An Algorithm for Reversible Logic Circuit Synthesis Based on Tensor Decomposition

HOCHANG LEE, The Affiliated Institute of ETRI, Daejeon, Korea (the Republic of)
KYUNG CHUL JEONG, The Affiliated Institute of ETRI, Daejeon, Korea (the Republic of)
DAEWAN HAN, The Affiliated Institute of ETRI, Daejeon, Korea (the Republic of)
PANJIN KIM, The Affiliated Institute of ETRI, Daejeon, Korea (the Republic of)

An algorithm for reversible logic synthesis is proposed. The task is, for a given n-bit substitution map, to find a sequence of reversible logic gates that implements the map. The gate library adopted in this work consists of multiple-controlled Toffoli gates with m control bits, where $m \in \{0, \ldots, n-1\}$. Controlled gates with large $m(> 2)$ are then further decomposed into smaller gates ($m \leq 2$). A primary goal in designing the algorithm is to reduce the number of Toffoli gates which is known to be universal.

The main idea is to view an n-bit substitution map as a rank-$2n$ tensor, and to transform it such that the resulting map can be written as a tensor product of a rank-$(2n-2)$ tensor and the 2×2 identity matrix. It can then be seen that the transformed map acts nontrivially on $n-1$ bits only, meaning that the map to be synthesized becomes $(n-1)$-bit substitution. This size reduction process is iteratively applied until it reaches a tensor product of only 2×2 matrices.

The time complexity of the algorithm is exponential in n as most previously known heuristic algorithms for reversible logic synthesis are, but it terminates within reasonable time for not too large n which may find practical uses. As stated earlier, our primary target is to reduce the number of Toffoli gates in the output circuit. Benchmark results show that the algorithm works well for hard benchmark functions, but it does not seem advantageous when the function is structured. As an application, the algorithm is applied to find reversible circuits for cryptographic substitution boxes, which are often required in quantum cryptanalysis.

CCS Concepts: • **Theory of computation** → **Design and analysis of algorithms**; **Circuit complexity**; *Quantum computation theory*.

Additional Key Words and Phrases: logic synthesis, reversible circuits, quantum computing

1 Introduction

Beginning from the mid-twentieth century, studies on reversible computing have been motivated by several factors such as power consumption, debugging, routing, performance issues in certain cases, and so on [60]. The circuit-based quantum computing model is also closely related to reversible computing, where every component of the circuit works as a unitary transformation except the measurement [49]. As in classical logic synthesis where abstract circuit behavior is designed by using a specified set of logic gates such as {AND, NOT}, quantum logic synthesis is a process of designing a logic circuit in terms of certain reversible gates.

Authors' Contact Information: Hochang Lee, The Affiliated Institute of ETRI, Daejeon, Korea (the Republic of); e-mail: lhc254@nsr.re.kr; Kyung Chul Jeong, The Affiliated Institute of ETRI, Daejeon, Korea (the Republic of); e-mail: jeongkc@nsr.re.kr; Daewan Han, The Affiliated Institute of ETRI, Daejeon, Korea (the Republic of); e-mail: dwh@nsr.re.kr; Panjin Kim, The Affiliated Institute of ETRI, Daejeon, Korea (the Republic of); e-mail: pansics@nsr.re.kr.

ACM 2643-6817/2024/6-ART
https://doi.org/10.1145/3673242

A bijection map of the form $P_n : \{0,1\}^n \to \{0,1\}^n$ often appears as a target behavior to be synthesized in various computational problems, for example in analyzing cryptographic substitution boxes (S-box) [39] or hidden weighted bit functions [15, 16], or permutation network routing [20]. We can also see that such a map can be interpreted as a permutation, for example, $\sigma = (7, 2, 0, 1, 5, 3, 6, 4)$ is a bijection map $\sigma : \{0,1\}^3 \to \{0,1\}^3$, which gives $\sigma(000) = 111$, $\sigma(001) = 010$, and so on. Let us call it a permutation map. Reversible logic synthesis on such maps has been studied intensively [45, 61, 73]. Interested readers may refer to benchmark pages [57] and [22], and related references therein. It is especially a nontrivial problem to synthesize a reversible circuit for a permutation behavior that does not have an apparent structure (Section 2.2). Most known algorithms for unstructured permutations with $n > 4$ are heuristic ones with the runtime exponential in n.

One of the well-known methods for synthesizing an n-bit permutation is to find a process of transforming P_n into $\sigma_{n,\mathrm{id}}$, where $\sigma_{n,\mathrm{id}}$ is the n-bit identity permutation. The reason why finding the process is equivalent to designing a reversible circuit is given in Section 2.2. A straightforward way to achieve the goal is to rewrite the permutation as a product of at most $2^n - 1$ transpositions and to synthesize each transposition in terms of specified gates [49]. This naive approach is easy to implement, but the resulting circuit involves a large number of Toffoli gates. Researchers have introduced various algorithms to improve the method, and what we have noticed is that instead of finding a direct map for identity, one may try a map that reduces the effective size of the permutation such that $P_n \mapsto P_{n-1}$ and iteratively apply it.

One of the merits of the proposed algorithm would be that it is applicable to any intermediate-size substitution map, whether or not it is structured. A structured function (for example, some finite field arithmetic operations) can certainly be optimized through human effort, but each result can hardly be generalized to other kinds of logic behaviors. From the perspective of productivity, it could be helpful to set a baseline by using automatic tools such as the proposed algorithm, and then look for optimizations.

This work is summarized as follows:

- An algorithm for reversible logic circuit synthesis is proposed. The algorithm is designed so that the output circuit involves as small number of Toffoli gates as possible. Comparisons with the previous results for benchmark functions are summarized in Table 1.
- The algorithm is applied to AES [51], Skipjack [50], KHAZAD [10], and DES [48] S-boxes. Except for AES, all other S-boxes do not have apparent structures where no known polynomial-time algorithm can be applied. Indeed, reversible circuits for these S-boxes have never been suggested prior to this work. It is also worth noting that the output circuits of the algorithm are always garbageless for the permutation maps.

An easy-to-use implementation of the algorithm written in Python is also available from GitHub [1].

The paper is organized as follows. Section 2 briefly covers the basics of reversible logic circuit synthesis and related works. Sections 3 and 4 are devoted to bringing out the design criteria of the algorithm. Benchmark results and applications to cryptographic S-boxes are presented in Section 5.

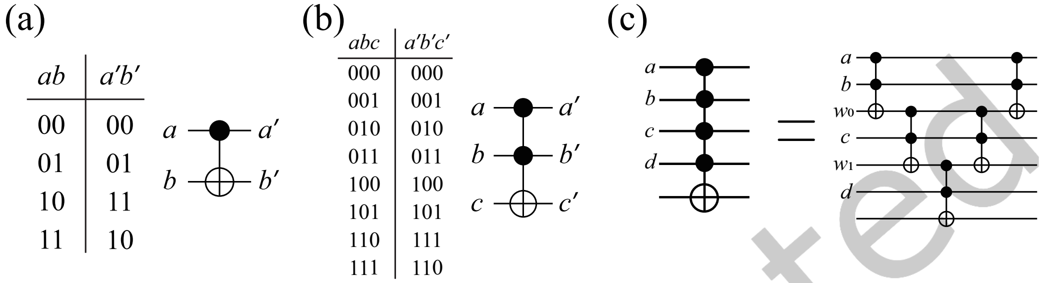
2 Background

Logic circuit synthesis involves a functionally complete set of logic gates such as {AND, NOT}, which we would call a universal gate set. In this section, we first briefly introduce widely adopted *reversible* gate sets, which are called multiple-controlled Toffoli (MCT) library and NOT, CNOT, Toffoli (NCT) library [57].

Throughout the paper, most numberings are zero-based for example ‘zeroth column of a matrix’, except for bit positions. In referring to the position of a bit we use one-based numberings, such as the ‘first’ for the first bit or the ‘n-th’ for the last bit.

2.1 Gate Library

Controlled-NOT, also known as CNOT, takes two input bits, one being a control and the other being a target. Similarly, a Toffoli gate takes three input bits, two being controls and the other being a target. Fig. 1 illustrates CNOT and Toffoli gates which are clearly reversible. NOT gate is already reversible, and NCT library is known to be a universal set, meaning that any reversible logic can be implemented by using these gates [68].



ab	$a'b'$
00	00
01	01
10	11
11	10

abc	$a'b'c'$
000	000
001	001
010	010
011	011
100	100
101	101
110	111
111	110

Fig. 1. Truth tables and circuit symbols for (a) CNOT and (b) Toffoli gates. (c) Decomposition of C^4X into five (= $2 \cdot 4 - 3$) Toffoli gates. Here a, b, c, d are input control bits and w_0, w_1 are zeroed work bits.

Naturally, CNOT can be generalized to take $m + 1$ input bits with m controls and one target. It is sometimes called an MCT gate [57]. Let C^mX denote such gates where m is a non-negative integer. Since NCT gates are already included in MCT library (as with $m = 0, 1, 2$), MCT library is also a universal set.

A C^mX gate with $m > 2$ can be decomposed into a few NCT gates with various tradeoffs. One may think of it as a conversion formula from an MCT gate to NCT gates. For example, C^mX can be constructed by using $2m - 3$ Toffoli gates with $m - 2$ zeroed work bits or by using $8m - 24$ Toffoli gates with one arbitrary work bit [8]. Fig. 1(c) illustrates the former formula. Note that there can be various ways to optimize such conversion.

