INFORMATION AND CONTROL 62, 129–143 (1984)

Optimal Decision Trees and One-Time-Only Branching Programs for Symmetric Boolean Functions*

INGO WEGENER

FB-20 Informatik, Johann Wolfgang Goethe-Universität, 6000 Frankfurt a.M., West Germany

Combinational complexity and depth are the most important complexity measures for Boolean functions. It has turned out to be very hard to prove good lower bounds on the combinational complexity or the depth of explicitly defined Boolean functions. Therefore one has restricted oneself to models where nontrivial lower bounds are easier to prove. Here decision trees, branching programs, and one-time-only branching programs are considered, where each variable may be tested on each path of computation only once. Efficient algorithms for the construction of optimal decision trees and optimal one-time-only branching programs for symmetric Boolean functions are presented. Furthermore, the following trade-off results are proved. An exponential lower bound on the decision tree complexity of some Boolean function is shown having linear formula size and linear one-time-only branching program complexity. Furthermore, a quadratic lower bound on the one-time-only branching program complexity of some Boolean function is shown having linear combinational complexity.

1. INTRODUCTION

One of the fundamental problems of complexity theory is to estimate the relative efficiency of different models of computation. In this paper we treat the complexity of Boolean functions. The most important model for the computation of Boolean functions is the Boolean network or circuit model. The proper complexity measures are the Boolean network complexity (combinational complexity) and the depth of Boolean functions. By a result of Spira [12] the formula size of Boolean functions is closely connected to the depth of these functions.

It is known that, for almost all functions, only networks of exponential size and linear depth exist; however, no nontrivial bounds for explicitly defined Boolean functions have been proved. Therefore one is interested in the complexity of Boolean functions in other models, for example,

* Parts of these results have been presented at the 9th Colloq. on Trees in Algebra and Programming (CAAP) at Bordeaux (3.3.–5.3.1984).

0019-9958/84 $3.00

monotone networks. Here we consider decision trees and branching programs for the computation of Boolean functions.

In Section 2 we summarize all necessary definitions and motivate two models of branching programs where the width (resp. the depth) is restricted. We prove some basic results for the different complexity measures.

In Section 3 we present an efficient algorithm for the construction of optimal decision trees for symmetric Boolean functions. The size of its running time equals the size of the constructed decision tree. By this algorithm we can deduce a trade-off result. We show that for some explicitly defined Boolean function each decision tree has exponential size while there exist Boolean formulae and one-time-only branching programs of linear size.

In Section 4 we treat one-time-only branching programs ($BP_1 - s$) which are the depth restricted branching programs motivated in Section 2. These programs fulfil the restriction that each variable may be tested on each path of computation only once. We are able to present also for this model an efficient algorithm for the construction of optimal programs for symmetric Boolean functions. Using this result we construct explicitly functions whose Boolean network complexity is linear and whose BP_1-complexity is quadratic. Furthermore we present the most complex symmetric Boolean functions with respect to decision trees or BP_1s.

Thus we obtain two classes of results. We get efficient algorithms for the construction of some optimal computation schemes. In the field of complexity theory there are only a few results of this type. On the other hand we obtain some interesting trade-off results.

2. A Comparison of the Different Computation Models

We assume that the reader is familiar with Boolean networks and formulae over a basis Ω (see, e.g., Savage [11]). We denote the Boolean network complexity, formula size, and depth of f by $C_\Omega(f)$, $L_\Omega(f)$, and $D_\Omega(f)$, respectively. We suppress the index Ω if we employ $\Omega = \{w \mid w\colon \{0,1\}^2 \to \{0,1\}\}$. Another well-known model for the computation of Boolean functions is the decision tree model.

Definition 1. A decision tree is a binary labelled tree, where the leaves are labelled by Boolean constants and the inner nodes by Boolean variables. The edges are directed from the root to the leaves. A decision tree computes a Boolean function in the following way. One starts at the root which may be labelled by x_i. If the input vector $a \in \{0,1\}^n$ has the property $a_i = 0$ we go to the left successor otherwise to the right successor.

If we reach a leaf labelled $b \in \{0, 1\}: f(a) = b$. By $DT(f)$ we denote the decision tree complexity of f.

Optimal decision trees may contain the same subtree more than once. It is more natural to define such a subtree only once and to point from different situations to the root of this subtree. In this way we obtain branching programs.

DEFINITION 2. A branching program is an acyclic labelled graph with one source and arbitrarily many sinks. Each node has outdegree 0 or 2. The labelling and the mode of computation are the same as for decision trees. By $BP(f)$ and $BPD(f)$ we denote the branching program complexity and the branching program depth of f.

