

TEL AVIV UNIVERSITY
THE IBY AND ALADAR FLEISCHMAN FACULTY OF ENGINEERING
Department of Electrical Engineering - Systems

**ON THE EXPECTED
CLASSIFICATION SPEED OF
BOOLEAN FUNCTIONS**

Thesis submitted toward the degree of
Master of Science in Electrical and Electronic Engineering
in Tel-Aviv University

by

Amir Rosenfeld

October 2004

TEL AVIV UNIVERSITY
THE IBY AND ALADAR FLEISCHMAN FACULTY OF ENGINEERING
Department of Electrical Engineering - Systems

**ON THE EXPECTED
CLASSIFICATION SPEED OF
BOOLEAN FUNCTIONS**

Thesis submitted toward the degree of
Master of Science in Electrical and Electronic Engineering
in Tel-Aviv University

by

Amir Rosenfeld

This research was carried out at Tel-Aviv University
in the Department of Electrical Engineering - Systems,
Faculty of Engineering
under the supervision of Dr. Dana Ron

October 2004

Contents

1	Introduction	1
1.1	The Basic Problem	1
1.2	Binary Decision Diagrams	3
1.3	Decision Maps	4
1.4	The Problems Studied in this Thesis	6
1.5	Background work	8
2	Preliminaries	9
2.1	Binary Decision Diagrams (BDDs)	9
2.2	Oblivious BDDs (OBDDs)	9
2.3	Decisive Subset Assignments and Deciding Hypercubes	10
2.4	Decision Trees	11
2.5	Decision Maps	11
3	Expected Decision Depth	13
3.1	Computing the Expected Decision Depth for Decision-Tree Functions is #P-Hard.	14
3.2	Approximating EDD for Decision Trees	19
4	The Oblivious Model	21
4.1	Introduction	21
4.2	Reduction to a Scheduling Problem	22
5	The Non-Oblivious Model	25
5.1	Introduction	25
5.2	An Algorithm (Sub-Optimal)	26

6 Further Research	29
References	30
A Technical Appendix	31

List of Figures

1	BDD of $g(x) = x_1x_2 \vee \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2\bar{x}_3$	4
2	BDD, OBDD and Decision Tree of $f(x) = x_1x_2 \vee \bar{x}_1\bar{x}_3 \vee \bar{x}_1x_2x_3$	12
3	Pruning the decision tree from figure 2 according to $x_2 = 1$	12
4	The two types of “waiting” gadgets used to construct T_1 and T_2	14
5	Transforming the CNF term $(x_1 \vee x_2 \vee x_3)$ into a subtree of T_1 and T_2	15
6	The structure of decision trees T_1 and T_2	16
7	A decision map in $\{0, 1\}^3$. The decision zones are bold. This decision map cannot be realized by a decision tree (whose leaves correspond to the zones).	26
8	An example of mapping of a decision map to a complete graph. Each edge can be labeled with at least one variable over which its two end nodes disagree. Each node is marked with the monomial describing its decision zone and the output value of the decision zone.	27
9	The remaining graph and monomials after the variables’ values $x_1 = 0$ and $x_3 = 1$ have been determined.	27

Abstract

In this thesis we initiate the study of the following new computational problem: Given a (Boolean) function f over $\{0,1\}^n$, we would like to design a rule for querying the variables of the inputs $p \in \{0,1\}^n$ so as to minimize the expected number of queries required to determine $f(p)$. The expectation is taken with respect to the uniform distribution over the inputs, and the rule may either be oblivious or non-oblivious (adaptive).

We start by observing that if there are no restrictions on the function f and the form in which it is given to the algorithm, then the problem is NP-hard. We hence consider restricted forms of functions and several variants of this problem.

In particular we first consider the case in which f is a (polynomial-size) decision tree, and study the following, seemingly more basic question: Suppose we are given an ordering of the input variables and we are required to compute the expected number of variables that have to be queried in order to obtain the value of the function when restricted to the given order. We prove that this problem is #P-hard. We observe though that an approximate answer can be obtained efficiently using a simple sampling algorithm.

We then turn to variants of the problem in which the function f is given in the form of a partition of the input space into (disjoint) “monochromatic” (with respect to f) hypercubes. The querying rule is required to “respect” this partition in a sense that we formalize in the sequel. We obtain results both for oblivious and for non-oblivious rules.

1 Introduction

1.1 The Basic Problem

Consider a Boolean classification machine. It receives a binary description of an object and classifies it into one of two sets. The machine consists of a processor and random access memory. It implements a Boolean function (the classification function) that operates on the object and usually depends on many details in its description. Normally, the processor is given a pointer to a memory space where the description of the object resides and is expected to return its classification. But, how much of the object must be seen by the processor for the classification process? This is the core question in our research, where we formalize it below. The result has direct implications on the classification speed in current computer architectures because the access time to the object's data is the bottleneck of the process.

For any classification function and for any binary description of a classified object, there is a minimal number of binary variables that must be queried in the description of the object before a resolution can be made. Consider the following function: $g(x) = x_1x_2 \vee \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2\bar{x}_3$. Obviously, given an object $p = (p_1, p_2, p_3) \in \{0, 1\}^3$ we must query at least one variable of p to decide on the output $g(p)$. This is because g is not a non-satisfiable ('all-zero') function, nor is it the 'all-one' function. Our response depends on the input. Generally we need not query all of the object's variables in order to make a decision. If we were to classify the object $p = (1, 1, 0)$ we could query p_2 first, and then, seeing that it is positive, we could query p_1 and make a resolution. The binary variable p_3 is irrelevant to the decision in this case. Now consider having to classify the object $q = (1, 0, 0)$: If we begin by querying q_2 , then we will have to query both q_1 and q_3 in order to make a resolution, but if we begin with querying q_1 and then we query q_3 we can make a resolution about q based on those two bits alone. Of course, a classifier does not know if it is faced

with p or q , and it also cannot begin by querying the first and second bits of the object simultaneously - it must choose one of them, as it must choose at every stage of the classification.

The classifier does not have any apriori knowledge about the classified object before the classification begins. The only thing that is known is the probability distribution on the objects. Here we assume that the probability distribution does not vary in time and the occurrence of objects is independently identically distributed. Hence, given a function f we would like to design a querying rule F , such that the expected number of queries it performs, taken over a uniform choice of the input, is as small as possible.

A querying rule F may have two forms: It may be *oblivious* or it may be *non-oblivious (adaptive)*. In the first case the rule can simply be described as a fixed ordering of a subset of the variables that determines the variable-query order regardless of the answers to the queries. In this case what varies with different inputs is only after how many queries the value of the function is fully determined. In the non-oblivious case, at each point, the next variable queried may depend on the values of the variables that were previously queried. Thus a non-oblivious rule may be more efficient, but an oblivious rule is simpler and has many practical applications. For example, consider a classification process, which takes the form of a client-server model: The clients, who do not possess processing power, stream descriptions of objects to a server, who only replies with the final decision per object, and the target is to minimize the mean response time.

We start by observing that unfortunately, in either case, the problem of finding an optimal rule is hard in general.

Observation 1 *Finding an optimal querying rule for a given Boolean function is NP-Hard.*

The reason for this hardness is simply that such a rule could be used to decide whether a given CNF formula is unsatisfiable: Given a formula ϕ , if the optimal

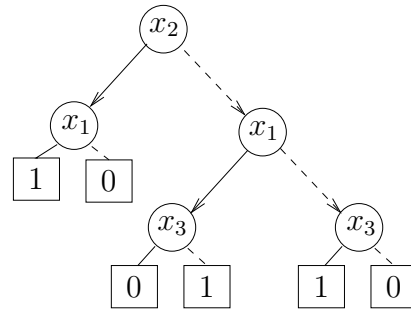
rule makes at least one query then we know that ϕ is not a tautology. Namely, some assignments satisfy it and some do not, and hence it is satisfiable. If the optimal rule makes no queries, then ϕ is either the all-zero function (it is unsatisfiable) or it is the all-one function. We can distinguish between the two cases by evaluating ϕ on an arbitrary input point.

