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

Related Version A full version of the paper is available at `https://arxiv.org/abs/1909.12658`.

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 standpoints (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 $f(x_1, \ldots, x_{2n}) = x_1x_2 + x_3x_4 + \cdots + x_{2n-1}x_{2n}$ 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 $f\colon \{0,1\}^n \to \{0,1\}$ 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 γ 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 $f\colon \{0,1\}^n \to \{0,1\}$ 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 γ 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 $\{1,\dots,n\}$, and $\mathcal{S}_n$ 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 $\pi \in \mathcal{S}_n$ such that the first $|I|$ members $\{\pi[1],\dots,\pi[|I|]\}$ constitutes I, i.e.,

$$\Pi_n(I) := \{\pi \in \mathcal{S}_n \colon \{\pi[1],\dots,\pi[|I|]\} = I\} \subseteq \mathcal{S}_n.$$

For simplicity, we omit the subscript n and write $\Pi(I)$. More generally, for any two disjoint subsets $I, J \subseteq [n]$, let

$$\Pi_n(\langle I, J\rangle) := \{\pi \in \mathcal{S}_n \colon \{\pi[1], \dots, \pi[|I|]\} = I, \{\pi[|I|+1], \dots, \pi[|I|+|J|]\} = J\} \subseteq \mathcal{S}_n.$$

For any disjoint subsets $I_1, \dots, I_m \subseteq [n]$ for $m \in [n]$, $\Pi_n(\langle I_1, \dots, I_m\rangle)$ is defined similarly. For simplicity, we may denote $\langle I\rangle$ 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 $x_1, \dots, x_n$, any set $I \subseteq [n]$, and any vector $b = (b_1, \dots, b_{|I|}) \in \{0,1\}^{|I|}$, x_I denotes the ordered set $(x_{j_1}, \dots, x_{j_{|I|}})$, where $\{j_1, \dots, j_{|I|}\} = I$ and $j_1 < \cdots < j_{|I|}$, and $x_I = b$ denotes $x_{j_i} = b_i$ for each $i = [|I|]$. For any Boolean function $f \colon \{0,1\}^n \to \{0,1\}$ with variables $x_1, \dots, x_n$, 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 $X \lessapprox Y$.

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 $\binom{n}{k} \le 2^{n\,\mathbf{H}(k/n)}$, where $\mathbf{H}(\cdot)$ represents the binary entropy function $\mathbf{H}(\delta) := -\delta \log_2 \delta - (1-\delta)\log_2(1-\delta)$.

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 $f \colon \{0,1\}^n \to \{0,1\}$ over variables $x_1, \dots, x_n$ and any permutation $\pi \in \mathcal{S}_n$ (called a *variable ordering*), an OBDD $\mathcal{B}(f, \pi)$ 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: $T = \{\mathsf{f}, \mathsf{t}\}$. The set N contains a unique source node r, called the *root*.
2. $\mathcal{B}(f, \pi)$ 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 $V_n = \{\mathsf{r}\}$ and $V_0 = T = \{\mathsf{t}, \mathsf{f}\}$, 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 $x_{\pi[i]}$.[1] For convenience, we define a map $\mathsf{var} \colon N \to [n]$ such that if $v \in V_i$ then $\mathsf{var} = \pi[i]$.
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 $F \colon V \to \mathcal{F}(f)$ as follows: (a) $F(\mathsf{r}) = f$ for $\mathsf{r} \in V_n$; (b) $F(\mathsf{t}) = \mathsf{true}$ and $F(\mathsf{f}) = \mathsf{false}$ for $\mathsf{t}, \mathsf{f} \in V_0$; (c) For every $u \in N$ and $b \in \{0,1\}$, $F(u_b)$ is the subfunction obtained from $F(u)$ by substituting $x_{\mathsf{var}(u)}$ with b, i.e., $F(u_b) = F(u)|_{x_{\mathsf{var}(u)}=b}$.

[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.

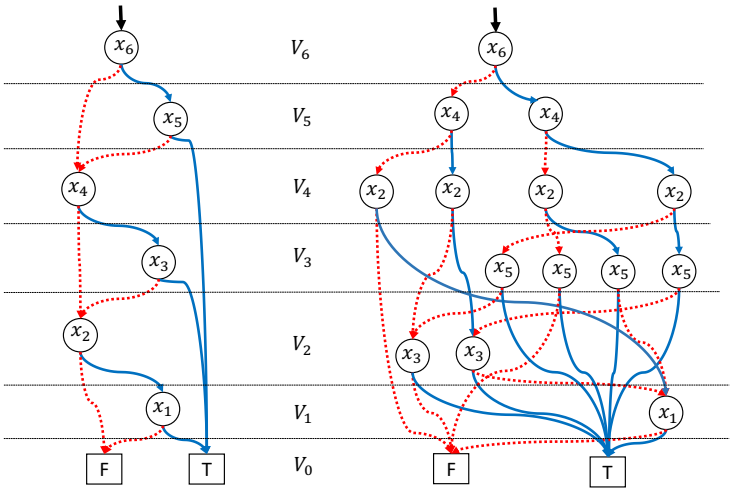

Figure 1 The OBDDs represent the function $f(x_1, x_2, x_3, x_4, x_5, x_6) = x_1x_2 + x_3x_4 + x_5x_6$ under two variable orderings: $(x_1, x_2, x_3, x_4, x_5, x_6)$ (left) and $(x_1, x_3, x_5, x_2, x_4, x_6)$ (right), where the solid and dotted arcs express 1-edges and 0-edges, respectively, and the terminal nodes for true and false are labeled with T and F, respectively. For each $n \in \mathbb{N}$, the function $f(x_1, \ldots, x_{2n}) = x_1x_2 + x_3x_4 + \cdots + x_{2n-1}x_{2n}$ has a $(2n+2)$-sized OBDD for the ordering $(x_1, \ldots, x_{2n})$ and a 2^{n+1}-sized OBDD for the ordering $(x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n})$.

5. $\mathcal{B}(f, \pi)$ must be minimal in the sense that the following reduction rules cannot be applied. In other words, $\mathcal{B}(f, \pi)$ 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* $\{u, v\} \subset N$, then remove v (i.e., any one of them) and its outgoing edges, and redirect all incoming edges of v to u, where u and v are equivalent if (1) $\mathsf{var}(u)$ is equal to $\mathsf{var}(v)$, and (2) u_0 and u_1 are the same nodes as v_0 and v_1, respectively.

