can be written on an oracle tape using logarithmic space only (see Appendix P), we can test 2-monotonicity in logarithmic space using the logarithmic space regularity test (see Appendix G) as an oracle.

As for the equivalence test, we again use the algorithm writing $\pi(\varphi)$ on the oracle tape and it is obvoius that a slight adaption could also write $\pi(\psi)$ on the oracle tape. Then the logarithmic space algorithm for MONE_{reg} is invoked as oracle. Since $\mathsf{L}^{\mathsf{L}} = \mathsf{L}$, the oracles do not increase the resource requirements.

Altogether, this is a logarithmic space algorithm deciding MONE_{2m}. □

4 Concluding Remarks

Easy instances of MONE are restrictions of the DNF that allow for a polynomial time solution of the corresponding restricted version of MONE. Many such instances are known but how easy are they? Nothing is known about hardness of the easy instances for classes like NL or P. But there is already an easy instance of MONE that can be decided using logarithmic space only. It restricts the DNF to contain only monomials of constant size [GHM05].

Our goal was to find more such logarithmic space instances of MONE improving the already known polynomial time bounds. Among our results are instances of MONE, where the DNF is allowed to contain only a constant number of monomials or MONE, where each monomial of the DNF is only allowed not to contain a constant number of variables.

As for the more structural restrictions, we have shown that MONE with a 2-monotonic or aligend DNF is decidable in logarithmic space improving the already known polynomial time bounds. This implies that also MONE with a regular DNF is solvable in logarithmic space.

Nevertheless, it would be very interesting to find logarithmic space algorithms for other easy instances of MONE or to prove hardness results for easy instances. Such hardness results for special cases may be useful when proving hardness of MONE. No hardness results for MONE are known yet. They should be adressed in future research.

References

[BHIK97] Endre Boros, Peter L. Hammer, Toshihide Ibaraki, and Kazuhiko Kawakami. Polynomial-time recognition of 2-monotonic positive Boolean functions given by an oracle. *SIAM Journal on Computing*, 26(1):93–109, 1997.

[BI95] Jan C. Bioch and Toshihide Ibaraki. Complexity of identification and dualization of positive Boolean functions. *Information and Computation*, 123(1):50–63, 1995.

[Bor94] Endre Boros. Dualization of aligned Boolean functions. Technical Report RRR 9-94, RUTCOR, Rutgers University, March 1994.

[BS87] Paola Bertolazzi and Antonio Sassano. An $O(mn)$ algorithm for regular set-covering problems. *Theoretical Computer Science*, 54:237–247, 1987.

[Dam06] Peter Damaschke. Parameterized enumeration, transversals, and imperfect phylogeny reconstruction. *Theoretical Computer Science*, 351(3):337–350, 2006.

[EG91] Thomas Eiter and Georg Gottlob. Identifying the minimal transversals of a hypergraph and related problems. Technical Report CD-TR 91/16, TU Wien, January 1991.

[EG95] Thomas Eiter and Georg Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278–1304, 1995.

[EG02] Thomas Eiter and Georg Gottlob. Hypergraph transversal computation and related problems in logic and AI. In Sergio Flesca, Sergio Greco, Nicola Leone, and Giovambattista Ianni, editors, *Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, September, 23-26, Proceedings*, volume 2424 of *Lecture Notes in Computer Science*, pages 549–564. Springer, 2002.

[EGM03] Thomas Eiter, Georg Gottlob, and Kazuhisa Makino. New results on monotone dualization and generating hypergraph transversals. *SIAM Journal on Computing*, 32(2):514–537, 2003.

[FK96] Michael L. Fredman and Leonid Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21(3):618–628, 1996.

[GB85] Hector Garcia-Molina and Daniel Barbará. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, 1985.

[GHM05] Judy Goldsmith, Matthias Hagen, and Martin Mundhenk. Complexity of DNF and isomorphism of monotone formulas. In Joanna Jedrzejowicz and Andrzej Szepietowski, editors, *Mathematical Foundations of Computer Science 2005, 30th International Symposium, MFCS 2005, Gdansk, Poland, August 29 - September 2, 2005, Proceedings*, volume 3618 of *Lecture Notes in Computer Science*, pages 410–421. Springer, 2005.

[GKMT97] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila, and Hannu Toivonen. Data mining, hypergraph transversals, and machine learning. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 209–216. ACM Press, 1997.

[KS03] Dimitris J. Kavvadias and Elias C. Stavropoulos. Monotone Boolean dualization is in $\mathsf{coNP}[\log^2 n]$. *Information Processing Letters*, 85(1):1–6, 2003.