1.2 Binary Decision Diagrams

We have said that the number of variables that need to be queried is a lower bound on the number of processor operations, but for it to have a meaning it must also be related to the upper bound. By using a Binary Decision Diagram (BDD) to implement the classifier this becomes also an upper bound. For our purposes a BDD is a graph with any number of decision nodes and output nodes. The output nodes are labeled with one of the possible output values, and in our case either ‘0’ or ‘1’. One of the nodes is the single entry point. Each decision node queries a single variable and according to it decides on one of two possible emanating paths. Whenever a particular variable’s value enables a resolution, there is an edge to one of the output nodes. See Figure 1 for an example BDD of the function g above.

We are interested in BDD representations because they allow us to analyze the expected number of variable-queries per computation. A BDD implementation of a function requires each variable to be queried at most once. The current decision node represents all the relevant information that could have been extracted from all previously queried input variables, and therefore no operations on previously acquired intermediate results of any kind is required. In fact, in a BDD calculation of f , for each node along a computation path only a limited amount of operations is necessary: (1) Fetch that node from memory, (2) Fetch the required object description bit and (3) Make a decision. Thus, the number of operations and memory accesses is bounded from above by a constant multiple of the number of required input bits.

The drawback of the model is that generally BDDs can be of a very large size



- 1 - Output node with its output value
- x_3 - Decision node with its queried variable

Figure 1: BDD of $g(x) = x_1x_2 \vee \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2\bar{x}_3$.

in relation to other representations of a given function. Note, however, that we are interested in BDDs for analyzing the expected number of queries and the variable ordering that will minimize it, and not necessarily as a means of actually implementing the classifier. Even if a BDD is used, one does not necessarily have to keep the entire decision diagram in memory. In particular, an oblivious rule is simply described by an ordering of a subset of the variables. The actual classification of objects given the values of the queried variables may be done by a procedure with a more compact description than the corresponding (oblivious) BDD.

1.3 Decision Maps

Given a Boolean classification function, what can we say about the minimal expected number of bits that need to be queried to make a resolution? Obviously if n is the number of binary variables in the object's description then n is also an immediate upper bound for the number of required queries to the input variables. Can we find a lower bound?

Observe that any given function f can be described by a partition of the input space, $\{0, 1\}^n$, into disjoint hypercubes, such that for each hypercube the value of f on all points in the hypercube is constant. We shall refer to such a partition as a

decision map and to each “monochromatic” hypercube, as a *decision zone*. We can attribute to each such decision map $\mathcal{M} = \{H_1, \dots, H_\ell\}$ the value:

$$\mathcal{H}(\mathcal{M}) = \sum_{i=1}^{\ell} \Pr(H_i) \cdot \log(1/\Pr(H_i)) \quad (1)$$

where $\Pr(H_i)$ is the probability that a point selected uniformly in $\{0, 1\}^n$ belongs to H_i . We refer to $\mathcal{H}(\mathcal{M})$ as the *entropy* of the decision map \mathcal{M} , Namely, we view the decision map as a distribution and obtain its entropy. Note that if we denote by $V(H_i)$ the number of variables that determine the hypercube H_i then $\Pr(H_i) = 2^{-|V(H_i)|}$ and so $\log(1/\Pr(H_i)) = |V(H_i)|$.

Clearly there are many possible decision maps for any given function f . In particular, each querying rule F for f induces a decision map \mathcal{M}_F , where the decision zones in the decision map are determined by the paths in the BDD which implements f using the querying rule F . For each point $p \in \{0, 1\}^n$, the number of queries performed using F in order to determine $f(p)$ is simply $|V(H)| = \log(1/\Pr(H))$, where H is the decision zone in \mathcal{M}_F to which p belongs. Hence we immediately get the following (implicit) lower bound:

Observation 2 *The minimum expected number of queries required for f is at least the minimum over all decision maps \mathcal{M} for f of $\mathcal{H}(\mathcal{M})$.*

Note that since not all decision maps can be exactly realized by BDDs, the minimum over all decision maps may be a strict lower bound.

We further use decision maps in some of our results in order to constrain the set of classifiers that we deal with. Consider a classifier viewed as a BDD. As noted above, its computation paths form a decision map. By definition, points within the same decision zone in this map follow the same computation path in the BDD, and points lying in different decision zones follow different computation paths in the BDD. A classifier is said to *respect* a decision map \mathcal{M} if its computation paths create a decision map \mathcal{N} such that for each decision zone of \mathcal{N} there is a decision

zone of \mathcal{M} that includes it. This means that all decision zones of the classifier are subsets of the decision zones of the respected decision map.

A direct consequence of Observation 1 is that if we wish to make progress in finding a querying rule, we must simplify the problem. We can either try to find an approximation algorithm for finding the querying rule in the general case, or we could restrict the solution to being competitive against a limited set of solutions and try to find a competitive solution for the restricted case. The latter is what we have done. Hence, we only compare the performance of classifiers that *respect a given decision map*. The reasoning is that we can expect that any (if at all) simplification of the classification function that can be made is done before the search for the fastest classifier. For a simple illustration of the simplified problem, let us describe two decision maps of the ‘all-one’ function. In the first case the function is described as $allone(x) = x_1 \vee \bar{x}_1$, where $\vec{x} \in \{0, 1\}^n$. Thus we are actually provided with a partition of the input space into two decision zones of $n - 1$ dimensions each: x_1 and \bar{x}_1 . All classifiers that respect this decision map must query at least the variable x_1 . Therefore, given the above decision map of ‘all-one()’, an optimal classifier must query that variable. If, on the other hand, we were told that *allone* is the ‘all-one’ function (i.e., given the decision map that includes a single hypercube), our response would be that we need not query any variable in an implementation of *allone*. Both results are optimal with respect to the decision map provided.

1.4 The Problems Studied in this Thesis

Following the discussion above we investigated the problems described below.

Expected Decision Depth. We start by considering the following basic problem. Given a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ and a fixed variable querying permutation, calculate the expected number of queries required for making a decision given this querying order.

Based on how Observation 1 is established, it directly follows that this problem

is NP-hard as well when there are no restrictions on the form of f .¹ We hence consider the case in which f is restricted, and in particular is given in the form of a decision tree.

We are able to prove a fairly strong negative result in this case: This problem is #P-hard for decision trees. The difficulty is with computing the *exact* expected decision depth: We show that there exists a simple approximation algorithm for this problem (in the case of decision trees), which uses sampling.

While the above hardness result does not directly imply that finding an optimal query order of the variables is hard when f is a decision tree function, it seems to indicate that this may be the case. Hence we continue our study by considering variants of the problem.

Respecting a Decision Map in the Oblivious Model. Given a decision map $\mathcal{M}(f)$ of a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$, we would like to find a fixed permutation π of the input variables such that a classifier that adheres to this permutation (i.e., queries the variables in the order determined by π) and respects $\mathcal{M}(f)$, minimizes the expected number of queried variables among all classifiers that respect $\mathcal{M}(f)$. Namely, here we would like to be competitive against a certain class of classifiers. Observe that the restriction to respecting the given decision map means that the number of queries performed for an input p (given the permutation π) may be more than the number of queries actually necessary to determine $f(p)$. Rather, if H is the hypercube in $\mathcal{M}(f)$ to which p belongs, and $V(H)$ is the subset of variables defining H , then it is the number of variables queried until all variables in $V(H)$ are queried.

We prove that there exists a solution with an average number of queries that is at most $2OPT$, where OPT is the minimal average number of variable-queries made by any classifier respecting the decision map. The result is based on a reduction to

¹Note that even if we are given one specific input and are required to determine the number of queries necessary for that input then the problem remains hard for a general f .

a known scheduling problem.

