Quantum Algorithm for Finding the Optimal Variable Ordering for Binary Decision Diagrams

Seiichiro Tani
NTT Communication Science Laboratories, NTT Corporation, Atsugi, Japan
seiichiro.tani.cs@hco.ntt.co.jp

Abstract
An ordered binary decision diagram (OBDD) is a directed acyclic graph that represents a Boolean function. Since OBDDs have many nice properties as data structures, they have been extensively studied for decades in both theoretical and practical fields, such as VLSI (Very Large Scale Integration) design, formal verification, machine learning, and combinatorial problems. Arguably, the most crucial problem in using OBDDs is that they may vary exponentially in size depending on their variable ordering (i.e., the order in which the variables are to be read) when they represent the same function. Indeed, it is NP hard to find an optimal variable ordering that minimizes an OBDD for a given function. Friedman and Supowit provided a clever deterministic algorithm with time/space complexity $O^*(3^n)$, where $n$ is the number of variables of the function, which is much better than the trivial brute-force bound $O^*(n!2^n)$. This paper shows that a further speedup is possible with quantum computers by presenting a quantum algorithm that produces a minimum OBDD together with the corresponding variable ordering in $O^*(2.77286^n)$ time and space with an exponentially small error probability. Moreover, this algorithm can be adapted to constructing other minimum decision diagrams such as zero-suppressed BDDs.

2012 ACM Subject Classification Theory of computation → Quantum computation theory; Theory of computation → Data structures design and analysis

Keywords and phrases Binary Decision Diagram, Variable Ordering, Quantum Algorithm

Digital Object Identifier 10.4230/LIPIcs.SWAT.2020.36


1 Introduction

1.1 Background

Ordered binary decision diagrams

The ordered binary decision diagram (OBDD) is one of the data structures that have been most often used for decades to represent Boolean functions in practical situations, such as VLSI design, formal verification, optimization of combinatorial problems, and machine learning, and it has been extensively studied from both theoretical and practical standpoint (see standard textbooks and surveys, e.g., Refs. [8, 12, 7]). Moreover, many variants of OBDDs have been invented to more efficiently represent data with properties observed frequently in specific applications. More technically speaking, OBDDs are directed acyclic graphs that represent Boolean functions and also known as special cases of oblivious read-once branching programs in the field of complexity theory. The reason for OBDDs’ popularity lies in their nice properties – they can be uniquely determined up to isomorphism for each function once variable ordering (i.e., the order in which to read the variables) is fixed and, thanks to this property, the equivalence of functions can be checked by just testing the isomorphism between the OBDDs representing the functions. In addition, binary operations such as AND and OR between two functions can be performed efficiently over the OBDDs representing those functions [2]. Since these properties are essential in many applications, OBDDs...
have gathered much attention from various research fields. To enjoy these nice properties, however, we actually need to address a crucial problem, which is that OBDDs may vary exponentially in size depending on their variable ordering. For instance, a Boolean function \(f(x_1, \ldots, x_{2n}) = x_1 x_2 + x_3 x_4 + \cdots + x_{2n-1} x_{2n}\) has a \((2n + 2)\)-sized OBDD for the ordering \((x_1, \ldots, x_{2n})\) and a \(2^{n+1}\)-sized OBDD for the ordering \((x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n})\) [8, Sec. 8.1] (see Figure 1 for the case where \(n = 6\)). This is not a rare phenomenon; it could happen in many concrete functions that one encounters. Thus, since the early stages of OBDD research, one of the most central problems has been how to find an optimal variable ordering, i.e., one that minimizes OBDDs. Since there are \(n!\) permutations over \(n\) variables, the brute-force search requires at least \(n! = 2^{\Omega(n \log n)}\) time to find an optimal variable ordering. Indeed, finding an optimal variable ordering for a given function is an NP hard problem (see a short survey in the full paper [11]).

To tackle this high complexity, many heuristics have been proposed to find an optimal variable ordering or a relatively good one. These heuristics work well for Boolean functions appearing in specific applications since they are based on very insightful observations, but they do not guarantee a worst-case time complexity lower than that achievable with the brute-force search. The only algorithm with a much lower worst-case time complexity bound, \(O^*(3^n)\) time (\(O^*(\cdot)\) hides a polynomial factor), than the brute-force bound \(O^*(n!2^n)\) for all Boolean functions with \(n\) variables was provided by Friedman and Supowit [5], and that was almost thirty years ago!

### 1.2 Our Results

In this paper, we show that quantum speedup is possible for the problem of finding an optimal variable ordering of the OBDD for a given function. This is the first quantum speedup for the OBDD-related problems. Our algorithms assume the quantum random access memory (QRAM) model [6], which is commonly used in the literature concerned with quantum algorithms. In the model, one can read contents from or write them into quantum memory in a superposition. We provide our main result in the following theorem.

> **Theorem 1.** There exists a quantum algorithm that, for a function \(f: \{0, 1\}^n \rightarrow \{0, 1\}\) given as its truth table, produces a minimum OBDD representing \(f\) together with the corresponding variable ordering in \(O^*(\gamma^n)\) time and space with an exponentially small error probability with respect to \(n\), where the constant \(\gamma\) is at most 2.77286. Moreover, the OBDD produced by our algorithm is always a valid one for \(f\), although it is not minimum with an exponentially small probability.

This improves upon the classical best bound \(O^*(3^n)\) [5] on time/space complexity. The classical algorithm achieving this bound is a deterministic one. However, there are no randomized algorithms that compute an optimal variable ordering in asymptotically less time complexity as far as we know.

It may seem somewhat restricted to assume that the function \(f\) is given as its truth table, since there are other common representations of Boolean functions such as DNFs, CNFs, Boolean circuits and OBDDs. However, this is not the case. Our algorithm actually works in more general settings where the input function \(f\) is given as any representation such that the value of \(f\) on any specified assignment can be computed over the representation in polynomial time in \(n\), such as polynomial-size DNFs/CNFs/circuits and OBDDs of any size. This is because, in such cases, the truth table of \(f\) can be prepared in \(O^*(2^n)\) time and the minimum OBDD is computable from that truth table with our algorithm. We restate Theorem 1 in a more general form as follows.
Corollary 2. Let $R(f)$ be any representation of a Boolean function $f$ with $n$ variables such that the value of $f(x)$ on any given assignment $x \in \{0,1\}^n$ can be computed on $R(f)$ in polynomial time with respect to $n$. Then, there exists a quantum algorithm that, for a function $f: \{0,1\}^n \rightarrow \{0,1\}$ given as $R(f)$, produces a minimum OBDD representing $f$ together with the corresponding variable ordering in $O^*(\gamma^n)$ time and space with an exponentially small error probability with respect to $n$, where the constant $\gamma$ is at most $2.77286$. Possible representations as $R(f)$ are polynomial-size DNFs/CNFs/circuits and OBDDs of any size for function $f$.

There are many variants of OBDDs, among which the zero-suppressed BDDs (ZDDs or ZBDDs) introduced by Minato [9] have been shown to be very powerful in dealing with combinatorial problems (see Knuth’s famous book [7] for how to apply ZDDs to such problems). With slight modifications, our algorithm can construct a minimum ZDD with the same time/space complexity. We believe that similar speedups are possible for many other variants of OBDDs (adapting our algorithm to multiterminal BDDs (MTBDDs) [8] is almost trivial).

Technical Contribution

Recently, Ambainis et al. [1] has introduced break-through quantum techniques to speed up classical dynamic programming approaches. Inspired by their technique, our quantum algorithm speeds up the classical one (called FS) discovered by Friedman and Supowit [5]. Ambainis et al.’s results depend on the property that a large problem can be divided into subproblems that can be regarded as a scale-down version of the original problem and can be solved with the same algorithm, as is often the case with graph problems. In our case, firstly, it is unclear whether the problem can be divided into subproblems. Secondly, subproblems would be to optimize the ordering of variables starting from the middle variable or even from the opposite end, i.e., from the variable to be read first, toward the one to be read last. Such subproblems cannot be solved with the algorithm FS, and, in particular, optimizing in the latter case essentially requires the equivalence check of subfunctions of $f$, which is very costly. Our technical contribution is to find, by carefully observing the unique properties of OBDDs, that it is actually possible to even recursively divide the original problem into not the same but somewhat similar kinds of subproblems, to generalize the algorithm FS so that it can solve the subproblems, and to use the quantum minimum finding algorithm to efficiently select the subproblems that essentially contribute to the optimal variable ordering. In the full paper [11], we provide the technical outline of our algorithm, which would help readers understand the structure of our algorithm.