For each $j \in [n]$, $\mathsf{Cost}_j(f, \pi)$ 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 π_I be a permutation π in $\Pi(I)$ that minimizes the number of nodes in level 1 to level $|I|$:

$$\pi_I := \arg\min \left\{ \sum_{j=1}^{|I|} \mathsf{Cost}_{\pi[j]}(f, \pi) \colon \pi \in \Pi(I) \right\}. \tag{1}$$

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

$$\pi_{\langle I_1, \ldots, I_m \rangle} := \arg\min \left\{ \sum_{j=1}^{|I_1|+\cdots+|I_m|} \mathsf{Cost}_{\pi[j]}(f, \pi) \colon \pi \in \Pi(\langle I_1, \ldots, I_m \rangle) \right\}. \tag{2}$$

Note that $\min \sum_{j=1}^{|I_1|+\cdots+|I_m|} \mathsf{Cost}_{\pi[j]}(f, \pi) = \sum_{i \in I_1 \sqcup \cdots \sqcup I_m} \mathsf{Cost}_i(f, \pi)$ for any $\pi \in \Pi(\langle I_1, \ldots, I_m \rangle)$. 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 π, 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 $\pi \in \Pi(\langle I \setminus \{i\}, \{i\}\rangle)$, $\mathsf{Cost}_{\pi[|I|]}(f, \pi) \equiv \mathsf{Cost}_i(f, \pi) = c_f$.*

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

$$\textsc{mincost}_{\langle I_1,\ldots,I_m\rangle}[I'] := \sum_{i \in I'} \mathsf{Cost}_i(f, \pi_{\langle I_1,\ldots,I_m\rangle}).$$

As a special case, we denote $\textsc{mincost}_{\langle I_1,\ldots,I_m\rangle}[I_1 \sqcup \cdots \sqcup I_m]$ by $\textsc{mincost}_{\langle I_1,\ldots,I_m\rangle}$. We define $\textsc{mincost}_\emptyset$ 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 $f\colon \{0,1\}^n \to \{0,1\}$, the following holds:*
$\mathit{MINCOST}_I = \min_{k \in I} \left(\mathit{MINCOST}_{I \setminus k} + \mathsf{Cost}_k(f, \pi_{\langle I \setminus k, k\rangle})\right) = \min_{k \in I} \left(\mathit{MINCOST}_{\langle I \setminus k, k\rangle}\right)$.

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

Before sketching algorithm FS, we provide several definitions. For any $I \subseteq [n]$, $\textsc{table}_I$ 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 $\textsc{table}_I[b]$ stores (the pointer to) the unique node of $\mathcal{B}(f, \pi_I)$ associated via F with function $f|_{x_{[n]\setminus I} = b}$. Hence, we may write $\textsc{table}_I[x_{[n]\setminus I} = b]$ instead of $\textsc{table}_I[b]$ to clearly indicate the value assigned to each variable x_j for $j \in [n] \setminus I$. The purpose of $\textsc{table}_I$ is to relate all subfunctions $f|_{x_{[n]\setminus I} = b}$ ($b \in \{0,1\}^{n-|I|}$) 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, $\textsc{table}_\emptyset$ is merely the truth table of f.

Algorithm FS computes $\textsc{table}_I$ together with π_I, $\textsc{mincost}_I$, and another data structure, $\textsc{node}_I$ for all $I \subseteq [n]$, starting from $\textsc{table}_\emptyset$ via dynamic programming. $\textsc{node}_I$ 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 $\mathsf{var}(u) = \pi_I[|I|]$, and (u, u_0) and (u, u_1) are the 0-edge and 1-edge of u, respectively. Thus, $\textsc{node}_I$ contains the structure of the subgraph of $\mathcal{B}(f, \pi_I)$ induced by $V_{|I|}$. The purpose of the $\textsc{node}_I$ 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, $\textsc{node}_\emptyset$ is the empty set. We assume that $\textsc{node}_I$ 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 $\textsc{node}_I$. An example of $\textsc{table}_I$ and $\textsc{node}_I$ is shown in Figure 4 in Appendix.

More generally, for disjoint subset $I_1, \ldots, I_m \subseteq [n]$, $\textsc{table}_{\langle I_1,\ldots,I_m\rangle}$ is an array with $2^{n-|I_1 \sqcup \cdots \sqcup I_m|}$ cells such that, for $b \in \{0,1\}^{n-|I_1 \sqcup \cdots \sqcup I_m|}$, $\textsc{table}_{\langle I_1,\ldots,I_m\rangle}[b]$ stores the nodes of $\mathcal{B}(f, \pi_{\langle I_1,\ldots,I_m\rangle})$ associated with the function $f|_{x_{[n]\setminus I_1 \sqcup \cdots \sqcup I_m} = b}$. $\textsc{node}_{\langle I_1,\ldots,I_m\rangle}$ is defined similarly for $\mathcal{B}(f, \pi_{\langle I_1,\ldots,I_m\rangle})$. For simplicity, we hereafter denote by $\mathcal{FS}(\langle I_1, \ldots, I_m\rangle)$ the quadruplet $(\pi_{\langle I_1,\ldots,I_m\rangle}, \textsc{mincost}_{\langle I_1,\ldots,I_m\rangle}, \textsc{table}_{\langle I_1,\ldots,I_m\rangle}, \textsc{node}_{\langle I_1,\ldots,I_m\rangle})$.

2.3.2 Sketch of Algorithm FS

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

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

$$u_0 \leftarrow \textsc{table}_{I\setminus i}[x_{[n]\setminus I} = b, x_i = 0], \qquad u_1 \leftarrow \textsc{table}_{I\setminus i}[x_{[n]\setminus I} = b, x_i = 1].$$

If $u_0 = u_1$, then store u_0 in $\textsc{table}_{\langle I\setminus i, i\rangle}[b]$. Otherwise, test whether (u, u_0, u_1) for some u is a member of $\textsc{node}_{I\setminus i}$. If it is, store u in the $\textsc{table}_{\langle I\setminus i, i\rangle}[b]$; otherwise create a new triple (u', u_0, u_1), insert it to $\textsc{node}_{\langle I\setminus i, i\rangle}$ and increment $\textsc{mincost}_{\langle I\setminus i, i\rangle}$. Since u' is the pointer to the new node, u' must be different from any pointer already included in $\textsc{node}_{\langle I\setminus i, i\rangle}$ and from any pointer to a node in $V_1 \sqcup \cdots \sqcup V_{k-1}$ in $\mathcal{B}(f, \pi_{\langle I\setminus i\rangle})$, where $k = |I|$. Such u' can be easily chosen by setting u' to two plus the value of $\textsc{mincost}_{\langle I\setminus i, i\rangle}$ before the increment, since the $\textsc{mincost}_{\langle I\setminus i, i\rangle}$ is exactly the number of triples in $\textsc{node}_{\langle I\setminus i, i\rangle}$ plus $|V_1 \sqcup \cdots \sqcup V_{k-1}|$, 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 $\textsc{table}_{\langle I\setminus i\rangle}$. We also mean it by "folding $\textsc{table}_{\langle I\setminus i\rangle}$ with respect to x_i".