An Algorithm for the Non-Oblivious Model. Given a decision map $\mathcal{M}(f)$ of a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$, we would like to find a rule for determining the variable that is to be queried at step j according to the values of the bits queried in steps $1, \dots, j - 1$. Here we do not obtain a result that is competitive with the best rule that respects the decision map, but rather describe a rule and analyze its performance as a function of properties of the decision map. Recall that any rule that respects the decision map performs an expected number of queries that is at least the entropy of the decision map (that is, $\sum_{H \in \mathcal{M}(f)} \Pr(H) \cdot \log(1/\Pr(H))$). We describe how to obtain a rule that performs an expected number of queries that is upper bounded by $\sum_{H \in \mathcal{M}(f)} \Pr(H) \cdot O(\log^2(1/\Pr(H)))$ (and respects the decision map).

1.5 Background work

The origins of decision maps may be found in existing lower bound techniques for communication complexity, where the number of monochromatic rectangles in a function is used to produce a lower bound for size of the resulting BDD (see e.g., [KN96]). Our analysis utilizes BDDs. BDDs have been extensively researched, but the focus of the research was predominantly concerned with their size, i.e., the number of nodes, not the depth nor the expected length of computation paths. When the depth was considered, it was in the context of questions regarding computational complexity where the total size of the BDD is bounded and the question was whether one could create a BDD of polynomial size and small depth for computing a function. In [BSSW03], Bollig et. al. discuss lower bounds for BDDs and prove that BDDs of limited size cannot represent too many functions. Communication complexity and also other methods are used to show that the size of oblivious BDDs of some functions is large. Looking into closely related fields, we could mention Hastad's switching lemma [Has86], which was used to prove a lower bound on the depth of polynomially bounded size circuits.

2 Preliminaries

In this section we further formalize some of the notions described in the introduction and introduce some useful notation.

2.1 Binary Decision Diagrams (BDDs)

Let $f : \{0, 1\}^n \rightarrow \{0, 1\}$ be a Boolean function. A Binary Decision Diagram is a non-uniform program for computing f based on a directed acyclic graph with two types of nodes:

Decision nodes. A decision node v has two successor nodes $zero(v)$ and $one(v)$, where based on the value of some input variable x_i , the computation path continues to one of them. We define $index(v) = i$, the index of the input variable relevant to the node.

Output nodes. Also known as sinks. An output node v has no successors. It has a $value(v) \in \{0, 1\}$ representing the result of the computation.

Any computation of $f(x_1, x_2, \dots, x_n)$ using a BDD model of f can be represented by a path from the root node to some output node. In an actual implementation of a BDD, each decision node is interpreted as a single “if” instruction, and each output node to a “return”.

2.2 Oblivious BDDs (OBDDs)

For a fixed permutation π , we can order the input variables according to the permutation: $(x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(n)})$. A π -OBDD is defined to be a BDD obeying the following restriction: For any decision node v , if $index(v) = \pi(i)$ then $index(zero(v)) = index(one(v)) = \pi(i+1)$. (It may be the case that for every input point the value of the function is already determined after querying $x_{\pi(1)}, \dots, x_{\pi(\ell)}$ where $\ell < n$, but for simplicity we think of a complete ordering of the variables rather than an ordering of a subset of the variables.) For each node v we let

$depth(v) = \pi^{-1}(index(v))$. For every Boolean function f and a permutation π , there exists a minimal-sized π -OBDD, unique up to isomorphism. For a more comprehensive discussion of Binary Decision diagrams see [B86, W00].

2.3 Decisive Subset Assignments and Deciding Hypercubes

Definition 1 *Given a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ and a set of input variables $X = \{x_1, x_2, \dots, x_n\}$, a partial assignment is a function $\tau : \{1, 2, 3, \dots, n\} \rightarrow \{0, 1, *\}$.*

We define the set of variables restricted by the partial assignment τ :

$$X_\tau = \{x_i \mid i \in [1, n], \tau(i) \neq *\} \quad . \quad (2)$$

We also define the set of all input points commensurate with it:

$$H(\tau) = \{x \in \{0, 1\}^n \mid \forall_{i \in [1, n]} \tau(i) \neq * \Rightarrow x_i = \tau(i)\} \quad . \quad (3)$$

The set $H(\tau)$ is a hypercube of dimension $m = |\{i \mid \tau(i) = *\}|$ within the n -dimensional space of the entire input set. The number of variables whose values are restricted by τ is given by $n - m$ and is defined to be the size of the partial assignment. The set of variables X_τ is said to *decide* the hypercube $H(\tau)$.

Given an input point \vec{a} such that $x_1 = a_1, x_2 = a_2, \dots, x_n = a_n$, knowing the values of the variables in X_τ allows us to determine whether \vec{a} lies inside $H(\tau)$ or outside of it.

Definition 2 *Decisive Partial Assignment.* *The partial assignment τ is said to be decisive with respect to f if and only if there exists a value $b \in \{0, 1\}$ such that for every $x \in H(\tau)$, $f(x) = b$. That is, f is constant on $H(\tau)$.*

Definition 3 *Decision Zone.* *The hypercube $H(\tau)$ is said to be a decision zone, if and only if τ is decisive.*

For example, in a DNF formula, every one of the monomial terms is a Hypercube Decision Zone (HDZ). The set of points for which the function returns a ‘0’ (i.e., the points not covered by the monomials in the DNF) are also partitioned into HDZs according to the computation paths. Given a function f , we can partition its entire input space into a set of disjoint HDZs. Any algorithm computing f , can terminate once it determines that its input is commensurate with a decisive partial assignment.

2.4 Decision Trees

A decision tree is a BDD with indegree at most 1. We say that a decision tree *respects* a permutation π if it is also a π -OBDD. A subtree of the decision tree is *homogenous* if all the output nodes reachable from its root have the same value. Homogenous subtrees can be reduced to a single output node without altering the function. Decision trees are useful in our context because their output nodes define a decision map. The analysis of the computation paths is then simple, since each path is distinctly differentiated from the other paths by its terminating output node.

Definition 4 *Reduced Decision Trees.* *A reduced decision tree is a decision tree such that each homogeneous subtree has at most 1 node and this node is an output node. That is, it has no un-reduced homogenous subtrees.*

Given a decision tree T and a partial assignment τ , we can prune all the computation paths of T that are not commensurate with τ to get a modified tree T^τ such that all of its paths are reachable by at least one input point in $H(\tau)$. For a given tree T of the function f and assignment τ , τ is a decisive assignment of f if and only if T^τ is homogenous. See Figures 2 and 3 for an example.

2.5 Decision Maps

A *decision map* is a set of disjoint HDZs covering the entire input space.

Definition 5 *Respecting a Decision Map.* *We say that a BDD, G , respects a decision map \mathcal{M} , if for any two HDZs $H_1, H_2 \in \mathcal{M}$ and for any two points*