2 Preliminaries

2.1 Basic Terminology

Let $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$ be the sets of natural numbers, integers, and real numbers, respectively. For each $n \in \mathbb{N}$, let $[n]$ be the set $\{1, \ldots, n\}$, and $S_n$ be the permutation group over $[n]$. We may denote a singleton set $\{k\}$ by $k$ for notational simplicity if it is clear from the context; for instance, $I \setminus \{k\}$ may be denoted by $I \setminus k$, if we know $I$ is a set. For any subset $I \subseteq [n]$, let $\Pi_n(I)$ be the set of $\pi \in S_n$ such that the first $|I|$ members $\{\pi[1], \ldots, \pi[|I|]\}$ constitutes $I$, i.e.,

$$\Pi_n(I) := \{\pi \in S_n : \{\pi[1], \ldots, \pi[|I|]\} = I\} \subseteq S_n.$$
For simplicity, we omit the subscript $n$ and write $\Pi(I)$. More generally, for any two disjoint subsets $I, J \subseteq [n]$, let

$$\Pi_n((I, J)) := \{ \pi \in S_n : [\pi[1], \ldots, \pi[I]] = I, [\pi[I + 1], \ldots, \pi[I + |J|]] = J \} \subseteq S_n.$$  

For any disjoint subsets $I_1, \ldots, I_m \subseteq [n]$ for $m \in [n]$, $\Pi_n((I_1, \ldots, I_m))$ is defined similarly. For simplicity, we may denote $(I)$ by $I$, if it is clear from the context.

We denote the union operation over disjoint sets by $\sqcup$ (instead of $\cup$) when we emphasize the disjointness of the sets.

For $n$ Boolean variables $x_1, \ldots, x_n$, any set $I \subseteq [n]$, and any vector $b = (b_1, \ldots, b_{|I|}) \in \{0, 1\}^{|I|}$, $x_I$ denotes the ordered set $(x_{j_1}, \ldots, x_{j_{|I|}})$, where $\{j_1, \ldots, j_{|I|}\} = I$ and $j_1 < \cdots < j_{|I|}$, and $x_I = b$ denotes $x_{j_i} = b_i$ for each $i = |I|$. For any Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ with variables $x_1, \ldots, x_n$, we denote by $f_{|x_I=b}$ the function obtained by restricting $f$ with $x_I = b$. If $I$ is a singleton set, say, $I = \{i\}$, we may write $x_i$ and $f_{|x_i=b}$ to mean $x_{(i)}$ and $f_{|x_{(i)}=b}$, respectively, for notational simplicity. We say that $g$ is a subfunction of $f$ if $g$ is equivalent to the function $f_{|x_I=b}$ for some $I \subseteq [n]$ and $b \in \{0, 1\}^{|I|}$.

For any function $g(n)$ in $n$, we use the notation $O^*(g(n))$ to hide a polynomial factor in $n$. We further denote $X = O^*(Y)$ by $X \lesssim Y$.

We use the following upper bound many times in this paper. For $n \in \mathbb{N}$ and $k \in [n] \cup \{0\}$, it holds that

$$\binom{n}{k} \leq 2^n H(k/n),$$

where $H(\cdot)$ represents the binary entropy function $H(\delta) := -\delta \log_2 \delta - (1-\delta) \log_2 (1-\delta)$.

### 2.2 Ordered Binary Decision Diagrams

We provide a quick review of OBDDs. For more details, consult standard textbooks (e.g., Refs. [8, 12]).

For any Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ over variables $x_1, \ldots, x_n$ and any permutation $\pi \in S_n$ (called a variable ordering), an OBDD $B(f, \pi)$ is a single-rooted directed acyclic graph $G(V, E)$ that is unique up to isomorphism, defined as follows (examples are shown in Figure 1).

1. The node set $V$ is the union of two disjoint sets $N$ and $T$ of non-terminal nodes with out-degree two and terminal nodes with out-degree zero, respectively, where $T$ contains exactly two nodes: $T = \{t, f\}$. The set $N$ contains a unique source node $r$, called the root.
2. $B(f, \pi)$ is a leveled graph with $n+1$ levels. Namely, the node set can be partitioned into $n$ subsets: $V := V_0 \sqcup V_1 \sqcup \cdots \sqcup V_n$, where $V_n = \{r\}$ and $V_0 = T = \{t, f\}$, such that each directed edge $(u, v) \in E$ is in $V_i \times V_j$ for a pair $(i, j) \in [n] \times ([0] \sqcup [n-1])$ with $i > j$. For each $i \in [n]$, subset $V_i$ (called the level $i$) is associated with the variable $x_{\pi[i]}$, or, alternatively, each node in $V_i$ is labeled with $x_{\pi[i]}$. For convenience, we define a map $\text{var}: N \rightarrow [n]$ such that if $v \in V_i$ then $\text{var} = \pi[i]$.
3. The two edges emanating from every non-terminal node $v$ are called the 0-edge and the 1-edge, which are labeled with 0 and 1, respectively. For every $u \in N$, let $u_0$ and $u_1$ be the destinations of the 0-edge and 1-edge of $u$, respectively.
4. Let $\mathcal{F}(f)$ be the set of all subfunctions of $f$. Define a bijective map $F : V \rightarrow \mathcal{F}(f)$ as follows: (a) $F(r) = f$ for $r \in V_0$; (b) $F(t) = \text{true}$ and $F(f) = \text{false}$ for $(t, f) \in V_0$; (c) For every $u \in N$ and $b \in \{0, 1\}$, $F(u_b)$ is the subfunction obtained from $F(u)$ by substituting $x_{\pi[u]}$ with $b$, i.e., $F(u_b) = F(u)_{|x_{\pi[u]}=b}$.

---

1 In the standard definition, $V_i$ is associated with the variable $x_{\pi[n-i]}$. Our definition follows the one given in [5] to avoid complicated subscripts of variables.
The following well-known lemma captures the essential property of OBDDs. It states that the number of nodes at level $\pi$ is equal to the number of nodes in level $\pi + 1$ for any $\pi$ in $[n]$, where $[n]$ denotes the set of nodes at level $n$. This property underlines the fact that the OBDDs represent the function $f(x_1, x_2, x_3, x_4, x_5, x_6) = x_1 x_2 + x_3 x_4 + x_5 x_6$ under two variable orderings: $(x_1, x_2, x_3, x_4, x_5, x_6)$ (left) and $(x_1, x_3, x_5, x_2, x_4, x_6)$ (right), where the solid and dotted arcs express $0$-edges and $1$-edges, respectively, and the terminal nodes for true and false are labeled with T and F, respectively. For each $n \in \mathbb{N}$, the function $f(x_1, \ldots, x_{2n}) = x_1 x_2 + x_3 x_4 + \cdots + x_{2n-1} x_{2n}$ has a $(2n + 2)$-sized OBDD for the ordering $(x_1, \ldots, x_{2n})$ and a $2^{n+1}$-sized OBDD for the ordering $(x_1, x_2, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n})$.

5. $B(f, \pi)$ must be minimal in the sense that the following reduction rules cannot be applied. In other words, $B(f, \pi)$ is obtained by maximally applying the following rules:
   a. if there exists a redundant node $u \in N$, then remove $u$ and its outgoing edges, and redirect all the incoming edges of $u$ to $u_0$, where a node $u$ is redundant if $u_0$ is the same node as $u_1$.
   b. if there exist equivalent nodes $\{u, v\} \subseteq N$, then remove the possibly redundant node $u$ and its incoming edges, and redirect all the incoming edges of $v$ to $u$, where $u$ and $v$ are equivalent if (1) $\operatorname{var}(u) = \operatorname{var}(v)$, and (2) $u_0$ and $u_1$ are the same nodes as $v_0$ and $v_1$, respectively.

For each $j \in [n]$, $\operatorname{Cost}_j(f, \pi)$ denotes the width at the level associated with the variable $x_j$, namely, the number of nodes in the level $\pi^{-1}[j]$ (see Figure 2 in Appendix). For $I \subseteq [n]$, let $\pi_I$ be a permutation $\pi$ in $\Pi(I)$ that minimizes the number of nodes in level 1 to level $|I|$: \[
\pi_I := \arg \min \left\{ \sum_{j=1}^{|I|} \operatorname{Cost}_{\pi_I}(j, \pi) : \pi \in \Pi(I) \right\}.
\] (1)

Note that $\sum_{j=1}^{|I|} \operatorname{Cost}_{\pi_I}(j, \pi) = \sum_{\pi \in \Pi(I)} \operatorname{Cost}_{\pi_I}(f, \pi)$ for $\pi \in \Pi(I)$. More generally, for disjoint subsets $I_1, \ldots, I_m \subseteq [n]$, $\pi_{(I_1, \ldots, I_m)}$ is a permutation in $\Pi(I_1, \ldots, I_m)$ that minimizes the number of nodes in level 1 to level $|I_1| + \cdots + |I_m|$ over all $\pi \in \Pi(I_1, \ldots, I_m)$: \[
\pi_{(I_1, \ldots, I_m)} := \arg \min \left\{ \sum_{j=1}^{|I_1|+\cdots+|I_m|} \operatorname{Cost}_{\pi_{I_1, \ldots, I_m}}(j, \pi) : \pi \in \Pi(I_1, \ldots, I_m) \right\}.
\] (2)

Note that $\min \sum_{j=1}^{|I_1|+\cdots+|I_m|} \operatorname{Cost}_{\pi_{I_1, \ldots, I_m}}(f, \pi) = \sum_{\pi \in \Pi(I_1, \ldots, I_m)} \operatorname{Cost}_{\pi_I}(f, \pi)$ for any $\pi \in \Pi(I_1, \ldots, I_m)$. The following well-known lemma captures the essential property of OBDDs. It states that the number of nodes at level $i \in [n]$ is constant over all $\pi$, provided that the two sets $\{\pi[1], \ldots, \pi[i-1]\}$ and $\{\pi[i+1], \ldots, \pi[n]\}$ are fixed (see Figure 3 in Appendix).
Lemma 3 ([5]). For any non-empty subset \( I \subseteq [n] \) and any \( i \in I \), there exists a constant \( c_f \) such that, for each \( \pi \in \Pi((I \setminus \{i\}, \{i\})) \), \( \text{Cost}_\pi[I](f, \pi) \equiv \text{Cost}_\pi[I](f) = c_f \).

