ALGORITHMS FOR BOOLEAN FUNCTION QUERY PROPERTIES

SCOTT AARONSON*

Abstract. We investigate efficient algorithms for computing Boolean function properties relevant to query complexity. Such properties include, for example, deterministic, randomized, and quantum query complexities; block sensitivity; certificate complexity; and degree as a real polynomial. The algorithms compute the properties given an n-variable function's truth table (of size $N = 2^n$) as input.

Our main results are the following:

- $O(N^{\log_2 3} \log N)$ algorithms for many common properties.

- An $O(N^{\log_2 5} \log N)$ algorithm for block sensitivity.

- An $O(N)$ algorithm for testing 'quasisymmetry.'

- A notion of a 'tree decomposition' of a Boolean function, proof that the decomposition is unique, and an $O(N^{\log_2 3} \log N)$ algorithm for finding it.

- A subexponential-time approximation algorithm for space-bounded quantum query complexity. To develop this algorithm, we give a new way to search systematically through unitary matrices using finite-precision arithmetic.

The algorithms discussed have been implemented in a linkable library.

Key words. algorithm, Boolean function, truth table, query complexity, quantum computation.

AMS subject classifications. 68Q10, 68Q17, 68Q25, 68W01, 81P68.

1. Introduction.

The query complexity of Boolean functions, also called black-box or decision-tree complexity, has been well studied for years [7, 9, 18]. Counting how many queries are needed to evaluate a function is easier than counting how many computational steps are needed; thus, nontrivial lower bounds are more readily shown for the former measure than for the latter. Also, query complexity has proved to be a powerful tool for studying the capabilities of quantum computers [2, 3, 4, 7, 13].

Numerous Boolean function properties relevant to query complexity have been defined, such as sensitivity, block sensitivity, randomized and quantum query complexity, and degree as a real polynomial. But many open questions remain concerning the relationships between the properties. For example, are sensitivity and block sensitivity polynomially related? How small can quantum query complexity be, relative to randomized query complexity? Lacking answers to these questions, we may wish to gain insight into them by using computer analysis of small Boolean functions. But to perform such analysis, we need efficient algorithms to compute the properties in question. Such algorithms are the subject of the present paper.

Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function, and let $N = 2^n$ be the size of the truth table of f. We seek algorithms that have modest running time as a function of N, given the truth table as input. The following table lists some properties important for query complexity, together with the complexities of the most efficient algorithms for them of which we know. In the table, 'LP' stands for linear programming reduction, and 'SDP' for semidefinite programming reduction.

---

* Computer Science Division, UC Berkeley, Berkeley, CA USA 94720-1776. Email: aaronson@cs.berkeley.edu. Work done at Bell Laboratories. Supported by a California MICRO Fellowship; an NSF Graduate Fellowship; and the Defense Advanced Research Projects Agency (DARPA) and Air Force Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-01-2-0524.

Query Property	Complexity	Source
Deterministic query complexity $\mathrm{D}(f)$	$O(N^{1.585}\log N)$	[12]
Certificate complexity $\mathrm{C}(f)$	$O(N^{1.585}\log N)$	[10]
Degree as a real polynomial $\deg(f)$	$O(N^{1.585}\log N)$	This paper
Approximate degree $\widetilde{\deg}(f)$	LP	Obvious
Randomized query complexity $\mathrm{R}_0(f)$	LP	This paper
Block sensitivity $\mathrm{bs}(f)$	$O(N^{2.322}\log N)$	This paper
Quasisymmetry	$O(N)$	This paper
Tree decomposition	$O(N^{1.585}\log N)$	This paper
Quantum query complexity $\mathrm{Q}_2(f)$	SDP	[5]
Randomized certificate complexity $\mathrm{RC}(f)$	LP	Obvious

There is also a complexity-theoretic rationale for studying algorithmic problems such as those considered in this paper. Much effort has been devoted to finding Boolean function properties that do not naturalize in the sense of Razborov and Rudich [20], and that might therefore be useful for proving circuit lower bounds. In our view, it would help this effort to have a better general understanding of the complexity of problems on Boolean function truth tables—both upper and lower bounds. This paper is a step towards such an understanding.

In an earlier version of this paper, we raised as our "most interesting" open question whether there exists a polynomial-time algorithm to compute (or even approximate) quantum query complexity. Subsequently Barnum, Saks, and Szegedy [5] managed to answer this question in the affirmative, using semidefinite programming. In Section 7 we present a weaker result, namely an $O(N^{\mathrm{polylog}(N)})$ constant-factor approximation algorithm for bounded-error quantum query complexity if the memory of the quantum computer is restricted to $O(\log n)$ qubits. The main issue here, which is only partly addressed by results of Bernstein and Vazirani [8], is how to represent unitary operators with limited-precision arithmetic. Our results on this issue are used in the detailed analysis of Barnum et al.'s algorithm.

We have implemented most of the algorithms discussed in this paper in a linkable C library, which is available for download[1].

The paper is organized as follows. Section 2 gives preliminaries, and Section 3 reviews simple algorithms for deterministic query complexity, certificate complexity, degree as a real polynomial, randomized query complexity, and randomized certificate complexity. Section 4 gives an $O(N^{\log_2 5}\log N)$ algorithm for computing block sensitivity, and Section 5 gives an $O(N)$ algorithm for testing 'quasisymmetry.' Section 6 defines a notion of 'tree decomposition' of a Boolean function, proves that the decomposition is unique, and gives an $O(N^{\log_2 3}\log N)$ algorithm for constructing it. Section 7 presents our results on algorithms for quantum query complexity, and Section 8 concludes with some open problems.

2. Preliminaries. A Boolean function f is a total function from $\{0,1\}^n$ onto $\{0,1\}$. We use V_f to denote the set of variables of f, and use X, or alternatively $x_1,\ldots,x_n$, to denote an input to f. The restriction of f to R is denoted $f_{|R}$. If X is an input, $|X|$ denotes the Hamming weight of X; if S is a set, $|S|$ denotes the cardinality of S. Particular Boolean functions to which we refer are AND_n, OR_n, and XOR_n, the AND, OR, and XOR functions respectively on n inputs.

Throughout we assume a RAM model of computation, in which for any input X, $f(X)$ can be obtained in $O(1)$ time.

[1] Available at `http://www.cs.berkeley.edu/~aaronson/bfw`

3. Basic Properties. To our knowledge, no algorithms for block sensitivity, quasisymmetry, or tree decomposition have been previously published. But algorithms for simpler query properties have appeared in the literature.

3.1. Deterministic Query Complexity. A *decision tree* T is a binary tree in which each non-leaf vertex is labeled with an index (1 through n) and each leaf vertex is labeled with an output (0 or 1). Evaluation begins at the root. At a vertex v labeled with i, if $x_i = 0$ we evaluate the left subtree of v, while if $x_i = 1$ we evaluate the right subtree. When we reach a leaf vertex we halt and return the appropriate output. T represents a Boolean function f if, for all settings of $x_1, \ldots, x_n$, the output of T equals $f(x_1, \ldots, x_n)$. Then the deterministic query complexity $\mathrm{D}(f)$ is the minimum height of a decision tree for f.

Guijarro et al. [12] give a simple $O(N^{1.585} \log N)$ dynamic programming algorithm to compute $\mathrm{D}(f)$. We present a similar algorithm for completeness. The idea is that, at any time, a decision tree for f has reduced f to one of its 3^n possible restrictions: each of the n variables either (1) has been queried and is a 0, (2) has been queried and is a 1, or (3) has not yet been queried. Thus we can represent a restriction S by an element of $\{0, 1, *\}^n$, where the asterisk represents 'not yet queried.' We can also impose a lexicographic ordering on restrictions by stipulating that 0 comes before 1 comes before asterisk.

The algorithm consists of two loops, both of which proceed through all states in lexicographic order. The first loop fills in an array called A, which stores, for each restriction, whether it is a constant function and if so what its value is. The second loop uses A to fill in an array called D, which stores the deterministic query complexity of each restriction. In the algorithm, $S(i)$ represents the ith element of S, and $S_{S(i)=k}$ represents S with $S(i)$ set to the value k.