[MR92] Heikki Mannila and Kari-Jouko Räihä. On the complexity of inferring functional dependencies. *Discrete Applied Mathematics*, 40(2):237–243, 1992.

[Mur71] Saburo Muroga. *Threshold Logic and Its Applications*. Wiley-Interscience, New York, 1971.

[Qui53] Willard van Orman Quine. Two theorems about truth functions. *Boletin de la Sociedad Matemática Mexicana*, 10:64–70, 1953.

[SS98] Saswati Sarkar and Kumar N. Sivarajan. Hypergraph models for cellular mobile communication systems. *IEEE Transactions on Vehicular Technology*, 47(2):460–471, 1998.

[Win62] Robert O. Winder. *Threshold Logic*. PhD thesis, Mathematics Department, Princeton University, 1962.

A Well-formed Test

We say that an input is a well-formed formula if and only if it can be interpreted as a Boolean formula. That means that only variable symbols, "$\wedge$", "$\vee$", "$\neg$", "(", and ")" are allowed symbols, that the parenthesis structure is correct, and that the sequence of variables and connectives does not include situations like "$x_1 \vee \vee x_2$" or "$(\vee$" or "$x_1 \wedge x_2 x_3 \vee x_4$" etc.

Lemma A.1. *Whether an input is a well-formed formula can be decided in logarithmic space.*

Proof. Let n be the length of the input. We describe the work of an appropriate machine. During a first scan of the input the machine tests whether only allowed symbols are involved. No additional space is needed.

A second scan of the input is used to examine the parenthesis structure. Therefor, a counter is used that is initially set to 0. The machine scans through the input searching parentheses. When a left parenthesis is found, the counter is incremented by 1. When a right parenthesis is found, the counter is decremented by 1. The parenthesis structure of the input is correct if and only if the counter always contains a value ≥ 0 and the counter is 0 when the input scan is completed. Since at most n parentheses may be contained in the input of length n, the space needed for the counter is bounded logarithmically.

In a last scan of the input the sequence of variables and connectives has to be tested as well. There are only few syntactic rules that have to be checked. After the occurrence of a variable only "$\vee$", "$\wedge$" or ")" are allowed. After a left parenthesis only a variable or a negation are allowed. After a right parenthesis only "$\wedge$" and "$\vee$" are allowed. After a "$\wedge$" or a "$\vee$" only "$\neg$" or a variable are allowed. And after a "$\neg$" only a variable or a left parenthesis are allowed. No additional space is needed while scanning the input and checking these syntactic rules.

All three scans of the input could also be tested in only one scan. Altogether, logarithmic space suffices. □

From now on, we assume that input formulas are well-formed since we could test it in advance using only logarithmic space.

B Monotony Test

Lemma B.1. *Whether a given Boolean formula is monotone can be decided without using any additional space.*

Proof. An appropriate machine scans through the input searching a negation sign. The formula is monotone if and only if no negation sign is found. One cycle through the input is enough and no additional space is needed. □

C Normal Form Test

A monotone formula is in normal form if and only if it is a conjunction of disjunctions of variables (CNF) or a disjunction of conjunctions of variables (DNF).

Lemma C.1. *Whether a given monotone formula is in normal form can be decided without using any additional space.*

Proof. An appropriate machine has to perform two tests. First, it checks whether the parenthesis structure of the input formula is "()() ... ()". Secondly, the correct usage of the connectives is tested. The machine checks whether the connectives within a pair "()" are "$\wedge$" (respectively "$\vee$" for a CNF) and whether between two pairs of parentheses "()()" the connectives are "$\vee$" (respectively "$\wedge$" for a CNF). For both tests the machine does not need any additional space. □

D Irredundancy Test

Lemma D.1. *Whether a given monotone formula in normal form is irredundant can be decided in logarithmic space.*

Proof. We give an algorithm that solves the problem and uses logarithmic space. A term is a monomial respectively a clause of the normal form.

---

input: monotone formula ϱ in normal form with variable set V
output: Yes, if ϱ is irredundant, and No, otherwise

```
01   b = 1;
02   for all terms t_i of ϱ do
03      for all other terms t_j of ϱ do
04         count = 0;
05         for all variables x ∈ t_i do
06            if x ∈ t_j then count = count + 1;
07         endfor
08         if count = |t_i| then b = 0;
09      endfor
10   endfor
11   if b = 1 then output Yes;
12   else output No;
```

The correctness proof of the above algorithm is straightforward. We have to analyse the space requirement.