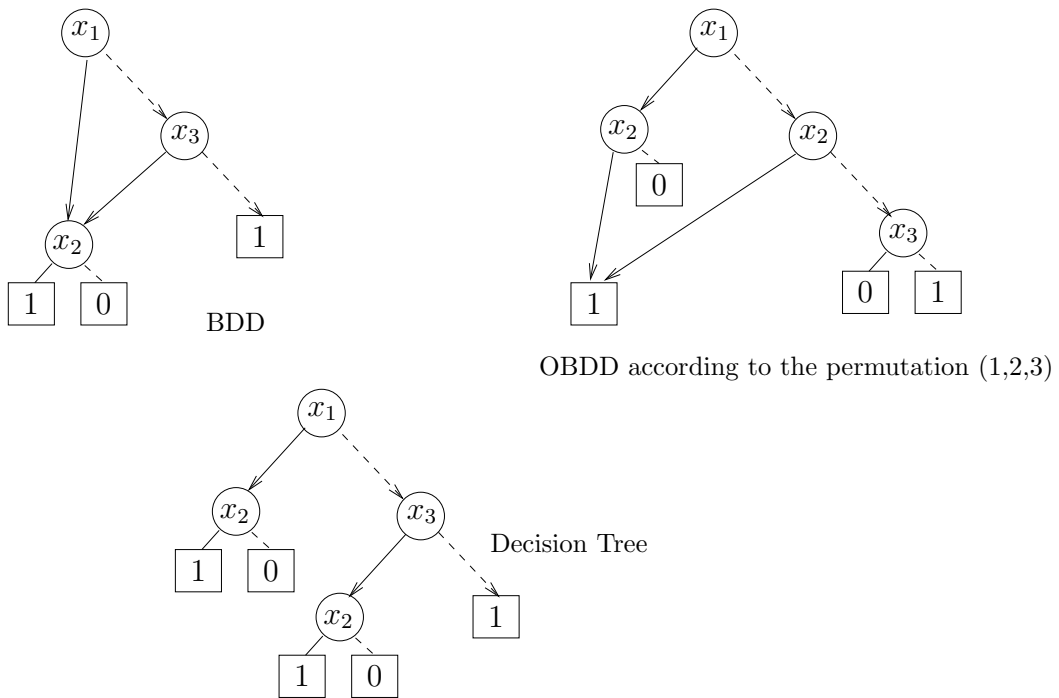


Figure 2: BDD, OBDD and Decision Tree of $f(x) = x_1x_2 \vee \bar{x}_1\bar{x}_3 \vee \bar{x}_1x_2x_3$.

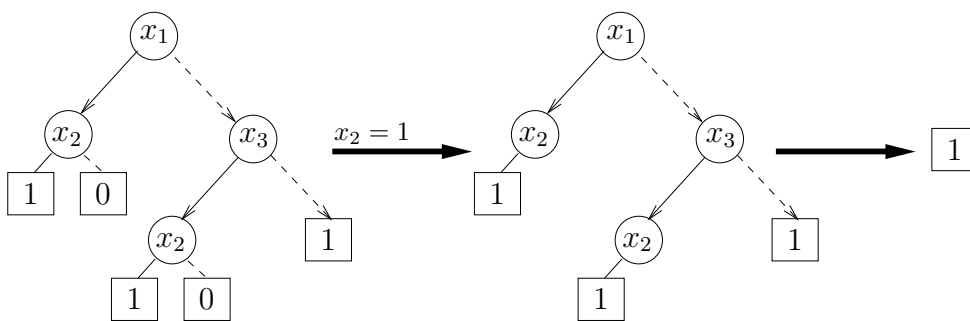


Figure 3: Pruning the decision tree from figure 2 according to $x_2 = 1$.

$p_1 \in H_1, p_2 \in H_2$, a simulation of the calculation of $f(p_1)$ using G traverses a path that is different from the path traversed in G during the calculation of $f(p_2)$.

In other words, G respects \mathcal{M} if it respects the difference between the HDZs in \mathcal{M} . It may be noted that partitioning HDZs of \mathcal{M} into smaller subspaces conserves the “respecting” property. Looking at Figure 2 you can see that the BDD, the OBDD and the Decision Tree all respect the same decision map outlined by the sum of products definition of the function $f(x) = x_1x_2 \vee \bar{x}_1\bar{x}_3 \vee \bar{x}_1x_2x_3$ (the implied 0 HDZs are: $x_1\bar{x}_2$, $\bar{x}_1\bar{x}_2x_3$). However, if we were to rewrite the function like this: $f(x) = x_2 \vee \bar{x}_1\bar{x}_2\bar{x}_3$ (the implied 0 HDZs are \bar{x}_2x_1 , $\bar{x}_2\bar{x}_1x_3$), in effect outlining a different decision map of the same function (the two are logically identical), and then produce a BDD according to this decision map - it will not respect the decision map in Figure 2. If we assume the inputs to be uniformly distributed then the second decision map will yield a classifier with a lower a-priori expected number of queries.

3 Expected Decision Depth

Let $f : \{0, 1\}^n \rightarrow \{0, 1\}$ be a Boolean function in some standard form and π a permutation defining the variable querying order (perhaps the result of some optimization algorithm seeking to minimize the number of queried variables). We are not required to respect any decision map. Are we able to calculate the expected number of variable-queries for deciding f ?

Definition 6 *Expected Decision Depth.* Given a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ represented as a decision tree (of polynomial size) and a permutation π , we define the functional $EDD(f, \pi)$ as follows:

For any input point $q \in \{0, 1\}^n$ let $depth_{f, \pi}(q)$ be the minimum number of queries required to decide $f(q)$ when the queries are ordered according to π . Then,

$$EDD(f, \pi) = \sum_{q \in \{0, 1\}^n} depth_{f, \pi}(q) \cdot 2^{-n} \quad (4)$$

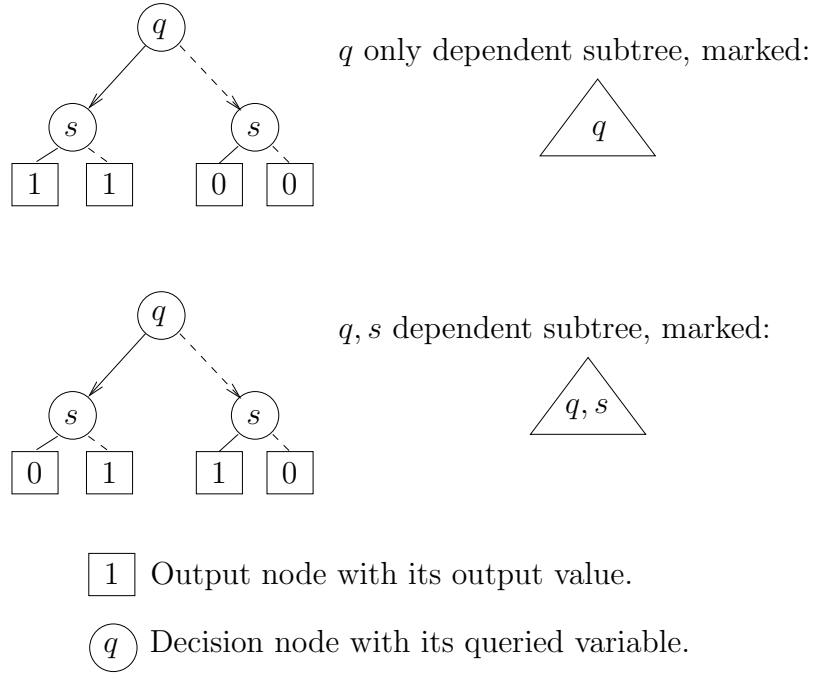


Figure 4: The two types of “waiting” gadgets used to construct T_1 and T_2 .

3.1 Computing the Expected Decision Depth for Decision-Tree Functions is #P-Hard.

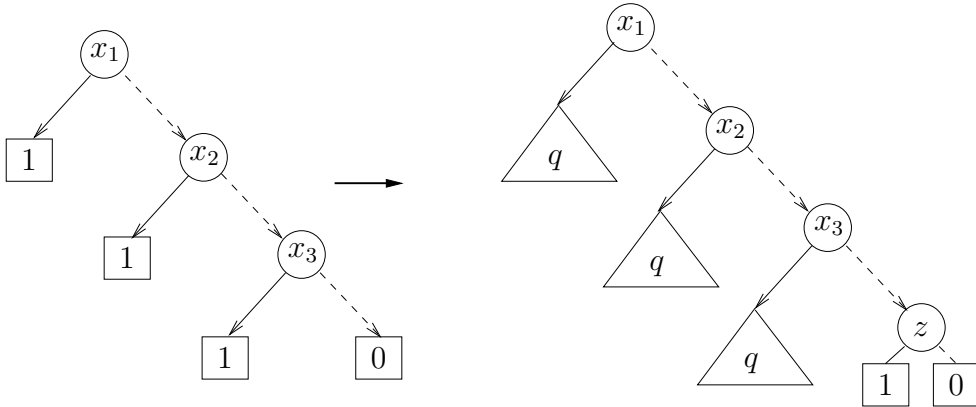
In all that follows, when we refer to a decision tree T , we mean both the function it computes and the tree structure itself. Our main theorem in this section is the following:

Theorem 1 *Given a decision tree T and an arbitrary variable querying permutation π , calculating $EDD(T, \pi)$ is #P-hard.*

Proof of Theorem 1. We first establish the hardness of the following related problem:

Theorem 2 *EDD Equality* *Given two decision trees T_1 and T_2 and a variable querying permutation π , computing $(EDD(T_1, \pi) - EDD(T_2, \pi))$ is #P-hard.*

Constructing a subtree of T_1 :



Constructing a subtree of T_2 :

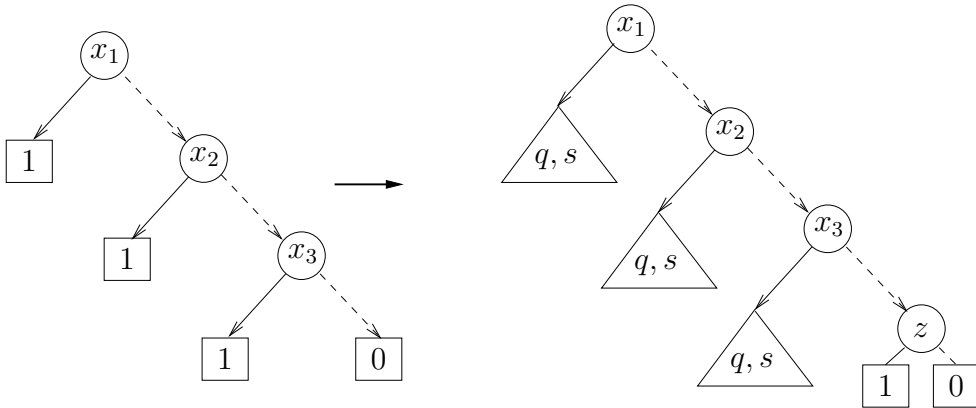


Figure 5: Transforming the CNF term $(x_1 \vee x_2 \vee x_3)$ into a subtree of T_1 and T_2 .

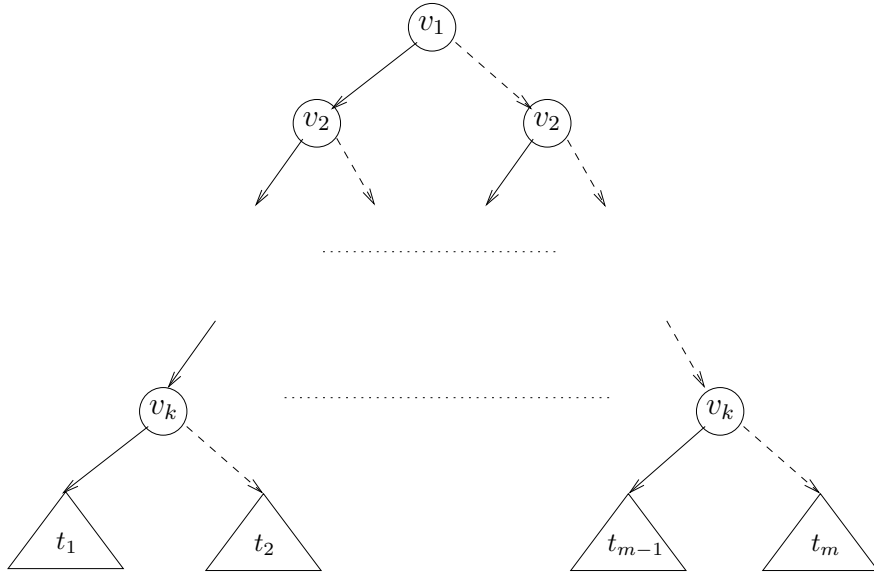


Figure 6: The structure of decision trees T_1 and T_2 .

Proof of Theorem 2. We reduce #SAT (counting the number of satisfying assignments of a formula) to the problem described in Theorem 2: Given a CNF function f with m clauses over n variables x_1, x_2, \dots, x_n we construct two different decision trees and a permutation π , and then prove that the number of satisfying assignments of f can be computed exactly using the difference between the expected decision depths of the two trees (with respect to π).

We begin by describing two gadgets used in the construction of the trees (see Figure 4). The q -gadget is a subtree that becomes homogenous only after the variable q is queried. The (q, s) -gadget is a subtree that becomes homogenous only after both q and s have been queried. Both may be called “waiting” gadgets, since they force the decision to wait until some variables are queried. The decision tree T_1 is constructed as follows: For each term in f choose any ordering of the variables in the term, and create a sub-tree rooted by the first variable (see the top of Figure 5). Instead of each output node with the value ‘1’, place a q -gadget. Instead of every ‘0’ output node (each sub-tree has exactly one such leaf), place a decision node marked with the new variable z . Each z node has two output node children: $value(zero(z)) = 0$

and $value(one(z)) = 1$. We refer by t_i both to the term t_i and its resulting subtree. For constructing a single tree out of all the sub-trees, we construct a tree out of $k = O(\log m)$ new tree variables, v_1, v_2, \dots, v_k and hang the individual sub-trees in place of its leaves (see Figure 6).

We construct the second decision tree T_2 the same way T_1 was constructed except that instead of the q -gadgets, we use (q, s) -gadgets (see the bottom of Figure 5). For completeness of the reduction we note that the size of T_1 and T_2 is polynomial (actually it is linear) with respect to the number of clauses in the original formula.

The variable set for T_1 and T_2 is composed of $x_1, \dots, x_n, v_1 \dots v_k, q, s$ and z . We set the variable querying order π as follows: First we query the first n original variables x_1, \dots, x_n in some arbitrary order, then we query q , then s , then the variables v_1, \dots, v_k in some arbitrary order and finally we query z . Now, assuming we have an algorithm to compute expected decision depth, we compute the expected decision depth of T_1 and T_2 : $EDD(T_1, \pi)$ and $EDD(T_2, \pi)$. We claim that f is satisfiable if and only if $EDD(T_1, \pi) < EDD(T_2, \pi)$ and furthermore, $EDD(T_1, \pi) - EDD(T_2, \pi)$ equals the number of satisfying assignments of f divided by 2^n .

Definition 7 Assignment Restriction. *An assignment $B = \{(x_1 = b_1), (x_2 = b_2), \dots, (x_m = b_m)\}$ restricted to a set $X' \subseteq \{x_1, x_2, \dots, x_n\}$ is another assignment $B' = \{(x_i = b_i) \mid x_i \in X'\}$.*

Lemma 1 *For any fixed assignment τ to all the variables in π , if the restriction of τ to x_1, \dots, x_n , denoted τ' , is a satisfying assignment of the original CNF f , then the number of variables that must be queried in order to determine $T_1(\tau)$ when the order of the queries is determined by π is exactly $n + 1$, and the number of variables that must be queried to determine $T_2(\tau)$ is exactly $n + 2$.*

Proof of Lemma 1. A CNF, being a conjunction of clauses, is satisfied by an assignment if and only if all clauses restricted to the assignment are satisfied. That is, at least one of the literals in every term must be satisfied by the assignment. By the construction of the sub-trees, if τ contains a satisfying assignment of a term t_i ,

then a traversal of sub-tree t_i using the values in τ must reach a waiting gadget (q or (q, s) , depending on the tree). To prove that point, we notice that if the latter does not happen, we must reach a z node, meaning that τ does not satisfy any of the literals in t_i . By the construction of the decision trees and the above argument, because τ' is a satisfying assignment of f , any traversal of the two decision trees must reach a waiting gadget regardless of the assignment of the rest of the variables.

