Electronic Colloquium on Computational Complexity, Report No. 45 (1998)

A Separation of Syntactic and Nonsyntactic $(1, +k)$-Branching Programs

Detlef Sieling[*]
FB Informatik, LS II, Univ. Dortmund
44221 Dortmund, Fed. Rep. of Germany
sieling@ls2.cs.uni-dortmund.de

Abstract For $(1, +k)$-branching programs and read-k-times branching programs syntactic and nonsyntactic variants can be distinguished. The nonsyntactic variants correspond in a natural way to sequential computations with restrictions on reading the input while lower bound proofs are easier or only known for the syntactic variants. In this paper it is shown that nonsyntactic $(1, +k)$-branching programs are really more powerful than syntactic $(1, +k)$-branching programs by presenting an explicitly defined function with polynomial size nonsyntactic $(1, +1)$-branching programs but only exponential size syntactic $(1, +k)$-branching programs. Another separation of these variants of branching programs is obtained by comparing the complexity of the satisfiability test for both variants.

1 Introduction

In complexity theory branching programs and lower bound methods for branching programs are investigated due to the relationship between branching programs and sequential computations. Superpolynomial lower bounds on the size of branching programs imply superlogarithmic lower bounds on the space complexity of the corresponding model of sequential computation. However, up to now superpolynomial lower bounds are known only for some quite restricted variants of branching programs. A survey is given in Razborov [8]. Hence, one tries to refine the proof methods in order to obtain lower bounds for less restricted variants of branching programs. In this paper we contribute to the refinement of lower and upper bound proofs for restricted branching programs by separating the syntactic and nonsyntactic variant of $(1, +k)$-branching programs.

Before we discuss some restrictions of branching programs in order to motivate the distinction between syntactic and nonsyntactic variants we repeat the definition of branching programs. A branching program for some Boolean function $f(x_1, \ldots, x_n)$ is a directed acyclic graph with one source node and two sinks labeled by 0 and 1. The nonsink nodes, also called interior nodes, are labeled by one of the variables $x_1, \ldots, x_n$ and have two outgoing edges, one labeled by 0 and the other one labeled by 1. The computation path for some input $(a_1, \ldots, a_n) \in \{0, 1\}^n$ starts at the source. At each interior node labeled by x_i the outgoing edge labeled by a_i is chosen. The label of the sink that is finally reached is equal to $f(a_1, \ldots, a_n)$.

Most restrictions of branching programs for which exponential lower bounds are known are restrictions on the multiplicity of reading a variable during a computation. In a read-k-times branching

[*]The author was supported in part by DFG grant We 1066/8.

ISSN 1433-8092

program each variable may be tested at most k times during a computation. Early lower bounds on the branching program complexity were proved for read-once branching programs by Wegener [16] and Žák [18]. However, there is no progress in proving superpolynomial lower bounds even for read-twice branching programs. The reason might be that such branching programs may contain paths that do not correspond to any computation so that on such paths variables may be tested more than k times. Paths not corresponding to any computation contain for some variable x_i a test whether x_i is equal to 0 and a test whether x_i is equal to 1. Such paths are called inconsistent paths or null chains. In order to overcome the difficulties caused by inconsistent paths the read-k-times restriction was extended also to inconsistent paths. We obtain a more restricted type of read-k-times branching programs called syntactic read-k-times branching programs, while the variant where the read-k-times restriction only applies to consistent paths is called nonsyntactic read-k-times. Exponential lower bounds for syntactic read-k-times branching programs and some generalizations were proved by Okol'nishnikova [5, 6, 7], Borodin, Razborov and Smolensky [1], Jukna [2], Sauerhoff [9] and Thathachar [15]. In the last paper even a hierarchy result is proved, i.e., a function is presented that separates polynomial size syntactic read-$(k+1)$-times branching programs from polynomial size syntactic read-k-times branching programs.

Another restriction on the multiplicity of reading variables is used in $(1,+k)$-branching programs. Now a set of at most k variables may be tested more than once (even arbitrarily often) where this set may depend on the chosen path in the branching program. Again there are nonsyntactic $(1,+k)$-branching programs where the restriction applies to computations or, equivalently, to consistent paths, and the syntactic variant, where the restriction applies to consistent and inconsistent paths. An exponential lower bound and a hierarchy result for syntactic $(1,+k)$-branching programs were proved by Sieling [14]. An exponential lower bound for the nonsyntactic variant was proved first by Žák [19] and later on improved by Savický and Žák [11, 12, 13] and by Jukna and Razborov [3]. Savický and Žák [11, 13] also obtained a hierarchy result for nonsyntactic $(1,+k)$-branching programs.

Jukna and Razborov [3] define a possibly stronger variant of $(1,+k)$-branching programs (and read-k-times branching programs as well) which they call semantic $(1,+k)$-branching programs. In such a branching program during a computation variables may be tested illegally, i.e., variables may be tested repeatedly even if at least k variables have been tested repeatedly before. But then the result of the computation must not depend on the results of illegal readings of variables. Jukna and Razborov [3] provide exponential lower bounds also for semantic $(1,+k)$-branching programs.