Obviously $BPD(f)$ could have been defined also with respect to decision trees. It has been shown by Cobham [3] and Pudlák and Zák [9] that the logarithm of branching program complexity is a lower bound on space requirements and obviously branching program depth is a lower bound on time requirements for the computation of f in any reasonable model of sequential computation. These results motivate the investigation of branching programs. At first we connect the complexity of branching programs and networks. By sel we denote the selection function:

$$\text{sel}(x, y, z) = y, \quad \text{if} \quad x = 0 \qquad \text{and} \qquad \text{sel}(x, y, z) = z, \quad \text{if} \quad x = 1.$$

THEOREM 1. (i) $C(f) \leqslant 3C_{\{\text{sel}\}}(f) \leqslant 3BP(f)$.

(ii) $D(f) \leqslant 2D_{\{\text{sel}\}}(f) \leqslant 2BPD(f)$.

(iii) $L_{\{\text{sel}\}}(f) \leqslant DT(f)$.

(iv) $BP(f) \leqslant L_{\{\wedge, \vee, \neg\}}(f) + 1$

(v) $D(f) \leqslant c \log(DT(f) + 1)$, *where* $c := 4/(\log 3 - 1) \approx 6.84$.

Proof. The inequalities on $C(f)$ and $D(f)$ are obvious since $\text{sel}(x, y, z) = \bar{x}y \vee xz$ which implies $C(\text{sel}) = 3$ and $D(\text{sel}) = 2$. Given a decision tree for f we may construct a Boolean formula for f over $\{\text{sel}\}$ with the same underlying tree. The same procedure works for branching programs and Boolean networks. At first we reverse the direction of the edges. A node G with predecessors G_1 and G_2 and label x_i gets a new predecessor G_0 labelled x_i. G is labelled by sel and the order of the predecessors is G_0, G_1, G_2. We may easily prove by induction on the number of gates of the decision tree (resp. formula) that this formula again computes f. This proves (i)–(iii).

Inequality (iv) may be proved by induction on $l := L_{\{\wedge, \vee, \neg\}}(f)$. The result is obvious for $l = 0$. For the induction step let f be a Boolean

function of formula size l. We consider an optimal formula for f. The result follows easily if the last gate is an $\neg$-gate since $BP(f) = BP(\bar{f})$. Otherwise $f = g \wedge h$ or $f = g \vee h$ and $L_{\{\wedge,\vee,\neg\}}(g) + L_{\{\wedge,\vee,\neg\}}(h) = l - 1$. By induction hypothesis $BP(g) + BP(h) \leqslant l + 1$. Thus it is sufficient to show that $BP(f) \leqslant BP(g) + BP(h)$. If the last gate is an $\wedge$-gate (resp. $\vee$-gate) we use an optimal branching program for g. We combine all 1-sinks (resp. 0-sinks) and use this node as source of an optimal branching program for h. This branching program obviously computes f.

For inequality (v) the theorem of Spira [12] yields $D_{\{\text{sel}\}}(f) \leqslant (c/2)\log(L_{\{\text{sel}\}}(f) + 1)$. By (ii) and (iii) we get the desired result. Q.E.D.

Remark 1. We note an interesting difference between Boolean networks and formulae on one hand and branching programs and decision trees on the other hand. The information flow is reversed. Thus some ideas for the construction of efficient algorithms for f may work better for one model while others may work better for the other. By Theorem 1 we see that it will perhaps be easier (in any case not harder) to prove lower bounds on the complexity of Boolean functions using branching programs or decision trees.

Remark 2. Inequality (v) cannot be improved to $BPD(f) \leqslant c^* \log(DT(f) + 1)$ for some constant c^* which would be a translation of the result of Spira to decision trees. Such an inequality cannot hold since our results will imply that $BPD(f) = DT(f) = n$ for $f = x_1 \vee \cdots \vee x_n$.

We have seen that complexity and depth of branching programs are interesting complexity measures. Until now we cannot prove large lower bounds on the branching program complexity of Boolean functions. The largest bound of Nechiporuk [8] is of size $n^2/\log^2 n$ and holds for functions with many subfunctions. Thus we obtain by this method only linear lower bounds for symmetric Boolean functions. In this situation one has considered restricted branching programs. Borodin, Dolev, Fich, and Paul [2] introduced branching programs of width 2 which have also been treated by Yao [14]. These are levelled branching programs such that the number of nodes on each level is bounded by 2. Thus the complexity is at most factor 2 larger than the depth. Since depth and complexity are related to time and space these are branching programs whose "space complexity" is minimum with respect to its "time complexity." This is an interesting model since space lower bounds in excess of $\log n$ are a fundamental challenge and since one may obtain time–space trade-offs.

On the other hand one should restrict the "time complexity" of branching programs in order to prove lower bounds on the "space complexity" of Boolean functions under this restriction. This suggests the investigation of branching programs of depth $BPD(f)$. Instead of that we

consider one-time-only branching programs introduced by Masek [7] and investigated also by Pudlák and Zák [9] and Wegener [13].

DEFINITION 3. One-time-only branching programs (BP_1s) are branching programs where on each path at most one node is labelled x_i. The proper complexity measure is also denoted by BP_1.