For convenience, we define shorthand for the minimums of the sums in Eqs. (1) and (2). For \( I' \subseteq I \subseteq [n] \), \( \text{Mincost}_I[I'] \) is defined as the number of nodes in the levels associated with variables indexed by elements in \( I' \) under permutation \( \pi_I \), namely, \( \text{Mincost}_I[I'] := \sum_{i \in I'} \text{Cost}_i(f, \pi_I) \). More generally, for disjoint subsets \( I_1, \ldots, I_m \subseteq [n] \) and \( I' \subseteq I_1 \cup \cdots \cup I_m \),

\[
\text{Mincost}(I_1, \ldots, I_m)[I'] := \sum_{i \in I'} \text{Cost}_i(f, \pi(I_1, \ldots, I_m)).
\]

As a special case, we denote \( \text{Mincost}(I_1, \ldots, I_m)[I_1 \cup \cdots \cup I_m] \) by \( \text{Mincost}(I_1, \ldots, I_m) \). We define \( \text{Mincost}_\emptyset \) as 0.

2.3 The Algorithm by Friedman and Supowit

This subsection reviews the algorithm by Friedman and Supowit [5]. We will generalize their idea later and heavily use the generalized form in our quantum algorithm. Hereafter, we call their algorithm FS.

2.3.1 Key Lemma and Data Structures

The following lemma is the basis of the dynamic programming approach used in FS.

Lemma 4. For any non-empty subset \( I \subseteq [n] \) and any Boolean function \( f : \{0,1\}^n \rightarrow \{0,1\} \), the following holds:

\[
\text{Mincost}_I = \min_{k \in I} \left( \text{Mincost}_{I \setminus k} + \text{Cost}_k(f, \pi(I \setminus k)) \right) = \min_{k \in I} \left( \text{Mincost}_{I \setminus k} \right).
\]

The proof is given in the full paper [11].

Before sketching algorithm FS, we provide several definitions. For any \( I \subseteq [n] \), \( \text{table}_I \) is an array with \( 2^{n-|I|} \) cells each of which stores a non-negative integer. Intuitively, for \( b \in \{0,1\}^{n-|I|} \), the cell \( \text{table}_I[b] \) stores (the pointer to) the unique node of \( \mathcal{B}(f, \pi_I) \) associated via \( f \) with function \( f|_{[n] \setminus I} = b \). Hence, we may write \( \text{table}_I[x_{[n] \setminus I} = b] \) instead of \( \text{table}_I[b] \) to clearly indicate the value assigned to each variable \( x_j \) for \( j \in [n] \setminus I \). The purpose of \( \text{table}_I \) is to relate all subfunctions \( f|_{[n] \setminus I} = b \) (\( b \in \{0,1\}^{n-|I|} \)) to the corresponding nodes of \( \mathcal{B}(f, \pi_I) \). We assume without loss of generality that the pointers to nodes of \( \mathcal{B}(f, \pi_I) \) are non-negative integers and, in particular, those to the two terminal nodes corresponding to false and true are the integers 0 and 1, respectively. Thus, \( \text{table}_\emptyset \) is merely the truth table of \( f \).

Algorithm FS computes \( \text{table}_I \) together with \( \pi_I \), \( \text{mincost}_I \), and another data structure, \( \text{node}_I \) for all \( I \subseteq [n] \), starting from \( \text{table}_\emptyset \) via dynamic programming. \( \text{node}_I \) is the set of all triples of (the pointers to) nodes, \( (u, u_0, u_1) \in N \times (N \sqcup T) \times (N \sqcup T) \), in \( \mathcal{B}(f, \pi_I) \), where \( \text{var}(u) = \pi_I[I] \), and \( (u, u_0) \) and \( (u, u_1) \) are the 0-edge and 1-edge of \( u \), respectively. Thus, \( \text{node}_I \) contains the structure of the subgraph of \( \mathcal{B}(f, \pi_I) \) induced by \( V[I] \). The purpose of the \( \text{node}_I \) is to prevent the algorithm from duplicating existing nodes, i.e., creating nodes associated with the same subfunctions as those with which the existing nodes are associated. By the definition, \( \text{node}_\emptyset \) is the empty set. We assume that \( \text{node}_I \) is implemented with an appropriate data structure, such as a balanced tree, so that the time complexity required for membership testing and insertion is the order of logarithm in the number of triples stored in \( \text{node}_I \). An example of \( \text{table}_I \) and \( \text{node}_I \) is shown in Figure 4 in Appendix.
More generally, for disjoint subset $I_1, \ldots, I_m \subseteq [n]$, $\text{TABLE}(I_1, \ldots, I_m)$ is an array with $2^{n-|I_1|} \cdots \cdot 2^{n-|I_m|}$ cells such that, for $b \in \{0,1\}^{n-|I_1|} \cdots \cdot 2^{n-|I_m|}$, $\text{TABLE}(I_1, \ldots, I_m)[b]$ stores the nodes of $B(f, \pi_{(I_1, \ldots, I_m)})$ associated with the function $f|_{x[I_1,\ldots,I_m]=b}$. NODE$(I_1, \ldots, I_m)$ is defined similarly for $B(f, \pi_{(I_1, \ldots, I_m)})$. For simplicity, we hereafter denote by $\mathcal{FS}((I_1, \ldots, I_m))$ the quadruplet $(\pi_{(I_1, \ldots, I_m)}, \text{MINCOST}(I_1, \ldots, I_m), \text{TABLE}(I_1, \ldots, I_m), \text{NODE}(I_1, \ldots, I_m))$.

2.3.2 Sketch of Algorithm $\mathcal{FS}$

Algorithm $\mathcal{FS}$ performs the following operations for $k = 1, \ldots, n$ in this order. For each $k$-element subset $I \subseteq [n]$, compute $\mathcal{FS}((I \setminus i, i))$ from $\mathcal{FS}((I \setminus i))$ for each $i \in I$ in the manner described later (note that, since the cardinality of the set $I \setminus i$ is $k-1$, $\mathcal{FS}((I \setminus i))$ has already been computed). Then set $\mathcal{FS}(I) \leftarrow \mathcal{FS}((I \setminus i^*, i^*))$, where $i^*$ is the index $i \in I$ that minimizes $\text{MINCOST}(I \setminus i, i)$, implying that $\pi_I = \pi_{(i^*, i^*)}$. This is justified by Lemma 4. A schematic view of the algorithm is shown in Figure 5 in Appendix.

To compute $\mathcal{FS}((I \setminus i, i))$ from $\mathcal{FS}((I \setminus i))$, do the following. First set NODE$(I \setminus i, i) \leftarrow \emptyset$ and $\text{MINCOST}(I \setminus i, i) \leftarrow \text{MINCOST}_{(I \setminus i)}$ as their initial values. Then, for each $b \in \{0,1\}^{n-|I|}$, set

$$u_0 \leftarrow \text{TABLE}_{I \setminus i}[x[I_1 \setminus i]=b, x_i = 0], \quad u_1 \leftarrow \text{TABLE}_{I \setminus i}[x[I_1 \setminus i]=b, x_i = 1].$$

If $u_0 = u_1$, then store $u_0$ in $\text{TABLE}_{(I \setminus i,i)}[b]$. Otherwise, test whether $(u, u_0, u_1)$ for some $u$ is a member of $\text{NODE}_{(I \setminus i,i)}$. If it is, store $u$ in the $\text{TABLE}_{(I \setminus i,i)}[b]$; otherwise create a new triple $(u', u_0, u_1)$, insert it into $\text{NODE}_{(I \setminus i,i)}$ and increment $\text{MINCOST}_{(I \setminus i,i)}$. Since $u'$ is the pointer to the new node, $u'$ must be different from any pointer already included in $\text{NODE}_{(I \setminus i,i)}$ and from any pointer to a node in $V_1 \cup \cdots \cup V_{k-1}$ in $B(f, \pi_{(I \setminus i)})$, where $k = |I|$. Such $u'$ can be easily chosen by setting $u'$ to two plus the value of $\text{MINCOST}_{(I \setminus i,i)}$ before the increment, since the $\text{MINCOST}_{(I \setminus i,i)}$ is exactly the number of triples in $\text{NODE}_{(I \setminus i,i)}$ plus $|V_1 \cup \cdots \cup V_{k-1}|$, and the numbers $0$ and $1$ are reserved for the terminal nodes. We call the above procedure table folding with respect to $x_i$, because it halves the size of $\text{TABLE}_{(I \setminus i)}$. We also mean it by “folding $\text{TABLE}_{(I \setminus i)}$ with respect to $x_i$.”