All these lower bound results imply the question whether the syntactic, nonsyntactic and semantic variants of branching programs are really of different power. The only result in this direction is due to Jukna [2] who considers read-once switching and rectifier networks, which are a nondeterministic variant of branching programs. We remark that there is no difference between syntactic and nonsyntactic read-once branching programs, so that read-once switching and rectifier networks are the weakest variant of branching programs where the distinction between a syntactic and a nonsyntactic variant makes sense. Jukna [2] gives an example of a function for which nonsyntactic read-once switching and rectifier networks are of polynomial size while syntactic read-once switching and rectifier networks are of exponential size.

The main result of this paper is the separation of syntactic and nonsyntactic $(1,+k)$-branching programs. We present an explicitly defined function with polynomial size nonsyntactic $(1,+1)$-branching programs but only exponential size syntactic $(1,+k)$-branching programs, if $k \leq n^{1/2}/(6\log n)$ where n is the number of variables. Hence, for $(1,+k)$-branching programs the distinction between a syntactic and a nonsyntactic variant is justified. We remark that the lower bound also holds for nondeterministic syntactic $(1,+k)$-branching programs. Hence, we also obtain the result that nondeterministic nonsyntactic $(1,+k)$-branching programs are exponentially more powerful than the syntactic variant.

Another setting where variants of branching programs are considered is their use as a data structure for the efficient representation and manipulation of Boolean functions. Besides the size of the representation for particular functions the complexity of operations on Boolean functions represented by the variants of branching programs is investigated. Examples for such operations are the satisfiability test, i.e., the test whether a function represented by the considered variant of branching programs is different from 0, or the equivalence test of two functions represented by the considered variant of branching programs. For a survey we refer to Wegener [17]. A rule of thumb that can be obtained by comparing several variants of branching programs is that variants with a larger class of functions with small size representations usually have less efficient algorithms, in particular, for the satisfiability test and the equivalence test. We show that this rule is true when comparing syntactic and nonsyntactic $(1,+k)$-branching programs by proving the NP-completeness of the satisfiability test and the coNP-completeness of the equivalence test for nonsyntactic $(1,+1)$-branching programs. This complements the results of Savický [10] who presents for syntactic $(1,+k)$-branching programs and constant k a polynomial time algorithm for the satisfiability test and a polynomial time probabilistic algorithm with one sided error for the equivalence test.

Both separation results on syntactic and nonsyntactic $(1,+k)$-branching programs, the separation by a function and by the complexity of the satisfiability test, are based on the same idea: Computations in nonsyntactic $(1,+k)$-branching programs may store information on the input not only by reaching different nodes for different inputs, but also by testing different sets of variables during the computation.

The paper is organized as follows. In Section 2 we define the function that separates polynomial size syntactic and nonsyntactic $(1,+k)$-branching programs and prove the upper bound for nonsyntactic $(1,+1)$-branching programs. In Section 3 we repeat the definition of nondeterministic branching programs and prove the lower bound for nondeterministic syntactic $(1,+k)$-branching programs. Finally, we prove the NP-completeness of the satisfiability test and some related problems for nonsyntactic $(1,+1)$-branching programs.

2 The Considered Function and the Upper Bound

We start with the definition of the considered function. Let $m \geq 2$ and let $n = \frac{7}{2}m^2 - \frac{5}{2}m$. In order to define the function $f_n : \{0,1\}^n \to \{0,1\}$ we first describe the set X of variables on which f_n depends. X consists of

i) variables a_i, where $i \in [m]$,

ii) variables $b_{i,j}, c_{i,j}, d_{i,j}$, where $i, j \in [m]$ and $i \neq j$,

iii) variables $e_{i,j}$, where $i, j \in [m]$ and $i < j$.

Then we define $f_n(a,b,c,d,e) = 1$ if and only if

$$\begin{aligned} & \forall i,j \in [m], i \neq j : a_i = b_{i,j} = c_{i,j} = d_{i,j} \\ \wedge \quad & \forall i,j \in [m], i < j : e_{i,j} = a_i \oplus a_j. \end{aligned}$$

For the proof of the lower bound it is helpful to associate the complete graph K_m on m vertices with f_n. We shall call the variables $a_i, b_{i,j}, c_{i,j}$ and $d_{i,j}$ corresponding to the ith vertex of K_m. Similarly, we call the variable $e_{i,j}$ corresponding to the edge between the vertices i and j. Then the function computes a one if for each vertex of K_m the corresponding variables take the same value and if for each edge the value of the corresponding variable is the parity of the variables of the endpoints. In order to avoid confusion of the nodes of the considered branching programs and of K_m we use the word node only for the branching programs and the word vertex only for the graph K_m.

Theorem 1 There is a nonsyntactic $(1,+1)$-branching program for f_n with $O(n)$ nodes.

Proof For each $i \in [m]$ we create the component G_i shown in Figure 1. In order to simplify the presentation we omit all edges leading to the 0-sink. For each pair (i,j) where $i < j$ we create the component $H_{i,j}$ shown in Figure 1. The branching program P_n for f_n computes the conjunction of all these components. This means that the components are combined in the following way. Let some ordering of the components G_i, $i \in [m]$, and $H_{i,j}$, $i,j \in [m], i < j$, be fixed. The source of P_n is the source of the first component in the ordering. The 1-sink of each component (except the last one in the ordering) is replaced by the source of the successor in the ordering. The 1-sink of the last component is the 1-sink of P_n.