Algorithm 1. ***(computes deterministic query complexity)***
loop over all $S \in \{0, 1, *\}^n$ *in lexicographic order* {
    if ($S \in \{0, 1\}^n$) *then set* $A[S] := f(S)$ *else* {
        choose an i *such that* $S(i) = *$;
        if ($A[S_{S(i)=0}] = A[S_{S(i)=1}]$) *then set* $A[S] := A[S_{S(i)=0}]$;
        else set $A[S] :=$ NONCONSTANT;
    }
}
loop over all $S \in \{0, 1, *\}^n$ *in lexicographic order* {
    if ($A[S] \neq$ NONCONSTANT) *then set* $D[S] := 0$;
    else set $D[S] := 1 + \min_{S(i)=*} \left[\max\left(D\left[S_{S(i)=0}\right], D\left[S_{S(i)=1}\right]\right)\right]$;
}
return $D[*^n]$;

That f is given as a truth table is crucial: if f is non-total and only the inputs for which f is defined are given, then deciding whether $\mathrm{D}(f) \leq k$ for some k is NP-complete [16], and NP-hard even to approximate within a constant factor [23].

3.2. Certificate Complexity. Given an input X to f, a *certificate* for X is a constant-valued restriction that agrees with X on the fixed variables. The *size* of the certificate is the number of fixed variables; note that querying these variables is sufficient to prove that $f(X) = 0$ or $f(X) = 1$, as the case may be. The certificate complexity $\mathrm{C}_X(f)$ of X is the minimum size of any certificate for X. The certificate complexity $\mathrm{C}(f)$ of f is the maximum of $\mathrm{C}_X(f)$ over all inputs X. (Equivalently, $\mathrm{C}(f)$ is the minimum height of a nondeterministic decision tree for f.)

Czort [10] gives an $O(N^{1.585}\log N)$ algorithm to compute $\mathrm{C}(f)$. We can obtain such an algorithm by reusing the same array A that was used for deterministic query complexity in Section 3.1. Consider a directed acyclic graph G in which the vertices are the 3^n possible restrictions of f, and an edge is drawn from S to T if and only if T is obtained from S by changing one 0 or 1 to an asterisk. The main loop of the algorithm fills in an array called q, proceeding in *reverse* lexicographic order. The array q stores, for each restriction S of f, the maximum length of any path in G from S to a non-constant function, given that the path must halt once it reaches a non-constant function. (Therefore, if S itself is non-constant then the length must be 0.) The certificate complexity is obtained by taking the maximum, over all $S \in \{0,1\}^n$, of $n - q[S] + 1$.

ALGORITHM 2. ***(computes certificate complexity)***
loop over all $S \in \{0,1,*\}^n$ *in reverse lexicographic order* {
if $(A[S] = \text{NONCONSTANT})$ *then*
set $q[S] := 1 + \max\left(-1, \max_{S(i)\in\{0,1\}} q\left[S_{S(i)=*}\right]\right)$;
else set $q[S] := 1 + \max_{S(i)\in\{0,1\}} q\left[S_{S(i)=*}\right]$;
}
return $\max_{S\in\{0,1\}^n} (n - q[S])$;

Again, if f is not given as a full truth table, then deciding whether $\mathrm{C}(f) \le k$ for some k is NP-complete [14].

3.3. Degree as a Polynomial. Let $\deg(f)$ be the minimum degree of an n-variate real multilinear polynomial p such that, for all $X \in \{0,1\}^n$, $p(X) = f(X)$. Degree was introduced to query complexity by Nisan and Szegedy [19], who observed the relationship $\deg(f) \le \mathrm{D}(f)$. Later Beals et al. [7] related degree to quantum query complexity by showing that $\deg(f) \le 2\,\mathrm{Q}_E(f)$.

The following lemma, adapted from Lemma 4 of [9], is easily seen to yield an $O(n3^n) = O(N^{1.585}\log N)$ dynamic programming algorithm for $\deg(f)$. Say that a function obeys the *parity property* if the number of inputs X with odd parity for which $f(X) = 1$ equals the number of inputs X with even parity for which $f(X) = 1$.

LEMMA 3.1 (folklore). $\deg(f)$ *equals the size of the largest restriction of* f *for which the parity property fails.*

Proof. Let c_S be the coefficient of the monomial $\prod_{v_k\in S} v_k$. By the Möbius formula,

$$c_S = \sum_{X\subseteq S} (-1)^{|X|+|S|} f(X),$$

where $X \subseteq S$ means that every i for which $x_i = 1$ is in S. Now f has degree less than d if and only if, for all S with $|S| \ge d$, $c_S = 0$. But for each fixed value of $|S|$, $|S|$ (in the formula above) can affect the sign of c_S but not whether $c_S = 0$. Therefore for all δ, $c_S = 0$ for all S with $|S| = \delta$ is equivalent to the parity property holding for all restrictions of size δ. □

Lemma 3.1 leads to an $O(N^{\log_2 3} \log N)$ dynamic programming algorithm for computing $\deg(f)$, as follows:

Algorithm 3. ***(computes degree as a real polynomial)***
loop over all $S \in \{0,1,*\}^n$ *in lexicographic order* {
if $(S \in \{0,1\}^n)$ *then set* $d[S] := f(S)$*;*
else set $d[S] := d\left[S_{S(i)=0}\right] - d\left[S_{S(i)=1}\right]$ *for some* i *such that* $S(i) = *$*;*
}
return $\max_{d[S]\neq 0}$ *(number of* $*$ *'s in* S*);*

3.4. Randomized Query Complexity. A randomized decision tree T_R is simply a collection $T_1, \ldots, T_k$ of ordinary decision trees, each T_i associated with a probability p_i satisfying $p_1 + \cdots + p_k = 1$. T_R represents f if each tree T_i in the collection represents f. Let $h(T, X)$ be the number of queries tree T makes on input X. Then we define

$$h(T_R) = \max_{X \in \{0,1\}^n} \left[p_1 h(T_1, X) + \cdots + p_k h(T_k, X)\right].$$

Then the zero-error randomized query complexity $\mathrm{R}_0(f)$ is the minimum height of a randomized decision tree that represents f. One can also discuss the bounded-error randomized query complexity $\mathrm{R}_2(f)$, which is the minimum height of a randomized decision tree that represents f with a probability of error at most $1/3$. Nisan showed that $\mathrm{R}_0(f)^2 \geq \mathrm{D}(f)$ and $\mathrm{R}_2(f)^3 = \Theta(\mathrm{D}(f))$ [18]. On the other hand, the best known separation between deterministic and randomized query complexity is $\mathrm{R}_0(f) = \mathrm{R}_2(f) = \mathrm{D}(f)^{0.753\ldots}$ [21, 22], for f an AND/OR tree with two children per node.

Whether better separations are possible is a long-standing open question, and one that might be fruitfully investigated with computer analysis[2]. Unfortunately, though, we do not know how to compute $\mathrm{R}_0(f)$ or $\mathrm{R}_2(f)$ in polynomial time without reliance on linear programming. Here we sketch the reduction to LP.

As before, at any time the query algorithm has reduced f to one of its 3^n restrictions. Also, for each restriction, the algorithm has up to $n+2$ possible moves: it can query any variable not yet queried, halt and return 0, or halt and return 1. So consider a directed acyclic graph in which the vertices are the restrictions $S_1, \ldots, S_{3^n}$ (together with halting states $S_{(0)}$ and $S_{(1)}$) and the edges $(S_{i_1}, S_{j_1}), \ldots, (S_{i_m}, S_{j_m})$ are the possible moves of the algorithm. With each edge e we associate a probability weight $p(e)$; these weights are the variables of the LP. Let $C(X)$ be the subset of N restrictions that are compatible with input X. There are four classes of constraints:

1. *Well-formedness:* The sum of the probability weights leaving the initial state must be 1. Formally $\sum_i p(S_0, S_i) = 1$, where S_0 is the initial state (no variables yet queried).
2. *Conservation of probability:* The sum of the probability weights entering each state must equal the sum of the probability weights leaving it. For all $j \neq 0$, $\sum_i p(S_i, S_j) = \sum_k p(S_j, S_k)$.

[2] We have done such analysis for all 4-variable Boolean functions dependent on all 4 inputs. The two functions exhibiting the largest deterministic/randomized complexity gap ($D(f) = 4$, $R_0(f) = 3$) are both AND/OR trees, namely A AND (B OR C OR D) and (A OR B) AND (C OR D). Randomization yields at least some speedup for 60 out of the 208 Boolean functions that are distinct up to negating inputs and outputs and permuting inputs.