The size n of ϱ is the number of variable occurences. We assume that ϱ consists of l terms. The `for`-loops in line `02` and `03` have to know the index of the current terms. Therefore, we can assume that the algorithm numbers the terms of ϱ by $1, 2, 3 \ldots, l$ (the first term of ϱ gets number 1, the second number 2, etc.) and the `for`-loops test the terms by increasing indices. Hence, the `for`-loops

can count the already tested terms and know the current term. These are two logarithmically in n space bounded counters. The `for`-loop in line `04` has to know the index of the current variable. We analogously assume that the variables have indices $1, 2, 3 \ldots, |V|$. Hence, the index is logarithmically in n space bounded. Two other logarithmically in n space bounded counters are needed for *count* and to evaluate the size $|t_i|$ needed in line `08`. The variable b only needs constant space.

As can be seen from the algorithm, the terms of ϱ need not to be copied to be compared. Hence, logarithmic space is enough. □

E Variable Set Test

Lemma E.1. *Whether a given irredundant, monotone DNF φ and a given irredundant, monotone CNF ψ have the same variable set can be decided in logarithmic space.*

Proof. Let the input size n be the number of variable occurences in φ and ψ. We assume that the variables have indices $1, 2, \ldots, |V|$.

An appropriate machine cycles through the DNF φ from the beginning to the end. For each variable occurence it tests whether this variable is present in ψ via looking for the index of the variable in ψ from the beginning to the end. When a variable is found in φ that is not present in ψ, the machine rejects. Otherwise it exchanges the roles and cycles through ψ. For each variable occurence of ψ the machine searches φ for this index. If also no variable of ψ is not present in φ, the variable sets are the same. If on the other hand the machine finds a variable in one formula that is not present in the other, the variable sets cannot be the same.

The machine needs two counters and a possibility to store a variable index. The first counter contains the number of variable occurences that were already checked in the first formula. Another storage is needed for the index of the current sought-after variable. The second counter contains the number of variable occurences in the second formula that were already tried to match the current index. Both counters are logarithmically in n space bounded. The variable index that has to be stored is also logarithmically in $|V|$ space bounded. Hence, the machine only needs logarithmic space. □

F Maxterm Test

A clause c is a *maxterm* of a monotone formula ϱ if it is contained in an irredundant, monotone CNF that is equivalent to ϱ.

Lemma F.1. *Let φ be an irredundant, monotone DNF and c be an irredundant, monotone clause. It can be decided in logarithmic space whether C is a maxterm of φ.*

Proof. Let $M_\varphi = \{m_1, \ldots, m_{|M_\varphi|}\}$ be the set of monomials of φ and $V = \{x_1, \ldots, x_{|V|}\}$ be the set of variables in φ and c. Let the input size n be the number of variable occurences in φ and c.

It has to be checked whether c has a non-empty intersection with every monomial of φ (lines `01` to `07` of the listing below). Thereafter, it has to be tested whether $c \setminus \{x\}$ has a non-empty intersection with all monomials for every variable $x \in c$ (lines `08` to `18`). If one such variable can be found, then c is not a maxterm. An appropriate algorithm is given in the following.

---

input: irredundant, monotone DNF φ and an irredundant, monotone clause c

output: Yes, if c is a maxterm of φ, and No, otherwise

```
01   for i = 1 to |M_φ| do
02      count = 0;
03      for j = 1 to |m_i| do
04         if x_j ∈ c then count = count + 1;
05      endfor
06      if count = 0 then output No and stop;
07   endfor
08   for i = 1 to |c| do;
09      hit = |M_φ|;
10      for j = 1 to |M_φ| do
11         count = 0;
12         for k = 1 to |m_j| do
13            if (x_k ∈ c) ∧ (k ≠ i) then count = count + 1;
14         endfor
15         if count > 0 then hit = hit − 1;
16      endfor
17      if hit = 0 then output No and stop;
18   endfor
19   output Yes;
```

The correctness proof of the above algorithm is straightforward. We have to analyse the space requirement.

To know the current monomial, the `for`-loops in line `01` and line `10` could manage counters that give the number of already checked monomials. These counters have to count till $|M_\varphi|$. Hence, they are logarithmically bounded in n. An analogous argumentation holds for the `for`-loops in line `03` and line `12`. To know the current variable, they manage counters that count till $|m|$ for the current monomial m, which is clearly logarithmic in n. And again, the `for`-loop in line `08` is handled analogously. Here, the counter has to count till $|c|$, which is also logarithmic in n.