The complexity analysis is fairly simple. For each k, we need to compute $\mathcal{FS}(I)$ for $\binom{n}{k}$ possible I's with $|I| = k$. For each I, it takes $O^*(2^{n-k})$ time since the the size of $\textsc{table}_{I\setminus i}$ is 2^{n-k+1} and each operation to $\textsc{node}_{I\setminus i}$ 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 $\textsc{table}_{I\setminus i}$ up to a polynomial factor. The space required by Algorithm FS during the process for k is dominated by that for $\textsc{table}_I$, $\textsc{table}_{I\setminus i}$ and $\textsc{node}_I$ for all I and $i \in I$, which is $O^*\left(2^{n-k}\binom{n}{k}\right)$. The space complexity is thus $O^*\left(\max_{k\in\{0\}\cup[n]} 2^{n-k}\binom{n}{k}\right) = O^*(3^n)$.

▶ **Theorem 5** (Friedman and Supowit [5]). *Suppose that the truth table of $f\colon \{0,1\}^n \to \{0,1\}$ is given as input. Algorithm* 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 $f\colon [N] \to \mathbb{Z}$ given as an oracle, finds an element $x \in [N]$ at which $f(x)$ achieves the minimum, with error probability at most ε 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 polynomially 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 ε 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 $I_1, \dots, I_m, J \subseteq [n]$ with $J \neq \emptyset$ and any Boolean function $f\colon \{0,1\}^n \to \{0,1\}$, the following holds:*

$$\begin{aligned} \mathit{MINCOST}_{\langle I_1,\dots,I_m,J\rangle} &= \min_{k\in J}\left(\mathit{MINCOST}_{\langle I_1,\dots,I_m,J\setminus\{k\}\rangle} + \mathsf{Cost}_k(f,\pi_{\langle I_1,\dots,I_m,J\setminus\{k\},\{k\}\rangle})\right) \\ &= \min_{k\in J}\left(\mathit{MINCOST}_{\langle I_1,\dots,I_m,J\setminus\{k\},\{k\}\rangle}\right). \end{aligned}$$

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 FS^* (its pseudo code is given below, and a schematic view of FS^* is shown in Figure 6 in Appendix).

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

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

Proof. We focuses on the simplest case of $m = 1$, for which our goal is to show an algorithm that produces $\mathcal{FS}(\langle I, J\rangle)$ from $\mathcal{FS}(I)$. It is straightforward to generalize the proof to the case of $m \geq 2$. Starting from $\mathcal{FS}(I)$, the algorithm first folds $\textsc{table}_I$ with respect to each variable in $\{x_j\colon j \in J\}$ to obtain $\mathcal{FS}(\langle I, j\rangle)$ for every $j \in J$, then fold $\textsc{table}_{\langle I,j_1\rangle}$ with respect to x_{j_2} and $\textsc{table}_{\langle I,j_2\rangle}$ with respect to x_{j_1} to obtain $\mathcal{FS}(\langle I, \{j_1, j_2\}\rangle)$ by taking the minimum of $\textsc{mincost}_{\langle I,j_1,j_2\rangle}$ and $\textsc{mincost}_{\langle I,j_2,j_1\rangle}$ for every $j_1, j_2 \in J$, and repeat this to finally obtain $\mathcal{FS}(\langle I, J\rangle)$. 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 $\mathcal{FS}(\langle I, K\rangle)$ from $\mathcal{FS}(\langle I, K - h\rangle)$ is linear to the size of $\textsc{table}_{\langle I,K\rangle}$, i.e., $2^{n-|I|-j}$ up to a polynomial factor. The total time is thus, up to a polynomial factor,

$$\sum_{j=1}^{|J|} 2^{n-|I|-j}\binom{|J|}{j} < 2^{n-|I|-|J|}\sum_{j=0}^{|J|} 2^{|J|-j}\binom{|J|}{j} = 2^{n-|I\sqcup J|}\cdot 3^{|J|}.$$

If we stop the algorithm at $j = k$, then the algorithm produces the set $\{\mathcal{FS}(\langle I, K\rangle)\colon K \subseteq J, |K| = k\}$. The time complexity in this case is at most $2^{n-|I|-|J|}\sum_{j=0}^{k} 2^{|J|-j}\binom{|J|}{j}$, up to a polynomial factor.

Since the space complexity is trivially upper-bounded by the time complexity, we complete the proof. ◀

▶ Remark 9. It is not difficult to see that the algorithm FS^* works even when the function f has a multivalued function: $f\colon \{0,1\}^n \to \mathbb{Z}$. 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.

Algorithm FS^* Composable variant of algorithm FS. "$A \leftarrow B$" means that B is substituted for A.

Input: disjoint subsets $I, J \in [n]$ and $\mathcal{FS}(I)$
Output: $\mathcal{FS}(\langle I, J\rangle)$

```
1  Function Main()
2      for ℓ := 1 to |J| do
3          for each ℓ-element subset K ⊆ J do
4              MINCOST_⟨I,K⟩ ← +∞;                                   // init.
5              for each k ∈ K do
6                  FS(⟨I, K \ k, k⟩) ← FOLD(I, K, k, FS(⟨I, K \ k⟩));
7                  if MINCOST_⟨I,K⟩ > MINCOST_⟨I,K\k,k⟩ then
8                      FS(⟨I, K⟩) ← FS(⟨I, K \ k, k⟩);
9                  end
10             end
11         end
12     end
13     return FS(⟨I, J⟩)
14 end
15 Function FOLD(I, K, k, FS(⟨I, K \ k⟩))   // produce FS(⟨I, K \ k, k⟩) from FS(⟨I, K \ k⟩)
16     π_⟨I,K\k,k⟩ ∈ {π ∈ Π(⟨I, K \ k, k⟩) : π[i] = π_⟨I,K\k⟩[i] (i = 1, ..., |I ⊔ K| − 1)} ;   // init.
17     MINCOST_⟨I,K\k,k⟩ ← MINCOST_⟨I,K\k⟩;                          // init.
18     NODE⟨I, K \ k, k⟩ ← ∅;                                        // init.
19     for b ∈ {0,1}^{n−|I|−|K|} do
20         u_0 ← TABLE_⟨I,K\k⟩[x_[n]\(I⊔K) = b, x_k = 0];
21         u_1 ← TABLE_⟨I,K\k⟩[x_[n]\(I⊔K) = b, x_k = 1];
22         if u_0 = u_1 then
23             TABLE_⟨I,K\k,k⟩[x_[n]\(I⊔K) = b] ← u_0
24         else if ∃u (u, u_0, u_1) ∈ NODE⟨I, K \ k, k⟩ then
25             TABLE_⟨I,K\k,k⟩[x_[n]\(I⊔K) = b] ← u
26         else                                                      // create a new node
27             u ← MINCOST_⟨I,K\k,k⟩ + 2;
28             TABLE_⟨I,K\k,k⟩[x_[n]\(I⊔K) = b] ← u;
29             MINCOST_⟨I,K\k,k⟩ ← MINCOST_⟨I,K\k,k⟩ + 1;
30             insert (u, u_0, u_1) into NODE⟨I, K \ k, k⟩
31         end
32     end
33     return FS(⟨I, K \ k, k⟩)
34 end
```

The following theorem is the basis of our quantum algorithms.