The complexity analysis is fairly simple. For each $k$, we need to compute $\mathcal{FS}(I)$ for $(\begin{smallmatrix} n \\ k \end{smallmatrix})$ possible $I$’s with $|I| = k$. For each $I$, it takes $O^*(2^{n-k})$ time since the the size of $\text{TABLE}_{I \setminus i}$ is $2^{n-k+1}$ and each operation to $\text{NODE}_{I \setminus i}$ takes a polynomial time in $n$. Thus, the total time is $\sum_{k=0}^{n} 2^{n-k+1} \binom{n}{k} = 2 \cdot 3^n$ up to a polynomial factor. The point is that computing each $\mathcal{FS}(I)$ takes time linear to the size of $\text{TABLE}_{I \setminus i}$ up to a polynomial factor. The space required by Algorithm $\mathcal{FS}$ during the process for $k$ is dominated by that for $\text{TABLE}_{I \setminus i}$, $\text{TABLE}_{I \setminus i}$ and $\text{NODE}_{I}$ for all $I$ and $i \in I$, which is $O^*(2^{n-k} \binom{n}{k})$. The space complexity is thus $O^*(\max_{k \in [0,1]} 2^{n-k} \binom{n}{k}) = O^*(3^n)$.

Theorem 5 (Friedman and Supowit [5]). Suppose that the truth table of $f$: $\{0,1\}^{n} \rightarrow \{0,1\}$ is given as input. Algorithm $\mathcal{FS}$ produces $\mathcal{FS}([n])$ in $O^*(3^n)$ time and space.

2.4 Quantum Computation

We assume that readers have a basic knowledge of quantum computing (e.g., Ref. [10]). We provide only a lemma used to obtain our results.

Lemma 6 (Quantum Minimum Finding [4, 3]). For every $\varepsilon > 0$ there exists a quantum algorithm that, for a function $f$: $[N] \rightarrow \mathbb{Z}$ given as an oracle, finds an element $x \in [N]$ at which $f(x)$ achieves the minimum, with error probability at most $\varepsilon$ by making $O(\sqrt{N \log(1/\varepsilon)})$ queries.
In this paper, the search space \( N \) is exponentially large in \( n \) and we are interested in exponential complexities, ignoring polynomial factors in them. We can thus safely assume \( \varepsilon = 1/2^p(n) \) for a polynomial \( p(n) \), so that the overhead is polynomial bounded. Since our algorithms use Lemma 6 a constant number of times, their overall error probabilities are exponentially small for a sufficiently large \( p(n) \). In the following proofs, we thus assume that \( \varepsilon \) is exponentially small whenever we use Lemma 6, and do not explicitly analyze the error probability for simplicity.

Our algorithms assume the quantum random access memory (QRAM) model [6], which is commonly used in the literature when considering quantum algorithms. In the model, one can read contents from or write them into quantum memory in a superposition.

### 3 Quantum Algorithm with Divide-and-Conquer

We generalize Lemma 4 and Theorem 5 and use them in our quantum algorithm.

**Lemma 7.** For any disjoint subsets \( I_1, \ldots, I_m, J \subseteq [n] \) with \( J \neq \emptyset \) and any Boolean function \( f : \{0, 1\}^n \to \{0, 1\} \), the following holds:

\[
\text{MINCOST}(I_1, \ldots, I_m, J) = \min_{k \in J} \left( \text{MINCOST}(I_1, \ldots, I_m, J \setminus \{k\}) + \text{Cost}_k(f, \pi(I_1, \ldots, I_m, J \setminus \{k\})) \right)
\]

\[
= \min_{k \in J} \left( \text{MINCOST}(I_1, \ldots, I_m, J \setminus \{k\}) \right).
\]

The proof of this lemma is very similar to that of Lemma 4 and given in the full paper [11]. Based on Lemma 7, we generalize Theorem 5 to obtain algorithm \( \text{FS}^* \) (its pseudo code is shown in Figure 6 in Appendix).

**Lemma 8 (Classical Composition Lemma).** For disjoint subsets \( I_1, \ldots, I_m, J \subseteq [n] \) with \( J \neq \emptyset \), there exists a deterministic algorithm \( \text{FS}^* \) that produces \( \text{FS}^*([I_1, \ldots, I_m, J]) \) from \( \text{FS}^*([I_1, \ldots, I_m]) \) for an underlying function \( f : \{0, 1\}^n \to \{0, 1\} \) in \( O^* \left( 2^{n-|I_1 \cup \cdots \cup I_m \cup J|} \cdot 3^{|J|} \right) \) time and space. More generally, for each \( k \in [|J|] \), the algorithm produces the set \( \{\text{FS}^*([I_1, \ldots, I_m, K]) : K \subseteq J, |K| = k \} \) from \( \text{FS}^*([I_1, \ldots, I_m]) \) in \( O^* \left( 2^{n-|I_1 \cup \cdots \cup I_m \cup J|} \sum_{j=0}^{k} 2^{|J|-j} \binom{|J|}{j} \right) \) time and space.

Note that if \( I_1 \cup \cdots \cup I_m = \emptyset \) and \( J = [n] \), then we obtain Theorem 5.

**Proof.** We focus on the simplest case of \( m = 1 \), for which our goal is to show an algorithm that produces \( \text{FS}^*([I, J]) \) from \( \text{FS}^*([I]) \). It is straightforward to generalize the proof to the case of \( m \geq 2 \). Starting from \( \text{FS}^*([I]) \), the algorithm first folds \( \text{TABLE}_f \) with respect to each variable in \( \{x_j : j \in J\} \) to obtain \( \text{FS}^*([I, j]) \) for every \( j \in J \), then fold \( \text{TABLE}_{(I, j_1)} \) with respect to \( x_{j_1} \) and \( \text{TABLE}_{(I, j_2)} \) with respect to \( x_{j_2} \) to obtain \( \text{FS}^*([I, \{j_1, j_2\}]) \) by taking the minimum of \( \text{MINCOST}([I, j_1, j_2]) \) and \( \text{MINCOST}([I, j_2, j_1]) \) for every \( j_1, j_2 \in J \), and repeat this to finally obtain \( \text{FS}^*([I, J]) \). This algorithms is justified by Lemma 7. For each \( j \in [|J|] \), \( K \subseteq J \) with \( |K| = j \), and \( h \in K \), the time complexity of computing \( \text{FS}^*([I, K - h]) \) is linear to the size of \( \text{TABLE}_{(I, K)} \), i.e., \( 2^{n-|I| - j} \) up to a polynomial factor. The total time is thus, up to a polynomial factor,

\[
\sum_{j=1}^{|J|} 2^{n-|I|-j} \binom{|J|}{j} < 2^{n-|I|-|J|} \sum_{j=0}^{|J|} 2^{j-|I|-j} \binom{|J|}{j} = 2^{n-|I|+J}, 3^{|J|}.
\]

If we stop the algorithm at \( j = k \), then the algorithm produces the set \( \{\text{FS}^*([I, K]) : K \subseteq J, |K| = k \} \). The time complexity in this case is at most \( 2^{n-|I|-|J|} \sum_{j=0}^k 2^{j-|I|-j} \binom{|J|}{j} \), up to a polynomial factor.
Since the space complexity is trivially upper-bounded by the time complexity, we complete the proof.

\textbf{Remark 9.} It is not difficult to see that the algorithm $FS^*$ works even when the function $f$ has a multivalued function: $f : \{0,1\}^n \rightarrow \mathbb{Z}$. The only difference from the Boolean case is that the truth table maps each Boolean assignment to a value in $\mathbb{Z}$. In this case, the algorithm produces a variant of an OBDD (called a multi-terminal BDD, MTBDD) of minimum size. In addition, our algorithm with slight modifications to the table folding rule in $FS^*$ can construct a minimum zero-suppressed BDD (ZDD) [9] for a given Boolean function. The details are described in the full paper [11]. These modifications are also possible for the quantum algorithms described later, since they perform table folding by running $FS^*$ as a subroutine.