Reaching a q -gadget in T_1 means that q must be known before the tree can be pruned to become homogenous. But once q is known $T_1^{\tau'}$ becomes homogenous, and the values of s, z and v_i, \dots, v_k are irrelevant to the decision. Because q is always the $n+1$ variable to be queried, we conclude that we need to query exactly the first $n+1$ variables in π to decide $T_1(\tau)$. On the other hand, T_2 uses the (q, s) -gadget instead of the q -gadget. This means that both q and s must be known before the tree can be pruned to become homogenous. Because s is always the $n+2$ variable queried, immediately after q , we conclude that we need to query exactly $n+2$ variables in π to decide $T_2(\tau)$. ■

Lemma 2 *For any fixed assignment τ to all the variables in π , if $\tau' = \tau$ restricted to x_1, \dots, x_n is a non-satisfying assignment of the original CNF f , then the number of variables that must be queried in order to decide $T_1(\tau)$ when the order of the queries is determined by π is equal to the number of variables that must be queried in order to decide $T_2(\tau)$.*

Proof of Lemma 2. We use a similar argument to the one used in the proof of the previous lemma. A CNF, being a conjunction of clauses is not satisfied by an assignment if and only if there exists at least one clause that is not satisfied by the assignment. That is, there exists a clause t_i such that none of its literals is satisfied by τ . By the construction of the sub-trees, because τ contains a non-satisfying assignment of t_i , then a traversal of the sub-tree t_i using the values in τ' must reach a z node. Because the value of z influences the decision of all sub-trees in both T_1 and T_2 , we cannot decide the value $T_1(\tau)$ or $T_2(\tau)$ until z is queried. Because z is

the last variable queried according to π , it follows that both trees can decide on τ at the same querying step and the lemma follows. ■

By Lemma 2 we conclude that if f is not satisfiable, then all assignments are decided by T_1 and T_2 at the same time, and in this case $EDD(T_1, \pi) = EDD(T_2, \pi)$ (so that the difference between them is 0 as required). By Lemma 1 we conclude that if f is satisfiable, then at least some of its input space can be decided by querying $n + 1$ variables in T_1 , but only $n + 2$ in T_2 . Now, looking at the difference $(EDD(T_2, \pi) - EDD(T_1, \pi))$ one sees that all the contributions of the non-satisfying assignments cancel out, and that each individual satisfying assignment contributes exactly $2^{-n} \cdot ((n + 2) - (n + 1)) = 2^{-n}$. Thus, $2^n \cdot (EDD(T_2, \pi) - EDD(T_1, \pi))$ equals the number of satisfying assignments of the CNF f . This problem, also known as #SAT is #P-complete. For the sake of completeness, note that from the restriction that the probability distribution is uniform, $O(n)$ bits should clearly suffice for each of the intermediate results of the calculation, and therefore the entire reduction process is linear in the size of the original problem. ■

(Theorem 2)

It immediately follows from the above proof that computing EDD is #P-hard as well. ■

(Theorem 1)

3.2 Approximating EDD for Decision Trees

Due to the hardness of computing the expected decision depth, we describe a simple approximation algorithm. We start with a procedure that given a specific input point $x \in \{0, 1\}^n$, a decision tree T description of f and a permutation π , computes the decision depth of x (when the order is restricted to π). We then use the procedure in a standard randomized algorithm to approximate $EDD(f, \pi)$ given a decision tree of f .

Procedure 1 *Decision-Depth*(x, T, π)

$T_0 \leftarrow T$

If T_0 is homogenous, return 0.

For $i \leftarrow 1$ to n

 Query the value of variable $x_{\pi(i)}$.

$T_i \leftarrow T_{i-1}$ pruned according to the value of $x_{\pi(i)}$.

 If T_i is homogenous, return i .

The running time of the procedure is $O(n|T|)$. Each pruning operation and homogeneity test is performed by a DFS traversal of the tree, requiring at most a full traversal of the tree for each variable-query.

Algorithm 1 *Rand-EDD*(T, π)

 Independently and uniformly select $m = \Theta(\ln \frac{1}{\delta} \cdot \frac{n^3}{\epsilon^2})$ input points and let the set of points be denoted X .

$D \leftarrow 0$

 For each $x \in X$

$depth \leftarrow \text{Decision-Depth}(x, T, \pi)$

$D \leftarrow D + \frac{1}{m} \cdot depth$

 Return D

Theorem 3 With probability at least $1 - \delta$ over the choice of the input points selected by Algorithm 1,

$$\left| \frac{D - EDD(T, \pi)}{EDD(T, \pi)} \right| < \epsilon \quad (5)$$

The theorem can be rephrased as follows:

$$\Pr(D > (1 + \epsilon) \cdot EDD(T, \pi)) + \Pr(D < (1 - \epsilon) \cdot EDD(T, \pi)) \leq \delta \quad (6)$$

Proof: For each random input point x_j , let $\chi_j = \text{Decision-Depth}(x_j, T, \pi)$. We are seeking to find $p = \mathbf{E}\chi_j$. In order to utilize the standard Chernoff inequality, we set $\chi'_j = \chi_j/n \in [0, 1]$, and accordingly set $p' = p/n \in [0, 1]$. Define $P_{\text{error}} = e^{-p'm(\epsilon/n)^2/2}$. According to the multiplicative Chernoff inequality,

$$\Pr\left(\frac{1}{m}\sum_{i=1}^m \chi'_i \geq (1 \pm \epsilon/n)p'\right) < P_{\text{error}} \quad (7)$$

Since $p' \geq 1/n$ (otherwise the function is all 0 or all 1), we get

$$2P_{\text{error}} \leq 2e^{-(1/n)m(\epsilon/n)^2/2} \quad (8)$$

Setting

$$m = 2 \ln \frac{2}{\delta} \cdot \frac{n^3}{\epsilon^2} \quad (9)$$

and since $D = \frac{1}{m} \sum_{i=1}^m \chi'_i n$, we have that

$$\Pr(D > (1 + \epsilon)p) + \Pr(D < (1 - \epsilon)p) < \delta \quad (10)$$

■

4 The Oblivious Model

4.1 Introduction

In this section we consider the following problem which is a restriction of our original (NP-hard) problem (in the oblivious model): Given a decision map \mathcal{M} of the Boolean function f , create a single fixed permutation π according to which the variables will be queried, such that the expected depth of a π -OBDD that respects \mathcal{M} is minimized. Recall that the restriction to respecting the given decision map means that the number of queries performed for an input p (given the permutation π) may be more than the number of queries actually necessary to determine $f(p)$. Rather, if H is the HDZ in \mathcal{M} to which p belongs, and $V(H)$ is the subset of variables defining H , then it is the number of variables queried until all variables in $V(H)$ are queried.

By reducing the problem to a well known scheduling problem, we are able to achieve a querying order with an expected number of queries per decision that is at most a factor of 2 larger than an optimal classifier respecting \mathcal{M} .

4.2 Reduction to a Scheduling Problem

Consider the following scheduling problem: Given a set of tasks $J = \{j_1, \dots, j_m\}$ with each task j_k having the following properties:

1. The task has a weight w_k .
2. The task has a predefined processing time p_k .
3. There is a set A_k of tasks that must precede it.

The objective is to minimize the weighted sum of completion times, $\sum_{k=1}^m w_k c_k$, where c_k is the completion time of the task j_k .

Given a decision map \mathcal{M} over a set of variables $x_1 \dots x_n$, we reduce our problem to the weighted sum of completion times scheduling problem as described below:

- For each variable x_k , $k = 1, \dots, n$, create a task j_k such that:
 1. $p_k = 1$
 2. $w_k = 0$
 3. $A_k = \emptyset$
 4. This task instructs to: “query variable x_k ”.

This creates n tasks enumerated from 1 to n . Their weight is 0 and their processing time is fixed at 1, which is the number of operations while querying a single variable.

- For each hypercube decision zone H_i in the decision map \mathcal{M} , create a distinct task j_{n+i} , $i = 1 \dots |\mathcal{M}|$ such that:

1. $p_{n+i} = 0$
2. $w_{n+i} = \Pr(H_i)$
3. $A_{n+i} = \{j_k | x_k \in V(H_i)\}$
4. This task instructs to: “decide H_i ”.