(Insert Figure 1 here)

We have to prove that the constructed branching program P_n computes the function f_n and that it is a nonsyntactic $(1,+1)$-branching program.

If we consider an input (a,b,c,d,e) where $f_n(a,b,c,d,e) = 1$, it is easy to verify that in each component the computation path leads from the source to the 1-sink. Hence, the branching program computes the value one.

Now consider an input (a,b,c,d,e) where the branching program computes the value one. First we show that for each $i,j \in [m]$, where $i \neq j$, we have $a_i = b_{i,j} = c_{i,j} = d_{i,j}$. Let some $i \in [m]$ be fixed. We assume that $a_i = 0$. For the other case the same proof can be used after exchanging the b- and the c-variables. Because of $a_i = 0$ in G_i the 1-sink is reached only if $b_{i,1} = \ldots = b_{i,m} = 0$. Hence, it remains to show that also the variables $c_{i,j}$ and $d_{i,j}$ take the value 0.

First, let $i < j$. We look at the component $H_{i,j}$. If $d_{i,j} = 1$, then a node labeled by $b_{i,j}$ is reached. Because of $b_{i,j} = 0$ the 0-sink is reached in contradiction to the assumption that the branching program computes one. Hence, $d_{i,j} = 0$. Then a test of $c_{i,j}$ is reached and again the 0-sink is reached if $c_{i,j} = 1$. Hence, also $c_{i,j} = 0$. If $i > j$, we look at the component $H_{j,i}$ and apply the same arguments on the nodes of the last two levels of $H_{j,i}$. Again we obtain that $c_{i,j} = d_{i,j} = 0$.

Now we may assume that for each i the variables a_i, $b_{i,j}$, $c_{i,j}$ and $d_{i,j}$ take the same value if the branching program computes one. Under this extra condition it is easy to see that in the component $H_{i,j}$ the 1-sink is reached only if $e_{i,j} = a_i \oplus a_j$.

It remains to show that the branching program is a nonsyntactic $(1,+1)$-branching program. The only variables that may be tested repeatedly during a computation are the $b_{i,j}$- and $c_{i,j}$-variables. If $b_{i,j}$ is tested twice during some computation, then it is tested in G_i and in $H_{i,j}$ if $i < j$ or in $H_{j,i}$ otherwise. In G_i the 1-sink is reached only if $b_{i,j} = 0$. In $H_{i,j}$ (or $H_{j,i}$, resp.) the 1-sink is reached only if $b_{i,j} = 1$, otherwise the computation paths runs to the 0-sink. In other words, if a variable is tested for a second time, then the computation is immediately aborted by going to the 0-sink. Then no other variable can be tested for a second time. $\square$

Let us assume that the first m components of P_n are the components G_i. Let v^* be the node that replaces the 1-sink of the last of these components. It is easy to see that there are 2^m computation paths leading from the source via v^* to the 1-sink. On each of these computation paths a different set of variables is tested between the source and v^*. We compare this property of P_n with a property of syntactic $(1, +k)$-branching programs for f_n which we shall prove below in Lemma 3: Let v be a node in such a branching program. Then the number of different sets of variables which are tested between the source and v on computation paths from the source via v to the 1-sink is bounded by some polynomial.

3 The Lower Bound

We shall prove an exponential lower bound even for nondeterministic syntactic $(1, +k)$-branching programs for (f_n). Different from deterministic branching programs as defined in the Introduction a nondeterministic branching program may contain “guessing nodes”, i.e., interior nodes with the outdegree 2 and without any label. At each guessing node there are two possibilities to continue a computation. Hence, for each input there may be several computation paths. As usual a nondeterministic branching program computes the value one on an input a, if there is at least one accepting computation path for a, i.e., a computation path from the source to the 1-sink.

Another nondeterministic variant of branching programs are switching and rectifier networks. Since we do not consider switching and rectifier networks in the lower bound proof we omit their definition, which can be found, e.g., in Jukna [2]. We only remark the observation of Jukna [2] that syntactic read-k-times switching and rectifier networks can be transformed into nondeterministic syntactic read-k-times branching programs where the size only increases polynomially. The same transformation is possible from syntactic $(1, +k)$-switching and rectifier networks into nondeterministic syntactic $(1, +k)$-branching programs. Hence, the following lower bound also holds for syntactic $(1, +k)$-switching and rectifier networks.

Theorem 2 Each nondeterministic syntactic $(1, +k)$-branching program for f_n has at least $2^m/(n^{3k}2^{k+2})$ nodes. This number grows exponentially, if $k \le n^{1/2}/(6 \log n)$.

Proof Let a nondeterministic syntactic $(1, +k)$-branching program Q_n for f_n be given. First we note that there are 2^m inputs $x \in \{0, 1\}^n$ with $f_n(x) = 1$. We shall present a cut-and-paste argument working only on these inputs. For each such input we fix one of the accepting computation paths and in the following we only consider these fixed paths. This idea was already used by Krause, Meinel and Waack [4]. Similar to the proof in Sieling [14] we shall prove for each node v that there is only a polynomial number of different sets of variables tested on accepting computation paths between the source and v (Lemma 3). Then we define a set of marked nodes of Q_n and show some properties of the marked nodes (Lemma 4, Lemma 5). Finally, we prove an upper bound on the number of markings of each node. Together with the total number of markings we obtain the lower bound for the number of marked nodes.