We believe that BP_1s are a better model than branching programs of depth $BPD(f)$. This will be established in the rest of this chapter. Furthermore we show that for symmetric functions and another class of functions these models indeed are the same.

In order to consider branching programs of minimum depth we have to know $BPD(f)$. This already is not an easy problem. Obviously the depth is bounded by the number of variables. Functions where the depth equals the number of variables are called exhaustive. Rivest and Vuillemin [10] proved that the branching program depth of Boolean functions deciding any nontrivial monotone graph property on n-vertex graphs is at least $n^2/16$. This proves the Aanderaa–Rosenberg conjecture. The generalized conjecture that these and other functions are exhaustive has been disproved by Illies [5]. Hedstück [4] tried to classify the exceptions. More results on the depth of branching programs may be found in Bollobás [1]. Though the general problem of computing $BPD(f)$ is hard, the problem is easy for symmetric functions.

PROPOSITION 1. *If f is nonconstant and symmetric $BPD(f)=n$. Each branching program for f contains a path on which all variables are tested.*

Proof. We describe symmetric functions f by their value vectors $v(f)=(v_0,\ldots, v_n)$, where $v_i=1$ iff $f(x_1,\ldots, x_n)=1$ for all $(x_1,\ldots, x_n)$, where $x_1+\cdots+x_n=i$. The claim holds for $n=1$. For $n>1$ we consider a branching program for f. The root is labelled (because of the symmetry of f) w.l.o.g. by x_n. Either $f_{|x_n=0}$ with value vector $(v_0,\ldots, v_{n-1})$ or $f_{|x_n=1}$ with value vector $(v_1,\ldots, v_n)$ is nonconstant. Following the proper edge we reach a branching program for a nonconstant and symmetric Boolean function of $n-1$ variables having by induction hypothesis depth $n-1$ and including a path where all $n-1$ variables are tested. Q.E.D.

This argument for BP_1s is based only on our inability of computing in general $BPD(f)$. The following argument is more significant. In BP_1s we are not allowed to gather information by combining several paths of computation if we would be forced to separate them again by repeating an old test. For branching programs of minimum depth the following may happen. $BPD(f)$ may be large but only one long path is necessary while most

of the computation paths may be short. In branching programs of depth $BPD(f)$ we may decrease the number of nodes by increasing the length of the paths which could be short. This is not possible for BP_1s. Thus BP_1s are branching programs where each path has an individual bound for its length. If we have to compute subfunctions where some variables are useless the bound may be smaller than for other paths. Thus also the average time complexity is bounded. We give an example for the described effect.

DEFINITION 4. The exactly-half function f^n_{exh} computes 1 iff the input contains exactly $\lceil n/2 \rceil$ ones, i.e., $\sum_{1 \leqslant i \leqslant n} x_i = \lceil n/2 \rceil$.

We show that f^n_{exh} may be computed efficiently. By the Chinese remainder theorem $f^n_{\text{exh}}(x) = 1$ iff $\sum_{1 \leqslant i \leqslant n} x_i \equiv \lceil n/2 \rceil \bmod p_j$ for primes $p_1, \ldots, p_m$, where $\prod_{1 \leqslant j \leqslant m} p_j \geqslant n$. Let $g_{n,p}(x) = 1$ iff $\sum_{1 \leqslant i \leqslant n} x_i \equiv \lceil n/2 \rceil \bmod p$. Obviously $BP_1(g_{n,p}) \leqslant pn$ since we may test the variables one after another and may combine all nodes where we have tested $x_1, \ldots, x_i$ and the number of ones mod p is the same. We get a branching program for f^n_{exh} of depth mn and size $n \sum_{1 \leqslant j \leqslant m} p_j$ by combining branching programs for g_{n,p_j}. The 1-sink of the BP_1 for g_{n,p_j} is the source of $g_{n,p_{j+1}}$ if $j < m$ and a 1-sink if $j = m$. For constant m we may use the m smallest primes larger than $n^{1/m}$. The depth is mn and the size $O(n^{1+1/m})$. It is also possible to use the $\lceil \ln n / \ln \ln n \rceil$ smallest primes. By the prime number theorem their product is larger than n but their size is at most $O(\log n)$. Thus we obtain a branching program of depth $O(n \log n / \log \log n)$ and size $O(n \log^2 n / \log \log n)$. This is for its own sake an interestig trade-off since we show in Section 4 that $BP_1(f^n_{\text{exh}}) = \theta(n^2)$.