The number of Toffoli gates involved in a synthesized circuit will be our cost metric for a few reasons. First, since the algorithm is for a permutation, not for a unitary operation, Toffoli gate itself is universal [68]. We may further decompose Toffoli into even smaller gates such as T or Hadamard, but such tasks can be conveyed to independent compilation. Secondly, we have in mind that the algorithm is applicable to quantum computing. To our best knowledge, one of the major huddles in realizing the circuit-based quantum hardware is to implement non-Clifford gates (such as T or Toffoli) fault tolerantly. As we are not considering synthesizing the unitaries, targeting Toffoli could be reasonable.

We say *quality* of a circuit $C_{\mathcal{A}}$ synthesized by an algorithm $\mathcal{A}$ is better than a circuit $C_{\mathcal{B}}$ synthesized by an algorithm $\mathcal{B}$, if the number of Toffoli gates involved is smaller in $C_{\mathcal{A}}$. Note that the quality gets better as the number of Toffoli gates decreases. An algorithm we design primarily uses MCT library, but then to measure the number of Toffoli gates, we need a conversion formula for all C^mX gates with $m > 2$. For this purpose, we will use a simple formula described in Fig.1(c).

2.2 Permutation Circuit Synthesis

Having an appropriate basis, an n-bit permutation written in one-line notation $(r_0, r_1, \ldots, r_{2^n-1})$ can be seen as a matrix in $\{0,1\}^{2^n \times 2^n}$. For example for $n = 3$ with the standard basis $\boldsymbol{b}_0 = (1\ 0\ 0\ 0\ 0\ 0\ 0\ 0)^\top$, $\boldsymbol{b}_1 = (0\ 1\ 0\ 0\ 0\ 0\ 0\ 0)^\top, \cdots$, a permutation $P_3 = (7, 2, 0, 1, 5, 3, 6, 4)$ can be written by a truth table and by a matrix.

$$\begin{array}{c|c} \text{in} & \text{out} \\ \hline 000 & 111 \\ 001 & 010 \\ 010 & 000 \\ 011 & 001 \\ 100 & 101 \\ 101 & 011 \\ 110 & 110 \\ 111 & 100 \end{array}, \quad P_3 = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \tag{1}$$

where we have used the same symbol P_n interchangeably to denote the one-line notation of an n-bit permutation and the corresponding matrix. Having this in mind, we define permutation circuit synthesis as follows:

Definition 2.1. For an n-bit permutation P_n, a finite universal gate set G, and a cost metric on a set, permutation circuit synthesis is to find a finite ordered set $R = \{g_i \mid g_i \in G\}$ such that $P_n \cdot (g_1 \cdot g_2 \cdot \cdots \cdot g_{|R|}) = I_{2^n}$, minimizing the cost on R.

Let X_i denote a NOT gate acting on i-th bit, CX_{ij} denote a CNOT gate controlled by i-th bit acting on j-th, CX_{ijk} denote a Toffoli gate controlled by i- and j-th bits acting on k-th bit. One can verify that $P_3 \cdot CX_{13} \cdot CX_{312} \cdot X_2 \cdot CX_{23} \cdot CX_{231} = I_{2^3}$ (gate action is described for example in Eq. (3)), and thus $CX_{231} \cdot CX_{23} \cdot X_2 \cdot CX_{312} \cdot CX_{13} = P_3$ since NCT gates are involutory ($A = A^{-1}, A \in \text{NCT}$).

The permutation circuit synthesis is closely related with symmetric groups. A set of n-bit permutations form the symmetric group $\mathcal{S}_{2^n}$, where $|\mathcal{S}_{2^n}| = 2^n!$.

2.3 Related Works

Study of algorithms for reversible logic circuit synthesis has been an active area of research over the last decades. One question that can be asked when one sees, for example the synthesis of P_3 as above, is whether the obtained circuit is optimal. In fact, the optimality of small-bit permutations (with a certain metric), or even larger permutations with linear structures, have been studied [25, 26, 38, 40, 55, 66, 72]. The algorithms *search* the target circuit such that its minimal cost can be guaranteed, for example by exhaustive search or meet-in-the-middle method.

From a theoretical point of view, a natural question is how small the synthesized circuit can be for given bijective function $f : \{0,1\}^n \to \{0,1\}^n$. Let G be a gate library and let K_n be a set of n-bit permutations that can be synthesized by the gates in G. It has been proven by [66] that the worst case n-bit permutation can be synthesized by $\Omega(|K_n|/\log|G|)$ gate-length circuit on n wires. If we choose $G = \text{NCT}$, then the lower bound reads $\Omega(n2^n/\log n)$. The paper has also shown that for a linear function, the worst-case synthesis can be as short as $\Omega(n^2/\log n)$ gate-length circuit. The asymptotically optimal algorithm for linear functions did come out later [53] with $O(n^2/\log n)$ gate-length output.

Apart from the worst case, when a permutation is structured, it is often possible to find an efficient circuit by exploiting its structure. Here by 'structured' we mean r_i from the previous subsection is efficiently computable such that $r_i = g(i)$ with some function g. If such a function is known, one may design an algorithm that computes g reversibly. Examples include arithmetic operations such as squaring in $\mathbb{F}_2^n$, or cryptographic S-boxes [27, 51].

On the other hand, when an n-bit permutation shows no apparent structure with n being larger than 4, then we are left with options using heuristic algorithms [60] or reversible lookup table methods [65]. A reversible lookup table implements each r_i for corresponding i one by one, reversibly. Although straightforward and possibly not less efficient than heuristic algorithms asymptotically, the reversible lookup table method is not preferred over heuristic algorithms for the problems at hand to our consideration.

There seems no clear criterion in classifying the known heuristic algorithms, but we briefly introduce a few well-known branches. When the synthesis is viewed as a search problem, since the search space grows exponentially as the number of bits increases, one may try *heuristic search*. For example the greedy method can be used as in [29], where priority-based tree search is applied with pruning. *Transformation-based* algorithms find an ordered set of gates that the sequential multiplications on the given permutation results in the identity function [45], as we have seen in Section 2.2. One obvious way is to write down the permutation as a composition of transpositions and implement each transposition by reversible gates. *Cycle-based* algorithms try to decompose the given permutation in terms of smaller cycles and then synthesize each cycle [61, 66]. Not being entirely independent of the above methods, but a number of approaches rely on *functional representation* of the given permutation, looking for reversible circuits corresponding to each part of the representation. Examples include decision diagram [36, 43], exclusive-or-sum-of-product [24, 63], and positive-polarity Reed-Muller expansion [3].

The algorithm we propose recursively looks for the decomposition of smaller tensors (Section 3). A similar notion has been investigated by [70], utilizing Young subgroups. Their algorithm makes use of the fact that for any $a \in \mathcal{S}_{2^n}$, there exists a decomposition $a = h_1 \cdot v \cdot h_2$ where h_i and v are elements of the certain Young subgroups. By finding efficiently implementable h_1 and h_2, and then by recursively applying the method to v, one obtains the reversible circuit. Therefore, the recursive procedure is always applied to the center element of the decomposition, resulting in a kind of two-way construction (gates are applied from the front (h_1) and from the end (h_2)). The proposed algorithm in this work is different from [70], as it is a kind of one-way construction (Section 3.1). Perhaps there could be a connection between [70] and ours, for example there is a chance that the tensor decomposition we look for in each recursion is related to Young subgroups, but it is inconclusive in our analysis. One advantage of the proposed algorithm is that since it recursively reduces the problem size, one bit at a time becomes free. More discussion is given in Section 3.1.

Due to the heuristic nature of the algorithms, it is not easy to analyze the pros and cons of each algorithm. Therefore, benchmark functions are often synthesized by the newly proposed algorithms for the comparisons [22, 57]. The results are given in Section 5 and [1].

A slightly off-topic but related problem is synthesis of unitary gates (Chapter 4, [49]). Similar to reversible logic synthesis, various approaches are examined for general unitaries [5, 9, 75], or structured (Clifford) ones [2, 14, 38]. For Clifford circuits, 6-bit functions are synthesized in near optimal ways [14].

3 Basic Idea

The basic strategy this work has taken is to reduce the effective size of a matrix at each step by looking for a size reduction process, leaving one bit completely excluded from subsequent steps. The size reduction idea naturally results in an algorithm for permutation circuit synthesis under an assumption. The assumption will be lifted in Section 4 , leading to an algorithm that works for any permutation.

3.1 Matrix Size Reduction

Reversible gates acting on n bits can be viewed as rank-$2n$ tensors [69]. Among them, there exist certain gates of which decomposition as a tensor product of smaller tensors can be found. For example, Walsh-Hadamard transformation acting on two bits can be represented as a 4×4 matrix (or equivalently rank-4 tensor), which can also be decomposed into two 2×2 matrices such as

$$H^{\otimes 2} = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \ . \tag{2}$$

However, CNOT cannot be written as a simple tensor product of two smaller tensors.

Now consider a permutation P_n that can be written as $P_{n-1} \otimes I_2$. It means the gate P_n acts nontrivially on $n-1$ bits only, effectively leaving one bit irrelevant from the operation. It is clear that if one is able to find a procedure for transforming an arbitrary rank-$2n$ tensor into one that can be decomposed into one rank-$(2n-2)$ tensor and one 2×2 identity matrix, the circuit can be synthesized by iterating the procedure at most $n-1$ times.

A somewhat related notion has been examined in synthesizing unitary matrices [47, 59], but apart from the use of the notion *block* (Section 3.2), the design criteria are quite different. In a nutshell, their method is to divide a large circuit into several smaller circuits and each one is recursively divided into even smaller ones, involving gate operations to every bit all the way through to the end. On the contrary, each time the size of the permutation gets smaller in our method, one bit becomes completely irrelevant from the circuit.

3.2 Definitions and Conventions

In n-bit permutation circuit synthesis, a gate set we use consists of C^mX gates, where the number of control bits ranges from 0 to $n-1$.