3. *Probability of correctness:* For each input, the probability of returning the correct answer must be at least $1-\epsilon$, where $\epsilon = 0$ for $\mathrm{R}_0(f)$ and $\epsilon = 1/3$ for $\mathrm{R}_2(f)$. For all X with $f(X) = 1$, $\sum_{S_i \in C(X)} p(S_i, S_{(1)}) \geq 1-\epsilon$. For $f(X) = 0$, substitute $p(S_i, S_{(0)})$.
4. *Minimum running time:* For each input, the expected running time must be at most T. For all X, $\sum_{S_i \in C(X)} Q(S_i) \left[p(S_i, S_{(0)}) + p(S_i, S_{(1)})\right] \leq T$, where $Q(S_i)$ is the number of queries that have been made in state S_i.

The objective is to minimize T.

Finally, let us briefly mention randomized and quantum certificate complexity. Given $X \in \{0,1\}^n$, Aaronson [2] defined the randomized certificate complexity $\mathrm{RC}_X(f)$ to be the minimum expected number of queries used by a randomized algorithm that accepts with $2/3$ probability given X as input, and rejects with $2/3$ probability given any Y for which $f(Y) \neq f(X)$. Then $\mathrm{RC}(f)$ is the maximum of $\mathrm{RC}_X(f)$ over all X. The quantum certificate complexity $\mathrm{QC}(f)$ is defined analogously, but with quantum instead of randomized algorithms. Aaronson showed that for any X, the optimal (up to a constant factor) randomized certificate for X queries the input nonadaptively. Furthermore, $\mathrm{QC}(f)$ is exactly characterized (again up to a constant factor) as the square root of $\mathrm{RC}(f)$. From these results, it readily follows that both $\mathrm{RC}(f)$ and $\mathrm{QC}(f)$ are approximable to within a constant factor in polynomial time, using linear programming.

4. Block Sensitivity. Block sensitivity, introduced by Nisan [18], is a Boolean function property that is used to establish lower bounds. There are several open problems that an efficient algorithm for block sensitivity might help to investigate [18, 7, 9].

Let X be an input to Boolean function f, and let B (a *block*) be a nonempty subset of V_f. Let $X(B)$ be the input obtained from X by flipping the bits of B.

Definition 4.1. *A block B is* sensitive *on X if $f(X) \neq f(X(B))$, and* minimal *on X if B is sensitive and no proper sub-block S of B is sensitive. Then the block sensitivity* $\mathrm{bs}_X(f)$ *of X is the maximum number of disjoint minimal (or equivalently, sensitive) blocks on X. Finally* $\mathrm{bs}(f)$ *is the maximum of* $\mathrm{bs}_X(f)$ *over all X.*

The obvious algorithm to compute $\mathrm{bs}(f)$ (compute $\mathrm{bs}_X(f)$ for each X using dynamic programming, then take the maximum) uses $\Theta(N^{2.585} \log N)$ time. Here we show how to reduce the complexity to $O(N^{2.322} \log N)$ by exploiting the structure of minimal blocks. Our algorithm has two main stages: one to identify minimal blocks and store them for fast lookup, another to compute $\mathrm{bs}_X(f)$ for each X using only minimal blocks. The analysis proceeds by showing that no Boolean function has too many minimal blocks, and therefore that if the algorithm is slow for some inputs (because of an abundance of minimal blocks), then it must be faster for other inputs.

Algorithm 4. ***(computes** $\mathrm{bs}(f)$**)*** *For each input X:*
1. *Construct an array M of all minimal blocks of X. To do this, loop over all blocks B in lexicographic order ($\{x_1\}, \{x_2\}, \{x_1, x_2\}, \{x_3\}, \ldots$), and mark (i) whether B is a minimal block, and (ii) whether B contains a minimal block. B is minimal if B is sensitive and, for all $x_i \in B$, $B \setminus \{x_i\}$ does not contain a minimal block. B contains a minimal block if B is minimal or $B \setminus \{x_i\}$ contains a minimal block for some $x_i \in B$.*
2. *Create $2^n - 1$ lists, one list L_S for each nonempty subset S of variables. Then, for each minimal block B in M, insert a copy of B into each list L_S such*

that $B \subseteq S$. The result is that, for each S, $L_S = 2^S \cap M$, where 2^S is the power set of S.

3. *Let a* state *be a partition (P, Q) of V_f. The set P represents a union of disjoint minimal blocks that have already been selected; the set Q represents the set of variables not yet selected. Then $\mathrm{bs}_X(f) = \theta(\emptyset, V_f)$, where $\theta(P, Q)$ is defined via the recursion $\theta(P, Q) \stackrel{\triangle}{=} 1 + \max_{B \in L_Q} \theta(P \cup B, Q \setminus B)$. Here the maximum evaluates to 0 if L_Q is empty. Compute $\theta(P, Q)$ using depth-first recursion, caching the values of $\theta(P, Q)$ so that each needs to be computed only once.*

The block sensitivity is then the maximum of $\mathrm{bs}_X(f)$ over all X.

Let $m(X, k)$ be the number of minimal blocks of X of size k. The analysis of Algorithm 4's running time depends on the following lemma, which shows that large minimal blocks are rare in any Boolean function.

Lemma 4.2. *$\sum_X m(X, k) \le 2^{n-k}\binom{n}{k}$ for $k \ge 2$.*

Proof. The number of positions that can be occupied by a minimal block of size k is $\binom{n}{k}$ for each input, or $2^n\binom{n}{k}$ for all inputs. Consider an input X with a minimal block $B = \{b_1, \ldots, b_k\}$ of size k. Block B has $2^k - 1$ nonempty subsets; label them $S_1, \ldots, S_{2^k-1}$. By the minimality of B, for each S_i the input $X(S_i)$ has $\{b_1\}, \ldots, \{b_k\}$ as minimal blocks if $S_i = B$, and $B \setminus S_i$ as a minimal block if $S_i \ne B$. Therefore, since $k \ge 2$, $X(S_i)$ cannot have B as a minimal block. So of the $2^n\binom{n}{k}$ positions, only one out of 2^k can be occupied by a minimal block of size k. □

Theorem 4.3. *Algorithm 4 takes $O(N^{2.322} \log N)$ time.*

Proof. Step 1 takes time $O(N^2 \log N)$, totaled over all inputs. Let us analyze step 2, which creates the $2^n - 1$ lists L_S. Since each minimal block B is contained in $2^{n-|B|}$ sets of variables, the total number of insertions is at most

$$\sum_X \sum_{k=0}^{n} m(X, k) 2^{n-k} = \sum_{k=0}^{n} \left[2^{n-k} \sum_X m(X, k) \right] \le \sum_{k=0}^{n} 2^{2n-2k}\binom{n}{k} = N^{\log_2 5}.$$

Since each insertion takes $O(\log N)$ time, the total time is $O\left(N^{2.322} \log N\right)$.

We next analyze step 3, which computes block sensitivity using the minimal blocks. Each $\theta(P, Q)$ evaluation is performed at most once, and involves looping through a list of minimal blocks contained in Q, with each iteration taking $O(\log n)$ time. For each block B, the number of distinct (P, Q) pairs such that $B \subseteq Q$ is at most $2^{n-|B|}$. Therefore, by the previous calculation, the time for each input X is at most $(\log N) \sum_{k=0}^{n} m(X, k) 2^{n-k}$ and a bound of $O\left(N^{2.322} \log N\right)$ follows. □

5. Quasisymmetry. A Boolean function $f(X)$ is *symmetric* if its output depends only on $|X|$. Query complexity is well understood for symmetric functions: for all non-constant symmetric f, $\mathrm{D}(f) = n$, $\mathrm{R}_0(f) = \Theta(n)$, and $\mathrm{Q}_E(f) = \Theta(n)$ [7]. Thus, a program for analyzing Boolean functions might first check whether a function is symmetric, and if it is, dispense with many expensive tests. We call f *quasisymmetric* if some subset of input bits can be negated to make f symmetric. There is an obvious $O(N^2)$ algorithm to test quasisymmetry; here we give a linear-time algorithm.

For an integer p, call a restriction of f a *p-left-restriction* if each variable v_i is fixed if and only if $i \le p$. The basic idea of the algorithm is to loop through all $2^{n+1} - 1$ such restrictions, with p decreasing from n to 0. Given a p-left-restriction S, let S_0 and S_1 be the two $(p+1)$-left-restrictions that agree with S. If either $f_{|S_0}$ or $f_{|S_1}$ is not quasisymmetric, then $f_{|S}$ (and hence f) cannot be quasisymmetric.