We start with the bound on the number of different sets of variables tested on accepting paths between the source and some node v. First we note that each input x with $f_n(x) = 1$ has the property that by flipping any bit of x we obtain an input x' with $f_n(x') = 0$. Hence, on each accepting path in Q_n all variables have to be tested.

Now let v be a node in Q_n and let x and x' be inputs with $f_n(x) = 1$ and $f_n(x') = 1$ and for which the fixed accepting computation paths run through v. Let $V(x)$ be the set of variables tested on the computation path for x before reaching the node v and let $V(x')$ be defined similarly. We claim that there are at most k variables in $V(x) - V(x')$ (and in $V(x') - V(x)$).

The claim is proved by contradiction. Assume that $V(x) - V(x')$ contains more than k variables. Now we consider the path from the source to v following the computation for x and from v to the 1-sink following the computation for x'. This path may be inconsistent, but because of the syntactic $(1, +k)$-restriction also on this path at most k variables may be tested more than once. Then among the at least $k + 1$ variables in $V(x) - V(x')$ there is a variable that must not be tested on the computation path for x' between v and the 1-sink. Hence, there is a variable not tested on the computation path for x' in contradiction to the fact that on accepting computation paths all variables have to be tested.

Lemma 3 Let v be a node in Q_n and let $N(v)$ be the number of different sets of variables tested on accepting paths between the source and v. Then $N(v) \leq n^{2k}$.

Proof We choose some set V of variables tested on accepting paths between the source and v of maximal size. Each other set of variables tested on accepting paths between the source and v can be obtained from V by removing at most k variables from V and adding at most k other variables. The number of sets that can be obtained in this way is bounded by n^{2k}. $\Box$

In order to define which nodes of Q_n are marked we describe a coloring of the vertices and edges of the complete graph K_m. Let $V \subseteq X$ be some set of variables. We say that an edge $\{i, j\}$ (where $i < j$) of K_m is colored for V if V contains the variable $e_{i,j}$. We say that a vertex i of K_m is colored for V if there is a vertex l of K_m so that

i) V contains at least one of the variables $a_l, b_{l,j}, c_{l,j}, d_{l,j}$, where $j \in [m], j \neq l$, and

ii) there is a path in K_m from l to i only consisting of colored edges.

The meaning of "i is colored for V" is the following: Assume that x' is an assignment to the variables in V. Let $A(x')$ be the set of assignments x to all variables in X so that x is an extension of x' and that $f_n(x) = 1$. Then for each input in $A(x')$ the value of a_l is determined by x' since by x' some variable corresponding to the lth vertex of K_m is fixed. Since the values of the variables corresponding to the edges of the path between l and i are fixed by x', for all assignments in $A(x')$ the variable a_i gets the same value.

Now we run for each input x where $f_n(x) = 1$ through Q_n on the accepting computation path fixed for x. For each node v reached on the computation path for x we compute the set $V(v)$ of variables tested on this computation path before v. We mark the first node v on this computation path for which all vertices of K_m are colored for $V(v)$. Furthermore, we associate the set $V(v)$ with v. Altogether, there are 2^m markings. Because of Lemma 3 for each node v there are at most n^{2k} different sets associated with v. In the following two lemmas we state the properties of the marking procedure which we shall use in the cut-and-paste argument.

Lemma 4 Let $x \in f_n^{-1}(1)$, let v be the node marked for x and let $V(v)$ be the set associated with v for x. For each assignment y' to the variables in $V(v)$ there is at most one extension y of y' so that $f_n(y) = 1$.

Proof By the choice of $V(v)$ each vertex of K_m is colored for $V(v)$. Hence, if y' can be extended to an assignment so that the function takes the value one, there is only one choice for the variables a_i, $i \in [m]$ and, therefore, only one choice for all other variables so that the function takes the value one. $\square$

Lemma 5 Let $x \in f_n^{-1}(1)$, let v be the node marked for x and let $V(v)$ be the set associated with v for x. For each assignment y' to the variables in $W(v) = X - V(v)$ there are at most two extensions y of y' so that $f_n(y) = 1$.

Proof Let u be the predecessor of v on the computation path for x. If $V(u) = V(v)$, then v is not marked since the first node v^* for which all vertices of K_m are colored for $V(v^*)$ gets a marking. Let z be the variable tested at u. Then $V(v) = V(u) \cup \{z\}$. We distinguish two cases.

Case 1 z is the variable $e_{i,l}$ corresponding to the edge $\{i, l\}$.

Then either i or l, w.l.o.g. i, is not colored for $V(u)$. If both vertices i and l were colored for $V(u)$ or if both vertices were uncolored for $V(u)$, by adding $e_{i,l}$ to $V(u)$ we would not obtain any new possibility to color vertices in K_m and, hence, v would not be marked.