It remains to check the variables *count* and *hit*. The maximal value of *count* is the size of a largest monomial of φ. Hence, *count* remains logarithmic in n. The maximal value of *hit* is $|M_\varphi|$. Hence, it is also logarithmic in n. Altogether, logarithmic space suffices to run the described algorithm. □

G Testing Regularity

A monotone formula ϱ is regular if for every pair of variable indices $i < j$ and every assignment $\mathcal{A}$ with $x_i \notin \mathcal{A}$ and $x_j \in \mathcal{A}$ it holds that $\mathcal{A}(\varrho) \leq \mathcal{A}'(\varrho)$, where $\mathcal{A}' = (\mathcal{A} \setminus \{x_j\}) \cup \{x_i\}$.

A monomial m is a prime implicant of ϱ if it is contained in the irredundant, monotone DNF of ϱ.

Muroga showed the following.

Proposition G.1 ([Mur71]). *A monotone formula ϱ with variable set $V = \{x_1, \ldots, x_{|V|}\}$ is regular if and only if for all prime implicants m of ϱ and all variables $x_i \notin m$ and $x_{i+1} \in m$ the assignment $m' = (m \setminus \{x_{i+1}\}) \cup \{x_i\}$ satisfies ϱ.*

We use Proposition G.1 to design a logspace regularity test for irredundant, monotone DNFs.

Lemma G.2. *The regularity test for an irredundant, monotone DNF φ can be implemented to run in logarithmic space.*

Proof. Note that the irredundant, monotone DNF φ already consists of all prime implicants of φ. Let $V = \{x_1, \ldots, x_{|V|}\}$ be the variable set and $M_\varphi = \{m_1, \ldots, m_{|M_\varphi|}\}$ be the set of monomials (prime implicants) of φ. A regularity tester for irredundant, monotone DNFs could process the following algorithm.

---

input: irredundant, monotone DNF φ
output: Yes, if φ is regular, and No, otherwise

```
01   for i = 1 to |M_φ| do
02      for j = 1 to |V| do
03         if x_j ∉ m_i and x_{j+1} ∈ m_i then
04            if (m_i \ {x_{j+1}}) ∪ {x_j}(φ) = 0 then output No
05         endif
06      endfor
07   endfor
08   output Yes
```

The correctness proof of the above algorithm is straightforward. The algorithm implements the test of the property stated in Proposition G.1. We have to analyse the space requirement.

Both `for`-loops could manage counters that contain the number of the currently tested monomial and the index of the current variable to know which are the current monomial and variable. Such counters stay logarithmic in the input size.

The two containedness tests of the `if` in line `03` require only one additional counter to store the index $j + 1$. This counter is logarithmic in the input size. The containedness tests just have to search the indices j and $j + 1$ in m_i. They need no additional storage other than the two logarithmic counters of the `for`-loops and the logarithmic index $j + 1$ to know the current monomial and the

current variables. The `if`-test in line `04` is answered by an oracle. The assignment $(m_i \setminus \{x_{j+1}\}) \cup \{x_j\}$ as a set together with φ is written on an oracle tape and the oracle answers "No" if and only if $(m_i \setminus \{x_{j+1}\}) \cup \{x_j\}(\varphi) = 0$. This oracle is a logspace oracle (see Appendix H) and since $\mathsf{L}^{\mathsf{L}} = \mathsf{L}$ the oracle does not increase the resource requirements.

Hence, the whole regularity test runs in logarithmic space. □

H Evaluate a DNF

Lemma H.1. *Given an irredundant, monotone DNF φ and an assignment $\mathcal{A}$, it can be decided in logarithmic space whether $\mathcal{A}(\varphi) = 1$.*

Proof. Let $V = \{x_1, \ldots, x_{|V|}\}$ be the set of variables and $M_\varphi = \{m_1, \ldots, m_{|M_\varphi|}\}$ be the set of monomials of φ. It has to be tested whether $\mathcal{A}$ contains at least one monomial of φ. An appropriate algorithm could look like the following.

---

input: irredundant, monotone DNF φ and an assignment $\mathcal{A}$
output: Yes, if $\mathcal{A}(\varphi) = 1$, and No, otherwise

```
01   for i = 1 to |M_φ| do
02      eval := 1;
03      for j = 1 to |m_i| do
04         if x_j ∉ A then eval := 0;
05      endfor
06      if eval = 1 then output Yes and stop;
07   endfor
08   output No
```

The correctness proof of the above algorithm is straightforward. We have to analyse the space requirement.