\textbf{Algorithm $FS^*$} Composable variant of algorithm $FS$. 

\textbf{Input:} disjoint subsets $I, J \in \mathbb{n}$ and $FS(I)\
\textbf{Output:} $FS((I, J))\
\begin{algorithm}
\begin{algorithmic}[1]
\Function{Main}{$n$}
\For {$\ell := 1$ \textbf{to} $|J|$}
\For {each $\ell$-element subset $K \subseteq J$}
\State $\text{MINCOST}(I, K) \leftarrow +\infty$; \hspace{1cm} // init.
\For {each $k \in K$}
\State $FS((I, K \setminus k, k)) \leftarrow FOLD(I, K, k, FS((I, K \setminus k)));$
\EndFor
\If {$\text{MINCOST}(I, K) > \text{MINCOST}(I, K \setminus k)$}
\State $FS((I, K)) \leftarrow FS((I, K \setminus k));$
\EndIf
\EndFor
\EndFor
\State $\text{return} FS((I, J))$
\EndFunction

\Function{FOLD}{$I, K, k, FS((I, K \setminus k))$} // produce $FS((I, K \setminus k))$ from $FS((I, K \setminus k))$
\State $\pi(I, K \setminus k) \in \{i \in \Pi(I, K \setminus k) : \pi[i] = \pi(I, K \setminus k)[i] \hspace{0.5cm} (i = 1, \ldots, |I \cup K| - 1)\}$; \hspace{1cm} // init.
\State $\text{MINCOST}(I, K \setminus k) \leftarrow \text{MINCOST}(I, K \setminus k);$ \hspace{1cm} // init.
\State $\text{NODE}(I, K \setminus k) \leftarrow \emptyset;$ \hspace{1cm} // init.
\For {$b \in \{0,1\}^{n-|I|}$}
\State $u_0 \leftarrow \text{TABLE}(I, K \setminus k)[x_n\setminus(I\cup K)] = b, x_k = 0;$
\State $u_1 \leftarrow \text{TABLE}(I, K \setminus k)[x_n\setminus(I\cup K)] = b, x_k = 1;$
\If {$u_0 \neq u_1$}
\State $\text{TABLE}(I, K \setminus k)[x_n\setminus(I\cup K)] = b \leftarrow u_0$
\ElseIf {$\exists u, u_0, u_1 \in \text{NODE}(I, K \setminus k)$}
\State $\text{TABLE}(I, K \setminus k)[x_n\setminus(I\cup K)] = b \leftarrow u$
\Else \hspace{1cm} // create a new node
\State $u \leftarrow \text{MINCOST}(I, K \setminus k) + 2;$
\State $\text{TABLE}(I, K \setminus k)[x_n\setminus(I\cup K)] = b \leftarrow u;$
\State $\text{MINCOST}(I, K \setminus k) \leftarrow \text{MINCOST}(I, K \setminus k) + 1;$
\State insert $(u, u_0, u_1)$ into $\text{NODE}(I, K \setminus k)$
\EndIf
\EndIf
\EndFor
\State $\text{return} FS((I, K \setminus k))$
\EndFunction
\end{algorithm}
\end{algorithm}

\textbf{SWAT 2020}
The following theorem is the basis of our quantum algorithms.

**Lemma 10 (Divide-and-Conquer).** For any disjoint subsets $I_1, \ldots, I_m, J \subseteq [n]$ with $J \neq \emptyset$ and any $k \in [\|J\|]$, it holds that $\text{Mincost}(I_1, \ldots, I_m, J)[J]$ is equal to

$$\min_{K: K \subseteq J, |K| = k} \left( \text{Mincost}(I_1, \ldots, I_m, K)[K] + \text{Mincost}(I_1, \ldots, I_m, K \setminus K)[J \setminus K] \right).$$

(3)

In particular, when $I_1 \cup \cdots \cup I_m = \emptyset$ and $J = [n]$, it holds that

$$\text{Mincost}[n] = \min_{K \subseteq [n], |K| = k} \left( \text{Mincost}_K + \text{Mincost}_{(K, [n] \setminus K)}[[n] \setminus K] \right).$$

(4)

**Proof.** We first prove the special case of $I_1 \cup \cdots \cup I_m = \emptyset$ and $J = [n]$. By the definition, we have

$$\text{Mincost}[n] = \sum_{j=1}^{n} \text{Cost}_{\pi[j]}(f, \pi) = \sum_{j=1}^{k} \text{Cost}_{\pi[j]}(f, \pi) + \sum_{j=k+1}^{n} \text{Cost}_{\pi[j]}(f, \pi)$$

for the optimal permutation $\pi = \pi[n]$. Let $K = \{\pi[1], \ldots, \pi[k]\}$. By Lemma 3, the first sum is independent of how $\pi$ maps $\{k+1, \ldots, n\}$ to $[n] \setminus K$. Thus, it is equal to the minimum of $\sum_{j=1}^{k} \text{Cost}_{\pi[j]}(f, \pi)$ over all $\pi_1 \in \Pi(K)$, i.e., $\text{Mincost}_K$. Similarly, the second sum is independent of how $\pi$ maps $\{k\}$ to $K$. Thus, it is equal to the minimum of $\sum_{j=k+1}^{n} \text{Cost}_{\pi[j]}(f, \pi)$ over all $\pi_2 \in \Pi((K, [n] \setminus K))$, i.e., $\text{Mincost}_{(K, [n] \setminus K)}[[n] \setminus K]$. This completes the proof of Eq. (4).

We can generalize this in a straightforward manner. Let $\pi = \pi(I_1, \ldots, I_m, J)$ and $\ell = |I_1 \cup \cdots \cup I_m|$. Then, we have

$$\text{Mincost}(I_1, \ldots, I_m, J)[J] = \sum_{j=1}^{k} \text{Cost}_{\pi[j]}(f, \pi) + \sum_{j=k+1}^{|I|} \text{Cost}_{\pi[j]}(f, \pi).$$

By defining $K := \{\pi[\ell+1], \ldots, \pi[\ell+k]\}$, the same argument as the special case of $\ell = 0$ implies that the first and second sums are $\text{Mincost}(I_1, \ldots, I_m, K)[K]$ and $\text{Mincost}(I_1, \ldots, I_m, K \setminus K)[J \setminus K]$, respectively. This completes the proof of Eq. (3). ▶

A schematic view of the above lemma is shown in Figure 7 in Appendix.

### 3.1 Simple Cases

We provide simple quantum algorithms on the basis of Lemma 10. The lemma states that, for any $k \in [n]$, $\text{Mincost}[n]$ is the minimum of $\text{Mincost}_K + \text{Mincost}_{(K, [n] \setminus K)}[[n] \setminus K]$ over all $K \subseteq [n]$ with $|K| = k$. To find $K$ from among $\binom{n}{k}$ possibilities that minimizes this amount, we use the quantum minimum finding (Lemma 6). To compute $\text{Mincost}_K + \text{Mincost}_{(K, [n] \setminus K)}[[n] \setminus K] = \text{Mincost}_{(K, [n] \setminus K)}$, it suffices to first compute $\mathcal{F}\mathcal{S}(K)$ (including $\text{Mincost}_K$), and then $\mathcal{F}\mathcal{S}((K, [n] \setminus K))$ (including $\text{Mincost}_{(K, [n] \setminus K)}$) from $\mathcal{F}\mathcal{S}(K)$. The time complexity for computing $\mathcal{F}\mathcal{S}(K)$ from $\mathcal{F}\mathcal{S}(\emptyset)$ is $O^*(2^n - 3^k)$ by Lemma 8 with $I_1 \cup \cdots \cup I_m = \emptyset$ and $J = K$, while that for computing $\mathcal{F}\mathcal{S}((K, [n] \setminus K))$ from $\mathcal{F}\mathcal{S}(K)$ is $O^*(3^k)$ by Lemma 8 with $m = 1$, $I_1 = K$, and $J = [n] \setminus K$. Thus, the time complexity for computing $\mathcal{F}\mathcal{S}((K, [n] \setminus K))$ from $\mathcal{F}\mathcal{S}(\emptyset)$ is $O^*(2^n - 3^k + 3^k)$. Thus, for $k = \alpha n$ with $\alpha \in [0, 1]$ fixed later, the total time complexity up to a polynomial factor is

$$T(n) = \sqrt{\binom{n}{\alpha n}} \left( 2^{(1-\alpha)n}3^{\alpha n} + 3^{(1-\alpha)n} \right) \leq 2^{\frac{1}{2}H(\alpha)n} \left\{ 2^{(1-\alpha)n+\alpha \log_2 3}n + 2^{(1-\alpha)n \log_2 3}n \right\}.$$
To balance the both terms, we set $1 - \alpha + \alpha \log_2 3 = (1 - \alpha) \log_2 3$ and obtain $\alpha = \alpha^*$, where $\alpha^* = \frac{\log_2 3 - 1}{2\log_2 3 - 1} \approx 0.269577$. We have