▶ **Lemma 10** (Divide-and-Conquer). *For any disjoint subsets $I_1, \dots, I_m, J \subseteq [n]$ with $J \neq \emptyset$ and any $k \in [|J|]$, it holds that $\mathit{MINCOST}_{\langle I_1,\dots,I_m,J\rangle}[J]$ is equal to*

$$\min_{K\colon K\subseteq J, |K|=k} \left(\mathit{MINCOST}_{\langle I_1,\dots,I_m,K\rangle}[K] + \mathit{MINCOST}_{\langle I_1,\dots,I_m,K,J\setminus K\rangle}[J\setminus K]\right). \tag{3}$$

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

$$\mathit{MINCOST}_{[n]} = \min_{K\subseteq[n], |K|=k} \left(\mathit{MINCOST}_K + \mathit{MINCOST}_{\langle K,[n]\setminus K\rangle}[[n]\setminus K]\right). \tag{4}$$

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

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

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

We can generalize this in a straightforward manner. Let $\pi = \pi_{\langle I_1,\dots,I_m,J\rangle}$ and $\ell = |I_1 \sqcup \cdots \sqcup I_m|$. Then, we have

$$\textsc{mincost}_{\langle I_1,\dots,I_m,J\rangle}[J] = \sum_{j=1}^{k} \mathsf{Cost}_{\pi[\ell+j]}(f,\pi) + \sum_{j=k+1}^{|J|} \mathsf{Cost}_{\pi[\ell+j]}(f,\pi).$$

By defining $K := \{\pi[\ell+1], \dots, \pi[\ell+k]\}$, the same argument as the special case of $\ell = 0$ implies that the first and second sums are $\textsc{mincost}_{\langle I_1,\dots,I_m,K\rangle}[K]$ and $\textsc{mincost}_{\langle I_1,\dots,I_m,K,J\setminus K\rangle}[J\setminus K]$, 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 $k \in [n]$, $\textsc{mincost}_{[n]}$ is the minimum of $\textsc{mincost}_K + \textsc{mincost}_{\langle K,[n]\setminus K\rangle}[[n]\setminus K]$ 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 $\textsc{mincost}_K + \textsc{mincost}_{\langle K,[n]\setminus K\rangle}[[n]\setminus K] = \textsc{mincost}_{\langle K,[n]\setminus K\rangle}$, it suffices to first compute $\mathcal{FS}(K)$ (including $\textsc{mincost}_K$), and then $\mathcal{FS}(\langle K, [n]\setminus K\rangle)$ (including $\textsc{mincost}_{\langle K,[n]\setminus K\rangle}$) from $\mathcal{FS}(K)$. The time complexity for computing $\mathcal{FS}(K)$ from $\mathcal{FS}(\emptyset)$ is $O^*(2^{n-k}3^k)$ by Lemma 8 with $I_1 \sqcup \cdots \sqcup I_m = \emptyset$ and $J = K$, while that for computing $\mathcal{FS}(\langle K, [n]\setminus K\rangle)$ from $\mathcal{FS}(K)$ is $O^*(3^{n-k})$ by Lemma 8 with $m = 1$, $I_1 = K$, and $J = [n]\setminus K$. Thus, the time complexity for computing $\mathcal{FS}(\langle K, [n]\setminus K\rangle)$ from $\mathcal{FS}(\emptyset)$ is $O^*(2^{n-k}3^k + 3^{n-k})$. Thus, for $k = \alpha n$ with $\alpha \in [0,1]$ fixed later, the total time complexity up to a polynomial factor is

$$T(n) = \sqrt{\binom{n}{\alpha n}} \left(2^{(1-\alpha)n}3^{\alpha n} + 3^{(1-\alpha)n}\right) \leq 2^{\frac{1}{2}\mathbf{H}(\alpha)n} \left\{2^{[(1-\alpha)+\alpha\log_2 3]n} + 2^{[(1-\alpha)\log_2 3]n}\right\}.$$

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

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

where $\gamma_0 = 2.98581\ldots$. [2] 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 $\mathcal{FS}(K)$ for every K with $|K|=\alpha n$ ($\alpha\in(0,1)$) by using Algorithm FS*. By Lemma 8, the preprocessing time is then

$$\sum_{j=1}^{\alpha n} 2^{n-j}\cdot\binom{n}{j} \le \alpha n\cdot\max_{j\in[\alpha n]} 2^{n-j}\binom{n}{j} \lessapprox \begin{cases} 2^{(1-\alpha)n+\mathbf{H}(\alpha)n} & (\alpha<1/3)\\ 2^{\frac{2}{3}n+\mathbf{H}(1/3)n} & (\alpha\ge 1/3),\end{cases} \tag{5}$$

since $2^{n-j}\binom{n}{j}$ increases when $j<n/3$ and decreases otherwise. Note that once this preprocess is completed, we can use $\mathcal{FS}(K)$ for free and assume that the cost for accessing $\mathcal{FS}(K)$ 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

$$T(n) = \sum_{j=1}^{\alpha n} 2^{n-j}\cdot\binom{n}{j} + \sqrt{\binom{n}{\alpha n}}\left(n^{O(1)}+3^{(1-\alpha)n}\right) \lessapprox 2^{[(1-\alpha)+\mathbf{H}(\alpha)]n} + 2^{[\frac{1}{2}\mathbf{H}(\alpha)+(1-\alpha)\log_2 3]n}.$$

To balance the both terms, we set $(1-\alpha)+\mathbf{H}(\alpha) = \frac{1}{2}\mathbf{H}(\alpha)+(1-\alpha)\log_2 3$ 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 $T(n)\lessapprox 2^{[(1-\alpha^*)+\mathbf{H}(\alpha^*)]n} = O^*(\gamma_1^n)$, where γ_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 $\boldsymbol{\alpha} := (\alpha_1,\ldots,\alpha_k)$ by $\mathsf{OptOBDD}(k,\boldsymbol{\alpha})$ 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 $x<y$, $f(x,y) := \frac{1}{2}y\cdot\mathbf{H}(x/y)+g(x,y)$ and $g(x,y) := (1-y)+(y-x)\log_2(3)$.

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

$$T(n) = \sum_{\ell=1}^{\alpha_1 n} 2^{n-\ell}\cdot\binom{n}{\ell} + L_{k+1}(n), \tag{6}$$

$$\begin{aligned} L_{j+1}(n) &= \sqrt{\binom{\alpha_{j+1}n}{\alpha_j n}}\left(L_j(n)+2^{(1-\alpha_{j+1})n}3^{(\alpha_{j+1}-\alpha_j)n}\right)\\ &= \sqrt{\binom{\alpha_{j+1}n}{\alpha_j n}}\left(L_j(n)+2^{g(\alpha_j,\alpha_{j+1})n}\right), \end{aligned} \tag{7}$$

[2] More precisely, α^* 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 α^*.