The predecessors set A_{n+i} for the task j_{n+i} that represents decision zone H_i has as its required predecessors the tasks that represent the variables $V(H_i)$ that must be queried for the decision of H_i . The processing time of the tasks representing hypercubes is fixed at 0, due to the fact that hypercubes are decided as a consequence of querying variables and apart from that exact no processing cost. The weight is set to equal the probability that the input resides inside the hypercube.

Definition 8 *A locally optimized schedule is a legitimate schedule where all tasks that have zero processing time have been moved to the earliest spot in the schedule where they can be legitimately executed.*

Claim 1 *In any locally optimized schedule of tasks in the resulting scheduling problem, if the task j_i , $i > n$ (“decide H_{i-n} ”) completes at time c_i , then by following the schedule and executing the tasks according to the instructions, the task “decide H_{i-n} ” happens exactly after all variables in $V(H_{i-n})$ have been queried.*

Proof: By construction of the scheduling problem, all variables in $V(H_{i-n})$ have caused the creation of a task instructing to query them, and those tasks are all predecessors of the task “decide H_{i-n} ”. So that if the schedule is legal, the required variable querying tasks must precede the decision. Because it is a locally optimized schedule, and “decide H_{i-n} ” is a zero processing time operation, it happens exactly after the last variable in $V(H_{i-n})$ has been queried. ■

Claim 2 *Let $S = \{s_1 \dots s_{n+|\mathcal{M}|}\}$ be a locally optimized schedule of tasks in the resulting scheduling problem and $c_k(S)$ be the completion time of task j_k according*

to the schedule S . Then $C(S) = \sum_{k=1}^{n+|\mathcal{M}|} w_k c_k(S)$ is equal to the expected number of queries of a classifier respecting the decision map \mathcal{M} , if it queries the variables according to the order defined in S .

Proof: Because the “query” tasks each have a processing time of 1 and the “decide” tasks have a processing time of 0, the completion time of each task is equal to the number of variable-queries before its completion. Because the “query” tasks have weight 0 they do not influence $C(S)$, and therefore $C(S) = \sum_{k=n+1}^{n+|\mathcal{M}|} w_k c_k(S)$. Because $\sum_{k=n+1}^{n+|\mathcal{M}|} w_k = 1$ and $\forall_k w_k \geq 0$, the values $w_{n+1} \dots w_{n+|\mathcal{M}|}$ form a probability distribution, and $C(S)$ becomes the expectation of the completion time. The probability distribution defined by the values $w_{n+1} \dots w_{n+|\mathcal{M}|}$, is identical to that of the a-priori probabilities the of the decision zones in \mathcal{M} .

For any decision zone H_j , a classifier respecting \mathcal{M} must query all of $V(H_j)$ before deciding H_j . Thus, considering the above claim, a classifier that respects \mathcal{M} and queries the variables according to the order defined in S , decides H_j exactly after $c_{j+n}(S)$ variable-queries.

Combining the above, the claim follows. ■

For the sake of completeness, we add that it is trivial to show that every solution to the original problem (i.e., a querying order of variables with its resulting expected number of queries of a classifier respecting \mathcal{M}) maps to a valid solution of the scheduling problem. As a result from the direct transformation between the problems, any approximation factor of the scheduling algorithm is also valid for the original expected number of queries problem. The above scheduling problem has a 2-approximation algorithm described in [H97].

Note that an actual classifier following the resulting querying schedule may not necessarily respect the original decision map \mathcal{M} . This is because decision zones of like decision may be unified causing the creation of a single bigger decision zone that may be decided by less variable-queries, improving the performance. However, respecting decision maps is our way of introducing structure that enables fair

comparison of solutions, it is not an implementation constraint.

5 The Non-Oblivious Model

5.1 Introduction

We are given a decision map \mathcal{M} of a Boolean classification function f and are asked to produce a rule F such that:

1. $F_1(\emptyset)$ outputs the index of the variable that needs to be queried first in order to minimize the expected number of queries.
2. Given a set $Y = \{(a_1, b_1), (a_2, b_2), \dots, (a_k, b_k)\}$ of tuples (a_i, b_i) where the a_i s are the indexes of the variables that have already been queried according to the output of the rule F in steps 1 to k , and the b_i s are their respective Boolean values, then $F_{k+1}(Y)$ outputs the index of the variable that is to be queried at step $k + 1$ such that the expected number of queries is minimized.

In the oblivious model we talked about *deciding* hypercube decision zones, where deciding meant that a certain decision zone specification is satisfied by the input data, and this could only occur after we examined all of the variables that form its specification. We did not care about recognizing situations where partial information enables us to *refute* decision zones before all of the variables required for deciding them are known. This was due to the fact that we could not use this information to improve our solution. It is not the case in the non-oblivious model. Thus, to clarify our analysis, we shall no longer use the term *deciding* that referred to a situation where all of the decision zone's specification has been met by the input data, and use the term *accepting* instead. We shall use the term *refuted* for the situation where the (possibly partial) input data disqualifies a decision zone from being accepted. Of course, we are still interested in the expected number of queries before we have an output, i.e., before some decision zone is accepted.

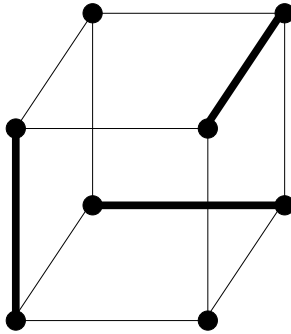


Figure 7: A decision map in $\{0, 1\}^3$. The decision zones are bold. This decision map cannot be realized by a decision tree (whose leaves correspond to the zones).

At this point it may be instrumental to clarify a few details about the connection between decision trees and decision maps. Consider a decision tree. It defines a decision map, where the output nodes of the decision tree form the decision zones. The variables on the path to each node are the variables that are required to be determined before the decision zone is accepted. The non-oblivious classifier respecting the decision map and having the lowest expected number of queries before a decision zone is accepted is a classifier that traverses the decision tree. To convince yourself of this, notice that for each subtree, the variable at the root of the tree is required for all decision zones at the leaves (remember the classifier respects the decision map). Thus, no unnecessary variables are ever queried, and it follows that this is the optimal solution for cases where the decision map is provided as a decision tree. This also follows from Observation 2. In such cases, the tree can be easily reconstructed from a list of decision zones by a simple non-oblivious algorithm that at each stage tests the variable that appears as a literal in all non-refuted decision zones. Many decision maps, however, cannot be realized by decision trees as is demonstrated in Figure 7, which brings us back to our more general question.

5.2 An Algorithm (Sub-Optimal)

Given a decision map \mathcal{M} , we can construct a complete graph $G(V, E)$ whose vertices are marked with a description of the elements of \mathcal{M} and their decision values (see

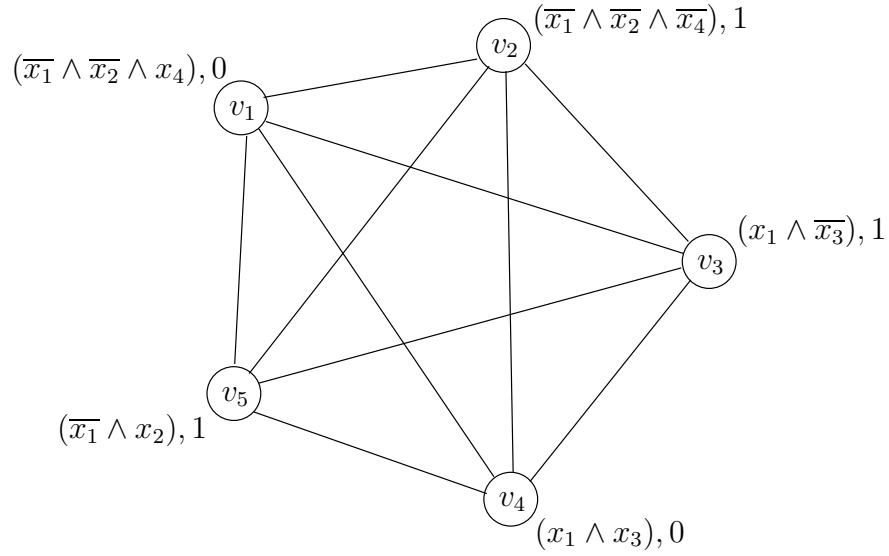


Figure 8: An example of mapping of a decision map to a complete graph. Each edge can be labeled with at least one variable over which its two end nodes disagree. Each node is marked with the monomial describing its decision zone and the output value of the decision zone.