$$\min_{\alpha \in [0, 1]} T(n) = O \left( 2^{\frac{1}{2} H(\alpha^*) n + (1 - \alpha^*) n + \alpha^* (\log_2 3) n} \right) = O(\gamma_0),$$

where $\gamma_0 = 2.98581 \ldots$ This slightly improves the classical best bound $O^*(3^n)$ on the time complexity. To improve the bound further, we introduce a preprocess that classically computes $FS(K)$ for every $K$ with $|K| = \alpha n$ ($\alpha \in (0, 1)$) by using Algorithm $FS^*$. By Lemma 8, the preprocessing time is then

$$\sum_{j=1}^{\alpha n} 2^{n-j} \cdot \begin{pmatrix} n \atop j \end{pmatrix} \leq \alpha n \cdot \max_{j \in [\alpha n]} 2^{n-j} \begin{pmatrix} n \atop j \end{pmatrix} \ll 2^{(1-\alpha)n + H(\alpha)n} \quad (\alpha < 1/3),$$

$$2^{\frac{1}{2} n + H(1/3)n} \quad (\alpha \geq 1/3),$$

since $2^{n-j} \begin{pmatrix} n \atop j \end{pmatrix}$ increases when $j < n/3$ and decreases otherwise. Note that once this preprocess is completed, we can use $FS(K)$ for free and assume that the cost for accessing $FS(K)$ is polynomially bounded for all $K \subseteq [n]$ with $|K| = \alpha n$.

Then, assuming that $\alpha < 1/3$, the total time complexity up to a polynomial factor is

$$T(n) = \sum_{j=1}^{\alpha n} 2^{n-j} \cdot \begin{pmatrix} n \atop j \end{pmatrix} + \sqrt{\frac{n}{\alpha n}} \left( n^{O(1)} + 3^{(1-\alpha)n} \right) \ll 2^{(1-\alpha)n + H(\alpha)n} + 2^{\frac{1}{2} n + H(1/3)n + (1-\alpha) \log_2 3 n}.$$

To balance the both terms, we set $(1 - \alpha) + H(\alpha) = \frac{1}{2} H(\alpha) = (1 - \alpha) \log_2 3$ and obtain the solution $\alpha = \alpha^*$, where $\alpha^* := 0.274863 \ldots$, which is less than $1/3$ as we assumed. At $\alpha = \alpha^*$, we have $T(n) \ll 2^{(1-\alpha)n + H(\alpha^*)n} = O^*(\gamma_1^n)$, where $\gamma_1$ is at most $2.97625$ ($\gamma_0$).

Thus, introducing the preprocess improves the complexity bound. A schematic view of the above algorithm is shown in Figure 8 in Appendix.

### 3.2 General Case

We can improve this bound further by applying Lemma 10 $k$ times. We denote the resulting algorithm with constant parameters $k \in \mathbb{N}$ and $\alpha := (\alpha_1, \ldots, \alpha_k)$ by $OptOBDD(k, \alpha)$ where $0 < \alpha_1 < \cdots < \alpha_k < 1$. Its pseudo code is given below. In addition, we assume $\alpha_1 < 1/3$ and $\alpha_{k+1} = 1$ in the following complexity analysis.

To simplify notations, define two function as follows: for $x, y \in (0, 1)$ such that $x < y$, $f(x, y) := \frac{1}{2} y \cdot H(x/y) + g(x, y)$ and $g(x, y) := (1 - y) + (y - x) \log_2 3$.

By Lemma 8, the time required for the preprocess is $\sum_{\ell=1}^{\alpha_1 n} 2^{n-\ell} \cdot \begin{pmatrix} n \atop \ell \end{pmatrix}$ up to a polynomial factor. Thus, the total time complexity can be described as the following recurrence:

$$T(n) = \sum_{\ell=1}^{\alpha_1 n} 2^{n-\ell} \cdot \begin{pmatrix} n \atop \ell \end{pmatrix} + L_{k+1}(n),$$

$$L_{j+1}(n) = \sqrt{\frac{\alpha_{j+1} n}{\alpha_j n}} \left( L_j(n) + 2^{(1-\alpha_{j+1})n} 3^{(\alpha_{j+1} - \alpha_j)n} \right)$$

$$= \sqrt{\frac{\alpha_{j+1} n}{\alpha_j n}} \left( L_j(n) + 2^{g(\alpha_j, \alpha_{j+1})n} \right),$$

---

More precisely, $\alpha^*$ must be rounded so that $\alpha^* n$ is an integer. We assume hereafter for simplicity of analysis that $n$ is sufficiently large so that the rounding error is negligible compared to the approximation error in the optimum parameter values, such as $\alpha^*$. 
Algorithm OptOBDD\((k, \alpha)\) Quantum OBDD-minimization algorithm with constant parameters \(k \in \mathbb{N}\) and \(\alpha := (\alpha_1, \ldots, \alpha_k) \in [0, 1]^k\) satisfying \(0 < \alpha_1 < \cdots < \alpha_k < 1\), where the quantum minimum finding algorithm is used in line 8, and \(F^*\) is used in lines 2 and 15. “\(A \leftarrow B\)" means that \(B\) is substituted for \(A\).

Input: \(\mathcal{F}\mathcal{S}(\emptyset) := \{\text{TABLE}_\emptyset, \pi_\emptyset, \text{MINCOST}_\emptyset, \text{NODE}_\emptyset\}\) (accessible from all Functions)

Output: \(\mathcal{F}\mathcal{S}(n)\)

Function Main()

1. compute the set \(\mathcal{F}\mathcal{S}(K) : K \subseteq [n], |K| = |\alpha|n\) by algorithm FS (or \(F^*\));
2. make the set of these \(\mathcal{F}\mathcal{S}(K)\) global (i.e., accessible from all Functions);
3. return DivideAndConquer\([n], k + 1\)

Function DivideAndConquer\((L, t)\) // Compute \(\mathcal{F}\mathcal{S}(L)\) with \(\alpha_1, \ldots, \alpha_t = \lfloor L/n \rfloor\)

4. if \(t = 1\) then return \(\mathcal{F}\mathcal{S}(L)\); // \(\mathcal{F}\mathcal{S}(L)\) has been precomputed.
5. Find \(K(\subset L)\) of cardinality \(|\alpha_1|n\), with Lemma 6, that minimizes MINCOST\((K, L)\),
6. which is computed as a component of \(\mathcal{F}\mathcal{S}(K, L \setminus K)\) by calling ComputeFS\((K, L \setminus K, t)\);
7. let \(K^*\) be the set that achieves the minimum;
8. return \(\mathcal{F}\mathcal{S}(K^* \setminus L, \mathcal{F}\mathcal{S}(K))\)

Function ComputeFS\((K, M, t)\) // Compute \(\mathcal{F}\mathcal{S}(K\cup M)\) with \(\alpha_1, \ldots, \alpha_t\)

9. \(\mathcal{F}\mathcal{S}(K) \leftarrow \) DivideAndConquer\((K, t - 1)\);
10. \(\mathcal{F}\mathcal{S}(K, M) \leftarrow \mathcal{F}\mathcal{S}(K, M, \mathcal{F}\mathcal{S}(K))\);
11. return \(\mathcal{F}\mathcal{S}(K)\)

Algorithm OptOBDD\((k, \alpha)\) Quantum OBDD-minimization algorithm with constant parameters \(k \in \mathbb{N}\) and \(\alpha := (\alpha_1, \ldots, \alpha_k) \in [0, 1]^k\) satisfying \(0 < \alpha_1 < \cdots < \alpha_k < 1\), where the quantum minimum finding algorithm is used in line 8, and \(F^*\) is used in lines 2 and 15. “\(A \leftarrow B\)" means that \(B\) is substituted for \(A\).

where \(j \in [k]\) and \(L_j(n) = O^*(1)\). Intuitively, \(L_j(n)\) is the time required for producing \(\mathcal{F}\mathcal{S}(K_1 \setminus K_2 \setminus \cdots \setminus K_1 \setminus K_j \setminus K_j - 1)\) such that MINCOST\((K_1, K_2 \setminus K_3, \ldots, K_j \setminus K_j - 1)\) is minimum over all \(K_1, \ldots, K_j - 1\) satisfying \(|K_\ell| = \alpha_\ell n\) for every \(\ell \in [k + 1]\) and \(K_\ell \subseteq K_\ell + 1\) for every \(\ell \in [k]\).