Dealing with a permutation is probably best comprehensible in matrix representation as in Eq. (1), with an obvious downside that it is inconvenient to write down. One-line notation is thus frequently adopted throughout the paper. For example, the action of X_1, CX_{21}, and CX_{312} on $(7, 2, 0, 1, 5, 3, 6, 4)$ reads

$$\begin{aligned} & \xmapsto{X_1} \left(\colorbox{lightgray}{$5, 3, 6, 4$}, \colorbox{lightgray}{$7, 2, 0, 1$} \right), \\ (7, 2, 0, 1, 5, 3, 6, 4) & \xmapsto{CX_{21}} \left(7, 2, \colorbox{lightgray}{$6, 4$}, 5, 3, \colorbox{lightgray}{$0, 1$} \right), \\ & \xmapsto{CX_{312}} \left(7, 2, 0, 1, 5, \colorbox{lightgray}{4}, 6, \colorbox{lightgray}{3} \right). \end{aligned} \tag{3}$$

When a permutation is written by $P_n = (r_0, r_1, ..., r_{2^n-1})$, the subscripts i and integers r_i are understood as *column numbers* and *row numbers*, respectively, in which nonzero values reside in the matrix representation. For example, $r_0 = 7$ means 1 is located at the 0th column and the 7th row in the matrix as in Eq. (1). Using these notions, a permutation P_n can be viewed as a function of a column number,

$$\begin{aligned} P_n : \{0, 1, \cdots, 2^n - 1\} & \rightarrow \{0, 1, \cdots, 2^n - 1\}, \\ i & \mapsto r_i. \end{aligned}$$

Column numbers will frequently be read as n-bit binary strings. Binary strings will be denoted by vector notation when necessary. In writing an integer x as an n-bit binary string $\boldsymbol{x} \in \{0, 1\}^n$, we let $x_i \in \{0, 1\}$ denote the i-th bit of $\boldsymbol{x}$ for $1 \leq i \leq n$.

One way to understand the action of a logic gate is to think of it as an operator that exchanges column numbers. When a gate is applied, it first reads column numbers as n-bit binary strings. Among the strings (columns), some must meet the condition for the activation of the gate. The row numbers that reside in these columns are swapped appropriately. For example in Eq. (3), each column number is read as $000, 001, \cdots, 111$ which are occupied by row numbers $7, 2, \cdots, 4$, respectively. The gate CX_{132} is activated when the value of the first and the third bits are both 1, which correspond to the 5th (101) and the 7th (111) columns. Row numbers 3 and 4 which preoccupied these positions are then swapped by the gate.

In addition to X_i, CX_{ij}, and CX_{ijk} gates defined in Section 2.2, we introduce a way to specify control and target bits of general C^mX gates with nonnegative integer m. Let $\mathcal{I}$ be a subset of $\{1, \cdots, n\}$. In denoting a controlled gate with an arbitrary number of control bits, an index set $\mathcal{I}$ will be used such that $CX_{\mathcal{I}:k}$ has $|\mathcal{I}|$ control bits specified by elements in $\mathcal{I}$ and targets k-th bit.

The following definition for *block* is the central notion in this work, and it is recommended to refer to the example in Eq. (4) before comprehending the formal definition.

Definition 3.1. For an n-bit permutation $(r_0, \ldots, r_{2^n-1})$, a pair of numbers r_{2i} and r_{2i+1} is defined as an even (odd) block if $r_{2i+1} - r_{2i} = 1\ (-1)$, where $i \in \{0, 1, ..., 2^{n-1} - 1\}$ is called a block-wise position.

It is unlikely that the number of blocks found in an unstructured permutation is large.[1] It is then our task to find a transformation to maximize the number. For example, assume there is a map applied to a permutation matrix in Eq. (1) as follows:

$$\begin{pmatrix} 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&1&0\\ 1&0&0&0&0&0&0&0 \end{pmatrix} \mapsto \left(\begin{array}{cc:cc:cc:cc} 0&0&0&0&0&0&\colorbox{lightgray}{0}&\colorbox{lightgray}{1}\\ 0&0&0&0&0&0&\colorbox{lightgray}{1}&\colorbox{lightgray}{0}\\ \hdashline \colorbox{lightgray}{1}&\colorbox{lightgray}{0}&0&0&0&0&0&0\\ \colorbox{lightgray}{0}&\colorbox{lightgray}{1}&0&0&0&0&0&0\\ \hdashline 0&0&0&0&\colorbox{lightgray}{1}&\colorbox{lightgray}{0}&0&0\\ 0&0&0&0&\colorbox{lightgray}{0}&\colorbox{lightgray}{1}&0&0\\ \hdashline 0&0&\colorbox{lightgray}{0}&\colorbox{lightgray}{1}&0&0&0&0\\ 0&0&\colorbox{lightgray}{1}&\colorbox{lightgray}{0}&0&0&0&0 \end{array}\right). \tag{4}$$

The 2×2 substructures in gray color are called blocks. Even (odd) blocks are 2×2 identity (off-diagonal) structures. In one-line notation of the resulting matrix $(2, 3, 7, 6, 4, 5, 1, 0)$, pairwise row numbers 2, 3 and 4, 5 are even blocks and 7, 6 and 1, 0 are odd blocks.[2]

These pairs of row numbers play an important role hereafter, but there is a chance the notions confuse readers. To avoid confusion, let us describe a way to construct an even block '2, 3' beginning from $(7, 2, 0, 1, 5, 3, 6, 4)$. Using the r_i notation, initially we have the pair $r_1 = 2$ and $r_5 = 3$. First, we want to put 2 in the 0th column. Applying $X_1 CX_{13} X_1$ leads to $(2, 7, 1, 0, 5, 3, 6, 4)$. This procedure can be interpreted as moving row number 2 from column position 1 to 0; $r_1 \mapsto r_0$. Similarly, applying CX_{31} then results in $(2, 3, 1, 4, 5, 7, 6, 0)$; $r_5 \mapsto r_1$. As already explained, applying gates can be interpreted as exchanging column numbers for fixed row numbers. All the algorithms and subroutines introduced below have similar descriptions that first targeting a certain pair of row numbers, and then changing their column positions. At this point, let us formally define such pairs.

Definition 3.2. A pair of row numbers $2j, 2j + 1$ for $0 \le j \le 2^{n-1} - 1$ is called a relevant row pair, denoted by $\langle 2j, 2j + 1 \rangle$.

Row numbers comprising a relevant row pair are called relevant row numbers. Constructing a block is thus putting relevant row numbers appropriately side-by-side that are originally located apart.

In addition, we further define the following:

Definition 3.3. For a relevant row pair $\langle r_i, r_j \rangle$, the row numbers r_i and r_j are said to be occupied at

- normal positions if $r_i \equiv i$ and $r_j \equiv j \mod 2$.
- inverted positions if $r_i \not\equiv i$ and $r_j \not\equiv j \mod 2$.
- interrupting positions otherwise.

To better understand the definitions, consider a permutation $(\colorbox{lightgray}{7}, \boxed{2}, \colorbox{lightgray}{0}, \boxed{1}, \colorbox{lightgray}{5}, \boxed{3}, \colorbox{lightgray}{6}, \boxed{4})$ and examine the relevant row pairs. Relevant row numbers $r_2 = 0$ and $r_3 = 1$ are at normal positions as $2 \equiv_2 0$ and $3 \equiv_2 1$. Another relevant numbers $r_7 = 4$ and $r_4 = 5$ are at inverted positions as $7 \not\equiv_2 4$ and $4 \not\equiv_2 5$. Other numbers are at interrupting positions. In simple terms, 0 and 1 are at normal positions as the small one is in the gray box, 4 and

[1] It is related with Hat matching (or Hat check) problem. For an even permutation, the asymptotic probability of an n-bit arbitrary permutation having k free blocks is $1/(k!e)$. (It does not depend on n.)

[2] Note that the resulting matrix in Eq. (4) is not a tensor product of two smaller tensors, yet. Further application of CX_{23} achieves four even blocks thereby completing the procedure.

5 are at inverted positions as the small one is in the white box, and other numbers are at interrupting positions as the numbers are in the same colored boxes.

It may seem plausible that the relevant numbers in normal (inverted) positions in the beginning likely end up in even (odd) blocks in the series of transformations for maximizing the number of blocks. Indeed if we are careful enough, all the row numbers in the normal (inverted) positions in the beginning form even (odd) blocks at the end. A formal description of the observation is as follows:

Remark 3.1. *When C^mX gate is applied, the number of normal, inverted, and interrupting positions is conserved unless the gate is targeting the n-th bit.*[3]

For example, a permutation $(7, 2, 0, 1, 5, 3, 6, 4)$ has four row numbers at interrupting positions; 2, 3 and 7, 6. The action of CX_{32} that does not *target* the 3rd bit leads to the permutation $(7, 1, 0, 2, 5, 4, 6, 3)$, which still has four interrupting positions. On the other hand, if CX_{13} is applied, the resulting permutation $(7, 2, 0, 1, 3, 5, 4, 6)$ will no longer involve any interrupting position, but instead, the number of normal and inverted positions will be increased by two each. Details will not be covered, but we would point out that handling the interrupting positions is typically more expensive than dealing with normal or inverted positions. Therefore in the next section, a preprocess that removes interrupting positions before getting into the main process will be introduced. A way to deal with the inverted positions will also be introduced in the next section. For now, let us restrict our attention to permutations that only have normal positions. It will soon turn out that reversible gates targeting the last bit are not required at all in this section.