If, on the other hand, $f_{|S_0}$ and $f_{|S_1}$ are both quasisymmetric, then the algorithm tries to fit them together in such a way that $f_{|S}$ itself is seen to be quasisymmetric. If the fitting-together process succeeds, then the algorithm returns a structure $g[S]$, containing both the output of $f_{|S}$ (encoded in compact form, as a symmetric function) and the *direction*, meaning the set of input bits that must be flipped to make $f_{|S}$ symmetric. Note that $g[S]$ occupies only $O(n-p)$ bits of space. Most of the algorithm deals with the special cases that $f_{|S}$ is a XOR function or a constant function; in both cases $f_{|S}$ is symmetric no matter which set of input bits is flipped. In the pseudocode, these cases are handled using the tags XOR (for a XOR function), CONSTANT (for a constant function), and NORMAL (for any other quasisymmetric function). Whenever the algorithm fails (meaning that f has been found not to be quasisymmetric), the whole algorithm terminates; whenever $g[S]$ is assigned a value, the current iteration terminates. To avoid ambiguity about whether $f_{|S}$ is a XOR, CONSTANT, or NORMAL function, we start p at $n-2$ rather than n.

ALGORITHM 5. ***(tests quasisymmetry)***
For all p-left-restrictions S for $p \leq n-2$, with p decreasing from $n-2$ to 0:

1. *If $p = n-2$, then let $g[S]$ be the appropriate* NORMAL, CONSTANT, *or* XOR *function for the 2-input Boolean function $f_{|S}$. If $f_{|S}$ is not quasisymmetric, then fail (meaning f is not quasisymmetric).*
2. *If $g[S_0]$ and $g[S_1]$ are* NORMAL *functions but have different directions, then fail.*
3. *Let 0^k be a string of k zeroes. If $g[S_0]$ and $g[S_1]$ are* CONSTANT *functions, then*
   - *If $f_{|S_0}\left(0^{n-p-1}\right) = f_{|S_1}\left(0^{n-p-1}\right)$, then let $g[S]$ be a* CONSTANT *function with output $f_{|S_0}\left(0^{n-p-1}\right)$; otherwise fail.*
4. *If $g[S_0]$ and $g[S_1]$ are* XOR *functions, then*
   - *If $f_{|S_0}\left(0^{n-p-1}\right) \neq f_{|S_1}\left(0^{n-p-1}\right)$, let $g[S]$ be a* XOR *function with $f_{|S}\left(0^{n-p}\right) = f_{|S_0}\left(0^{n-p-1}\right)$; otherwise fail.*
5. *For $i \in \{0,1\}$, if $g[S_i]$ is a* CONSTANT *function and $g[S_{1-i}]$ is a* XOR *function, then halt and return failure.*
6. *For $i \in \{0,1\}$, if $g[S_i]$ is a* CONSTANT *or* XOR *function, then make $g[S_i]$ a* NORMAL *function with the same direction as $g[S_{1-i}]$.*
7. *For $i \in \{0,1\}$ and $j \in \{0,\ldots,n-p-1\}$, let $a_j^{(i)} = f_{|S_i}(X)$ for all $X \in \{0,1\}^{n-p-1}$ of Hamming distance j from the direction string.*
   - *If the strings $a^{(0)}$ and $a^{(1)}$ overlap each other on $n-p-2$ bits, so that for either $i=0$ or $i=1$,*

     $$a_1^{(i)} = a_2^{(1-i)}, \ldots, a_{n-p-2}^{(i)} = a_{n-p-1}^{(1-i)},$$

     then let $g[S]$ be a NORMAL *function with outputs described by the $(n-p)$-bit overlap string, and appropriate direction. Otherwise fail.*

Since the time used by each invocation is linear in $n-p$, the total time used is

$$\sum_{p=0}^{n} 2^p(n-p) = O(N).$$

The following lemma shows that the algorithm deals with all of the ways in which a function can be quasisymmetric, which is key to the algorithm's correctness.

LEMMA 5.1. *Let f be a Boolean function on n inputs. If two distinct (and non-complementary) sets of input bits A and B can be flipped to make f symmetric, then f is either* XOR_n, $1 - \mathrm{XOR}_n$, *or a constant function.*

Proof. Assume without loss of generality that B is empty. Then A has cardinality less than n. We know that $f(X)$ depends only on $|X|$, and also that it depends only on $/X/ \triangleq \sum_{i=1}^{n} \kappa(x_i)$ where $\kappa(x) = 1 - x$ if $x_i \in A$ and $\kappa(x) = x$ otherwise. Choose any Hamming weight $0 \leq w \leq n-2$, and consider an input Y with $|Y| = w$ and with two variables v_i and v_j such that $v_i \in A$, $v_j \notin A$, and $Y(i) = Y(j) = 0$. Let Z be Y with $Y(i) = Y(j) = 1$. We have $|Z| = |Y| + 2$, but on the other hand $/Z/ = /Y/$, so $f(Y) = f(Z)$ by symmetry. Again applying symmetry, $f(P) = f(Q)$ whenever $|P| = w$ and $|Q| = w + 2$. Therefore f is either XOR_n, $1 - \mathrm{XOR}_n$, or a constant function. ∎

6. Tree Decomposition. Many of the Boolean functions of most interest to query complexity are naturally thought of as trees of smaller Boolean functions: for example, AND-OR trees and majority trees. Thus, given a function f, one of the most basic questions we might ask is whether it has a tree decomposition and if so what it is. In this section we define a sense in which every Boolean function has a unique tree decomposition, and we prove its uniqueness. We also sketch an $O(N^{1.585} \log N)$ algorithm for finding the decomposition.

DEFINITION 6.1. *A* distinct variable tree *is a tree in which*

(i) Every leaf vertex is labeled with a distinct variable (which may or may not be negated).

(ii) Every non-leaf vertex v is labeled with a Boolean function having one variable for each child of v, and depending on all of its variables.

(iii) Every non-leaf vertex has at least two children.

Such a tree represents a Boolean function in the obvious way. We call the tree *trivial* if it contains exactly one non-leaf vertex. For instance, the majority function on 3 inputs can only be represented by a trivial tree.

A *tree decomposition* of f is a separation of f into the smallest possible components, with the exception of $(\neg)\,\mathrm{AND}_k$, $(\neg)\,\mathrm{OR}_k$, and $(\neg)\,\mathrm{XOR}_k$ components (where $(\neg)$ denotes possible negation), which are left intact. The choice of AND, OR, and XOR components is not arbitrary; these are precisely the three components that "associate," so that, for example, $\mathrm{AND}\left(x_1, \mathrm{AND}\left(x_2, x_3\right)\right) = \mathrm{AND}\left(\mathrm{AND}\left(x_1, x_2\right), x_3\right)$. Formally:

DEFINITION 6.2. *A tree decomposition of f is a distinct variable tree representing f such that:*

(i) No vertex is labeled with a function f that can be represented by a nontrivial tree, unless f is $(\neg)\,\mathrm{AND}_k$, $(\neg)\,\mathrm{OR}_k$, or $(\neg)\,\mathrm{XOR}_k$ for some k.

(ii) No vertex labeled with $(\neg)\,\mathrm{AND}_k$ has a child labeled with AND_l.

(iii) No vertex labeled with $(\neg)\,\mathrm{OR}_k$ has a child labeled with OR_l.

(iv) No vertex labeled with $(\neg)\,\mathrm{XOR}_k$ has a child labeled with $(\neg)\,\mathrm{XOR}_l$.

(v) Any vertex labeled with a function that is constant on all but one input is labeled with $(\neg)\,\mathrm{AND}_k$ or $(\neg)\,\mathrm{OR}_k$.