Let us now consider the function $h_{n,l}$, where $h_{n,l}(x) := x_1 \wedge \cdots \wedge x_{n-1}$ if $x_n = 1$ and $h_{n,l}(x) := f^l_{\text{exh}}(x_1, \ldots, x_l)$ if $x_n = 0$. Since $BP_1(f^n_{\text{exh}}) = \theta(n^2)$ also $BP_1(h_{n,l}) = \Omega(l^2)$. If $l := \lfloor (n-1)/m \rfloor$ for a constant m even $BP_1(h_{n,l}) = \Omega(n^2)$. Considering the input of ones only it follows that $BPD(h_{n,l}) = n$. Let us restrict the depth by n. We may test at first x_n. If $x_n = 1$, $n-1$ further nodes are sufficient. If $x_n = 0$ we may test any of the variables $x_1, \ldots, x_l$ for m times. Thus as we have seen before $O(n^{1+1/m}) = o(n^2)$ nodes are sufficient. We obtain a branching program for $h_{n,l}$ of minimum depth which is much more efficient than the best BP_1. But $h_{n,l}$ is essentially an exactly half function. Its optimal time-bounded branching program should therefore be essentially an optimal time-bounded branching program for the exactly half function. This proves that BP_1s build a more natural computation model for time-bounded branching programs than branching programs of minimum depth.

Nevertheless for many functions and all symmetric functions branching programs of minimum depth and BP_1s are the same.

THEOREM 2. *For all symmetric Boolean functions, all functions whose prime implicants have length n, and all functions whose prime clauses have length n the size of optimal branching programs of minimum depth equals the size of optimal* BP_1s.

Proof. Let F be the class of functions considered in the Theorem. At first we show that $BPD(f)=n$ for $f\in F$. For symmetric functions this follows by Proposition 1. Otherwise let t be any prime implicant or prime clause of length n. We consider that path of computation where this prime implicant is 1 (resp. this prime clause is 0). This path stops at a 1-sink (resp. 0-sink). If we would have tested at this sink less than n variables the monom (resp. sum) consisting of exactly those variables and negated variables which must be 1 (resp. 0) if we reach this sink is an implicant (resp. a clause) and a proper shortening of t. Thus this path must have length n.

Since the depth of each BP_1 is at most n we know already that a BP_1 for $f\in F$ is always a branching program of minimum depth.

Let us now consider an optimal branching program of minimum depth n for $f\in F$. If it is not a BP_1 we look at a node N where some variable x_i is tested for the second time. We choose N as near as possible to the source. Then we may reach N on a path where x_i has already been tested. After the second test of x_i we have tested on this path l variables and the length of the paths starting here is bounded by $n-l-1$. Let g be the subfunction we have to compute. If g is constant the second test of x_i on our path was superfluous. The edge to N on this path may be replaced by an edge to a sink without increasing the size of the program. Otherwise g is symmetric on $n-l$ variables if f is symmetric. Or g has only prime implicants (resp. clauses) of length $n-l$ if f has only prime implicants (resp. clauses) of length n. Thus by our considerations above $BPD(g)=n-l$. It is impossible to compute g since we are allowed to use only paths of length $n-l-1$. Q.E.D.

Altogeher we have argued that BP_1s build the proper model for time-bounded branching programs. For symmetric functions they are equal to branching programs of minimum depth. By results on BP_1s we hope to gain more insight to the complexity of branching programs and to prove time–space trade-offs.

3. AN EFFICIENT ALGORITHM FOR THE CONSTRUCTION OF OPTIMAL DECISION TREES FOR SYMMETRIC FUNCTIONS

The main purpose of this chapter is indicated in the title. A further purpose is to prove a trade-off. The most important part of this theorem, the

lower bound on the decision tree complexity of p_n, is an easy special case of the proposed efficient algorithm.

THEOREM 3. *For the parity function* $p_n(x_1,..., x_n) = x_1 \oplus \cdots \oplus x_n$ *it holds that*

(i) $L(p_n) = n - 1$,

(ii) $BP(p_n) = 2n - 1$,

(iii) $DT(p_n) = 2^n - 1$.

Proof. The property $L(p_n) = n - 1$ is obvious. For all functions f on n variables it holds that $DT(f) \leqslant 2^n - 1$ since we may label the root by x_n and may construct two subtrees for $f_{|x_n=0}$ and $f_{|x_n=1}$. For the proof of (iii) we have still to show the lower bound. For $n = 1$ the result is obvious. For $n > 1$ a decision tree for p_n contains the source and disjoint subtrees for p_{n-1} and $\bar{p}_{n-1}$. Since $DT(p_n) = DT(\bar{p}_n)$ we obtain by induction hypothesis

$$DT(p_n) \geqslant 1 + 2DT(p_{n-1}) \geqslant 1 + 2(2^{n-1} - 1) = 2^n - 1.$$

Since $p_n(x) = 1$ iff $\sum_{1 \leqslant i \leqslant n} x_i \equiv 1 \bmod 2$ we obtain by the considerations of Section 2 a BP_1 for p_n with $2n - 1$ nodes. For the lower bound we prove at first $BP(p_n, \bar{p}_n) \geqslant 2n$. The assertion holds clearly for $n = 1$. For $n > 1$ we may assume w.l.o.g. that the source for the computation of $\bar{p}_n$ does not lie on a path starting from the source for p_n. In general in branching programs for more than one function some sources need not be sources of the graph. A similar property holds for Boolean networks for several outputs where outputs for some functions need not to be sinks of the graph. Because of the symmetry of p_n we may assume w.l.o.g. that the source for p_n is labelled by x_n. The two direct successors are sources for p_{n-1} and $\bar{p}_{n-1}$. These two nodes form the sources of a branching program for p_{n-1} and $\bar{p}_{n-1}$ which does not contain the necessarily different sources for p_n and $\bar{p}_n$. Thus $BP(p_n, \bar{p}_n) \geqslant 2 + BP(p_{n-1}, \bar{p}_{n-1})$ yielding $BP(p_n, \bar{p}_n) \geqslant 2n$. By the same argument we obtain