Let I be the set of vertices that are uncolored for $V(u)$ and let J be the set of vertices that are colored for $V(u)$. Then $i \in I$ and $l \in J$. Except the edge $\{i, l\}$ all edges between a vertex in I and a vertex in J are uncolored for $V(v)$. Hence, the variables corresponding to those edges are contained in $W(v)$. Since the vertices in I are uncolored for $V(u)$, the variables corresponding to these vertices are not contained in $V(u)$ and, hence, they are contained in $W(v)$.

Now let an assignment to the variables in $W(v)$ be given. If there is no extension of this assignment so that the function takes the value one, there is nothing to show. Otherwise, if we try to extend the given assignment to a total assignment so that the function takes the value one, we have no choice for the variables a_j, $j \in I$, because a variable corresponding to j is contained in $W(v)$. Furthermore, we have no choice for the variables a_j, $j \in J - \{l\}$, because there is an edge $\{j, j'\}$ to a node j' in I, where $e_{j,j'}, a_{j'} \in W(v)$. Hence, there are at most two extensions of the given assignment so that the function takes the value one, namely, we may choose $a_l = 0$ or $a_l = 1$. If $I \neq \{i\}$, there is even only one such extension.

Case 2 z is a variable corresponding to a vertex i.

Again we define I as the set of all vertices that are uncolored for $V(u)$, and J as the set of all vertices that are colored for $V(u)$. Then $i \in I$. Furthermore, i is connected with each vertex $j' \in J$ by an uncolored edge $\{i, j'\}$. Hence, $W(v)$ contains all variables $e_{i,j'}$ (or $e_{j',i}$, resp.) for such edges $\{i, j'\}$. The set $W(v)$ also contains for each $i' \in I$ at least one variable corresponding to i'. (This also holds for $i' = i$ since all variables corresponding to i except z are contained in $W(v)$.) Hence, there is at most one extension y for each assignment y' to the variables in $W(v)$ so that $f_n(y) = 1$. $\square$

Now let v be some node of Q_n. Because of Lemma 3 there are at most n^{2k} sets $V(v)$ of variables associated with v. Let V be one of these sets and let $W = X - V$. We only consider those inputs $x \in f_n^{-1}(1)$, for which the fixed paths run through v and for which v is associated with the set V. Let H_V be this set of inputs. We show $|H_V| \le n^k 2^{k+2}$. Then the number of marked nodes is at least the number of markings, i.e. 2^m, divided by n^{2k} for the number of choices for the set V and divided by $n^k 2^{k+2}$ for the maximum number of markings of v for each set V. This implies Theorem 2.

In order to obtain a bound on the size of H_V we perform some modifications of Q_n. By these modifications the represented function does not change and the resulting branching program has the $(1, +k)$-property. However, the size may increase exponentially. Similar to Sieling [14] we develop in Q_n the sub-branching program starting at the node v into a decision tree, i.e., afterwards each node reachable from v has an indegree of one. This can be done by running through Q_n in a top-down order starting at v and duplicating nodes so that the copies have an indegree of one afterwards. The second step is to reorder this decision tree DT with the root v so that in this tree all nodes labeled by W-variables are arranged before all guessing nodes and all nodes labeled by V-variables. This can be done in the following way. We construct a complete decision tree in which all W-variables are tested on each path. We replace each leaf of this decision tree by a copy of DT. Then we remove all redundant tests of variables. This is easily possible: If on some path starting at the root of the resulting decision tree a variable is tested for a second time, there is only one possible result of this second test so that this node and one subtree of this node can be removed. We obtain a branching program where all nodes reachable from v form a decision tree which has in the bottom small decision trees containing guessing nodes and tests of V-variables (see also Fig. 2). On each path of such a small decision tree at most k of the V-variables are tested since already Q_n has this property and for each path the number of tests of V-variables does not increase by all these modifications.

(Insert Figure 2 here)

Let T be one of the small decision trees in the bottom and let x_W be the assignment to the W-variables so that the root of T is reached from v. By Lemma 5 the assignment x_W can be combined with at most two assignments to the V-variables to an input where the function takes the value one. Hence, there are at most two assignments to the V-variables for which in Q_n the node v is reached and which is accepted by T. Thus it suffices only to consider the two corresponding accepting paths in T. On each such path at most k of the V-variables are tested. Hence, the computation of each such path is the evaluation of a monomial consisting of at most k literals over the V-variables. We may replace T by a simplified decision tree T^* without changing the function represented by the branching program. If there is only one accepting path through T, then T^* consists of a path of at most k tests of V-variables on which the corresponding monomial is evaluated. If there are two accepting paths through T, then T^* consists of a guessing node switching between two paths each consisting of at most k tests of V-variables and evaluating the corresponding monomial.