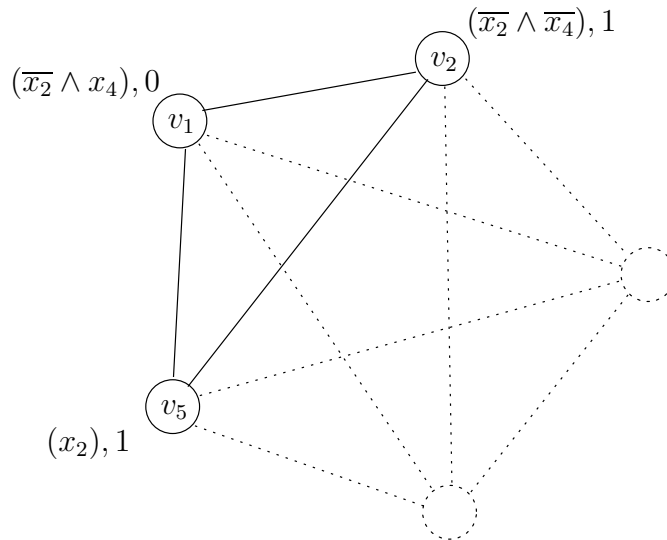


Figure 9: The remaining graph and monomials after the variables' values $x_1 = 0$ and $x_3 = 1$ have been determined.

Figure 8). Each of the decision zones, being a hypercube, can be represented by a monomial. Two vertices of G are connected by an undirected edge if and only if the monomials of the hypercubes they represent disagree on at least one of the literals (i.e., they are disjoint). Because all the decision zones in a decision map are disjoint, the graph is complete. For any decision zone H (or vertex of G), $V(H)$ is the set of variables in the monomial representation of H . Obviously, the dimension of the hypercube H is $n - |V(H)|$, and $\Pr(H) = 2^{-|V(H)|}$.

Algorithm 2

Let $\mathcal{H} \leftarrow \mathcal{M}$.

Repeat until \mathcal{H} becomes empty:

1. Let $H' = \arg \min_{V(H)} \{H \in \mathcal{H}\}$ be the HDZ from \mathcal{H} with the lowest number of literals in its representing monomial.
2. Query the set of variables in $V(H')$.
3. If during the process of querying the variables in $V(H')$, H' or some other HDZ in \mathcal{H} is accepted - Terminate and return the value of the accepted decision zone.
4. Remove from \mathcal{H} all of the decision zones $H \in \mathcal{H}$ whose partial assignments are refuted by one of the values of the variables just queried.
5. For all remaining HDZs in \mathcal{H} , remove the set of queried variables $V(H')$ from their monomial representation (thus modifying the HDZs).

Theorem 4 Algorithm 2 achieves an average number of variable-queries given by

$$\sum_{H_i \in \mathcal{M}} \frac{|V(H_i)|(|V(H_i)| + 1)}{2} \cdot \Pr(H_i) \quad (11)$$

Recall that the entropy of \mathcal{M} (which is a lower bound on the expected decision depth of a rule that respects \mathcal{M}) is $\mathcal{H}(\mathcal{M}) = \sum_{H_i \in \mathcal{M}} |V(H_i)| \cdot \Pr(H_i)$. Thus we get an upper bound that is reminiscent of the entropy. If the H_i 's are all approximately of the same size then this bound is roughly quadratic in the lower bound. However, in general it may be much larger.

Proof: Because every pair of decision zones disagree on at least one of the literals, querying the literals of a non-refuted monomial implies a reduction of at least 1 literal from all other non-refuted monomials (see Figure 9). To see why this property is kept during all steps of the algorithm, consider what would happen if at some iteration it were not the case: Assume we have two non-refuted monomials that represent two decision zones, and have no literal on which they disagree. Then, there exists an assignment that satisfies both of the original monomials representing the decision zones. This assignment consists of the variables already queried being equal to whatever value they have (not refuting either monomial), and the values of the other variables set so that they satisfy both monomials. This means that the decision zones overlap, in contradiction of their definition.

Thus, for any input to the classifier that belongs to a decision zone H_i , the maximum number of monomials that are queried before H_i is accepted is $|V(H_i)|$. Because the monomials are ordered in ascending order of number of literals, at most $|V(H_i)|^2$ literals are queried before the decision is made. ■

6 Further Research

The main question that remains open is whether there exists an approximation algorithm for the general case problem (without the limitation of respecting a given decision map). However, a polynomial time algorithm with an approximation factor expression of the form $k \cdot OPT + c$, where $c = O(\log n)$, is not likely to be found, since it could be used to solve the satisfiability problem in polynomial time. To verify this consider a non-satisfiable function f . One would simply run the approximation algorithm, and receive at most c variables in the set that should be queried. This means that at most 2^c decision zones exist and all that remains is to check them. Also note that an approximation factor that is purely multiplicative ($c = 0$) is also not likely for the same reason.

References

- [BW96] Bollig, B. and Wegener, I. (1996). Improving the variable ordering of OB-DDs is NP-complete. *IEEE Trans. on computers* 45, 993-1002.
- [BSSW03] Bollig B., Sauerhoff M., Sieling D., and Wegener I. (2003). Binary Decision Diagrams, submitted as a chapter of Boolean Functions, Volume II, edited by Y.Crama and P. Hammer.
- [B86] Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers* 35, 677-691.
- [GL81] Gens, G.V. and Levner, E.V. (1981). Fast approximation algorithms for job sequencing with deadlines. *Discrete Applied Mathematics* 3, 313-318.
- [H97] Hall, L.A. (1997). Approximation algorithms for scheduling. In: Hochbaum, D.S. (ed) Approximation algorithms for NP-hard problems. PWS publishing company, Boston, 1-45.
- [Has86] Hastad, J. (1986). Computational limitations for small depth circuits. PhD thesis, MIT Press, 1986.
- [KN96] Kushilevitz E., Nisan, N. (1996). *Communication Complexity*. Cambridge University Press.
- [W00] Wegener, I. (2000). *Branching Programs and Binary Decision Diagrams – Theory and Applications*. SIAM Monographs on Discrete Mathematics and Applications.

A Technical Appendix

Chernoff Bounds. The following probability inequality is due to Chernoff. There are several forms of the inequality. We state here the multiplicative form. Let χ_1, \dots, χ_m be independent identically distributed random variables in the range $[0, 1]$ with $\mathbf{E}\chi_i = p$. Then for any $\gamma \in [0, 1]$ the following bounds hold:

$$\Pr \left[\frac{1}{m} \sum_{i=1}^m \chi_i > (1 + \gamma)p \right] < \exp(-\gamma^2 pm/3)$$

$$\Pr \left[\frac{1}{m} \sum_{i=1}^m \chi_i < (1 - \gamma)p \right] < \exp(-\gamma^2 pm/2)$$

The Shannon Entropy. The Shannon entropy $\mathcal{H}(D)$ of a probability distribution $D = \{p_1, p_2, \dots, p_n\}$ such that $\sum_{i=1}^n p_i = 1$ is given by

$$\mathcal{H}(D) = - \sum_{i=1}^n p_i \log_2(p_i)$$