Since \(L_1(n) = O^*(1)\), we have \(L_2(n) \leq \sqrt{(\alpha_2 n)} \cdot 2g(\alpha_1, \alpha_2)n \leq 2f(\alpha_1, \alpha_2)n\). By setting \(f(\alpha_1, \alpha_2) = g(\alpha_2, \alpha_3)\), we have \(L_3(n) = \sqrt{(\alpha_3 n)} \cdot (L_2(n) + 2g(\alpha_2, \alpha_3)n) \leq \sqrt{(\alpha_3 n)} \cdot 2g(\alpha_2, \alpha_3)n \leq 2f(\alpha_2, \alpha_3)n\). In general, for \(j = 2, \ldots, k\), setting \(f(\alpha_j, \alpha_1) = \alpha_jn\) yields

where \(j \in [k]\) and \(L_j(n) = O^*(1)\). Intuitively, \(L_j(n)\) is the time required for producing \(\mathcal{F}\mathcal{S}(K_1 \setminus K_2 \setminus \cdots \setminus K_1 \setminus K_j \setminus K_j - 1)\) such that MINCOST\((K_1, K_2 \setminus K_3, \ldots, K_j \setminus K_j - 1)\) is minimum over all \(K_1, \ldots, K_j - 1\) satisfying \(|K_\ell| = \alpha_\ell n\) for every \(\ell \in [k + 1]\) and \(K_\ell \subseteq K_\ell + 1\) for every \(\ell \in [k]\).

Since \(L_1(n) = O^*(1)\), we have \(L_2(n) \leq \sqrt{(\alpha_2 n)} \cdot 2g(\alpha_1, \alpha_2)n \leq 2f(\alpha_1, \alpha_2)n\). By setting \(f(\alpha_1, \alpha_2) = g(\alpha_2, \alpha_3)\), we have \(L_3(n) = \sqrt{(\alpha_3 n)} \cdot (L_2(n) + 2g(\alpha_2, \alpha_3)n) \leq \sqrt{(\alpha_3 n)} \cdot 2g(\alpha_2, \alpha_3)n \leq 2f(\alpha_2, \alpha_3)n\). In general, for \(j = 2, \ldots, k\), setting \(f(\alpha_j, \alpha_1) = \alpha_jn\) yields

Therefore, the total complexity [Eq. (6)] is

where we use \(\alpha_1 < 1/3\), \(\alpha_{k+1} = 1\), and Eq. (5). To optimize the right-hand side, we set parameters so that \(1 - \alpha_1 + \mathbf{H}(\alpha_1) = f(\alpha_k, 1)\).

In summary, we need to find the values of parameters \(\alpha_1, \ldots, \alpha_k\) that satisfy the following system of equations and \(\alpha_1 < 1/3:\)

By numerically solving this system of equations, we obtain \(T(n) = O(\gamma_k^2)\), where \(\gamma_k\) is at most 2.83728 for \(k = 6\). The value of \(\gamma_k\) becomes smaller as \(k\) increases. However, incrementing \(k\) beyond 6 provides only negligible improvement of \(\gamma_k\). Since the space complexity is trivially upper-bounded by the time complexity, we have the following theorem. Note that the values
of $\alpha_i$’s are not symmetric with respect to $1/2$. This reflects the fact that optimizing cost is not symmetric with respect to $1/2$, contrasting with many other combinatorial problems.

- **Theorem 11.** There exists a quantum algorithm that, for the truth table of $f : \{0, 1\}^n \rightarrow \{0, 1\}$ given as input, produces $FS([n])$ with probability $1 - \exp(-\Omega(n))$ in $O^*(\gamma^n)$ time and space, where the constant $\gamma$ is at most 2.83728, which is achieved by $OptOBDD(k, \alpha)$ with $k = 6$ and $\alpha = (0.183791, 0.183802, 0.183974, 0.186131, 0.206480, 0.343573)$.

## 4 Quantum Algorithm with Composition

### 4.1 Quantum Algorithm Composition Lemma

By generalizing the quantum algorithm given in Theorem 11, we now provide a quantum version of Lemma 8, called the quantum composition lemma.

- **Lemma 12 (Quantum Composition: Base Part).** For any disjoint subsets $I_1, \ldots, I_m, J \subseteq [n]$ with $J \neq \emptyset$, there exists a quantum algorithm that, with probability $1 - \exp(-\Omega(n))$, produces $FS((I_1, \ldots, I_m, J))$ from $FS((I_1, \ldots, I_m))$ for an underlying function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ in $O^*(2^{n - |I_1 \cup \cdots \cup I_m|} \gamma^{2^m})$ time and space, where $\gamma$ is the constant defined in Theorem 11.

The proof is given in the full paper [11]. The proof idea is similar to that used in the proof of Lemma 8. A pseudo code of the algorithm provided in Lemma 12 is shown below as $OptOBDD^r(k, \alpha)$, where the subroutine $\Gamma$ appearing in line 17 is set to the deterministic algorithm $FS^*$, and $k$ and $\alpha$ are set to the values specified in Theorem 11.

- **Algorithm $OptOBDD^r(k, \alpha)$** Composable Quantum OBDD-minimization algorithm with subroutine $\Gamma$ and constant parameters $k \in \mathbb{N}$ and $\alpha = (\alpha_1, \ldots, \alpha_k) \in [0, 1]^k$ satisfying $0 < \alpha_1 < \cdots < \alpha_k < 1$, where the quantum minimum finding algorithm is used in line 9, and subroutine $\Gamma$ is used in line 17. “$A \leftarrow B$” means that $B$ is substituted for $A$. $\Gamma(I_1, I_2, J, FS(I_1, I_2))$ produces $FS((I_1, I_2, J))$ from $FS((I_1, I_2))$.

**Input:** $I \subseteq [n], J \subseteq [n], FS(I), (\text{accessible from all Functions})$

**Output:** $FS((I, J))$

```
1 Function Main()
2     $n' \leftarrow |J|; \quad // \text{init.}$
3     compute the set $\{FS((I, K)) : K \subseteq J, |K| = \lfloor \alpha_1 n' \rfloor\}$ by algorithm $FS^*$;
4     make $n'$ and the above set of $FS((I, K))$ global (i.e., accessible from all Functions);
5     return $\text{DivideAndConquer}(J, k + 1)$
6 end

7 Function DivideAndConquer(L, t) \quad // Compute $FS((I, L))$ with $\alpha_1, \ldots, \alpha_t$
8      if $t = 1$ then return $FS(I, L)$; \quad // $FS(I, L)$ has been precomputed.
9      Find $K(\subseteq L)$ of cardinality $|\alpha_{t-1} n'|$, with Lemma 6, that minimizes $\text{MINCOST}_{(I, K, L \setminus K)}$
10     which is computed as a component of $FS((I, K, L \setminus K))$
11     by calling $\text{ComputeFS}(I, K, L \setminus K, t)$;
12     let $K^*$ be the set that achieves the minimum;
13      return $FS((I, K^*, L \setminus K^*))$
14 end

15 Function ComputeFS(I, K, M, t) \quad // Compute $FS((I, K, M))$ with $\alpha_1, \ldots, \alpha_t$
16      $FS(I, K) \leftarrow \text{DivideAndConquer}(K, t - 1)$;
17      $FS(I, K, M) \leftarrow \Gamma(I, K, M, FS(I, K))$;
18      return $FS((I, K, M))$
19 end
```
Lemma 13 (Quantum Composition: Induction Part). Suppose that $\Gamma$ is a quantum algorithm that, for any disjoint subsets $I_1, \ldots, I_m, J \subseteq [n]$ with $J \neq \emptyset$, produces $FS((I_1, \ldots, I_m, J))$ from $FS((I_1, \ldots, I_m))$ with probability $1 - \exp(-\Omega(n))$ in $O^*\left(2^{n-|I_1\cup\cdots\cup I_m\cup J|} \cdot \beta_k^{|J|}\right)$ time and space for an underlying function $f: \{0,1\}^n \to \{0,1\}$. Then, for any constant parameters $k \in \mathbb{N}$ and $\alpha = (\alpha_1, \ldots, \alpha_k) \in [0,1]^k$ with $\alpha_1 < \cdots < \alpha_k$ and for any disjoint subsets $I_1, \ldots, I_m, J \subseteq [n]$ with $J \neq \emptyset$, $OptOBDD^\ast_T(k, \alpha)$ produces $FS((I_1, \ldots, I_m, J))$ from $FS((I_1, \ldots, I_m))$ with probability $1 - \exp(-\Omega(n))$ in $O^*\left(2^{n-|I_1\cup\cdots\cup I_m\cup J|} \cdot \beta_k^{|J|}\right)$ time and space for the function $f$, where $\beta_k^{|J|}$ upper-bounds, up to a polynomial factor, the time complexity required for $OptOBDD^\ast_T(k, \alpha)$ to compute $FS([n])$ from $FS(\emptyset)$, that is, $T(n) = O^*\left(\beta_k^n\right)$ for $T(n)$ that satisfies the following recurrence:

$$T(n) = \sum_{\ell=1}^{\alpha_1 n} 2^{n-\ell} \binom{n}{\ell} + L_{k+1},$$

$$L_{j+1} = \sqrt{\left(\frac{\alpha_{j+1} n}{\alpha_j n}\right)} \left(L_j + 2^{(1-\alpha_{j+1})n} \gamma^{(\alpha_{j+1}-\alpha_j)n}\right) = \sqrt{\left(\frac{\alpha_{j+1} n}{\alpha_j n}\right)} \left(L_j + 2^{g_j(\alpha_j, \alpha_{j+1})}\right),$$

where $j \in [k]$, $L_1 = O^*(1)$ and $g_j(x, y) := (1 - y) + (y - x) \log_2 \gamma$.