Now assume that during the transformation of two different small decision trees the same monomial M is constructed. Let x_W and x'_W be the assignments to the W-variables so that these decision trees are reached from v. If there is some assignment x_V to the V-variables so that $M(x_V) = 1$ and for which v is reached from the source of the branching program, then there are two extensions of x_V to a total assignment so that the function takes the value one, namely x_W and x'_W. This is a contradiction to Lemma 4. We conclude that each monomial occurs at most once during the above transformation. By Lemma 5 for each monomial M there are at most two assignments x_V to the V-variables so that

v is reached from the source and that $M(x_V) = 1$. Hence, we can estimate $|H_V|$ by two times the number of different monomials on at most k of the V-variables, i.e.,

$$|H_V| \leq 2 \cdot \sum_{i=0}^{k} \binom{n}{i} 2^i \leq n^k 2^{k+2}.$$

This completes the proof of Theorem 2. □

In order to summarize the results we use the following notation. Let P(syn $(1, +k)$-BP) be the set of functions with polynomial size syntactic $(1, +k)$-branching programs. Let P(nonsyn $(1, +k)$-BP), P(syn $(1, +k)$-NBP) and P(nonsyn $(1, +k)$-NBP) defined similarly for the nonsyntactic and nondeterministic variants, resp. Then we have proved for each $k \leq n^{1/2}/(6 \log n)$:

$$\begin{aligned}\text{P(syn } (1,+k)\text{-BP)} &\subsetneq \text{P(nonsyn } (1,+k)\text{-BP)},\\ \text{P(syn } (1,+k)\text{-NBP)} &\subsetneq \text{P(nonsyn } (1,+k)\text{-NBP)}.\end{aligned}$$

Since both separation results are obtained for the same function we also have

$$\text{P(nonsyn } (1,+k)\text{-BP)} \not\subseteq \text{P(syn } (1,+k)\text{-NBP)}.$$

We remark that the function used by Savický and Žák [11, 13] for the separation of P(nonsyn $(1, +k)$-BP) and P(nonsyn $(1, +(k + 1))$-BP) can be represented by polynomial size nondeterministic read-once branching programs and, hence, also by polynomial size nondeterministic syntactic $(1, +k)$-branching programs if k is a constant. On the other hand, this function cannot be represented by polynomial size nonsyntactic $(1, +k)$-branching programs. Hence, for constant k the classes P(syn $(1, +k)$-NBP) and P(nonsyn $(1, +k)$-BP) are incomparable.

4 The NP-Completeness of the Satisfiability Test

In this section we prove the following results.

Theorem 6

i) The problem to decide for a given branching program P whether P is a nonsyntactic $(1, +1)$-branching program is coNP-complete.

ii) The problem to decide for a given nonsyntactic $(1, +1)$-branching program P whether the function represented by P is satisfiable is NP-complete.

iii) The problem to decide for two given nonsyntactic $(1, +1)$-branching programs whether they represent the same function is coNP-complete.

Since by i) the test whether a given branching program is a nonsyntactic $(1, +1)$-branching program is intractable, we assume for the satisfiability test and the equivalence test that it is not necessary to check whether the instance is a nonsyntactic $(1, +1)$-branching program, i.e., we consider promise problems with the promise that the instance is a nonsyntactic $(1, +1)$-branching program. The results of Theorem 6 complement the results of Savický [10] who presents polynomial time algorithms for the test whether a given branching program is a syntactic $(1,+k)$-branching program and for the satisfiability test for syntactic $(1,+k)$-branching programs if k is a constant. For the equivalence test of syntactic $(1,+k)$-branching programs Savický [10] presents a probabilistic polynomial time algorithm with one-sided error.

Proof of Theorem 6 First we consider the satisfiability test for nonsyntactic $(1,+1)$-branching programs. It is obvious that the problem is contained in NP. We present a polynomial time reduction from 3-SAT. Let (X,C) be an instance of 3-SAT where $X = \{x_1,\dots,x_n\}$ is the set of variables and $C = \{c_1,\dots,c_m\}$ is the set of clauses. Let $p(i)$ be the number of unnegated occurrences of the variable x_i in the clauses and let $q(i)$ be the number of negated occurrences of x_i. We introduce new variables $b_{i,1},\dots,b_{i,p(i)}$ and $c_{i,1},\dots,c_{i,q(i)}$ and replace each unnegated occurrence of x_i by some $b_{i,\cdot}$-variable and each negated occurrence by some $c_{i,\cdot}$-variable so that each $b_{i,\cdot}$- and each $c_{i,\cdot}$-variable occurs exactly once. Furthermore, we introduce the variables $a_1,\dots,a_n,d_1,\dots,d_m,e_1,\dots,e_m$. Then we construct for each $i \in [n]$ the component G_i shown in Figure 3 and for each $j \in [m]$ the component H_j corresponding to the jth clause. An example of H_j if the jth clause is $b_{i_1,l_1} \vee c_{i_2,l_2} \vee b_{i_3,l_3}$ is also shown in Figure 3.

(Insert Figure 3 here)

The constructed branching program computes the conjunction of all these components and can be built as described in the proof of Theorem 1. Using the ideas of the proof of Theorem 1 it can be shown that the constructed branching program is really a nonsyntactic $(1,+1)$-branching program and that it represents a satisfiable function iff (X,C) is satisfiable.

The result for the equivalence test follows easily since the satisfiability test is the test whether the given branching program and a branching program for the function 0 represent different functions. Finally, for the test whether a given branching program is a nonsyntactic $(1,+1)$-branching program we use the reduction from 3-SAT as described above, but we replace the 1-sink by some branching program Q which is not a nonsyntactic $(1,+1)$-branching program. The resulting branching program is a nonsyntactic $(1,+1)$-branching program iff Q is not reached for any input, i.e., iff (X,C) is not satisfiable. $\square$

Acknowledgment

I thank Martin Sauerhoff, Petr Savický and Ingo Wegener for fruitful discussions, which inspired and helped to improve this work.