Let *double-negation* be the operation of negating the output of a function at some non-root vertex v, then negating the corresponding input of the function at v's parent. Double-negation is a trivial way to obtain distinct decompositions. This caveat aside, we can assert uniqueness:

THEOREM 6.3. *Every Boolean function has a unique tree decomposition, up to double-negation.*

We will build up to this uniqueness theorem via a sequence of preliminary results. Given a vertex v of a distinct variable tree, let $L(v)$ be the set of variables in the subtree of which v is the root. Assume that f is represented by two distinct tree decompositions, S and T, such that S has a vertex v_S and T has a vertex v_T with $L(v_S)$ and $L(v_T)$ incomparable (i.e. they intersect, but neither contains the other). We partition V_f into four sets of variables as follows: $A = L(v_S) - L(v_T)$, $B = L(v_T) - L(v_S)$, $I = L(v_S) \cap L(v_T)$, and $U = V_f - L(v_S) - L(v_T)$. Our strategy will be to derive increasingly strong constraints on how S and T can combine information from A, B, I, and U. We do this by repeatedly restricting variables—considering f as, say, a function of I only—and then exploiting the fact that S and T must produce the same output, even though information travels along different routes in the two trees. Ultimately (in Lemma 6.5) we show that f is a function of $s(A)$, $r(I)$, and $t(B)$ for some Boolean functions s, r, and t. The problem thereby reduces to which Boolean functions of *three* variables have non-unique decompositions—and we can check that the only possibilities, AND, OR, and XOR, are ruled out by the definition of a tree decomposition.

Call a set of variables *unifiable* if there exists a vertex v, in *any* decomposition of f, such that $L(v) = V$. The preceding results imply that no pair of unifiable sets V_S, V_T is incomparable (Lemma 6.6): either $V_S \cap V_T = \phi$, $V_S \subseteq V_T$, or $V_T \subseteq V_S$. From there, it is readily shown that any decomposition must contain a vertex v with $L(v) = V$ for *every* unifiable V, from which the uniqueness theorem follows.

A remark on notation: we use subscripts to name Boolean functions (i.e. s_0, s_1, etc.) in order of their appearance, and superscripts to list which of A, B, I, and U are currently being restricted.

Lemma 6.4. *There exist Boolean functions r, t_0^0, and t_0^1 such that f is a function of A, $r(I)$, $t_0^{r(I)}(B)$, and U.*

Proof. For any restriction u of U, we can write the output of S as $S^u[s_1(A,I),B]$, where S^u and s_1 are Boolean functions. Similarly we can write the output of T as $T^u[A, t_1(I,B)]$. We have that, for all settings of U,

$$S^u[s_1(A,I),B] = T^u[A,t_1(I,B)].$$

Consider a restriction b of B. This yields

$$S^{u,b}[s_1(A,I)] = T^u\left[A, t_2^b(I)\right].$$

for some Boolean function t_2^b. Therefore, for each b, s_1 depends on only a single bit obtained from I, namely $t_2^b(I)$. So we can write $s_1(A,I)$ as $s_3\left(A, t_2^b(I)\right)$ for some Boolean function s_3—or even more strongly as $s_3(A, s_4(I))$, since we know that s_1 does not depend on B. By analogous reasoning we can write $t_1(I,B)$ as $t_3(t_4(I),B)$ for some functions t_3 and t_4. So we have

$$S^u[s_3(A,s_4(I)),B] = T^u[A,t_3(t_4(I),B)].$$

Next we apply the restrictions $A = a$ and $B = b$, obtaining

$$S^{u,b}[s_3^a(s_4(I))] = T^{u,a}\left[t_3^b(t_4(I))\right],$$

which implies that, for some functions s_5 and t_5,

$$s_5(s_4(I)) = t_5(t_4(I))$$

for all I. This shows that $s_4(I)$ and $t_4(I)$ are equivalent up to negation of output, since S and T must depend on I for some restriction of A and B. So we have

$$S^u\left[s_0^{r(I)}(A), B\right] = T^u\left[A, t_0^{r(I)}(B)\right]. \tag{*}$$

for some Boolean functions $r(I)$, s_5^i, and t_5^i $(r \in \{0,1\})$. □

We will henceforth think of $r(I)$ as a single Boolean variable.

LEMMA 6.5. *There exist Boolean functions s and t such that f is a function of $s(A)$, $r(I)$, $t(B)$, and U.*

Proof. Starting from equation (*), we next apply the restrictions $A = a$ and $r(I) = i$:

$$S^{u,a,i}[B] = T^{u,a}\left[t_0^i(B)\right].$$

Thus, for all restrictions of A and $r(I)$, S depends on only a single bit obtained from B, namely $t_0^i(B)$. Note that t_0^i does not depend on A. Analogously, for both possible restrictions i of $r(I)$, T depends on only a single bit obtained from A, namely $s_0^i(A)$. So we can write

$$s_6^u\left[s_0^i(A), t_0^i(B)\right] = t_6^u\left[s_0^i(A), t_0^i(B)\right]$$

where s_6^u and t_6^u are two-input Boolean functions. We claim that $s_0^0 = s_0^1$ and $t_0^0 = t_0^1$.

There must exist a restriction u of U such that s_6^u depends on both s_0^i and t_0^i. Suppose there exists a restriction b of B such that $t_0^0(b) \neq t_0^1(b)$. Now, s_0^i must be a nonconstant function, so find a constant c such that $s_6^u\left[c, t_0^i(b)\right]$ depends on t_0^i, and choose restrictions $A = a$ and $r(I) = i$ such that $s_0^i(a) = c$. (If s_6^u is a XOR function, then either $c = 0$ or $c = 1$ will work, whereas if s_6^u is an AND or OR function, then only one value of c will work.) For s_6^u to be well-defined, we need that whenever $s_0^i(a) = c$, the value of i is determined—since

$$s_6^u\left[s_0^i(A), t_0^i(B)\right] = S^u\left[s_0^i(A), B\right]$$

and so the only access that s_6^u has to i is through s_0^i. This implies that s_0^i has the form $s(A) \wedge i$ or $s(A) \wedge \neg i$ for some function s. Therefore s_6^u can be written as $s_7^u\left[s(A), i, t_0^i(B)\right]$ for some function s_7^u. Now repeat the argument for t_6^u. We obtain that t_6^u can be written as $t_7^u\left[s_0^i(A), i, t(B)\right]$ for some functions t_7^u.and t. Therefore

$$s_7^u\left[s(A), i, t_0^i(B)\right] = t_7^u\left[s_0^i(A), i, t(B)\right].$$

So we can take $t_0^i(B) = t(B)$ and $s_0^i(A) = s(A)$, and write s_7^u (as well as t_7^u) as $s_7^u[s(A), r(I), t(B)]$. □

Recall that a set $V \subseteq V_f$ is *unifiable* if there exists a vertex v, in some decomposition of f, such that $L(v) = V$.

LEMMA 6.6. *If V_S and V_T are unifiable, then either $V_S \cap V_T = \phi$, $V_S \subseteq V_T$, or $V_T \subseteq V_S$.*

Proof. Let $V_S = L(v_S)$ in decomposition S and $V_T = L(v_T)$ in decomposition T, and suppose V_S and V_T are incomparable. Let g_S be the function at v_S and g_T the function at v_T. Defining A, I, B, and U as before, from Lemma 6.5 there exist Boolean functions $s(A)$, $r(I)$, and $t(B)$ such that

$$g_S = h_S(s(A), r(I)),$$
$$g_T = h_T(r(I), t(B))$$

for some two-variable Boolean h_S and h_T. Also, there exists a restriction $U = u$ for which f depends on all three of $s(A)$, $r(I)$, and $t(B)$. So the question reduces to which Boolean $\eta(a, i, b)$ dependent on all three inputs are *associative*, in the sense that there exist Boolean η_1, η_2 and h_S, h_T for which

$$\eta(a, i, b) = \eta_1(h_S(a, i), b) = \eta_2(a, h_T(i, b)).$$

It is easily checked that the only possibilities are

$$(\neg)\,\mathrm{XOR}(a, i, b) \quad \text{or} \quad (\neg)\,\mathrm{AND}((\neg)\,a, (\neg)\,i, (\neg)\,b),$$

where $(\neg)$ denotes possible negation. Furthermore, η is determined up to negation given h_S and h_T, so η cannot depend on u. In both the XOR and the AND (or equivalently OR) cases, v_S and v_T would have been collapsed to a single vertex in both S and T, by properties (ii)-(v) of a tree decomposition. Contradiction. □

Now that we have ruled out the possibility of incomparable subtrees, we can establish uniqueness.