Both `for`-loops could manage counters that contain the number of the currently tested monomial and the index of the current variable to know which are the current monomial and variable. Such counters stay logarithmic in the input size. The `if`-test in line `04` does not need any additional space, since it is just a search in $\mathcal{A}$ for the index of the variable. The *eval*-variable only needs constant space.

Hence, the algorithm runs in logarithmic space. □

I Test if a set is contained in a set of sets

Lemma I.1. *Given a set S of subsets of V and a subset $t \subseteq V$, it can be decided in logarithmic space whether t is contained in S.*

Proof. We give an algorithm with the desired properties.

---

input: set $S = \{s_1, \ldots, s_{|S|}\}$ of subsets of V and subset $t \subseteq V$
output: Yes, if $t \in S$, and No, otherwise

```
01   for i = 1 to |S| do
02      if |s_i| = |t| then
03         isin := 0;
04         for all x ∈ t do
05            if x ∈ s_i then isin := isin + 1;
06         endfor
07         if isin = |S| then output Yes and stop;
08      endif
09   endfor
10   output No;
```

The correctness proof of the above algorithm is straightforward.

The `for`-loops need two logarithmic counters to count till $|S|$ and $|t|$. The `if`-test in line `02` can be implemented using two other logarithmic counters. The $isin$-variable needs logarithmic space as well, since the largest value stored is $|t|$.

Altogether, this gives logarithmic space. □

J Function `pred`

Lemma J.1. *The function* `pred`*, used in the proof of Theorem 3.2, can be implemented to run in logarithmic space.*

Proof. The function `pred` should return the index p in the given ordering of monomials of the predecessor of the current monomial in a lexicographic ordering. We give an algorithm with the desired properties.

Function `pred`(m_i, M_φ)**:**

input: monomial m_i of an irredundant, monotone DNF φ with the set M_φ of monomials and variable set V
output: index of the lexicographic predecessor monomial of m_i in M_φ

```
01   p = i
02   for j = 1 to |M_φ| do
03      if leq_lex(m_i, m_j) = j then begin
04         if leq_lex(m_j, m_p) = p then p = j
05      endif
06   endfor
07   return p
```

The `for`-loop can be managed via a logarithmic counter and since p contains monomial indices, it is also logarithmic. Hence, it is obvious that `pred` works correctly and in logarithmic space if `leq_lex` does.

The function `leq_lex`(m_i, m_j) is intended to return the index of the lexicographic smaller of the two monomials. We give an appropriate algorithm in the following.

Function `leq_lex`(m_i, m_j)**:**

input: two monomials m_i, m_j of an irredundant, monotone DNF φ with
the set M_φ of monomials and variable set V
output: index of the lexicographically smaller monomial

```
01   for k = 1 to |V| do
02      if (x_k ∉ m_j) ∧ (x_k ∈ m_i) then return j
03      elseif (x_k ∈ m_j) ∧ (x_k ∉ m_i) then return i
04   endfor
05   return i
```

The correctness proof of the above algorithm is straightforward. Hence, `pred` is correct.

The `for`-loop can be managed via a logarithmic counter. The "$x \in m$"-tests can also be managed using a logarithmic counter. Hence, `leq_lex` runs in logarithmic space and so does `pred`. □

K Function `least_diff`

Lemma K.1. *The function* `least_diff`*, used in the proof of Theorem 3.2, can be implemented to run in logarithmic space.*

Proof. Given two monomials, the function `least_diff` should return the smallest index l of a variable that is contained in only one of the monomials. We give an algorithm with the desired properties.

Function `least_diff`(m_i, m_j)**:**

input: two monomials $m_i \geq_{lex} m_j$ of an irredundant, monotone DNF φ with
the set M_φ of monomials and variable set V
($m_i =_{lex} m_j$ only when m_i is the lexicographically first monomial of φ)
output: smallest index k, such that $x_k \in m_i$ and $x_k \notin m_j$

```
01   if i = j then return 0
02   for k = 1 to |V| do
03      if (x_k ∉ m_j) ∧ (x_k ∈ m_i) then return k
04   endfor
end
```

The correctness proof of the above algorithm is straightforward. As for the space requirement, the `for`-loop requires a logarithmic counter. The "$x \in m$"-tests can also be managed using a logarithmic counter. □

L Aligned Test

A monotone formula ϱ with variable set $V = \{x_1, \ldots, x_{|V|}\}$ is aligned if for all prime implicants m of ϱ and all variables $x_i \notin m$ with $i \leq max_m = \max\{j : x_j \in m\}$ the assignment $m' = (m \setminus \{x_{max_m}\}) \cup \{x_i\}$ satisfies ϱ.