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

Input: $\mathcal{FS}(\emptyset) := \{$ $\text{TABLE}_\emptyset$, $\pi_\emptyset$, $\text{MINCOST}_\emptyset$, $\text{NODE}_\emptyset$ $\}$ (accessible from all Functions)
Output: $\mathcal{FS}([n])$

```
1  Function Main()
2      compute the set {FS(K): K ⊆ [n], |K| = ⌊α_1 n⌋} by algorithm FS (or FS*);
3      make the set of these FS(K) global (i.e., accessible from all Functions);
4      return DivideAndConquer([n], k + 1)
5  end
6  Function DivideAndConquer(L, t)          // Compute FS(L) with α_1, ..., α_t(= |L|/n)
7      if t = 1 then return FS(L);                    // FS(L) has been precomputed.
8      Find K(⊂ L) of cardinality ⌊α_{t-1} n⌋, with Lemma 6, that minimizes MINCOST_{⟨K, L\K⟩},
9          which is computed as a component of FS(⟨K, L \ K⟩) by calling ComputeFS(K, L \ K, t);
10     let K* be the set that achieves the minimum;
11     return FS(⟨K*, L \ K*⟩)
12 end
13 Function ComputeFS(K, M, t)                 // Compute FS(⟨K, M⟩) with α_1, ..., α_t
14     FS(K) ← DivideAndConquer(K, t − 1);
15     FS(⟨K, M⟩) ← FS*(K, M, FS(K));
16     return FS(⟨K, M⟩)
17 end
```

where $j \in [k]$ and $L_1(n) = O^*(1)$. Intuitively, $L_j(n)$ is the time required for producing $\mathcal{FS}(\langle K_1, K_2 \setminus K_1, \ldots, K_j \setminus K_{j-1}\rangle)$ such that $\text{MINCOST}_{\langle K_1, K_2\setminus K_1, \ldots, K_j \setminus K_{j-1}\rangle}$ is minimum over all $K_1, \ldots, K_{j-1}$ satisfying $|K_\ell| = \alpha_\ell n$ for every $\ell \in [k+1]$ and $K_\ell \subset K_{\ell+1}$ for every $\ell \in [k]$.

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

$$L_{j+1}(n) \lessapprox 2^{f(\alpha_j,\alpha_{j+1})n}.$$

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

$$T(n) \lessapprox \sum_{\ell=1}^{\alpha_1 n} 2^{n-\ell} \cdot \binom{n}{\ell} + 2^{f(\alpha_k,\alpha_{k+1})n} \lessapprox 2^{(1-\alpha_1)n + \mathbf{H}(\alpha_1)n} + 2^{f(\alpha_k,1)n},$$

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 $\alpha_1 < 1/3$:

$$1 - \alpha_1 + \mathbf{H}(\alpha_1) = f(\alpha_k, 1), \tag{8}$$

$$f(\alpha_{j-1}, \alpha_j) = g(\alpha_j, \alpha_{j+1}) \quad (j = 2, \ldots, k). \tag{9}$$

By numerically solving this system of equations, we obtain $T(n) = O(\gamma_k^n)$, where γ_k is at most 2.83728 for $k = 6$. The value of γ_k becomes smaller as k increases. However, incrementing k beyond 6 provides only negligible improvement of γ_k. Since the space complexity is trivially upper-bounded by the time complexity, we have the following theorem. Note that the values

of α_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 $f\colon \{0,1\}^n \to \{0,1\}$ given as input, produces $\mathcal{FS}([n])$ with probability $1-\exp(-\Omega(n))$ in $O^*(\gamma^n)$ time and space, where the constant γ is at most 2.83728, which is achieved by $\mathsf{OptOBDD}(k,\boldsymbol{\alpha})$ with $k=6$ and $\boldsymbol{\alpha}=(0.183791, 0.183802, 0.183974, 0.186131, 0.206480, 0.343573)$.*

4 Quantum Algorithm with Composition

4.1 Quantum 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 $I_1,\dots,I_m,J\subseteq[n]$ with $J\neq\emptyset$, there exists a quantum algorithm that, with probability $1-\exp(-\Omega(n))$, produces $\mathcal{FS}(\langle I_1,\dots,I_m,J\rangle)$ from $\mathcal{FS}(\langle I_1,\dots,I_m\rangle)$ for an underlying function $f\colon\{0,1\}^n\to\{0,1\}$ in $O^*\left(2^{n-|I_1\sqcup\cdots\sqcup I_m\sqcup J|}\cdot\gamma^{|J|}\right)$ time and space, where γ 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 $\mathsf{OptOBDD}^*_\Gamma(k,\boldsymbol{\alpha})$, where the subroutine Γ appearing in line 17 is set to the deterministic algorithm FS^*, and k and $\boldsymbol{\alpha}$ are set to the values specified in Theorem 11.

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

Input: $I\subseteq[n]$, $J\subseteq[n]$, $\mathcal{FS}(I)$. (accessible from all Functions)
Output: $\mathcal{FS}(\langle I,J\rangle)$

1 **Function** Main()
2 $\quad n'\leftarrow|J|$; `// init.`
3 $\quad$ compute the set $\{\mathcal{FS}(\langle I,K\rangle)\colon K\subseteq J,\ |K|=\lfloor\alpha_1 n'\rfloor\}$ by algorithm FS^*;
4 $\quad$ make n' and the above set of $\mathcal{FS}(\langle I,K\rangle)$ global (i.e., accessible from all Functions);
5 $\quad$ **return** DivideAndConquer$(J,k+1)$
6 **end**
7 **Function** DivideAndConquer(L,t) `// Compute` $\mathcal{FS}(\langle I,L\rangle)$ `with` $\alpha_1,\dots,\alpha_t$
8 $\quad$ **if** $t=1$ **then return** $\mathcal{FS}(I,L)$; `//` $\mathcal{FS}(I,L)$ `has been precomputed.`
9 $\quad$ Find $K(\subset L)$ of cardinality $\lfloor\alpha_{t-1}n'\rfloor$, with Lemma 6, that minimizes $\textsc{mincost}_{\langle I,K,L\setminus K\rangle}$
10 $\quad\quad$ which is computed as a component of $\mathcal{FS}(\langle I,K,L\setminus K\rangle)$
11 $\quad\quad$ by calling ComputeFS$(I,K,L\setminus K,t)$;
12 $\quad$ let K^* be the set that achieves the minimum;
13 $\quad$ **return** $\mathcal{FS}(\langle I,K^*,L\setminus K^*\rangle)$
14 **end**
15 **Function** ComputeFS(I,K,M,t) `// Compute` $\mathcal{FS}(\langle I,K,M\rangle)$ `with` $\alpha_1,\dots,\alpha_t$
16 $\quad\mathcal{FS}(I,K)\leftarrow$ DivideAndConquer$(K,t-1)$;
17 $\quad\mathcal{FS}(\langle I,K,M\rangle)\leftarrow\Gamma(I,K,M,\mathcal{FS}(I,K))$;
18 $\quad$ **return** $\mathcal{FS}(\langle I,K,M\rangle)$
19 **end**