Proof. [of Theorem 6.3]It remains only to show that any decomposition must contain a vertex v with $L(v) = V$ for each unifiable V. Suppose that V is not represented in some decomposition F. Certainly $V \neq V_f$, so let V_P be the *parent set* of V in F: that is, the unique minimal set such that $V \subset V_P$ and there exists a vertex v_P in F with $L(v_P) = V_P$. Then the function at v_P is represented by a nontrivial tree, containing a vertex v with $L(v) = V$. For were it not so represented, then for any Boolean function g on V, there would exist a setting W of $V_P - V$ such that W, together with $g(V)$, would not suffice to determine the function h at v_P. Since f depends on h for some setting of $V_f - V_P$, it follows that v could not be a vertex in any decomposition. Furthermore, the function at v_P cannot be $(\neg)\,\mathrm{AND}_k$, $(\neg)\,\mathrm{OR}_k$, or $(\neg)\,\mathrm{XOR}_k$. If it were, then again v could not be a vertex in any decomposition, since it would need to be labeled correspondingly with $(\neg)\,\mathrm{AND}_k$, $(\neg)\,\mathrm{OR}_k$, or $(\neg)\,\mathrm{XOR}_k$. Having determined the unique set of vertices that comprise any decomposition, the vertices' labels are also determined up to double-negation. □

We now consider algorithms for finding the tree decomposition. First, given a subset G of V_f, there is a linear-time algorithm to decide whether a Boolean function tree representing f could have a vertex u with $L(u) = G$. Consider the set F of $2^{n-|G|}$ restrictions on G induced by setting all the variables in $V_f - G$ to constant values. A vertex could have $L(u) = G$ if and only if all restrictions in F are identical up to negation, omitting constant functions. This can be checked in $O(N)$ time, which leads to an $O(N^2)$ algorithm for finding all vertices in the tree decomposition. (As a postprocessing step, the algorithm prunes superfluous AND_k, OR_k, and XOR_k vertices.)

However, we can reduce the running time to $O(N^{\log_2 3} \log N)$ by being more careful about how we check whether all restrictions in F are identical. The idea is to represent each restriction by a concise *code number*, which takes up only $O(n)$ bits rather than $2^{|G|}$ bits. We create the code numbers recursively, starting with the smallest restrictions and working up to larger ones. The code numbers need to satisfy the following conditions:

1. Two restrictions S and T over the same set of variables get mapped to identical code numbers if and only if $S = T$.
2. If S is constant or S is the negation of T, then these facts are easy to tell given the code numbers of S and T.

We can satisfy these conditions by building up a binary tree of restrictions at each recursive call, then assigning each restriction a code number based on its position in the tree: 1 if it's the leftmost leaf, 2 if it's the second-to-leftmost, and so on. There are two exceptions: the constant 0 and 1 restrictions are assigned special code numbers Φ_0 and Φ_1 respectively; and if the negation of S was already assigned code number J, then S is assigned code number $-J$. For all $G \neq \phi$, each object inserted into the tree is two code numbers of size $|G| - 1$ restrictions concatenated together. Because this pair of code numbers is then 'hashed down' to a single number based on its position in the tree, the numbers always remain of size $O(n)$. In the pseudocode, B is the binary tree, $J[S]$ is the codeword of restriction S, and the operation $\odot$ denotes concatenation. The set VERTICES stores the final result: namely all sets $H \subseteq V_f$ such that there is a vertex u in the decomposition of f having $L(u) = H$. After the main loop of the algorithm, a postprocessing step deletes redundant AND_k, OR_k, and $(\neg)\,\text{XOR}_k$ vertices. This step looks for vertices u and v with $L(u)$ and $L(v)$ incomparable, which by Theorem 6.3 can only have arisen by AND_k, OR_k, or $(\neg)\,\text{XOR}_k$.

ALGORITHM 6. ***(decomposes a Boolean function)***
For all $G \subseteq V_f$ (in lexicographic order, starting with $G = \phi$):

1. *Initialize an empty self-balancing binary tree B.*
2. *Let Z be the set of all restrictions $S \in \{0, 1, *\}^n$ that fix exactly those variables not in G. Also, if $G \neq \phi$, then let k be the minimum i such that $v_i \in G$.*
3. *For all $S \in Z$,*
   - *If $G = \phi$ then insert $\Phi_{f(S)}$ into B. Otherwise, let S_0 and S_1 be further restrictions of S that fix v_k to 0 and 1 respectively, and insert $J[S_0] \odot J[S_1]$ into B.*
4. *For all $S \in Z$, assign S a code number $J[S]$ as follows.*
   - *For $i \in \{0, 1\}$, if $J[S_0] = J[S_1] = \Phi_i$, then $J[S] = \Phi_i$ also.*
   - *Otherwise, if $(-J[S_0]) \odot (-J[S_1])$ (corresponding to the negation $\neg S$ of S) is to the left of $J[S_0] \odot J[S_1]$ in B, then $J[S] = -J[\neg S]$.*
   - *Otherwise, if $J[S_0] \odot J[S_1]$ is the t^{th} leaf of B in left-to-right order, then $J[S] = t$.*
5. *If $|G| \geq 2$ and for all $S \in Z$, $|J[S]|$ is identical (omitting those S for which $J[S] = \Phi_0$ or $J[S] = \Phi_1$), then add G to* VERTICES*; otherwise do not.*

For each $i \in \{1, \ldots, n\}$, find all $G \in$ VERTICES such that $v_i \in G$. Attempt to sort them into an ascending sequence $G_1 \subset G_2 \subset \cdots$. If a G_i to be inserted is incomparable with some G_j in the sequence, then leave G_j in the sequence, do not insert G_i, and flag both G_i and G_j for removal.

Both the main loop and the second loop effectively perform an $O(\log N)$-time operation for all subsets of subsets of V_f. Therefore the total running time is $O(N^{\log_2 3} \log N)$.

7. Quantum Query Complexity. The quantum query complexity of a Boolean function f is the minimum number of oracle queries needed by a quantum computer to evaluate f. Here we are concerned only with the bounded-error query complexity $\mathrm{Q}_2(f)$ (defined in [7]), since approximating unitary matrices with finite precision introduces bounded error into any quantum algorithm. A quantum query algorithm Γ proceeds by an alternating sequence of $T+1$ unitary transformations and T query transformations: $U_0 \to Q_1 \to U_1 \to \cdots \to Q_T \to U_{T+1}$. Then $\mathrm{Q}_2(f)$ is the minimum of T over all Γ that compute f with bounded error.

As mentioned in Section 1, Barnum, Saks, and Szegedy have recently obtained a polynomial-time algorithm to approximate $Q_2(f)$. Here we show a weaker result: that if we limit the number of qubits, we can obtain a subexponential-time approximation algorithm via careful exhaustive search.

7.1. Overview of Result. For what follows, it will be convenient to extend the quantum oracle model to allow intermediate observations. With an unlimited workspace, this cannot decrease the number of queries needed [8]. In the space-bounded setting, however, it might make a larger difference.

We define a *composite algorithm* Γ' to be an alternating sequence $\Gamma_1 \to D_1 \to \cdots \to \Gamma_t \to D_t$. Each Γ_i is a quantum query algorithm that uses T_i queries and at most m qubits of memory for some $m \geq \log_2 n + 2$. When Γ_i terminates a basis state $|\psi_i\rangle$ is observed. Each D_i is a *decision point*, which takes as input the sequence $|\psi_1\rangle, \ldots, |\psi_i\rangle$, and as output decides whether to (1) halt and return $f = 0$, (2) halt and return $f = 1$, or (3) continue to Γ_{i+1}. (The final decision point, D_t, must select between (1) and (2).) There are no computational restrictions placed on the decision points. However, a decision point cannot modify the quantum algorithms that come later in the sequence; it can only decide whether to continue with the sequence. For a particular input, let p_k be the probability, over all runs of Γ', that quantum algorithm Γ_k is invoked. Then Γ' uses a total number of queries $\sum_{k=1}^{t} p_k T_k$.

We define the space-bounded quantum query complexity $SQ_{2,m}(f)$ to be the minimum number of queries used by any composite algorithm that computes f with error probability at most $1/3$ and that is restricted to m qubits. We give an approximation algorithm for $SQ_{2,m}(f)$ taking time $2^{O(4^m mn)}$, which when $m = O(\log n)$ is $O(N^{\mathrm{polylog}(N)})$. The approximation ratio is $\sqrt{22}/3 + \epsilon$ for any $\epsilon > 0$. The difficulty in proving the result is as follows.