Lemma L.1. *Whether an irredundant, monotone DNF φ is aligned can be decided in logarithmic space.*

Proof. We slightly adapt the algorithm of the regularity test (see Appendix G), since we do not have to test all what we have tested there (compare the definition of an aligned formula with Proposition G.1).

Note that the irredundant, monotone DNF φ, that is input for the test, already consists of all prime implicants of φ. Let $V = \{x_1, \ldots, x_{|V|}\}$ be the variable set and $M_\varphi = \{m_1, \ldots, m_{|M_\varphi|}\}$ be the set of monomials (prime implicants) of φ. Let furthermore max_m denote the largest variable index appearing in monomial m. An algorithm, testing whether φ is aligned, could look like the following.

---

input: irredundant, monotone DNF φ
output: Yes, if φ is aligned, and No, otherwise

```
01   for i = 1 to |M_φ| do
02      for j = 1 to max_{m_i} do
03         if x_j ∉ m_i then
04            if (m_i − {x_{max_{m_i}}}) ∪ {x_j}(φ) = 0 then output No
05         endif
06      endfor
07   endfor
08   output Yes
```

The correctness proof of the above algorithm is straightforward, since the algorithm just tests the property given in the definition of aligned formulas. We have to analyse the space requirement.

Both `for`-loops could manage counters that contain the number of the currently tested monomial and the index of the current variable to know which are the current monomial and variable. Such counters stay logarithmic in the input size. The variable index max_{m_i} can be stored using logarithmic space, too.

The containedness test of the `if` in line `03` just has to search the index j in m_i which can be done with logarithmic space as described in the proof of Lemma G.2. Analogously to the regularity testing algorithm, the `if`-test in line `04` is answered by an oracle. The assignment $(m_i - \{x_{max_{m_i}}\}) \cup \{x_j\}$ as a set together with φ is written on the oracle tape and the oracle answers whether this assignment satisfies φ. The oracle machine using logarithmic space is described in Appendix H. Since $\mathsf{L}^{\mathsf{L}} = \mathsf{L}$ the usage of the oracle does not increase the space requirement.

Hence, the whole algorithm runs in logarithmic space.

M Where are the False-Leaves?

Lemma M.1. *There is no false-leave in a BDT T that is not the false-son of a father contained in $D(v)$ for a true-leave v.*

Proof. Assume that we could find a false-leave u that is the true-son of a node w. Then $true(u)$ is a leftmost assignment of a formula represented by T. But then u cannot be a false-leave. A contradiction. Hence, false-leaves are false-sons of their fathers.

Assume now that the father w of the false-leave u is not contained in any $D(v)$ for a true-leave v. Hence, the leaves in $T(w)$ all are false-leaves. Again, a contradiction. □

N Subimplicant Test

A monomial m is an implicant of a formula ϱ if $m(\varrho) = 1$. An implicant is prime if none of its subsets is an implicant. A monomial m is a subimplicant of ϱ if m is subset of a prime implicant of ϱ.

Lemma N.1. *Whether a subset s of the set $V = \{x_1, \ldots, x_{|V|}\}$ of variables of an irredundant, monotone DNF φ is a subimplicant of φ can be decided in logarithmic space.*

Proof. Note that the DNF φ contains all prime implicants of φ. Let the input size n be the number of variable occurences in φ and s. We give an algorithm with the desired properties.

---

input: irredundant, monotone DNF φ with the set $M_\varphi = \{m_1, \ldots, m_{|M_\varphi|}\}$ of monomials and a subset s of the set $V = \{x_1, \ldots, x_{|V|}\}$ of variables
output: Yes, if s is a subimplicant of φ, and No, otherwise

```
01   for i = 1 to |M_φ| do
02      test := 1;
03      for all x ∈ s do
04         if x ∉ m_i then test := 0;
05      endfor
06      if test = 1 then output Yes and stop;
07   endfor
08   output No;
```

The correctness proof of the above algorithm is straightforward. We have to analyse the space requirement.

Both `for`-loops can be managed using logarithmic counters to know the current monomial or variable. The *test*-variable needs constant space. Altogether, logarithmic space suffices. □

O Superclause Test

A clause c is a superclause of a formula ϱ if c contains a maxterm of ϱ.