Proof. Recall that algorithm $FS^\ast$ is used as a subroutine in $OptOBDD(k, \alpha)$ provided in Theorem 11. Since the input and output of $\Gamma$ assumed in the statement are the same as those of algorithm $FS^\ast$, one can use $\Gamma$ instead of algorithm $FS^\ast$ in $OptOBDD(k, \alpha)$ (compromising on an exponentially small error probability). Let $OptOBDD^\ast_T(k, \alpha)$ be the resulting algorithm. Then, one can see that the time complexity $T(n)$ of $OptOBDD^\ast_T(k, \alpha)$ satisfies the recurrence: Eqs. (10)-(11), which are obtained by just replacing $g(x, y)$ with $g_j(x, y)$ in Eqs. (6)-(7). Suppose that $T(n) = O^*(\beta_k^n)$ follows from the recurrence.

Next, we generalize $OptOBDD^\ast_T(k, \alpha)$ so that it produces $FS((I_1, \ldots, I_m, J))$ from $FS((I_1, \ldots, I_m))$ for any disjoint subsets $I_1, \ldots, I_m, J \subseteq [n]$ with $J \neq \emptyset$. The proof is very similar to that of Lemma 12. The only difference is that the time complexity of $\Gamma$ is $O^*\left(2^{n-|I_1\cup\cdots\cup I_m\cup J|} \cdot \gamma^{|J|}\right)$, instead of $O^*\left(2^{n-|I_1\cup\cdots\cup I_m\cup J|} \cdot g^{|J|}\right)$. Namely, when $m = 1$ and $n' = |J|$, the time complexity of $OptOBDD^\ast_T(k, \alpha)$ satisfies the following recurrence: for each $j \in [n]$,

$$T'(n, n') = 2^{n-|I|} \sum_{\ell=1}^{\alpha_1 n'} 2^{n'-\ell} \binom{n'}{\ell} + L_{k+1}(n, n'),$$

$$L_{j+1}(n, n') = \sqrt{\left(\frac{\alpha_{j+1} n'}{\alpha_j n'}\right)} \left(L'_j(n, n') + 2^{n-|I|} \gamma^{(\alpha_{j+1}-\alpha_j)n'}\right) \quad [j \in [n]],$$

$$L_1(n, n') = O^*(1),$$

from which it follows that $T'(n, n') = 2^{n-|I|} T(n') = O^*\left(2^{n-|I|} \cdot \beta_k^{|J|}\right)$. It is straightforward to generalize to the case of $m \geq 2$.

The total error probability is exponentially small by union bound, if the error probabilities of $\Gamma$ and the quantum minimum finding (Lemma 6) are made sufficiently small.

4.2 The Final Algorithm

Lemmas 12 and 13 naturally lead to the following algorithm. We first define $\Gamma_1$ as $OptOBDD^\ast_T(k^{(0)}, \alpha^{(0)})$ for some $k^{(0)} \in \mathbb{N}$ and $\alpha^{(0)} \in [0,1]^{k^{(0)}}$. Then, we define $\Gamma_2$ as
We therefore obtain the following theorem.

Fix $k^{(i)} = 6$ for every $i$. Note that, in the proof of Lemmas 12 and 13, parameter $\alpha^{(i)} = (\alpha_1^{(i)}, \ldots, \alpha_6^{(i)}) \in [0,1]^6$ is set for each $i$ so that it satisfies the system of equations, a natural generalization of Eqs. (8)-(9),

\begin{align}
1 - \alpha_1^{(i)} + H(\alpha_1^{(i)}) &= f_\gamma(\alpha_0^{(i)}, 1), \\
f_\gamma(\alpha_j^{(i-1)}, \alpha_j^{(i)}) &= g_\gamma(\alpha_j^{(i)}, \alpha_{j+1}^{(i)}) & (j = 2, \ldots, 6),
\end{align}

where $f_\gamma(x,y) := \frac{1}{2}y \cdot H(x/y) + g_\gamma(x,y)$ and $g_\gamma(x,y) := (1 - y) + (y - x) \log_2 \gamma$.

By numerically solving this system of equations for $\gamma = 3$, we have $\beta_0 < 2.83728$ as shown in Theorem 11. Then, numerically solving the system of equations with $\gamma = 2.83728$, we have $\beta_0 < 2.79364$. In this way, we obtain a certain $\gamma$ less than 2.77286 at the tenth composition. We therefore obtain the following theorem.

**Theorem 14.** There exists a quantum algorithm that, for the truth table of $f : \{0,1\}^n \to \{0,1\}$ given as input, produces $\mathcal{FS}([n])$ in $O^*(\gamma^n)$ time and space with probability $1 - \exp(-\Omega(n))$, where the constant $\gamma$ is at most 2.77286.

### References

A Appendix

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure2}
\caption{Schematic expression of $\text{Cost}_j(f, \pi)$.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure3}
\caption{Schematic expression of Lemma 3: For any two permutations $\pi, \pi' \in S_n$ such that $\{\pi[1], \ldots, \pi[|I| - 1]\} = \{\pi'[1], \ldots, \pi'[|I| - 1]\}$ and $\pi[|I|] = \pi'[|I|]$, it holds that the number of nodes labeled with $x_i$ in $B(f, \pi)$ is equal to that of nodes labeled with $x_i$ in $B(f, \pi')$.}
\end{figure}
Figure 4 Examples of data structures used in Algorithm FS: TABLE and NODE with $I = \{1,3,5\}$ for the OBDD (rhs) representing $f(x_1, \ldots, x_6) = x_1x_2 + x_3x_4 + \cdots + x_5x_6$ for the variable ordering $(x_1, x_3, x_5, x_2, x_4, x_6)$. The pointers (integers) to the nodes labeled with $x_1, x_3, x_5$ are each shown at the top-left positions of the nodes.

Figure 5 Schematic view of Friedman-Supowit Algorithm. The algorithm goes from the left to the right. On the vertical line indicated by $k$, there are $\binom{n}{k}$ dots, each of which corresponds to $\mathcal{FS}(I)$ for a subset $I \subseteq [n]$ of size $k$. $\mathcal{FS}(I)$ is computed from $\mathcal{FS}(I \setminus i)$ for all $i \in I$, which are arranged as dots on the line indicated by $k - 1$ and have already been computed.
Lemma

There is a deterministic algorithm \( \text{FS}^* \) that, given the partial OBDD for \( \pi! \) and \( J \subseteq n \setminus K \), produces the partial OBDD for \( \pi! \sqcup \# \) in time \( O^* (2^n \cdot |J| \cdot |J|) \) time/space.

\[ \mathcal{F}(\mathcal{S}(\mathcal{F}(\mathcal{S}(I) \cup J))) \mathcal{S}(\mathcal{F}(I) \cup J) \]

\[ |I \sqcup J| \]

\[ |I| \]

\[ 0 \]

\[ n \]

\[ \mathcal{F}(\mathcal{S}(\emptyset)) \]

\[ \mathcal{F}(\mathcal{S}((I, [n] \setminus I))) \]

\[ \text{Output } \pi_{(I,[n]\setminus I)} \]

**Figure 6** Schematic view of \( \text{FS}^* \). This view corresponds to the case where \( m = 1 \) and \( J \subset [n] \setminus I \) in Lemma 8. The shaded area is the one that \( \text{FS}^* \) sweeps to produce \( \mathcal{F}(\mathcal{S}((I, J))) \).

**Figure 7** Schematic view of Eq. (4) in Lemma 10. Intuitively, the lemma says that it is possible to decompose \( \text{FS}^* \) into the parts each of which consists of the two shaded rectangles that share the dot corresponding to \( \mathcal{F}(\mathcal{S}(I)) \) on the line indicated by \( |I| \) for a subset \( I \subseteq [n] \) of some fixed size. The optimal variable ordering is induced by one of the parts.
Quantumly find the minimum

\[ \mathcal{F} \mathcal{S}(\emptyset) \]

Composed in the classical preprocess (a truncation of FS*)

\[ \mathcal{F} \mathcal{S}(\lfloor n \rfloor) \]

Figure 8 Schematic view of our algorithm in the simplest case (one-parameter case). The dotted area is computed in the classical preprocess, which is realized by truncating the process of FS* as stated in Lemma 8. The shaded area is computed by using FS*. The actual algorithm runs the quantum minimum finding, which calls FS* to coherently compute the shaded area corresponding to every dot on the vertical line indicated by k.