▶ **Lemma 13** (Quantum Composition: Induction Part)**.** *Suppose that* Γ *is a quantum algorithm that, for any disjoint subsets* $I_1, \dots, I_m, J \subseteq [n]$ *with* $J \neq \emptyset$*, produces* $\mathcal{FS}(\langle I_1, \dots, I_m, J\rangle)$ *from* $\mathcal{FS}(\langle I_1, \dots, I_m\rangle)$ *with probability* $1 - \exp(-\Omega(n))$ *in* $O^*\left(2^{n-|I_1 \sqcup \cdots \sqcup I_m \sqcup J|} \cdot \gamma^{|J|}\right)$ *time and space for an underlying function* $f\colon \{0,1\}^n \to \{0,1\}$*. Then, for any constant parameters* $k \in \mathbb{N}$ *and* $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_k) \in [0,1]^k$ *with* $\alpha_1 < \cdots < \alpha_k$ *and for any disjoint subsets* $I_1, \dots, I_m, J \subseteq [n]$ *with* $J \neq \emptyset$*,* $\mathsf{OptOBDD}^*_\Gamma(k, \boldsymbol{\alpha})$ *produces* $\mathcal{FS}(\langle I_1, \dots, I_m, J\rangle)$ *from* $\mathcal{FS}(\langle I_1, \dots, I_m\rangle)$ *with probability* $1 - \exp(-\Omega(n))$ *in* $O^*\left(2^{n-|I_1 \sqcup \cdots \sqcup I_m \sqcup J|} \cdot \beta_k^{|J|}\right)$ *time and space for the function* f*, where* β_k^n *upper-bounds, up to a polynomial factor, the time complexity required for* $\mathsf{OptOBDD}^*_\Gamma(k, \alpha)$ *to compute* $\mathcal{FS}([n])$ *from* $\mathcal{FS}(\emptyset)$*, that is,* $T(n) = O^*(\beta_k^n)$ *for* $T(n)$ *that satisfies the following recurrence:*

$$T(n) = \sum_{\ell=1}^{\alpha_1 n} 2^{n-\ell} \binom{n}{\ell} + L_{k+1}, \tag{10}$$

$$L_{j+1} = \sqrt{\binom{\alpha_{j+1} n}{\alpha_j n}} \left(L_j + 2^{(1-\alpha_{j+1})n} \gamma^{(\alpha_{j+1}-\alpha_j)n}\right) = \sqrt{\binom{\alpha_{j+1} n}{\alpha_j n}} \left(L_j + 2^{g_\gamma(\alpha_j, \alpha j+1)}\right), \tag{11}$$

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

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

Next, we generalize $\mathsf{OptOBDD}_\Gamma(k, \boldsymbol{\alpha})$ so that it produces $\mathcal{FS}(\langle I_1, \dots, I_m, J\rangle)$ from $\mathcal{FS}(\langle I_1, \dots, I_m\rangle)$ for any disjoint subsets $I_1, \dots, I_m, J \subseteq [n]$ with $J \neq \emptyset$. The proof is very similar to that of Lemma 12. The only difference is that the time complexity of Γ is $O^*\left(2^{n-|I_1 \sqcup \cdots \sqcup I_m \sqcup J|} \cdot \gamma^{|J|}\right)$, instead of $O^*\left(2^{n-|I_1 \sqcup \cdots \sqcup I_m \sqcup J|} \cdot 3^{|J|}\right)$. Namely, when $m = 1$ and $n' = |J|$, the time complexity of $\mathsf{OptOBDD}^*_\Gamma(k, \boldsymbol{\alpha})$ satisfies the following recurrence: for each $j \in [n]$,

$$\begin{aligned}
T'(n,n') &= 2^{n-|I|-n'} \sum_{\ell=1}^{\alpha_1 n'} 2^{n'-\ell} \binom{n'}{\ell} + L'_{k+1}(n,n'), \\
L'_{j+1}(n,n') &= \sqrt{\binom{\alpha_{j+1} n'}{\alpha_j n'}} \left(L'_j(n,n') + 2^{n-|I|-\alpha_{j+1} n'} \gamma^{(\alpha_{j+1}-\alpha_j)n'}\right) \qquad [j \in [n]], \\
L'_1(n,n') &= O^*(1),
\end{aligned}$$

from which it follows that $T'(n,n') = 2^{n-|I|-n'} T(n') = O^*\left(2^{n-|I \sqcup J|} \cdot \beta_k^{|J|}\right)$. 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 Γ 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 Γ_1 as $\mathsf{OptOBDD}^*_{\mathsf{FS}^*}(k^{(0)}, \boldsymbol{\alpha}^{(0)})$ for some $k^{(0)} \in \mathbb{N}$ and $\boldsymbol{\alpha}^{(0)} \in [0,1]^{k^{(0)}}$. Then, we define Γ_2 as

$\mathsf{OptOBDD}^*_{\Gamma_1}(k^{(1)}, \boldsymbol{\alpha}^{(1)})$ for some $k^{(1)} \in \mathbb{N}$ and $\boldsymbol{\alpha}^{(1)} \in [0,1]^{k^{(1)}}$. In this way, we can define Γ_{i+1} as $\mathsf{OptOBDD}^*_{\Gamma_i}(k^{(i)}, \boldsymbol{\alpha}^{(i)})$ for some $k^{(i)} \in \mathbb{N}$ and $\boldsymbol{\alpha}^{(i)} \in [0,1]^{k^{(i)}}$.

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

$$1 - \alpha_1^{(i)} + \mathbf{H}(\alpha_1^{(i)}) = f_\gamma(\alpha_6^{(i)}, 1), \tag{12}$$

$$f_\gamma(\alpha_{j-1}^{(i)}, \alpha_j^{(i)}) = g_\gamma(\alpha_j^{(i)}, \alpha_{j+1}^{(i)}) \qquad (j = 2, \ldots, 6), \tag{13}$$

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

By numerically solving this system of equations for $\gamma = 3$, we have $\beta_6 < 2.83728$ as shown in Theorem 11. Then, numerically solving the system of equations with $\gamma = 2.83728$, we have $\beta_6 < 2.79364$. In this way, we obtain a certain γ 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* $f\colon \{0,1\}^n \to \{0,1\}$ *given as input, produces* $\mathcal{FS}([n])$ *in* $O^*(\gamma^n)$ *time and space with probability* $1 - \exp(-\Omega(n))$*, where the constant* γ *is at most* 2.77286*.*