Lemma O.1. *Whether a subset s of the set $V = \{x_1, \ldots, x_{|V|}\}$ of variables of an irredundant, monotone CNF ψ is a superclause of ψ can be decided in logarithmic space.*

Proof. Note that the CNF ψ contains all maxterms of ψ. Let the input size n be the number of variable occurences in ψ and s. We give an algorithm with the desired properties.

input: irredundant, monotone CNF ψ with the set $C_\psi = \{c_1, \ldots, c_{|C_\psi|}\}$ of clauses and a subset s of the set $V = \{x_1, \ldots, x_{|V|}\}$ of variables
output: Yes, if s is a superclause of ψ, and No, otherwise

```
01  for i = 1 to |C_ψ| do
02     test := 1;
03     for all x ∈ c_i do
04        if x ∉ s then test := 0;
05     endfor
06     if test = 1 then output Yes and stop;
07  endfor
08  output No;
```

The algorithm is very similar to the one given in Appendix N. Hence, an analogous argumentation gives the logarithmic space bound. □

P Winder-Permutation

Lemma P.1. *Let φ be an irredundant, monotone DNF with the variable set $V = \{x_1, \ldots, x_{|V|}\}$ and the set $M_\varphi = \{m_1, \ldots, m_{|M_\varphi|}\}$ of monomials. The w-permuted $\pi(\varphi)$ can be written on an oracle tape using logarithmic space only.*

Proof. We give an algorithm with the desired properties. Let $s_1 \circ s_2$ denote the concatenation of the two strings s_1, s_2 on the oracle tape.

```
01  π(φ) := "(";
02  for i = 1 to |M_φ| do
03     c_1 := 0;
04     if i = 1 then π(φ) := π(φ) ∘ "(";
05     else π(φ) := π(φ) ∘ " ∨ (";
06     for j = 1 to |V| do
07        if (x_j ∈ m_i) ∧ (c_1 ≠ 0) then π(φ) := π(φ) ∘ " ∧ ";
08        if x_j ∈ m_i then
09           c_1 := c_1 + 1;
10           max = get_max(φ);
11           c_2 = 1;
12           while max ≠ i do
13              max := get_next(φ, max);
14              c_2 := c_2 + 1;
15           endwhile
16           π(φ) := π(φ) ∘ "x_{c_2}";
17        endif
18     endfor
19     π(φ) := π(φ) ∘ ")";
20  endfor
21  π(φ) := π(φ) ∘ ")";
```

The algorithm writes the string $\pi(\varphi)$ on an oracle tape recomputing new variable indices each time they are needed. The algorithm does not store already computed indices, since that would need more than logarithmic space. For each variable occurence x_i in φ the algorithm counts where in the lexicographic ordering of the α-vectors the vector $\alpha^{(i)}$ appears. A variable with the corresponding index is written on the oracle tape instead of x_i. To derive the new index of x_i the algorithm computes the lexicographically last α-vector (`get_max` in line `10`). As long as $\alpha^{(i)}$ is not found the algorithm computes the next element in the ordering of the α-vectors (`get_next` in line `13`) and adds one to the counter c_2 that should contain the number of $\alpha^{(i)}$ in the lexicographic ordering of Proposition 3.6. When $\alpha^{(i)}$ is found, the counter c_2 contains the number of $\alpha^{(i)}$ in the ordering of Proposition 3.6. The formula $\pi(\varphi)$ is composed as a string on the oracle tape (lines `01`, `04`, `05`, `07`, `16`, `19`, and `21`).

The counters c_1 (largest value is the size of a largest monomial) and c_2 (largest value is $|V|$) stay logarithmic in n. Both `for`-loops can also be managed via logarithmically space bounded counters that contain the number of the current monomial or the index of the current variable. And last but not least, the variable max is logarithmically space bounded, since it only contains variable indices.

We have to analyse the listings of the functions `get_max` and `get_next` to fully describe the algorithm computing $\pi(\varphi)$. The function `get_max` should return the index i of the lexicographic largest of the $\alpha^{(i)}$.