References

[1] Borodin, A., Razborov, A. and Smolensky, R. (1993). On lower bounds for read-k-times branching programs. *Computational Complexity* 3, 1–18.

[2] Jukna, S. (1995). A note on read-k times branching programs. *RAIRO Theoretical Informatics and Applications* 29, 75–83.

[3] Jukna, S. and Razborov, A. (1996). Neither reading few bits twice nor reading illegally helps much. ECCC Report TR96-037.

[4] Krause, M., Meinel, C. and Waack, S. (1991). Separating the eraser Turing machine classes L_e, NL_e, *co*-NL_e and P_e. *Theoretical Computer Science* 86, 267–275.

[5] Okol'nishnikova, E.A. (1993). On lower bounds for branching programs. *Siberian Advances in Mathematics* 3, 152–166.

[6] Okol'nishnikova, E.A. (1997). On the hierarchy of nondeterministic branching k-programs. In *Proc. of Fundamentals of Computing Theory*, Lecture Notes in Computer Science 1279, 376–387.

[7] Okol'nishnikova, E.A. (1997). On comparison between the sizes of read-k-times branching programs. *Operations Research and Discrete Analysis*, Korshunov, A.D., Ed., Kluwer Academic Publishers, 205–225.

[8] Razborov, A.A. (1991). Lower bounds for deterministic and nondeterministic branching programs. In *Proc. of Fundamentals of Computing Theory*, Lecture Notes in Computer Science 529, 47–60.

[9] Sauerhoff, M. (1998). Lower bounds for randomized read-k-times branching programs. In *Proc. of 15th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 1373, 105–115.

[10] Savický, P. (1998). A probabilistic nonequivalence test for syntactic $(1,+k)$-branching programs. Preprint.

[11] Savický, P. and Žák, S. (1996). A hierarchy for $(1,+k)$-branching programs with respect to k. In *Proc. of Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 1295, 478–487.

[12] Savický, P. and Žák, S. (1997). A lower bound on branching programs reading some bits twice. *Theoretical Computer Science* 172, 293–301.

[13] Savický, P. and Žák, S. (1998). A read-once lower bound and a $(1,+k)$-hierarchy for branching programs. To appear in *Theoretical Computer Science*.

[14] Sieling, D. (1996). New lower bounds and hierarchy results for restricted branching programs. *Journal of Computer and System Sciences* 53, 79–87.

[15] Thathachar, J.S. (1998). On separating the read-k-times branching program hierarchy. ECCC Report TR98-002.

[16] Wegener, I. (1988). On the complexity of branching programs and decision trees for clique functions. *Journal of the Association for Computing Machinery* 35, 461–471.

[17] Wegener, I. (1994). Efficient data structures for Boolean functions. *Discrete Mathematics* 136, 347–372.

[18] Žák, S. (1984). An exponential lower bound for one-time-only branching programs. In *Proc. of Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 176, 562–566.

[19] Žák, S. (1995). A superpolynomial lower bound for $(1, +k(n))$-branching programs. In *Proc. of Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 969, 319–325.