A natural way to design an algorithm for the size reduction could be to build blocks one-by-one, without losing already built ones. Recall that in one-line notation for a given permutation $(\boxed{r_0, r_1}, \boxed{r_2, r_3}, \boxed{r_4, r_5}, \boxed{r_6, r_7}, \cdots)$, a block is a pair of row numbers r_i that differ by 1 (with the odd one being larger) and both reside in one of the designated column positions (boxes). Applying reversible gate swaps positions of at least two row numbers as in Eq. (3). What we want to do is to construct a new block while maintaining already constructed ones, which very much resembles solving Rubik's Cube. Each rotation in Rubik's Cube corresponds to the application of a logic gate. The difference is that while the number of rotations is to be minimized typically in Rubik's Cube, we try to minimize the number of Toffoli gates.

3.3 Glossary

All necessary definitions and notations have been introduced, which we summarize below.

X_i, CX_{ij}, CX_{ijk}	NCT gates (Section 2.2)
$CX_{\mathcal{I}:k}$	MCT gates (Section 2.1)
R	An ordered set of gates
P_n	An n-bit substitution map
$(r_0, r_1, \ldots, r_{2^n-1})$	One-line notation of P_n
Row number r_i	Below Eq. (3)
Column number i (of r_i)	Below Eq. (3)
Block	Definition 3.1
Block-wise position	Definition 3.1
Relevant row pair	Definition 3.2
Relevant row numbers	Below Definition 3.2
Normal position	Definition 3.3
Inverted position	Definition 3.3
Interrupting position	Definition 3.3
$\boldsymbol{x}$	Binary string of an integer x
x_i	The i-th bit of $\boldsymbol{x}$ for $1 \leq i \leq n$
Quality	In Section 2.1

[3] More specifically, each row number remains still in its respective position unless the specified gates are involved, but we only need the fact that the number of such positions is conserved.

3.4 Size Reduction Algorithm

In this section we restrict our attention to a procedure $\mathcal{A}'_{\text{red}} : \mathcal{P}'_n \to \{P_n : P_n = P_{n-1} \otimes I_2\} \times \mathcal{R}$, where $\mathcal{P}'_n$ is a set of all n-bit permutations of which row numbers are only in normal positions and $\mathcal{R}$ is a set of all ordered set of MCT gates. The algorithm $\mathcal{A}'_{\text{red}}$ is designed to obey three rules.

- A block is constructed one-by-one.
- The constructed block is allocated to the left in one-line notation.
- The number of left-allocated blocks should not decrease upon the action of any gate.

The meaning of the construction and the allocation is given in an example. Assume a three-bit permutation $P_3 = (0, 1, 6, 3, 2, 5, 4, 7)$ is at hand. The first block is already there, and we target the next block $(4, 5)$. Applying CX_{312} results in $P_3 \cdot CX_{312} = (0, 1, 6, 3, 2, 7, 4, 5)$. One can see that a new block is *constructed* in the 6th and the 7th columns at the cost of one Toffoli gate. The constructed block is then *allocated* in the 2nd and the 3rd columns by applying CX_{21}, i.e., $P_3 \cdot CX_{312} \cdot CX_{21} = (0, 1, 4, 5, 2, 7, 6, 3)$. In this example, only the construction costs a Toffoli gate whereas the allocation does not, but in general allocation also costs Toffoli gates. In spite of the seemingly unnecessary costs of the allocation, we conclude that the left-allocation is more beneficial than leaving a block where it is constructed. Detailed reasonings for the left-allocation will not be covered[4].

In describing algorithms and subroutines, operations that update the permutation or the gate sequence will be frequently involved. For a permutation P_n, an ordered set of gates $R = (a_1, \ldots, a_{|R|})$, and another ordered set of gates $S = (b_1, \ldots, b_{|S|})$, define a symbol $\cdot$ such that

$$(P_n, R) \cdot S = \left(P_n \cdot b_1 \cdot b_2 \cdot \ldots \cdot b_{|S|}, (R; S)\right),$$
$$(R; S) = (a_1, \ldots, a_{|R|}, b_1, \ldots, b_{|S|}).$$

High-level description of the size reduction algorithm is as follows:

Algorithm 1 $\mathcal{A}'_{\text{red}}$

```
Input $P_n$                                   ▷ $P_n \in \mathcal{P}'_n$
Output $(P, R)$                               ▷ $P = P_{n-1} \otimes I_2$
$R \leftarrow ()$                             ▷ An empty ordered set
$P \leftarrow P_n$
for $i$ from 0 to $2^{n-1} - 1$ do
    $\langle a, b \rangle \leftarrow$ PICK$(P, i)$          ▷ Pick a relevant row pair
    $S_C \leftarrow$ CONS$(P, i, \langle a, b \rangle)$
    $(P, R) \leftarrow (P, R) \cdot S_C$
    $S_A \leftarrow$ ALLOC$(P, i, a)$
    $(P, R) \leftarrow (P, R) \cdot S_A$
return $(P, R)$
```

Three subroutines PICK, CONS, and ALLOC will be examined in detail in the next subsection, and the working example is given in Appendix I, but it will be helpful to have some intuition behind the details here. Define m as the smallest positive integer satisfying

$$2l \leq \sum_{j=1}^{m-1} 2^{n-j}, \tag{5}$$

[4] Roughly explained, if the blocks are located randomly across the possible positions, it will get more and more difficult to construct a new block while maintaining the already constructed ones since cheaper logic gates tend to stir many positions. One way to mitigate the difficulty is to make blocks share the same bit value in the binary string of the column numbers so that certain controlled-gates leave such blocks unaffected by using that bit as a control.

where l is the number of left-allocated blocks and we define $m = 1$ for $l = 0$. A pattern of how m is determined depending on l is illustrated in Fig. 2. Now notice that a group of left-allocated blocks distinguished by dashed lines in the figure shares the same bit values in their column numbers as n-bit binary strings. For example, from $i = 0$ to $2^{n-2} - 1$ in the figure, the first bit of the column numbers is zero. From $i = 2^{n-2}$ to $2^{n-2} + 2^{n-3} - 1$, the first bit is one and the second bit is zero. The point is, in constructing a new block, the (already) left-allocated blocks have some common properties. The properties can be exploited to build a block without too much cost being paid, which we shall prove in Section 3.5 to be upper bounded by one C^mX and a few CX and X gates.

In constructing a new block in the i-th iteration where i is the iterator in Algorithm 1, not all remaining relevant row pairs can be constructed as a block meeting the bound. However, it can be shown that at least one pair that meets the bound always exists for all i. Subroutine PICK takes care of passing an appropriate pair to CONS and ALLOC such that the quality bound can be met.

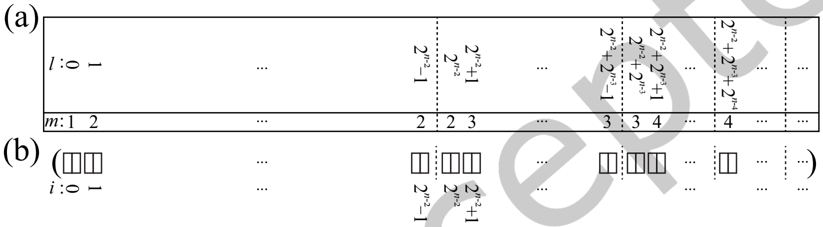


Fig. 2. (a) Relation between m and l. (b) Column positions (rectangle) and block-wise positions (two conjoined rectangles) in a permutation. Constructed blocks are to be allocated to the left, occupying block-wise positions denoted by i. When one tries to construct and allocate a new block at i-th position, l equals i and thus m is determined as specified. Dashed vertical lines divide the number of column positions to be as ratios 1:1, 3:1, 7:1, and so on.

Allocation is similar to construction. Let $\oplus$ denote the bit-wise XOR of binary strings. Suppose we are to construct a block by conjoining two relevant row numbers of which column numbers are α and β, respectively, where $\boldsymbol{\alpha} \oplus \boldsymbol{\beta} \in \{0,1\}^n$. Construction can be understood as a process of changing α and β so that $\boldsymbol{\alpha} \oplus \boldsymbol{\beta} = 00\cdots01$, without breaking left-allocated blocks. By breaking, we mean some of the previously left-allocated blocks are no longer left-allocated, or the number of left-allocated blocks decreases. Now, suppose we successfully constructed a block with the aforementioned relevant row pair, and we want to allocate it to the i-th block-wise position. Let c_i denote one of the column numbers at i-th block-wise position (regular squares in Fig. 2). Allocation is a process of changing α so that $\boldsymbol{c_i} \oplus \boldsymbol{\alpha} = 00\cdots0*$ $(* \in \{0,1\})$, without breaking left-allocated blocks and with preserving $\boldsymbol{\alpha} \oplus \boldsymbol{\beta} = 00\cdots01$. Therefore ALLOC is more or less the same as CONS, and it will be shown in detail that in allocating a constructed block to i-th block-wise position, the cost is upper bounded by $C^{s(i)}X$ and a few CX and X gates, where $s(i)$ is a function such that $s(i) \leq HW(\boldsymbol{i})$, where $HW(\boldsymbol{i})$ is the hamming weight of $\boldsymbol{i}$.

How PICK, CONS, and ALLOC are designed so that the number of Toffoli gates involved is bounded is given in Section 3.5 as proofs. The subroutine PICK is described below. The role of PICK is to output a relevant row pair that requires the bounded number of Toffoli gates for the construction and the allocation.

Subroutine 1 PICK

```
    Input $P_n$, $i$
    Output $\langle a, b \rangle$
 1: $m \leftarrow$ FINDM($i$)
 2: $k \leftarrow 2^n - 2^{n-m+1}$
 3: for $j$ from $k$ to $2^n - 2$ do
 4:     $a \leftarrow r_j$
 5:     $b \leftarrow \boldsymbol{a} \oplus \mathbf{1}$                        ▷ $\mathbf{1} = 00 \ldots 01$
 6:     for $t$ from $j + 1$ to $2^n - 1$ do
 7:         $c \leftarrow r_t$
 8:         if $b = c$ then return $\langle a, b \rangle$
```

Termination of the subroutine will be shown to be guaranteed in the next subsection. In line 1, FINDM computes m from i by letting $l = i$ in Eq. (5). Next, CONS is described below. It takes an input pair from PICK, and then constructs a block. It is designed following the proof of Lemma 3.5, in a way that the number of Toffoli gates involved is bounded.