$$BP(p_n) \geqslant 1 + BP(p_{n-1}, \bar{p}_{n-1}) \geqslant 1 + 2(n-1) = 2n - 1. \qquad \text{Q.E.D.}$$

Remark 3. The proof above shows that the parity function is the most difficult Boolean function with respect to decision trees. Furthermore we get the largest possible gap between formula size and decision tree complexity.

We have seen that it is rather simple to construct optimal decision trees for p_n. In the following we present an efficient algorithm for the construction of an optimal decision tree for each symmetric Boolean function. Such algorithms do not exist for Boolean networks or formulae. In general there

are known only a few efficient algorithms for the construction of optimal computation schemes. Therefore our result offers the problem of finding more classes of functions where efficient algorithms for the computation of optimal decision trees of BP_1s exist.

We assume that a symmetric function is represented by its value vector $v(f)=(v_0,..., v_n)$ (see Section 2). Because of the symmetry of the function we may assume w.l.o.g. that the nodes of level i are labelled by x_{n+1-i}. The graph structure of a decision tree is determined. Therefore we have to decide only whether we may stop the computation. This is possible iff the partial function which has to be computed at the considered node is constant.

ALGORITHM 1. Input: $v(f)\in\{0,1\}^{n+1}$, the value vector of a symmetric function f: $\{0,1\}^n\to\{0,1\}$.

(1) We determine the constant parts of $v(f)$. If $v_{i-1}\neq v_i=\cdots=v_j\neq v_{j+1}$, where $v_{-1}:=v_{n+1}:=-1$ we store the interval $[i,j]$ together with v_i in an array at place i and at place j. Thus each interval is stored twice if $i\neq j$ and once if $i=j$. At each array place we store at most one interval.

(2) Perform procedure $[0,n]$. We define procedure $[k,l]$ for all $0\leqslant k\leqslant l\leqslant n$.

PROCEDURE $[k,l]$. (1) Check whether some interval $[k',l]$ where $k'\leqslant k$ or $[k,l']$, where $l\leqslant l'$ is stored. In the positive case go to 2, else go to 3.

(2) Label the node by the value stored together with the interval and STOP.

(3) Label the node by x_{l-k} and construct two direct successors. For the left successor call procedure $[k,l-1]$ and for the right successor call procedure $[k+1,l]$.

THEOREM 4. *Algorithm* 1 *constructs optimal decision trees for symmetric Boolean functions. If f is not constant the size of the running time equals the size of the constructed decision tree and is therefore minimal.*

Proof. At first we prove the correctness of the algorithm. It is obvious that on the ith level we label the nodes by a constant or by x_{n+1-i}. If a node is considered by procedure $[k,l]$ the current value vector, that is, the value vector of the symmetric subfunction which has to be computed by the subtree rooted at this node, is $(v_k,..., v_l)$. The intervals are correctly updated in step (3). Therefore we can conclude that the algorithm works correctly if it labels some node by a constant. The length of the intervals is always decreased by one. If we reach for the first time an interval of con-

stant values at least the right or left endpoint of the interval coincides with the right or left endpoint of the proper maximum interval. Therefore we label a node by a constant at the first point of time where this is correct. The constructed decision tree is optimal.

Now we consider the running time of the algorithm. Using the obvious procedure we may store the appropriate intervals $[i, j]$ in time $O(n)$. For each node procedure $[k, l]$ has constant running time since the check of step (1) can be done by two tests looking at the array places k and l. Thus the running time of procedure $[0, n]$ altogether is only by a constant factor larger than the number of nodes of the constructed decision tree. By Proposition 1 we know for all nonconstant symmetric functions f that $DT(f) \geqslant BPD(f) = n$. Thus we get the proposed result for the running time. Q.E.D.

4. An Efficient Algorithm for the Construction of Optimal BP_1s for Symmetric Functions

We present an efficient algorithm for the construction of optimal BP_1s for symmetric Boolean functions. Furthermore we obtain trade-offs. For several functions we get quadratic lower bounds including the exactly half function and the majority function. Borodin *et al.* [2] proved an $\Omega(n^2/\log n)$ lower bound on the complexity of branching programs of width 2 for the majority function.