References

1. Andris Ambainis, Kaspars Balodis, Jānis Iraids, Martins Kokainis, Krišjānis Prūsis, and Jevgēnijs Vihrovs. Quantum speedups for exponential-time dynamic programming algorithms. In *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 1783–1793, 2019. doi:10.1137/1.9781611975482.107.
2. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
3. Harry Buhrman, Richard Cleve, Ronald de Wolf, and Christof Zalka. Bounds for small-error and zero-error quantum algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science, FOCS*, pages 358–368, 1999.
4. Christoph Dürr and Peter Høyer. A quantum algorithm for finding the minimum. *arXiv.org e-Print archive*, quant-ph/9607014, 1996. URL: http://arxiv.org/abs/quant-ph/9607014.
5. Steven J. Friedman and Kenneth J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. Comput.*, 39(5):710–713, 1990.
6. Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Phys. Rev. Lett.*, 100:160501, 2008.
7. Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley Professional, 1 edition, 2009.
8. Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications.* Springer, 1998.
9. Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. 30th ACM/IEEE Design Automation Conference*, pages 272–277, 1993.
10. Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 2000.
11. Seiichiro Tani. Quantum algorithm for finding the optimal variable ordering for binary decision diagrams. *arXiv.org e-Print archive*, arXiv:1909.12658, 2019. URL: https://arxiv.org/abs/1909.12658.
12. Ingo Wegener. *Branching Programs and Binary Decision Diagrams.* SIAM Monographs on Discrete Mathematics and Applications. SIAM, 2000.

A Appendix

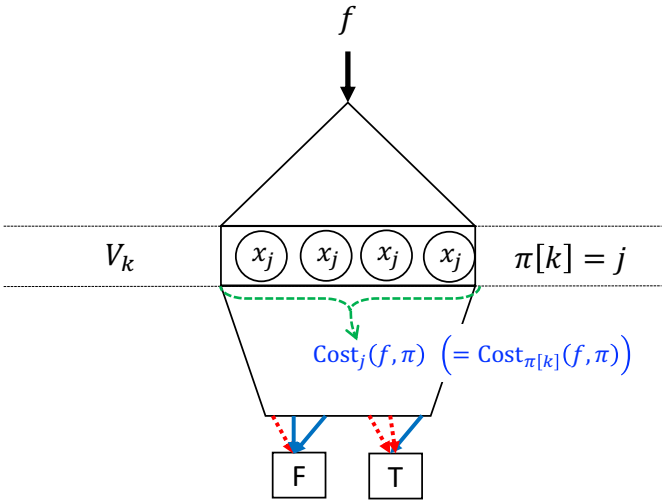

Figure 2 Schematic expression of $\mathsf{Cost}_j(f,\pi)$.

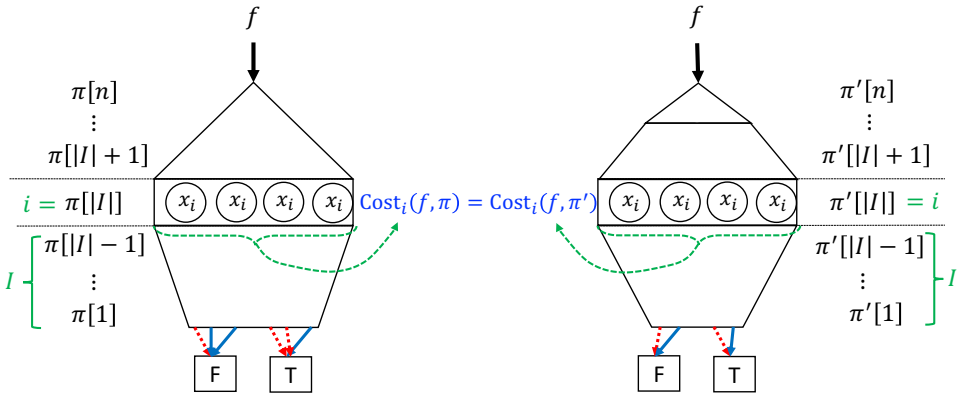

Figure 3 Schematic expression of Lemma 3: For any two permutations $\pi, \pi' \in \mathcal{S}_n$ such that $\{\pi[1], \ldots, \pi[|I|-1]\} = \{\pi'[1], \ldots, \pi'[|I|-1]\}$ and $\pi[|I|] = \pi'[|I|]$, it holds that the number of nodes labeled with x_i in $\mathcal{B}(f,\pi)$ is equal to that of nodes labeled with x_i in $\mathcal{B}(f,\pi')$.

$I = \{1,3,5\}$ $\pi_I = (1,3,5,2,4,6)$

Table$_I$

$x_6x_4x_2$	pointer
000	0
001	2
010	3
011	4
100	6
101	7
110	5
111	8

NODE$_I$

u	u_0	u_1
5	3	1
6	0	1
7	2	1
8	4	1

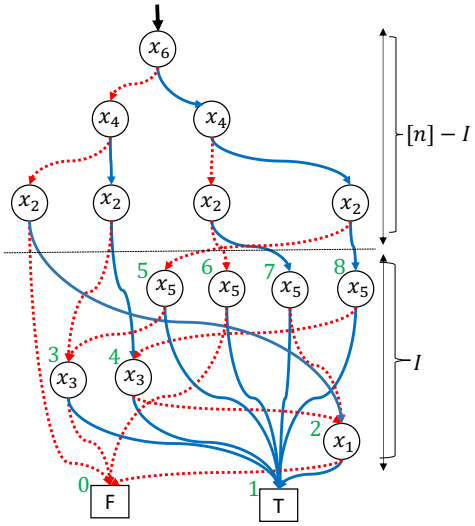

Figure 4 Examples of data structures used in Algorithm FS: TABLE$_I$ and NODE$_I$ with $I = \{1,3,5\}$ for the OBDD (rhs) representing $f(x_1,\ldots,x_6) = x_1x_2 + x_3x_4 + \cdots + x_5x_6$ for the variable ordering $(x_1,x_3,x_5,x_2,x_4,x_6)$. The pointers (integers) to the nodes labeled with x_1,x_3,x_5 are each shown at the top-left positions of the nodes.

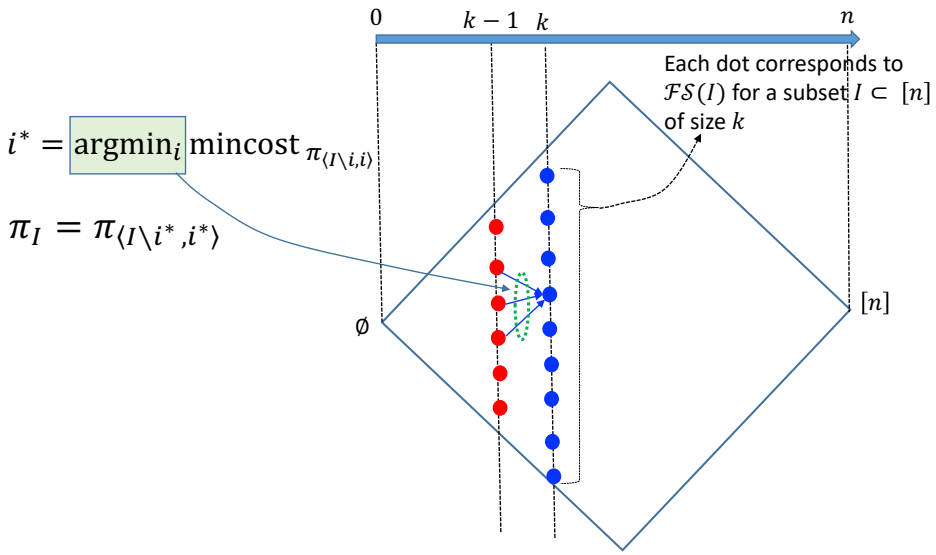

Figure 5 Schematic view of Friedman-Supowit Algorithm. The algorithm goes from the left to the right. On the vertical line indicated by k, there are $\binom{n}{k}$ dots, each of which corresponds to $\mathcal{FS}(I)$ for a subset $I \subseteq [n]$ of size k. $\mathcal{FS}(I)$ is computed from $\mathcal{FS}(\langle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