---

input: irredundant, monotone DNF φ with the set $M_\varphi = \{m_1, \ldots, m_{|M_\varphi|}\}$
of monomials and the set $V = \{x_1, \ldots, x_{|V|}\}$ of variables
output: index max of the variable with the lexicographic largest α-vector

```
01    max := 1;
02    for i = 2 to |V| do
03       k := 0;
04       while k ≤ |V| do
05          k := k + 1
06          c3 := |{m ∈ Mφ : x_max ∈ m, |m| = k}|;
07          c4 := |{m ∈ Mφ : x_i ∈ m, |m| = k}|;
08          if c4 > c3 then
09             max := i;
10             k := |V| + 1;
11          elseif c4 < c3 then k := |V| + 1;
12       endwhile
13    endfor
14    return max
```

Each variable is a candidate for having the lexicographic largest α-vector. Hence, all variables are tested systematically by the above algorithm. In the `while`-loop (line `04`) the vector $\alpha^{(max)}$ which is so far the lexicographic largest vector and the vector $\alpha^{(i)}$ of the current variable are tested componentwise to decide which one is lexicographic larger. If it is $\alpha^{(i)}$, then i is the new maximum so far (line `09`). The correctness proof is straightforward.

As for the space requirement, both counters c_3 and c_4 remain logarithmic in n, since their largest value is $|M_\varphi|$. They can be computed by checking the monomials systematically whether they contain the tested variable. If so, another counter is used to get the size of the current monomial. This counter is compared to k. The largest value stored in variable k is $|V|+1$ which is logarithmic in n. Another logarithmically space bounded counter is used for the `for`-loop. The variable max contains variable indices. Hence, it is logarithmically space bounded. Altogether, the function `get_max` can be computed using logarithmic space.

The function `get_next` should return the index of the variable whose α-vector is the succesor in the lexicographic ordering of Proposition 3.6 of the current $\alpha^{(max)}$.

---

input: irredundant, monotone DNF φ with the set $M_\varphi = \{m_1, \ldots, m_{|M_\varphi|}\}$ of monomials and the set $V = \{x_1, \ldots, x_{|V|}\}$ of variables and a variable index max

output: index of the variable whose α-vector is the successor in the lexicographic ordering of Proposition 3.6 of $\alpha^{(max)}$

```
01    next := 0;
02    for i = 1 to |V| do
03       k := 0;
04       while k ≤ |V| do
05          k := k + 1
06          c5 := |{m ∈ Mφ : xmax ∈ m, |m| = k}|;
07          c6 := |{m ∈ Mφ : xi ∈ m, |m| = k}|;
08          if (c6 < c5) then
09             next := i;
10             k := |V| + 1;
11          elseif (c6 > c5) then k := |V| + 1;
12       endwhile
13    endfor
14    for i = 1 to |V| do
15       k := 0;
16       while k ≤ |V| do
17          k := k + 1
18          c5 := |{m ∈ Mφ : xmax ∈ m, |m| = k}|;
19          c6 := |{m ∈ Mφ : xnext ∈ m, |m| = k}|;
19          c7 := |{m ∈ Mφ : xi ∈ m, |m| = k}|;
20          if (c6 < c7) ∧ (c7 < c5) then
21             next := i;
22             k := |V| + 1;
23          elseif (c6 > c7) ∨ (c7 > c5) then k := |V| + 1;
24       endwhile
25    endfor
26    return next
```

In lines `01` to `13` the algorithm searches a variable whose α-vector is lexicographically smaller than $\alpha^{(max)}$. All variables whose α-vector is lexicographically smaller than $\alpha^{(max)}$ are candidates for the variable having the lexicographically next largest vector. Our algorithm simply checks all candidates. In the `while`-loop of line `16` the algorithm tries to find a variable whose α-vector vector is smaller than $\alpha^{(max)}$ but larger than $\alpha^{(next)}$, the successor so far of $\alpha^{(max)}$. If $\alpha^{(i)}$ lies lexicographically in between $\alpha^{(max)}$ and $\alpha^{(next)}$, then i is a new candidate for the successor (lines `20` to `22`). The correctness proof is straightforward.

The space needed for the three counters c_5, c_6, and c_7 is logarithmically bounded in n, since their largest value is $|M_\varphi|$. The values of these counters are derived analogously to the counters c_3 and c_4. Hence, we only need three other logarithmically space bounded counters that count size of monomials. The `for`-loops in lines `02` and `14` manage two other logarithmic counters to know the current variable. The variable k is also logarithmically space bounded, since the largest value to store is $|V| + 1$. The variables $next$ and max contain variable indices which are logarithmically space bounded in n. Hence, the function `get_next` can be computed using logarithmic space.

Altogether, we can conclude that the w-permutation $\pi(\varphi)$ of φ can be written on an oracle tape using logarithmic space. □