Lemma 1. *For each symmetric Boolean function f there exists an optimal BP_1 which is levelled and where each node on level l is labelled by a constant or by x_{n+1-l}.*

Proof. We consider any optimal BP_1 for f. A linkage node is a node with more than one direct predecessor. Let p_1 and p_2 be the two paths which are linked at the linkage node L. Either this node is labelled by a constant or by a variable. In the first case we may destroy the linkage without increasing the cost of the program. We like to prove for linkage nodes L in optimal BP_1s which are labelled by a variable that the set of labellings is on all paths from the source to L the same. Let us assume on the contrary that p_1 and p_2 are paths from the source to L where some variable x_k is tested on p_1 but not on p_2. Let f' be the function computed by the BP_1 starting at L. Considering the path p_2, f' has to be a non-constant, symmetric function on a set of variables containing x_k. But the branching program for f' starting at L cannot test x_k since x_k has been tested on p_1. This contradicts Proposition 1.

Since on all paths from the source to L we have tested the same set of

variables and each variable only once, all these paths have the same length. That means that the underlying graphs is levelled.

We fix the graph and rename the labels of the inner nodes. An inner node on level l gets label x_{n+1-l}. Obviously we again obtain a BP_1. It remains to show that it again computes f. Let $a \in \{0,1\}^n$ and p be the path we have to follow for input a in the reconstructed program. Let π be a permutation on $\{1,...,n\}$ such that for the old program the node on level l on p does not exist or is labelled by $x_{\pi(n+1-l)}$. Let $a' := (a_{\sigma(1)},..., a_{\sigma(n)})$ for $\sigma := \pi^{-1}$. Because of the symmetry of f we have $f(a) = f(a')$. If the input is a', the old program forces us to follow p. Thus the endpoint of p is labelled by $f(a')$. This label has not been changed during the reconstruction of the program. Thus the new program computes correctly $f(a)$. Q.E.D.

By Lemma 1 we know a lot about the structure of optimal BP_1s for symmetric Boolean functions f. We start with the source labelled by a constant if f is constant or labelled by x_n otherwise. After having labelled all nodes on level l by appropriate constants or x_{n+1-l}, the nodes labelled by x_{n+1-l} get two direct successors, one for $x_{n+1-l}=0$ and one for $x_{n+1-l}=1$. At first we decide for each new node whether it may be labelled by a constant. This can be done by the same tests as in Algorithm 1. For the nodes which do not become labelled by constants we decide which nodes may be merged to one node. By Lemma 1 we have to decide this question only for nodes on the same level. For each node N at level $l+1$ not labelled by a constant we have to compute the nonconstant and symmetric function f_N on $\{x_1,..., x_{n-l}\}$ which results from f by fixing $x_{n-l+1},..., x_n$ in a way that we reach N. By the proof of Lemma 1 we may merge N and N' iff $f_N = f_{N'}$. The value vectors of f_N and $f_{N'}$ are substrings of $v(f)$, the value vector of f. If we have merged as many nodes as possible, we label all the unlabelled nodes on level $l+1$ by x_{n-l}. Then we go on in the same way until there are no nodes without successors labelled by a variable. This happens on level $n+1$.

Thus the following algorithm constructs an optimal BP_1 for any symmetric function f with value vector $v(f)$.

ALGORITHM 2. Input: $v(f) \in \{0,1\}^{n+1}$, the value vector of a symmetric function f: $\{0,1\}^n \to \{0,1\}$.

(1) We determine the constant parts of $v(f)$. If $v_{i-1} \neq v_i = \cdots = v_j \neq v_{j+1}$, where $v_{-1} := v_{n+1} := -1$, we store the interval $[i,j]$ together with v_i in an array at place i and at place j.

(2) Create a node on level 1, label the node by $[0,n]$, $l=0$.

(3) $l := l+1$. For all nodes on level l check whether for its label $[i,j]$ (where $j-i=n+1-l$) some interval $[i',j]$, where $i' \leqslant i$ or $[i,j']$, where

$j \leqslant j'$ is stored. In the positive case relabel the node by the value stored together with the interval.

(4) Stop if $l = n+1$.

(5) Otherwise perform a maximal merging on the set of nodes on level l labelled by intervals (this step will be explained later in more detail).

(6) Consider all nodes on level l labelled by an interval. A node with label $[i,j]$ gets a left successor with label $[i,j-1]$ and a right successor with label $[i+1,j]$. The node itself is relabelled by x_{n-l+1}. Go to 3.

If Step 5 would be given for free, we again would obtain an algorithm where the size of the running time equals, for all nonconstant symmetric functions, the size of the constructed BP_1.