A unitary transformation is represented by a continuous-valued matrix, which might suggest that the quantum model of computation is analog rather than digital. But Bernstein and Vazirani [8] showed that, for a quantum computation taking T steps, the matrix entries need to be accurate only to within $O(\log T)$ bits of precision in the bounded-error model. However, when we try represent unitary transformations on a computer with finite precision, a new problem arises. On the one hand, if we allow only matrices that are exactly unitary, we may not be able to approximate every unitary matrix. So we also need to admit matrices that are *almost* unitary. For example, we might admit a matrix if the norm of each row is sufficiently close to 1, and if the inner product of each pair of distinct rows is sufficiently close to 0. But how do we know that every such matrix is close to some actual unitary matrix? If it is not, then the transformation it represents cannot even approximately be realized by a quantum computer.

We resolve this issue as follows. First, we show that every almost-unitary matrix is close to some unitary matrix in a standard metric. Second, we show that every unitary matrix is close to some almost-unitary matrix representable with limited precision. Third, we upper-bound the precision that suffices for a quantum algorithm, given a fixed accuracy that the algorithm needs to attain.

An alternative approach to approximating $SQ_{2,m}(f)$ would be to represent each unitary matrix as a product of elementary gates. Kitaev [17] and independently Solovay [24] showed that a $2^m \times 2^m$ unitary matrix can be represented with arbitrary accuracy $\delta > 0$ by a product of $2^{O(m)\,\mathrm{polylog}(1/\delta)}$ unitary gates. But this yields a $2^{2^{O(m)\,\mathrm{polylog}(mn)}}$ algorithm, which is slower than ours. Perhaps the construction or its analysis can be improved; in any case, though, this approach is less natural for the setting of query complexity.

7.2. Almost-Unitary Matrices. Let $u \bullet v$ denote the conjugate inner product of u and v. The distance $|A - B|$ between matrices $A = (a_{ij})$ and $B = (b_{ij})$ in the $L_{\max}$ norm is defined to be $\max_{i,j} |a_{ij} - b_{ij}|$.

DEFINITION 7.1. *A matrix A is* q-almost-unitary *if* $|I - AA^{\dagger}| < q$.

In the following lemma, we start with an almost-unitary matrix A and construct an actual unitary matrix U that is close to A in the $L_{\max}$ norm.

LEMMA 7.2. *Let A be a q-almost-unitary $s \times s$ matrix, with $s \geq 2$ and $q \leq 1/4s$. Then there exists a unitary matrix U such that $|A - U| < 4.91q\sqrt{s}$.*

Proof. We first normalize each row A_i so that $A_i \bullet A_i = 1$. For each entry a_{ij},

$$|a_{ij}/(A_i \bullet A_i) - a_{ij}| = |a_{ij}||1 - (A_i \bullet A_i)|/|A_i \bullet A_i| < q\,(1+q)\,/\,(1-q)\,.$$

We next form a unitary matrix B from A by using the Classical Gram-Schmidt (CGS) orthogonalization procedure. The idea is to project A_2 to make it orthogonal to A_1, then project A_3 to make it orthogonal to both A_1 and A_2, and so on. Initially we set $B_1 \leftarrow A_1$. Then for each $2 \leq i \leq s$, we set $B_i \leftarrow A_i - \sum_{j=1}^{i-1} (A_i \bullet B_j) B_j$. Therefore

$$A_i \bullet B_k = (A_i \bullet A_k) - \textstyle\sum_{j=1}^{k-1} (A_i \bullet B_j)(A_k \bullet B_j).$$

We need to show that the discrepancy between A and B does not increase too drastically as the recursion proceeds. Let $\sigma_k = \max_i A_i \bullet B_k$. By hypothesis, $\sigma_1 < q$. Then $\sigma_k \leq \sigma_1 + \sum_{j=1}^{k-1} \sigma_j^2$. Assume that $\sigma_k < q + 4q^2 s$ for all $k \leq K$. By induction,

$$\sigma_{K+1} < q + K\left(q + 4q^2 s\right)^2 \leq q + 4q^2 s$$

since $q \leq 1/4s$ and $K \leq s$. So for all k, $\sigma_k < q + 4q^2 s$.

Let $\phi = |A - B|$. By the definition of B, $\phi \leq \sigma_1 |w_1| + \cdots + \sigma_s |w_s|$ where w is a column of B. Since $|w_1|^2 + \cdots + |w_s|^2 = 1$, ϕ is maximized when $w_i = \sigma_i \sqrt{s}/\left(\sigma_1 + \cdots + \sigma_s\right)$, or

$$\phi \leq \sigma_1^2 + \cdots + \sigma_s^2 \sqrt{s}/(\sigma_1 + \cdots + \sigma_s) \leq \left(q + 4q^2 s\right)^2 \sqrt{s}/q.$$

Adding $q\,(1+q)\,/\,(1-q)$ from normalization yields a quantity less than $\left(4 + 9\sqrt{2}/14\right) q\sqrt{s} \approx 4.91q\sqrt{s}$. This can be seen by working out the arithmetic for the worst case of $s = 2$, $q = 1/4s$. ∎

The next lemma, which is similar to Lemma 6.1.3 of [8], is a sort of converse to Lemma 7.2: we start with an arbitrary unitary matrix, and show that truncating its entries to a precision $\delta > 0$ produces an almost-unitary matrix.

LEMMA 7.3. *Let U and V be $s \times s$ matrices with $s \geq 2$ and $|U - V| < \delta$. If U is unitary, then V is $\left(2\delta\sqrt{s} + \delta^2 s\right)$-almost-unitary.*

Proof. First,

$$U_i \bullet U_i = \sum_{k=1}^{s} |u_k + \gamma_k|^2 = 1 + \sum_{k=1}^{s} \left(u_k \gamma_k^* + u_k^* \gamma_k + \gamma_k \gamma_k^*\right)$$

where the u_k's are entries of U and the γ_k's are error terms satisfying $|\gamma_k| < \delta$. So by the Cauchy-Schwarz inequality, $U_i \bullet U_i$ differs from 1 by at most $2\delta\sqrt{s} + \delta^2 s$. Second, for $i \neq j$,

$$U_i \bullet U_j = \sum_{k=1}^{s} (u_k + \gamma_k)(u_k + \eta_k)^*$$

where the γ_k's and η_k's are error terms, and the argument proceeds analogously. ∎

7.3. Searching for Quantum Algorithms. In this section we use the results on almost-unitary matrices to construct an algorithm. First we need a lemma about error buildup in quantum algorithms, which is similar to Corollary 3.4.4 of [8] (though the proof technique is different).

LEMMA 7.4. *Let $U_1, \ldots U_T$ be $s \times s$ unitary matrices, $\widehat{U_1}, \ldots \widehat{U_T}$ be $s \times s$ arbitrary matrices, and v be an $s \times 1$ vector with $\|v\|_2 = 1$. Suppose that, for all i, $\left|\widehat{U_i} - U_i\right| < 1/cs$, where $c > T/2$. Then $\widehat{U_1} \cdots \widehat{U_T} v$ differs from $U_1 \cdots U_T v$ by at most $2T/\left[\sqrt{s}\,(2c - T)\right]$ in the L_2 norm.*

Proof. For each i, let $E_i = \widehat{U_i} - U_i$. By hypothesis, every entry of E_i has magnitude at most $1/cs$; thus, each row or column w of E_i has $\|w\|_2 \leq 1/\left(c\sqrt{s}\right)$. Then

$$\widehat{U_1} \cdots \widehat{U_T} v = \left(U_1 + E_1\right) \cdots \left(U_T + E_T\right) v.$$

The right-hand side, when expanded, has 2^T terms. Any term containing k matrices E_i has L_2 norm at most $s^{-1/2} c^{-k}$, and can therefore add at most $c^{-k}/\sqrt{s}$ to the discrepancy with $U_1 \cdots U_T v$. So the total discrepancy is at most

$$s^{-1/2} \sum_{k=1}^{T} \binom{T}{k} (1/c)^k < s^{-1/2} \left(e^{T/c} - 1\right).$$

Since $d \ln t / dt$ evaluated at $t = 2c$ is $1/2c$ and since $\ln t$ is concave,

$$\ln(2c + T) - \ln(2c - T) \geq 2T/2c = T/c$$

when $T < 2c$. Therefore $e^{T/c} \leq (2c + T)/(2c - T)$ and the discrepancy is at most $2T/\left[\sqrt{s}\,(2c - T)\right]$ in the L_2 norm. ∎

Applying Lemmas 7.2, 7.3, and 7.4, we now prove the main theorem.