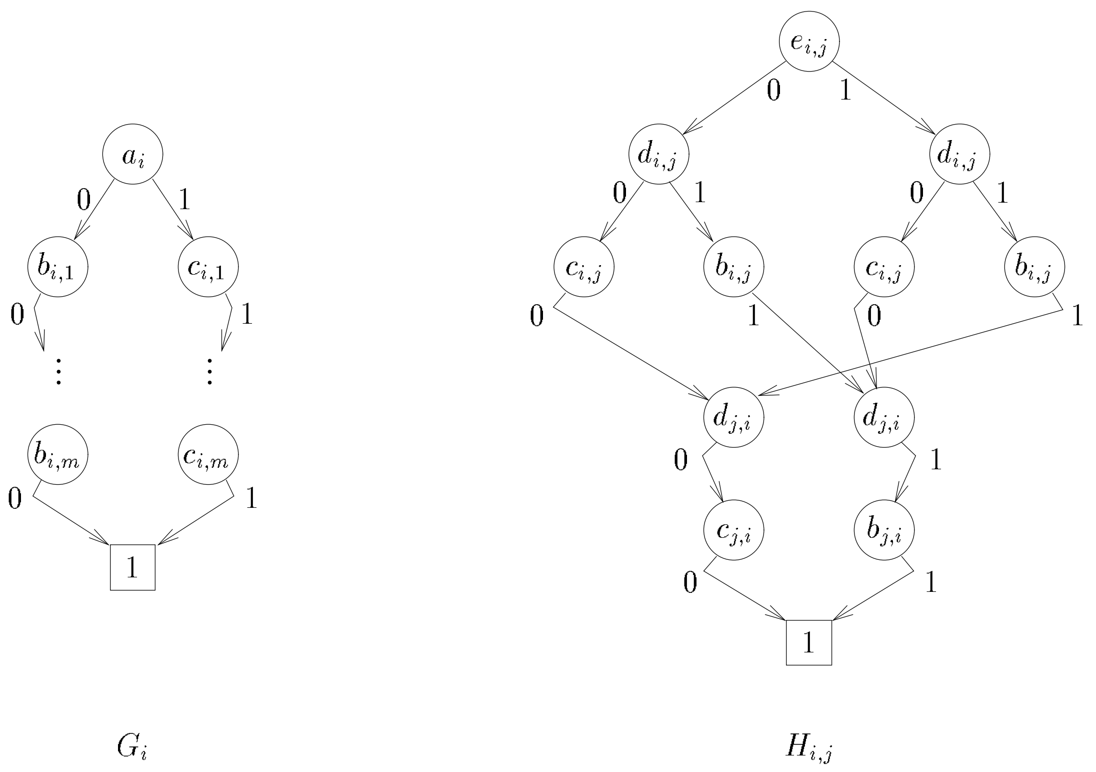
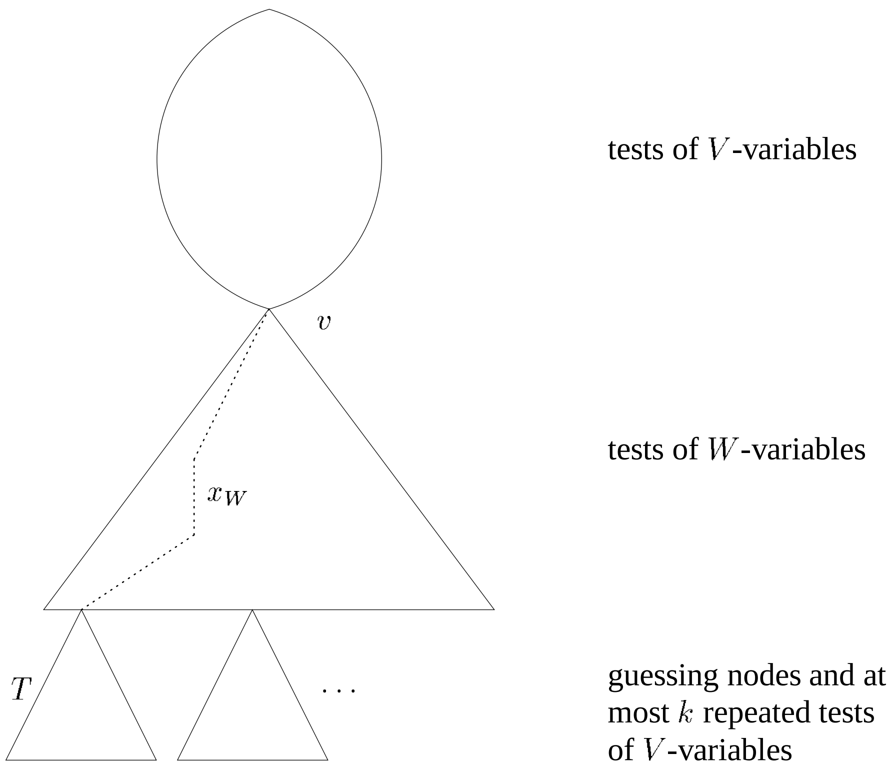


Figure 1: The components G_i and $H_{i,j}$ of the branching program P_n for f_n. All omitted edges in this figure are edges leading to the 0-sink.



tests of V-variables
v
tests of W-variables
x_W
T
$\dots$
guessing nodes and at most k repeated tests of V-variables

Figure 2: The branching program Q_n after constructing and reordering the decision tree starting at v.

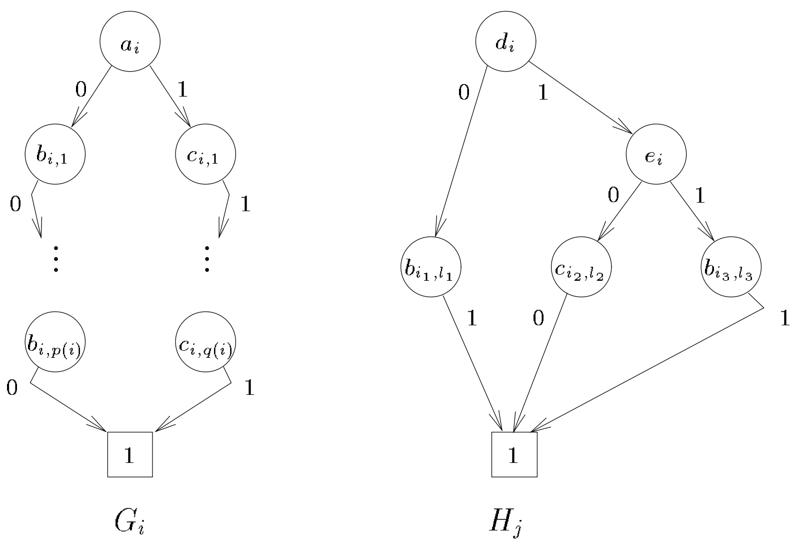


Figure 3: The components G_i and H_j of the branching program constructed in the NP-completeness proof. The component G_i is constructed for the ith variable and the component H_j shown in this figure is constructed for the clause $b_{i_1,l_1} \vee c_{i_2,l_2} \vee b_{i_3,l_3}$.