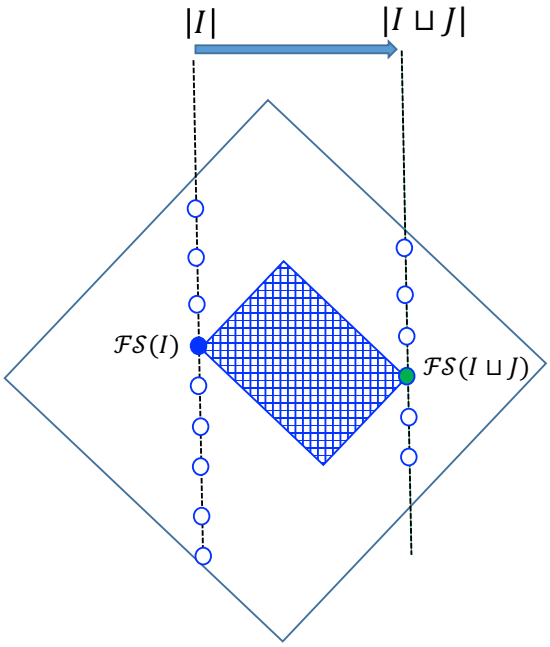

Figure 6 Schematic view of FS*. This view corresponds to the case where $m = 1$ and $J \subset [n] \setminus I$ in Lemma 8. The shaded area is the one that FS* sweeps to produce $\mathcal{FS}(\langle I, J\rangle)$.

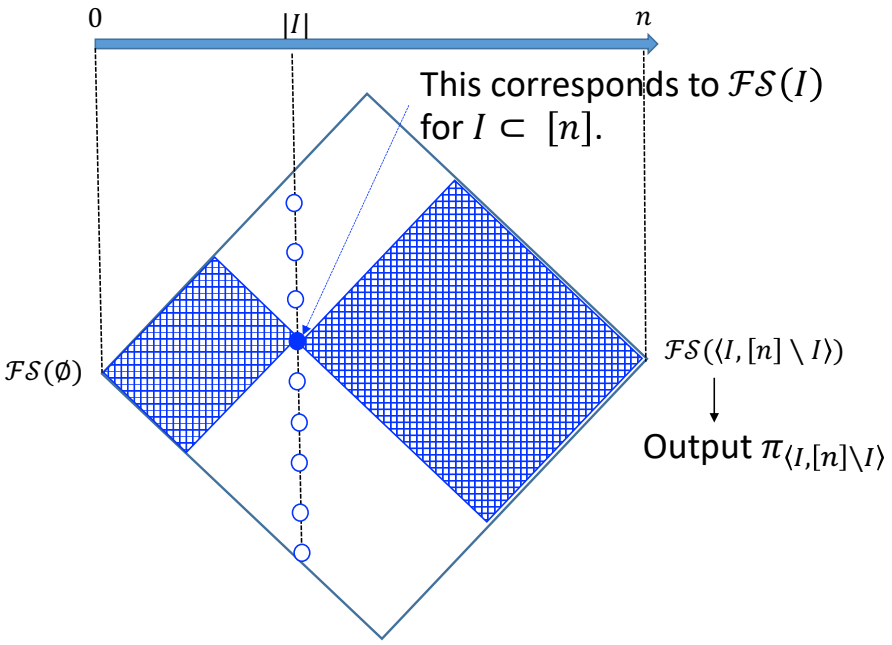

Figure 7 Schematic view of Eq. (4) in Lemma 10. Intuitively, the lemma says that it is possible to decompose FS* into the parts each of which consists of the two shaded rectangles that share the dot corresponding to $\mathcal{FS}(I)$ on the line indicated by $|I|$ for a subset $I \subseteq [n]$ of some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

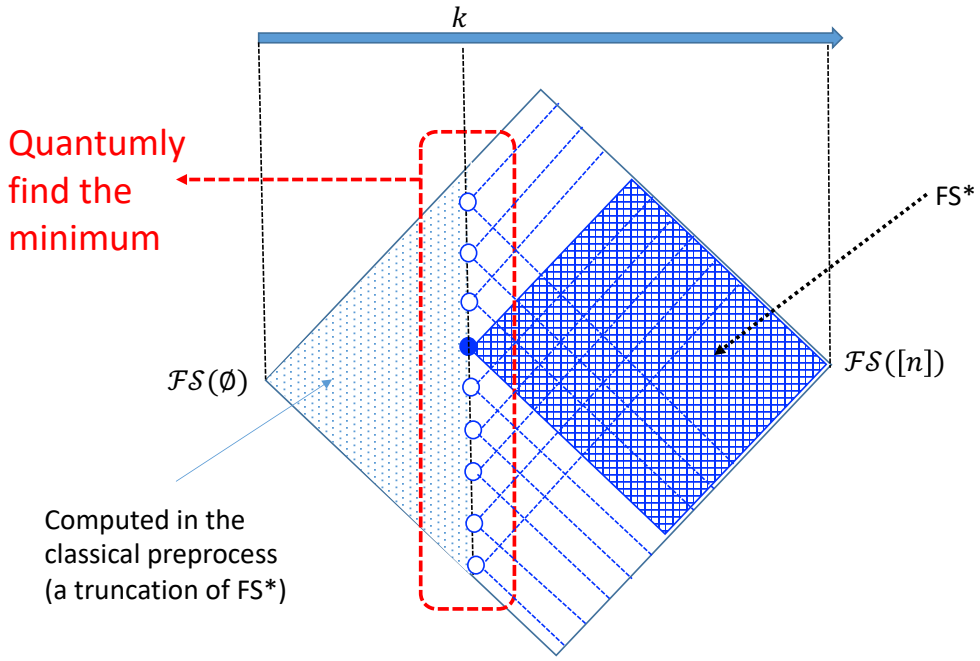

Figure 8 Schematic view of our algorithm in the simplest case (one-parameter case). The dotted area is computed in the classical preprocess, which is realized by truncating the process of FS^* as stated in Lemma 8. The shaded area is computed by using FS^*. The actual algorithm runs the quantum minimum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