THEOREM 5. *Algorithm 2 constructs optimal BP_1s for symmetric Boolean functions. There exists an implementation such that the size of the running time equals the size of the constructed BP_1 and the storage space is of size $O(2^n)$. For another implementation the running time is by a linear factor larger but the storage space is of the same size as the constructed BP_1.*

Proof. By Lemma 1 we can conclude that the algorithm constructs optimal BP_1s for symmetric Boolean functions. For the implementation we have to consider only step (5). At first we use much storage space ($O(2^n)$) and handle numbers of length n. Let $z[i,j]$ be the number whose binary representation is $(v_i,\ldots,v_j)$. $z[0,n]$ can be computed in linear time. $z[i,j-1]$ and $z[i+1,j]$ have to be computed only in a situation where we know already $z[i,j]$.

$$\begin{aligned} z[i+1,j] &= z[i,j]/2 && \text{if } z[i,j] \text{ is even,}\\ &= (z[i,j]-1)/2 && \text{if } z[i,j] \text{ is odd,}\\ z[i,j-1] &= z[i,j] && \text{if } z[i,j] < 2^{j-i},\\ &= z[i,j]-2^{j-i} && \text{if } z[i,j] \geqslant 2^{j-i}. \end{aligned}$$

Thus the computation of the z-values does not increase the size of the running time of the algorithm. We can conclude that two nodes labelled $[i,j]$ and $[i',j']$ on the same level may be merged iff $z[i,j] = z[i',j']$. We may realize step (5) in the following way. We handle the nodes on level l labelled by intervals in an arbitrary order. We use a second array A of length 2^n. Nothing has to be done for $l=1$. If we handle for $l>1$ a node labelled $[i,j]$ we look at $A[z[i,j]]$. If this place is empty we store there $[i,j]$. If this place is occupied by $[i',j']$ we test whether $j-i=j'-i'$. In the negative case we store $[i,j]$ instead of $[i',j']$. In this and the preceding case we handle the first node on level l with z value equal to $z[i,j]$. If

$j-i=j'-i'$ we merge $[i,j]$ and $[i',j']$ by replacing the edge to $[i,j]$ by an edge to $[i',j']$.

The procedure is correct. For each node we need only a constant amount of time. For each node of the optimal BP_1 we do not create more than two nodes which may become eliminated (merged) afterwards. Thus by this procedure the size of the running time of our algorithm equals the size of the constructed BP_1 if f is not constant.

This approach may be criticized because of its large amount of storage space and its assumption that we may handle very large numbers very quickly. Therefore we will present shortly a second realization of step (5). Here our purpose is to describe briefly an efficient procedure using bit operations rather than looking for optimal procedures.

The binary representations of the values $z[i+1,j]$ and $z[i,j-1]$ are the substrings $(v_{i+1},\ldots,v_j)$ and $(v_i,\ldots,v_{j-1})$ of $v(f)$. By an easy bucket sort on the left endpoint of the interval labels we may merge all nodes having the same label. Afterwards we consider the strings $z[k,l]$ for all nodes for constant $l-k$. In the first step we divide the set into two subclasses of strings starting with 0 or with 1. These sets are treated one after another (or in parallel). They are divided in further subsets looking at the second, third,..., component. Empty sets and sets of one element need not be examined further. The set of nodes forming one class at the end of this classification is a maximal set for merging. Here we do not create more than l classes on level l since there are not more than l different intervals $[i,j]$ where $j-i=n+1-l$. Now it is easy to implement the procedure such that our claim holds. For the running time we count here even bit operations.

Q.E.D.

COROLLARY 1. $BP_1(f) \leqslant \sum_{1\leqslant l\leqslant n} \min\{l,\ 2^{n-l+2}-2\} = n^2/2 - n\log n + O(n)$ *for all symmetric* f: $\{0,1\}^n \to \{0,1\}$.

Proof. We count the maximum number of nodes on level l in an optimal BP_1 for a symmetric Boolean function f. At level l we have tested $l-1$ variables and the value vectors of the subfunctions we have to compute have length $n-l+2$. There exist only $2^{n-l+2}-2$ nonconstant vectors of this length. On the other hand these vectors are subvectors of $v(f)$ which has length $n+1$. There exist at most l positions for the beginning of a subvector of length $n-l+2$. Thus the upper bound follows. It is easy to show that the upper bound equals $n^2/2-n\log n+O(n)$. Q.E.D.

We omit the exercise of evaluating the upper bound exactly. We compute the BP_1-complexity of the exactly half function and of the majority function. Finally we show that the upper bound of Corollary 1 is exact for some symmetric function f^*. Therefore f^* is the most complex symmetric

Boolean function with respect to BP_1s. Furthermore we obtain the proposed trade-offs since it is well known (see Savage [11]) that all symmetric Boolean functions have linear network complexity.

DEFINITION 5. The majority function f^n_{maj} computes 1 iff the input contains at least $\lceil n/2 \rceil$ ones.

THEOREM 6. $BP_1(f^n_{\text{exh}}) = k^2 + 2k$ *if* $n = 2k$, $BP_1(f^n_{\text{exh}}) = k^2 + 3k + 1$ *if* $n = 2k+1$, $BP_1(f^n_{\text{maj}}) = k^2 + k$ *if* $n = 2k$, $BP_1(f^n_{\text{maj}}) = k^2 + 2k + 1$ *if* $n = 2k+1$.