THEOREM 7.5. *There exists an approximation algorithm for $\mathrm{SQ}_{2,m}(f)$ taking time $2^{O(4^m mn)}$, with approximation ratio $\sqrt{22}/3 + \epsilon$.*

Proof. Given f, we want, subject to the following two constraints, to find an algorithm Γ that approximates f with a minimum number of queries. First, Γ uses at most m qubits, meaning that $s = 2^m$ and the relevant matrices are $2^m \times 2^m$. Second, the correctness probability of Γ is known to a constant accuracy $\pm\varepsilon$. Certainly the number T of queries never needs to be more than n, for, although each quantum algorithm is space-bounded, the *composite* algorithm need not be. Let λ be the $L_{\max}$ error we can tolerate in the matrices, and let Δ be the resultant L_2 error in the final states. Setting $c = 1/\left(\lambda 2^m\right)$, by Lemma 7.4 we have

$$\Delta \leq 2n/\left[2^{m/2}\left(2^{1-m}/\lambda - n\right)\right].$$

From the Cauchy-Schwarz inequality, one can show that $\varepsilon \leq 2\Delta$. Then solving for $1/\lambda$, $1/\lambda \leq 2^{m/2} n\,(2/\varepsilon + 1)$ which, since ε is constant, is $O\left(2^{m/2} n\right)$. Solving for c, we can verify that $c > T/2$, as required by Lemma 7.4. If we generate almost-unitary matrices, they need to be within λ of actual unitary matrices. By Lemma 7.2 we can use $\lambda/\left(4.91\sqrt{s}\right)$-almost-unitary matrices. Finally we need to ensure that we approximate every unitary matrix. Let δ be the needed precision. Invoking Lemma 7.3, we set $\lambda/\left(4.91\sqrt{s}\right) \geq 2\delta\sqrt{s} + \delta^2 s$ and obtain that

$$\delta \leq \max\left[\lambda/\left(9.82s\right), \lambda^{1/2}/\left(2.22 s^{3/4}\right)\right]$$

is sufficient.

Therefore the number of bits of precision needed per entry, $\log(1/\delta)$, is $O(m)$. We thus need only $O(4^m mn)$ bits to specify Γ, and can search through all possible Γ in time $2^{O(4^m mn)}$. The amount of time needed to evaluate a composite algorithm Γ' is polynomial in m and n, and is absorbed into the exponent. The approximation algorithm is this: first let $\varepsilon > 0$ be a constant at most 0.0268, and let $\omega = \frac{22}{9} + \frac{4}{3}\varepsilon - 8\varepsilon^2$. Then find the smallest T such that the maximum probability of correctness over all T-query algorithms Γ' is at least $2/3 - \varepsilon$ (subject to $\pm\varepsilon$ uncertainty), and return $T\sqrt{\omega}$. The algorithm achieves an approximation ratio of $\sqrt{\omega}$, for the following reason. First, $T \leq \mathrm{SQ}_{2,m}(f)$. Second, $\omega T \geq \mathrm{SQ}_{2,m}(f)$, since by repeating the optimal algorithm Γ^* until it returns the same answer twice (which takes either two or three repetitions), the correctness probability can be boosted above $2/3$. Finally, a simple calculation reveals that Γ^* returns the same answer twice after expected number of invocations ω. $\square$

8. Open Problems. Implicit in the paper of Ambainis [3] is a novel Boolean function property, which is used to obtain lower bounds on quantum query complexity. To take a special case: given a function f and a set S of inputs, let the "Ambainis density" $\mathrm{AD}_S(f)$ be the minimum, over all $X \in S$, of the number of i such that $X^{(i)} \in S$ and $f(X) \neq f\left(X^{(i)}\right)$. (Here $X^{(i)}$ denotes X with the i^{th} bit negated.) Then let $\mathrm{AD}(f)$ be the maximum of $\mathrm{AD}_S(f)$ over all S. Ambainis shows that $\mathrm{Q}_2(f) = \Omega(\mathrm{AD}(f))$. How efficient an algorithm can we find for $\mathrm{AD}(f)$? Additionally, Bar-Yossef, Kumar, and Sivakumar [6] have defined "approximate" versions of measures such as block sensitivity. Can we extend the algorithms given in this paper to compute those measures?

9. Acknowledgments. I thank Rob Pike and Lorenz Huelsbergen for sponsoring the internship during which this work was done and for helpful discussions; Andris Ambainis, Wim van Dam, Michael Saks, Umesh Vazirani, and the anonymous reviewers for their comments; Ronald de Wolf for discussions of space-bounded quantum query complexity; and Peter Bro Miltersen for correspondence.

REFERENCES

[1] S. Aaronson, *Quantum certificate complexity*, to appear in Proc. IEEE Conf. on Comp. Complexity, 2003.
[2] S. Aaronson, *Quantum lower bound for the collision problem*, in Proc. 34th ACM STOC, pp. 635–642, 2002.
[3] A. Ambainis, *Quantum lower bounds by quantum arguments*, J. Comput. System Sci., 64, pp. 750–767, 2002. Earlier version in STOC'2000. quant-ph/0002066.
[4] A. Ambainis and R. de Wolf, *Average-case quantum query complexity*, in Proc. Symposium on Theoretical Aspects of Comp. Sci., 2000. quant-ph/9904079.
[5] H. Barnum, M. Saks, and M. Szegedy. *Quantum decision trees and semidefinite programming*, to appear in Proc. IEEE Conf. on Comp. Complexity, 2003.
[6] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, *Sampling algorithms: lower bounds and applications*, in Proc. 33rd ACM STOC, pp. 266–275, 2001.
[7] R. Beals, H. Buhrman, R. Cleve, M. Mosca, and R. de Wolf, *Quantum lower bounds by polynomials*, J. ACM, 48(4), pp. 778–797, 2001. quant-ph/9802049. Earlier version in FOCS'98.
[8] E. Bernstein and U. Vazirani, *Quantum complexity theory*, SIAM J. Comput., 26:5, pp. 1411–1473, 1997.
[9] H. Buhrman and R. de Wolf, *Complexity measures and decision tree complexity: a survey*, Theoretical Comp. Sci., 288, pp. 21–43, 2002.
[10] S. L. A. Czort, *The complexity of minimizing disjunctive normal form formulas*, Master's Thesis, University of Aarhus, 1999.

[11] W. van Dam, *Quantum oracle interrogation: getting all the information for almost half the price*, in Proc. 39th IEEE FOCS, pp. 362–367, 1998. quant-ph/9805006.

[12] D. Guijarro, V. Lavín, and V. Raghavan, *Exact learning when irrelevant variables abound*, Information Proc. Lett., 70, pp. 233–239, 1999.

[13] L. K. Grover, *A fast quantum mechanical algorithm for database search*, in Proc. 28th ACM STOC, pp. 212–219, 1996. quant-ph/9605043.

[14] T. Hancock, T. Jiang, M. Li, and J. Tromp, *Lower bounds on learning decision lists and trees*, Information and Computation, 126, pp. 114–122, 1996.

[15] K. Hoffman and R. Kunze, *Linear Algebra*, Prentice Hall, 1971.

[16] L. Hyafil and R. L. Rivest, *Constructing optimal binary decision trees is NP-complete*, Information Proc. Lett., 5, pp. 15–17, 1976.

[17] A. Yu. Kitaev, *Quantum computations: algorithms and error correction*, Russian Math. Surveys, 52:6, pp. 1191–1249, 1997.

[18] N. Nisan, *CREW PRAMs and decision trees*, SIAM J. Comput., 20:6, pp. 999–1007, 1991.

[19] N. Nisan and M. Szegedy, *On the degree of Boolean functions as real polynomials*, Computational Complexity, 4(4), pp. 301–313, 1994. Earlier version in STOC'92.

[20] A. A. Razborov and S. Rudich, *Natural proofs*, J. Comput. System Sci., 55, pp. 24–35, 1997.

[21] M. Saks and A. Wigderson, *Probabilistic Boolean decision trees and the complexity of evaluating game trees*, in Proc. 27th IEEE FOCS, pp. 29–38, 1986.

[22] M. Santha, *On the Monte Carlo decision tree complexity of read-once formulae*, Random Structures and Algorithms 6:1, pp. 75–87, 1995.

[23] D. Sieling, *Minimization of decision trees is hard to approximate*, to appear in Proc. IEEE Conf. on Comp. Complexity, 2003.

[24] R. Solovay, *Lie groups and quantum circuits*, talk at workshop on Mathematics of Quantum Computation, Mathematical Sciences Research Institute, Spring 2000.