Subroutine 2 CONS

```
    Input $P_n, i, \langle a, b \rangle$
    Output $S_C$
 1: $S_C \leftarrow (\ )$                                   ▷ An empty ordered set
 2: $m \leftarrow$ FINDM($i$)
 3: $(\alpha, \beta) \leftarrow$ COL($P_n, a, b$)                      ▷ Find column numbers
 4: $\boldsymbol{\gamma} \leftarrow \boldsymbol{\alpha} \oplus \boldsymbol{\beta}$
 5: $\delta \leftarrow 0$
 6: for $j$ from 1 to $n$ do
 7:     if $\gamma_j = 1$ then $\delta \leftarrow j$; break
 8: if $\delta = n$ then return $S_C$                          ▷ Already a block
 9: if $i_\delta = 1$ then $S_C \leftarrow S_C; X_\delta$                    ▷ Append $X_\delta$ to $S_C$
10: for $j$ from $\delta + 1$ to $n - 1$ do
11:     if $\gamma_j = 1$ then $S_C \leftarrow S_C; CX_{\delta j}$
12: if $i_\delta = 1$ then $S_C \leftarrow S_C; X_\delta$
13: $\mathcal{I} \leftarrow \{n\}$
14: for $j$ from 1 to $m - 1$ do
15:     $\mathcal{I} \leftarrow \mathcal{I} \cup \{j\}$
16: $S_C \leftarrow S_C; CX_{\mathcal{I}:\delta}$
17: return $S_C$
```

In line 3, COL outputs column numbers α, β corresponding to the relevant row numbers a, b. This procedure is necessary since what CONS essentially does is to swap columns as mentioned earlier. Next, ALLOC is similarly described, which allocates the constructed block to the left. It is designed following the proof of Lemma 3.6, in a way that the number of Toffoli gates involved is bounded.

Subroutine 3 ALLOC

```
Input $P_n, i, a$
Output $S_A$
$S_A \leftarrow (\,)$
$m \leftarrow \mathsf{FINDM}(i)$
$\alpha \leftarrow \mathsf{COL}(P_n, a)$
$\boldsymbol{\gamma} \leftarrow \boldsymbol{\alpha} \oplus \boldsymbol{i}$
$\delta \leftarrow 0$
for $j$ from 1 to $n$ do
    if $\gamma_j = 1$ then $\delta \leftarrow j$; break
if $\delta = 0$ then return $S_A$                                    ▷ Already allocated
$\mathcal{I} \leftarrow \{\,\}$
for $j$ from $\delta + 1$ to $n - 1$ do
    if $\gamma_j = 1$ then $S_A \leftarrow S_A; CX_{\delta j}$
    if $i_j = 1$ then $\mathcal{I} \leftarrow \mathcal{I} \cup \{j\}$
$S_A \leftarrow S_A; CX_{\mathcal{I}:\delta}$
return $S_A$
```

3.5 Complexity

It has been sketched in Section 3.4 how the size reduction algorithm works. Here we evaluate the time complexity and the quality of the output by introducing two lemmas, one for the construction and the other for the allocation.

Proposition 3.4. *Let l be the number of left-allocated blocks, and m be the smallest positive integer satisfying Eq. (5). Define a function $h_n : \mathbb{N} \rightarrow \mathbb{Z}$, $h_n(x) = 2^n - 2^{n-(x-1)}$. There exists at least one relevant row pair r_α, r_β such that*

$$h_n(m) < \min(\alpha, \beta), \tag{6}$$

where α, β are column numbers of r_α, r_β, respectively.

Proof. It is nothing more than Pigeonhole principle. It is trivial for $m = 1$. For $m > 1$, there exist $2^n - 2l$ row numbers that do not form left-allocated blocks yet. Let us call them the remaining row numbers (row pairs). Assume there is no remaining row pair satisfying Eq. (6). Then at least one relevant row number in every remaining pair has to sit in column numbers no greater than $h_n(m)$. However, since $2l$ column numbers are already occupied by left-allocated blocks, there only exist $h_n(m) - 2l$ columns available for the row numbers to sit in. If we subtract $h_n(m) - 2l$ from half the number of remaining row numbers $(2^n - 2l)/2$,

$$\frac{(2^n - 2l)}{2} - (h_n(m) - 2l) = l - \frac{h_n(m-1)}{2} > 0,$$

where the inequality follows from the fact that $h_n(m-1) < 2l \leq h_n(m)$ by the definition of m. Therefore the assumption cannot be true and there must exist at least one pair that satisfies Eq. (6). □

Lemma 3.5. *Let i be an iterator used in Algorithm 1, l be the number of left-allocated blocks, and m be the smallest positive integer satisfying Eq. (5). For a given permutation, a new block can be constructed by using at most $n - m - 1$ CX, two X, and one C^mX gates, without breaking left-allocated blocks.*

Proof. Let $C = \{0, 1, \cdots, 2^{n-1} - 1\}$ be a set of block-wise positions of an n-bit permutation, $\boldsymbol{i}$ be the binary string of $i \in C$, and r_α, r_β be relevant row numbers satisfying Eq. (6). Viewing their column numbers as binary strings $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, the first $m - 1$ bits of them are 1 (none if $m = 1$),

$$\alpha_1 = \beta_1 = \alpha_2 = \beta_2 = \cdots = \alpha_{m-1} = \beta_{m-1} = 1. \tag{7}$$

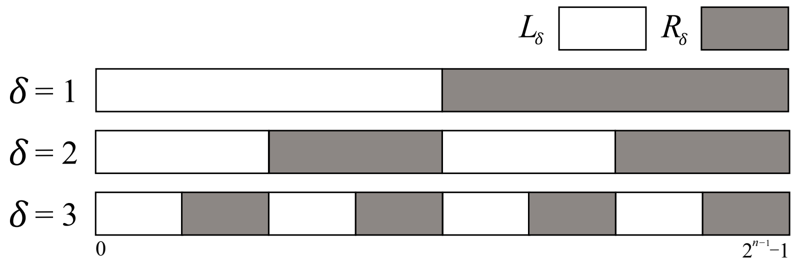


Fig. 3. L_δ and R_δ for $\delta = 1, 2, 3$.

Let $\boldsymbol{\gamma} = \boldsymbol{\alpha} \oplus \boldsymbol{\beta}$ and δ be the smallest integer such that $\gamma_\delta = 1$. It can be seen that $\delta > m - 1$. Assume $\delta \neq n$, otherwise, the pair is already a block. Let L_δ, R_δ be disjoint subsets of C such that $C = L_\delta \bigcup R_\delta$ and the blocks residing in L_δ and R_δ are preserved under the actions of $CX_{\delta x}$ and $X_\delta CX_{\delta x} X_\delta$ gates for $x \in \{1, \ldots, \delta-1, \delta+1, \ldots, n\}$, respectively. A few L_δ and R_δ are illustrated in Fig. 3.

At i-th iteration, note that i (as a block-wise number) is either included in L_δ or R_δ. If $i \in L_\delta$, an action of $CX_{\delta x}$, $\delta < x < n$ gate does not break left-allocated blocks since the blocks in L_δ are not affected at all and the blocks in R_δ change their respective positions within $< i$. If on the other hand $i \in R_\delta$, similarly an action of $X_\delta CX_{\delta x} X_\delta$, $\delta < x < n$ gate does not break left-allocated blocks since the blocks in R_δ are conserved and the blocks in L_δ change their respective positions within $< i$. Because i_δ tells if it is included in L_δ or R_δ, we are able to apply the aforementioned gates without breaking left-allocated blocks.

First $\delta - 1$ bits of $\boldsymbol{\gamma}$ are zeros due to the definition of δ. Since $CX_{\delta x}$ or $X_\delta CX_{\delta x} X_\delta$, $\delta < x < n$ gate can be applied without breaking the left-allocated blocks, it is always possible to achieve $\gamma_\delta = \gamma_n = 1$ and $\gamma_k = 0$ for all $k \notin \{\delta, n\}$, without involving any Toffoli gate. Once it is achieved, by applying $CX_{\mathcal{I}:\delta}$, $\mathcal{I} = \{1, 2, \cdots, m-1, n\}$, we have $\boldsymbol{\gamma} = 00 \cdots 01$, which is by definition a block. Blocks located left to i are unaffected by $CX_{\mathcal{I}:\delta}$ gate since the binary strings of their positions have at least one zero among the first $m - 1$ bits. □

Lemma 3.6. *Let i be an iterator used in Algorithm 1, l be the number of left-allocated blocks, and m be the smallest positive integer satisfying Eq. (5). For given permutation with a block constructed by* CONS, *the block can be allocated to the i-th block-wise position by using at most $(n - m)$ CX and one $C^{HW(i')}X$ gates, without breaking left-allocated blocks, where $i' = i - h_n(m-1)/2$ for $m > 1$ and $i' = i$ otherwise.*