Proof. We use Algorithm 2 for the construction of optimal BP_1s. Let us consider f^n_{exh}, where $n = 2k$. On level $l \in \{1,..., k+1\}$ we have l nodes distinguishing the inputs with 0,..., $l-1$ ones in $x_1,..., x_{l-1}$. If we have found more than k ones the function computes 0. The same holds if we have found less than $k-m$ ones and only m variables are not tested. Thus we have on level $k+l$ $(2 \leqslant l \leqslant k)$ exactly $k+2-l$ nodes distinguishing the inputs with $l-1,..., k$ ones in $x_1,..., x_{k+l-1}$. Thus $BP(f^n_{\text{exh}}) = 1 + \cdots + k + (k+1) + k + \cdots + 2 = k^2 + 2k$. The other results follow by similar considerations. Q.E.D.

DEFINITION 6. A de Bruijn sequence is a 0–1-sequence of length $2^k + k - 1$ containing any 0–1-sequence of length k exactly once as subsequence.

It is well known that de Bruijn sequences exist and are easy to construct (see, e.g., Knuth [6]).

THEOREM 7. *Let f^*_n be a symmetric function whose value vector is a de Bruijn sequence of length $n+1 = 2^k + k - 1$. Then* $BP_1(f^*_n) = \sum_{1 \leqslant l \leqslant n} \min\{l, 2^{n-l+2} - 2\}$.

Proof. The upper bound follows from Corollary 1. Since all subsequences of length k of a de Bruijn sequence are different there is no constant subsequence of a length larger than k. Also all subsequences of length larger than k are different. The function has $n = 2^k + k - 2$ variables. At level $2^k - 1$ we have tested $2^k - 2$ variables. Since k variables are left the value vectors of the subfunctions we have to compute have length $k+1$ and are different and nonconstant. This holds also for the levels $l \leqslant 2^k - 1$. Thus we have on level $l \leqslant 2^k - 1$ at least l nodes in an optimal BP_1 for f^*_n. At level $l = 2^k + m$, where $0 \leqslant m \leqslant k-2$ there are $k-m-1$ variables which we have not tested yet. The value vectors of the subfunctions have length $k-m$. A de Bruijn sequence obviously contains all $2^{k-m} - 2$ nonconstant sequences of length $k-m$ as subsequences. Thus we have at these levels in

an optimal BP_1 at least $2^{k-m}-2=2^{n-l+2}-2$ nodes. The lower bound follows since we have shown that we have on level l at least $\min\{l, 2^{n-l+2}-2\}$ nodes. Q.E.D.

ACKNOWLEDGMENT

I would like to thank an unknown referee for suggesting the definition of symmetric Boolean functions by de Bruijn sequences.

RECEIVED: November 17, 1983; ACCEPTED: October 31, 1984

REFERENCES

1. BOLLOBÁS, B. (1978), "Extremal Graph Theory," Academic Press, New York.
2. BORODIN, A., DOLEV, D., FICH, F. E., AND PAUL, W. (1983). Bounds for width two branching programs, *in* "15th Annu. Sympos. on Theory of Computing," pp. 87–93.
3. COBHAM, A. (1966), The recognition problem for the set of perfect squares, *in* "7th Sympos. on Switching and Automata Theory," pp. 78–87.
4. HEDTSTÜCK, U. (1983) On the argument complexity of multiply transitive Boolean functions, Univ. of Stuttgart, preprint.
5. ILLIES, N. (1978), A counterexample to the generalized Aanderaa–Rosenberg conjecture, *Inform. Process. Lett.* **7**, 154–155.
6. KNUTH, D. F. (1981), "The Art of Computer Programming," Vol. 2, Addison–Wesley, Reading, Mass.
7. MASEK, W. (1976), "A fast Algorithm for the String Editing Problem and Decision Graph Complexity," M.Sc. thesis, MIT.
8. NECHIPORUK, E. I. (1966), A Boolean function, *Soviet Math. Dokl.* **7**, 999–1000.
9. PUDLÁK, P. AND ZÁK, S. (1983), Space complexity of computations, Univ. of Prague, preprint.
10. RIVEST, R. L. AND VUILLEMIN, J. (1976), On recognizing graph properties from adjacency matrices, *Theoret. Comput. Sci.* **3**, 371–384.
11. SAVAGE, J. E. (1976), "The Complexity of Computing," Wiley, New York.
12. SPIRA, P. M. (1971), On time-hardware complexity tradeoffs for Boolean functions, *in* "Proc. of 4th Hawaii Int. Sympos. on System Sciences," pp. 525–527.
13. WEGENER, I. (1984), On the complexity of branching programs and decision trees for clique functions, *J. Assoc. Comput. Mach.*
14. YAO, A. C. (1983), Lower bounds by probabilistic arguments, *in* "24th Annu. Sympos. on Found. of Comput. Sci." pp. 420–428.