Proof. Let $\boldsymbol{j}$ be a binary string of the (block-wise) position j of the block constructed by CONS. Assume $j \neq i$, otherwise, the block is already left-allocated. Let $\boldsymbol{\gamma} = \boldsymbol{i} \oplus \boldsymbol{j}$ and δ be the smallest integer such that $\gamma_\delta = 1$. Due to the property of the column numbers we begin with in CONS, i.e., Eq. (6), the first $m - 2$ bits of $\boldsymbol{i}$ and $\boldsymbol{j}$ are 1 (none if $m \leq 2$),

$$i_1 = j_1 = i_2 = j_2 = \cdots = i_{m-2} = j_{m-2} = 1, \tag{8}$$

and thus $\delta > m - 2$. Since $i < j$ and $\gamma_1 = \gamma_2 = \cdots = \gamma_{\delta-1} = 0$, it is guaranteed that $i_\delta = 0$ and $j_\delta = 1$. Similar to the technique used in Lemma 3.5, $\boldsymbol{j}$ can be transformed such that $\gamma_\delta = 1$ and $\gamma_k = 0$ for all $k \notin \{\delta\}$ by using $CX_{\delta x}$ gates for $\delta < x < n$. Note that unlike in Lemma 3.5 where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ both can be transformed by gates, here only $\boldsymbol{j}$ is allowed to change. Once it is done, by applying $CX_{\mathcal{I}:\delta}$, $\mathcal{I} = \{x \mid x > \delta,\ i_x = 1\}$, the left-allocation is completed. Note that blocks residing in L_σ, $\sigma \in \{x \mid x < \delta,\ i_x = 1\}$ are conserved by an action of a gate whose target is δ-th bit, up to changes of block-wise positions within. In addition, by the definition of L_σ, i cannot be included in any of L_σ. Therefore, $CX_{\mathcal{I}:\delta}$ can be applied without breaking the left-allocated blocks. □

Upper bounds on the number of Toffoli gates in the output circuit naturally follow from two lemmas. Recall that we have adopted the conversion formula $C^mX : C^2X = 1 : 2m - 3$. By counting the worst-case number of Toffoli gates in CONS and ALLOC for all iterations, it can be summarized as follows:

Theorem 3.7. *The number of Toffoli gates in the output R of Algorithm 1 is upper bounded by $\mathcal{N}_c(n) + \mathcal{N}_a(n)$ for $n \geq 3$, where*

$$\mathcal{N}_c(n) = \sum_{i=2}^{n-1} (2i-3) \cdot 2^{n-i},$$
$$\mathcal{N}_a(n) = \sum_{j=2}^{n-2} \sum_{i=2}^{n-j} (2i-3) \cdot \binom{n-j}{i}, \tag{9}$$

where $\mathcal{N}_a(3) = 0$.

The time complexity of Algorithm 1 can also be estimated by the lemmas. *Assuming all gates exchange 2^n column numbers* (Section 6 for more discussion), the time complexity of the algorithm simply reads the total number of gates times 2^n. The maximum number of gates applied in each CONS and ALLOC is $O(n)$, and the number of CONS and ALLOC called in Algorithm 1 is 2^{n-1} each, and thus the asymptotic time complexity reads $O(n2^{2n})$.

4 Algorithm

Key ideas have been introduced in Section 3. The only downside of Algorithm 1 is its inability to decompose arbitrary permutations. This section is therefore mostly devoted to lifting the assumption that an input permutation consists only of row numbers in normal positions. Lifting the assumption costs an extra number of Toffoli gates which is bounded by

$$5 \cdot 2^{n-4} + 2n - 5 + \sum_{i=2}^{n-3} (2i-3) \cdot \binom{n-3}{i}. \tag{10}$$

Lifting the assumption takes several independent steps, which we shall separately deal with in each subsection. Details on various formulas appearing in this section are more or less the same as in the previous section, and thus will mostly be dropped.

4.1 Heuristic Mixing

Throughout Section 4 we will frequently refer to the ratio of the number of normal, inverted, and interrupting positions as the ratio $x : y : z$ such that $x + y + z = 1$. For example, all permutations handled in Section 3.4 have the ratio $1 : 0 : 0$.

The goal of the first step, Heuristic Mixing, is to transform an arbitrary permutation into one with the ratio $x : y : 0.5$, where x and y are arbitrary. The reason for the target ratio is as follows. In the next step, we will use an algorithm to make the output ratio $0.5 : 0.5 : 0$ so that the problem effectively becomes two smaller instances. The algorithm however requires that for its input ratio $x : y : z$, x and y should not be larger than 0.5. One way to meet the requirement is to make $z = 0.5$, thus Heuristic Mixing is applied first.

The method is to apply CX gates a few times until the desired ratio is (nearly) achieved. The following assumption is based on an observation: unstructured or randomly chosen permutations have a ratio close to $x : y : 0.5$.

Assumption 4.1. *There exists a composite gate consisting of at most four CX and one $C^{n-1}X$ gates that transforms a permutation such that the resulting permutation exhibits the ratio $x : y : 0.5$.*

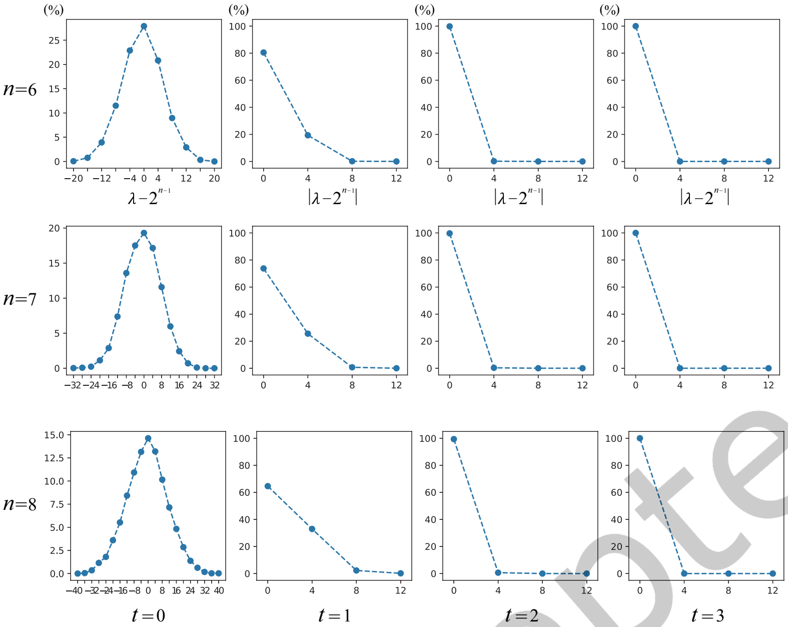


Fig. 4. Sampling tests for $n \in \{6, 7, 8\}$ and $t \in \{0, 1, 2, 3\}$. Each figure shows the percentage of samples as a function of $\lambda - 2^{n-1}$ (or $|\lambda - 2^{n-1}|$), where λ is defined in the main text. In other words, figures show how far the permutations are from the ratio $x : y : 0.5$ (and especially $\lambda - 2^{n-1} = 0$ means the ratio is exactly $x : y : 0.5$). For $t = 2$ where two CX gates have been applied, we were already able to find the desired permutations with high probability. The margin of error is ±1.55% at 95% confidence level.

Here we count $X_i CX_{ij} X_i$, $j \neq i$ (so called negative controlled gates) as CX gate, too. Let us call a composite gate consisting of t CX gates a depth-t composite. For example, counting the meaningful composites, there exist no depth-0 composites, $2n - 2$ depth-1 composites,[5] at most $(2n - 2) \cdot 4\binom{n}{2}$ depth-2 composites, and so on. As an algorithm, we may apply depth-t composites one by one from $t = 0$ to $t = 4$ until the ratio hits $x : y : 0.5$ exactly, or until all the composites are exhausted. When the latter happens, we choose a permutation among the tried ones that are closest to the desired ratio, and then apply one $C^{n-1}X$ gate to meet $x : y : 0.5$.

We have carried out numerical tests on random samples for $t \in \{0, 1, 2, 3\}$, plotting $\lambda - 2^{n-1}$ (or $|\lambda - 2^{n-1}|$), where λ is the number of interrupting positions in a permutation we can get with depth-t composites that is closest to the desired ratio. As the results in Fig. 4 show, applying a few CX composites likely leads to the desired ratio.

4.2 Preprocessing

The output of Heuristic Mixing is a permutation with the ratio $x : y : 0.5$. It is then further processed by an algorithm $\mathcal{A}_{\text{pre}}$ to be the ratio $0.5 : 0.5 : 0$.

The idea is simple and we give an example first. Consider the following 4-bit permutation:

$$(3, 10, 14, 6, 12, 2, 0, 15, 5, 8, 13, 9, 1, 4, 7, 11),$$

[5] The number of interrupting positions can be varied by CX gate if the gate's target is the n-th bit. There exist $2n - 2$ such CX gates.

where eight interrupting positions are highlighted. Move four of them to the left by a sequence of gates CX_{13}, X_1, CX_{12}, CX_{13}, CX_{241}, CX_{32} in order, leading to

$$(13, 9, 1, 10, 7, 6, 5, 8, 0, 15, 3, 4, 14, 11, 12, 2).$$

Applying another sequenceX_1, X_2, CX_{124}, X_1, X_2 results in

$$(9, 13, 10, 1, 7, 6, 5, 8, 0, 15, 3, 4, 14, 11, 12, 2).$$

Here we only sketch the mechanism, rather than elaborating on it. Due to the way the interrupting position is defined, it can only be an even number; (hypothetical) change of a 'single' row number leads to either $-2, +0$, or $+2$ change to the number of interrupting positions. Since at least two row numbers are exchanged by the logic gates, the number of interrupting positions can only vary by multiples of 4. Note that an action of $C^{n-1}X$ gate that swaps two row numbers can change the number of interrupting positions by at most 4. Considering along the line, an action of C^2X gate that swaps 2^{n-2} row numbers can change the number by at most 2^{n-1}. Since the number of interrupting positions of an input permutation is 2^{n-1}, at best a single Toffoli can achieve the desired ratio. However, the best case hardly happens naturally, and we need to manipulate the input before applying the Toffoli gate. It is complicated to give details, but we claim that a procedure similar to repeating CONS and ALLOC one-quarter times of that of $\mathcal{A}'_{\text{red}}$ is enough. In addition to its ability to eliminate interrupting positions, the procedure can also fine-tune the number of normal and inverted positions. In fact, $\mathcal{A}_{\text{pre}}$ can turn the ratio $x : y : 0.5$ into $x \pm \Delta_x : y \pm \Delta_y : 0$ where $0 \leq \Delta_x, \Delta_y \leq 0.5$ as desired. In the following algorithm $\mathcal{A}_{\text{pre}}$, we may simply set the output ratio to be 0.5:0.5:0.

The procedure is described as follows:

Algorithm 2 $\mathcal{A}_{\text{pre}}$

```
Input $P_n$                                              ▷ The ratio is $x : y : 0.5$
Output $(P, R)$
$R \leftarrow (\ )$
$P \leftarrow P_n$
for $i$ from 0 to $2^{n-3} - 1$ do
    $\langle a, b \rangle \leftarrow$ PRE_PICK($P$, $i$)
    $S_C \leftarrow$ CONS($P$, $i$, $\langle a, b \rangle$)
    $(P, R) \leftarrow (P, R) \cdot S_C$
    $S_A \leftarrow$ ALLOC($P$, $i$, $a$)
    $(P, R) \leftarrow (P, R) \cdot S_A$
$(P, R) \leftarrow (P, R) \cdot (X_1, X_2, CX_{12n}, X_2, X_1)$
return $(P, R)$
```

PRE_PICK works in a similar way to PICK, but the output is not a relevant row pair but a certain pair of row numbers at interrupting positions. We will not cover the details on the pair, but with abuse of notation we let $\langle \cdot, \cdot \rangle$ denotes the pair, too. Interested readers may refer to the implemented code [1]. CONS and ALLOC work exactly in the same way as in Section 3.4. The number of Toffoli gates involved in Algorithm 2 is bounded by $3 \cdot 2^{n-4} - 1 + \sum_{i=2}^{n-3} (2i - 3) \cdot \binom{n-3}{i}$.

4.3 Generalized Size Reduction Algorithm

After the mixing and the preprocessing, the permutation has the ratio $0.5 : 0.5 : 0$. Now the problem can be thought of as two subproblems. We will first construct and allocate only *even blocks* out of row numbers that are in normal positions. Since CONS and ALLOC do not require any controlled gate targeting the n-th bit, the number of normal or inverted positions is conserved by Remark 3.1. Once 2^{n-2} even blocks are left-allocated, the remaining row numbers that are only in inverted positions are constructed and allocated to be *odd blocks*. In the end, we would have 2^{n-2} even blocks on the left half and 2^{n-2} odd blocks on the right half. Applying one CX_{1n} gate completes the size reduction.

The size reduction algorithm is described as follows:

Algorithm 3 $\mathcal{A}_{\text{red}}$

```
Input $P_n$                                              ▷ The ratio is $0.5 : 0.5 : 0$
Output $(P, R)$                                          ▷ $P = P_{n-1} \otimes I_2$
$R \leftarrow (\ )$
$P \leftarrow P_n$
for $i$ from $0$ to $2^{n-2} - 1$ do                     ▷ First Part
    $\langle a, b \rangle \leftarrow$ N_PICK$(P, i)$         ▷ Pick a normal pair
    $S_C \leftarrow$ CONS$(P, i, \langle a, b \rangle)$
    $(P, R) \leftarrow (P, R) \cdot S_C$
    $S_A \leftarrow$ ALLOC$(P, i, a)$
    $(P, R) \leftarrow (P, R) \cdot S_A$
for $i$ from $2^{n-2}$ to $2^{n-1} - 1$ do               ▷ Second Part
    $\langle a, b \rangle \leftarrow$ PICK$(P, i)$
    $S_C \leftarrow$ CONS$(P, i, \langle a, b \rangle)$
    $(P, R) \leftarrow (P, R) \cdot S_C$
    $S_A \leftarrow$ ALLOC$(P, i, a)$
    $(P, R) \leftarrow (P, R) \cdot S_A$
$(P, R) \leftarrow (P, R) \cdot (CX_{1n})$
return $(P, R)$
```

The first part deals with row numbers in normal positions to construct and allocate even blocks only. Therefore, N_PICK has to output a relevant row pair in normal positions. Accordingly, the working mechanism of N_PICK is a bit different from that of PICK [1]. Quality bounds given by Proposition 3.4, Lemma 3.5, and Lemma 3.6 are no longer valid in the first part, but overall it only adds 2^{n-3} to the quality bound of $\mathcal{A}'_{\text{red}}$. Details on N_PICK and related quality bound are not discussed here. The second part works exactly the same as Algorithm 1 except now the row numbers are only in inverted positions instead of normal positions.

The additional quality factor, 2^{n-3}, can hardly be met in average instances, and in practice the quality difference between outputs of $\mathcal{A}'_{\text{red}}$ and $\mathcal{A}_{\text{red}}$ is negligible. Note that the point of mixing and preprocessing is to transform an arbitrary permutation so that the result has an appropriate form for the decomposition. If a given permutation is already well-suited for $\mathcal{A}'_{\text{red}}$, or $\mathcal{A}_{\text{red}}$, or something similar, mixing and preprocessing can be skipped.

4.4 Algorithm for Synthesis

Combining Heuristic Mixing, Preprocessing, and $\mathcal{A}_{\text{red}}$, one can easily come up with an algorithm for reversible logic circuit synthesis as follows:

Algorithm 4 $\mathcal{A}_{\text{syn}}$

```
    Input $P_n$
    Output $R$
$R \leftarrow ()$
for $i$ from $n$ to 3 do
    $(P_i, R_{\text{mix}}) \leftarrow \mathcal{A}_{\text{mix}}(P_i)$                ▷ Heuristic Mixing
    $(P_i, R_{\text{pre}}) \leftarrow \mathcal{A}_{\text{pre}}(P_i)$
    $(P_i, R_{\text{red}}) \leftarrow \mathcal{A}_{\text{red}}(P_i)$
    $P_{i-1} \leftarrow$ REDUCE$(P_i)$                                ▷ $P_i = P_{i-1} \otimes I_2$
    $R \leftarrow R;R_{\text{mix}};R_{\text{pre}};R_{\text{red}}$                      ▷ Append all
$R \leftarrow R;$SEARCH$(P_2)$                                ▷ Exhaustive search for $P_2$
return $R$
```

Let the quality bound given by Theorem 3.7 be $A(n)$ and the cost Eq. (10) be $B(n)$ for an n-bit permutation. The number of Toffoli gates required for the size reduction $P_n \mapsto P_{n-1}$ is upper bounded by

$$A(n) + B(n). \tag{11}$$

Therefore the quality bound on the output R of Algorithm 4 is given by $\sum_{x=n}^{3} (A(x) + B(x))$.

Time complexity of $\mathcal{A}_{\text{mix}}$ can be estimated by counting the number of composites times 2^n, which reads $O(n^7 2^n)$. Preprocessing roughly does one-quarter of what the size reduction performs, thus the time complexity is $O(n2^{2n-2})$. Time complexities of $\mathcal{A}'_{\text{red}}$ and $\mathcal{A}_{\text{red}}$ are the same. In $\mathcal{A}_{\text{syn}}$, the first size reduction step dominates the overall running time because each time the effective size gets smaller, time to transform the permutation becomes exponentially faster. The time complexity of size reduction steps in $\mathcal{A}_{\text{syn}}$ is dominated by $\mathcal{A}_{\text{red}}$ with $O(n2^{2n})$, thus the time complexity of $\mathcal{A}_{\text{syn}}$ is $O(n2^{2n})$.

5 Benchmark and Application

Algorithm 4 can be considered as a very basic algorithm that makes use of the size reduction idea. Indeed we have focused on simplifying the algorithm so that the design criteria are as transparently brought out as possible. When it comes to optimizations we have noticed and indeed gone through a number of directions, for example exploiting partial search or statistical properties or undoing and retrying, but then what we confronted at the end was way too complicated descriptions for the algorithm. Most optimization options we have considered earlier thus have been dropped at the end, not only from the paper but also from the source code so that the code can be a natural reflection of the descriptions given in Sections 3 and 4.

Nevertheless, one optimization option is still implemented in our code. Briefly explained, as shown in Subroutines CONS and ALLOC, there is a possibility that a block is already existing at the iteration. We call such cases free constructions or free allocations, or simply free blocks without distinction. To utilize free blocks, let us review how the algorithm works.

In i-th iteration in Algorithm 3, N_PICK or PICK chooses one relevant row pair and it is processed to be a left-allocated block. The resulting permutation may or may not have free blocks in $(i + 1)$-th iteration. Notice at this point that instead of specifying only one relevant row pair for the block, we may try all possible relevant row pairs and rank them based on certain criteria, for example by the number of free blocks upon the left-allocation of the candidate pair (block).[6] Let us call it depth-1 partial search at i-th iteration. Fig. 5 shows the number of Toffoli gates resulting from the size reduction of 1000 randomly generated permutations for $n = 4, 5, 6, 7, 8$ employing depth-0 and depth-1 partial search for all blocks, together with the upper bound for the reference.

[6] In fact, it is much more complicated than just counting the number of free blocks. Finding a sophisticated cost metric for evaluating free blocks is already a nontrivial task for optimization.

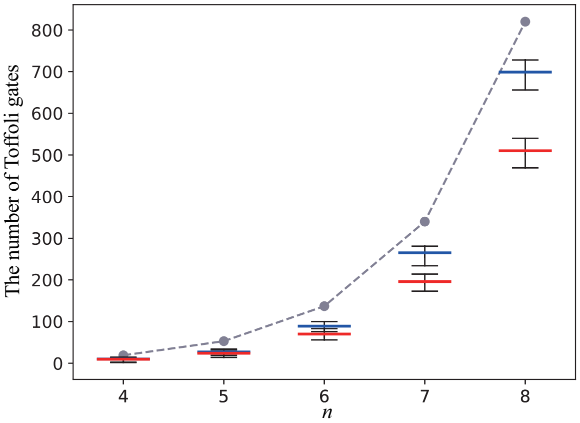


Fig. 5. Quality bounds on size reduction of n-bit permutations (Eq. (11)) joined by dashed lines, and the statistical averages of the output qualities denoted by the red and blue bars for 1000 randomly generated instances for each n with depth-0 (blue) and depth-1 (red) partial search. Min-Max values are also marked with black bars.

If computational resources allow, one may try searching for a better pair by considering free blocks upon the construction and allocation of blocks not only at the i-th block-wise position but also up to even more later blocks.

To be more specific, define $d(i)$ a search depth at position i. For $d(i) = 0$, the partial search option is off. For $d(i) = 1$, we do depth-1 partial search at i as described earlier. For $d(i) > 1$, we try all the possible combinations of the blocks up to $i + d(i) - 1$ block-wise position and pick the best one, and call it depth-$d(i)$ partial search at i. For example, setting $d(i) = 2$ for all $i \in \{0, 1, \ldots, 2^{n-1} - 2\}$ is to carry out depth-2 partial search in constructing and allocating each block. The quality of the output circuit gets better as $d(i)$ gets larger, but the time complexity becomes worse. Roughly estimated, for a constant $d(i) = d$ for all i, the time complexity reads $O\left(n2^{(2+d)n}\right)$.

Since the number of remaining relevant row numbers decreases as the iteration goes on, it becomes even easier to do the search for larger search depth in later iterations. Let $r(i)$ denote the number of remaining relevant row numbers in normal positions at i-th iteration. Let d_j be a search depth at i-th iteration such that $2^{j-1} < r(i) \leq 2^j$. In constructing and allocating even blocks, we may apply depth-d_j partial search. In the early stages, one should set small d_j and let $d(i) = d_j$ so that the searching is tractable, but in later stages of left-allocating even blocks, d_j can be larger. Similar search depth can be defined for odd blocks, and we use the same symbol d_j for either case. In our code, we basically set $d(i) = d_j$ and let users control d_j as external parameters for each $j \in \{1, \ldots, n\}$, except $d(i)$ for $i \geq 2^{n-1} - 9$.[7] When d_j is set as a constant d for all j, we simply say depth-d partial search is applied.

Reversible logic circuits for known functions have been synthesized by the algorithm, which are summarized and compared with the previous results when available in Table 1. Depth-2 partial search option is applied to the functions. Benchmark results with different d_j options are to be posted on [1].[8] Notice that the algorithm works

[7] It is set such that the last few blocks are exhaustively searched, which has a nontrivial effect on the output quality considering the statistical properties and Lemma 3.5. Especially for even permutations, output circuits contain either zero or two $C^{n-1}X$ gates. The setting likely eliminates the chance of the latter taking place. About the statistical properties, for example, it can be shown that the last four blocks cost the *minimal* number of Toffoli gates if no free block exists upon the construction and allocation of a block in $(2^{n-1} - 5)$-th iteration. It seems many iteration stages can utilize such statistical properties.

[8] To get a sense of the running time, on a commercial laptop, eight-bit permutation URF2 takes 6s, 16s, 228s for the depth-0, 1, 2 partial search, respectively. Better runtime data can be found in [1].

Table 1. Synthesis results for various structured and unstructured functions with depth-2 partial search. In subcolumns, #in, #out, #grb read the number of input, output, and garbage bits, respectively. QC stands for Quantum Cost which is one of the widely accepted cost metrics [6, 22], and #TOF is the number of Toffoli gates. Most previous results have not reported #TOF, some of which we were unable to retrieve ourselves. Ratio columns show the ratio of this work to the previous result in each respective metric. DES, Skipjack, and KHAZAD are S-box functions used in cryptography. For DES, the values are averaged since there are eight different S-boxes.

	Functions	Previous work					This work					Ratio	
		#in	#out	#grb	QC	#TOF	#in	#out	#grb	QC	#TOF	QC	#TOF
Unstructured	URF1	9	9	0	17002 [62]	2225	9	9	0	13318	2029	78%	91%
	URF2	8	8	0	7083 [62]	892	8	8	0	5510	803	78%	90%
	URF3	10	10	0	37517 [62]	4997	10	10	0	33541	4898	89%	98%
	URF4	11	11	0	160020 [58]	32004	11	11	0	77616	11706	49%	37%
	URF5	9	9	0	14549 [62]	1964	9	9	0	9863	1366	68%	70%
	*n*thPrime7	7	7	0	2284 [73]	334	7	7	0	1887	281	83%	84%
	*n*thPrime8	8	8	0	6339 [73]	930	8	8	0	5165	691	81%	74%
	*n*thPrime9	10	9	1	17975 [62]	2392	9	9	0	12189	1762	68%	74%
	*n*thPrime10	11	10	1	40299 [62]	5302	10	10	0	28630	4003	71%	76%
	*n*thPrime11	12	11	1	95431 [62]	12579	11	11	0	63314	9269	66%	74%
	DES	-	-	-	-	-	6	4	2	743.88	102.63	-	-
	Skipjack	-	-	-	-	-	8	8	0	5562	791	-	-
	KHAZAD	-	-	-	-	-	8	8	0	5411	794	-	-
Structured	HWB7	7	7	0	*poly(n)* [15]	249	7	7	0	1979	288	-	-
	HWB8	8	8	0	*poly(n)* [15]	586	8	8	0	5439	788	-	-
	HWB9	9	9	0	*poly(n)* [15]	1853	9	9	0	13333	2010	-	-
	HWB10	10	10	0	*poly(n)* [15]	3411	10	10	0	33120	4885	-	-
	HWB11	11	11	0	*poly(n)* [15]	8536	11	11	0	75961	11497	-	-
	aj-e	4	4	0	30 [57]	5	4	4	0	108	12	360%	240%
	graycode6	6	6	0	5 [71]	0	6	6	0	13	0	260%	-
	ham3	3	3	0	9 [71]	1	3	3	0	18	1	200%	100%
	ham7	7	7	0	49 [42]	6	7	7	0	325	22	663%	367%
	mod5mils	5	5	0	13 [71]	6 [19]	5	5	0	94	8	723%	133%
	cycle10	12	12	0	1198 [44]	98	12	12	0	78448	11905	6548%	130%
	plus127mod2^{13}	13	13	0	35348 [46]	-	13	13	0	13669	609	39%	-
	plus63mod2^{12}	12	12	0	14652 [46]	-	12	12	0	8771	536	60%	-
	plus63mod2^{13}	13	13	0	19566 [46]	-	13	13	0	18517	1177	95%	-

well only for the unstructured functions compared with the previous constructions. About the hidden weighted bit (HWB) function, it has recently been shown by [15] that the circuit of size $O(n^{6.42})$ can be synthesized by using the reversible gates, or the circuit of size $O(n^2)$ can be constructed by using quantum unitary gates. We were unable to generate the concrete circuit of [15] ourselves for the comparison, but since their circuit is polynomial in n in size, ours should be less efficient.

The algorithm is applied to find reversible circuits for cryptographic S-boxes. Among various S-boxes, one that has attracted the most attention from the research community is undoubtedly AES S-box [27, 34, 39, 76]. Table 1 in Jaques et al.'s work [34] neatly summarizes a few notable circuit designs from the literature, and we add one

Table 2. Comparison of different circuit designs for AES S-box given in [34] and a circuit obtained by the algorithm with depth-4 partial search. The third to the sixth columns read the number of CNOT gates, single qubit Clifford gates, T gates, and measurements. TD and W mean T-depth and the number of logical qubits, respectively.

Source	Type	#CX	#1qC	#T	#M	TD	W
[27]	out-of-place	8683	1023	3854	0	217	44
[39]	out-of-place	818	264	164	41	35	41
[34]	out-of-place	654	184	136	34	6	137
This work	in-place	12466	1787	5089	0	579	15

more entry to it as in Table 2 with the same Q# options being used as the first entry. The design we have added in the table is first obtained by the algorithm proposed, and then it went through manual optimizations [8, 30] before Q# estimates the resources.[9] A detailed circuit can be found in [1]. Note that we could get an even smaller T-depth at the cost of increasing the number of measurements, but we post the one without any measurements. Here we would point out that our design works *in-place* [23], meaning that it does not need extra bits for writing the outputs. In-place design of the S-box likely leads to a quantum oracle with a minimal number of qubits, but we leave the task of constructing an oracle as one of the future directions to explore. Note however that in Grover's algorithm it is generally more important to optimize depth than width considering the quadratic speedup [28, 37, 74].

Efficient reversible circuits can be found for S-boxes with algebraic structures, but if an S-box does not have an apparent structure, Algorithm 4 can be a good candidate for a solution. We have applied the algorithm to Skipjack [50], KHAZAD [10], and DES [48] S-boxes, all of which are known not to have efficient classical implementations. The results are summarized in Table 1, and the circuits can be found from [1]. For DES (which is an abbreviation for data encryption standard), there are eight different '6-bit input, 4-bit output' S-boxes, and in Table 1 the average values are filled. Details are given in Appendix II. To our knowledge, none of them have been analyzed before possibly due to the difficulty of handling unstructured permutation maps. Note that the algorithm is generally applicable to any substitution maps with not too large n, giving rise to in-place circuits.

Comparisons with S-boxes that have been previously studied are also given in Appendix II. These S-boxes are either structured or small enough for near optimal synthesis.

6 Summary and Discussion

In summary, we have developed an algorithm for reversible circuit synthesis based on a transformation-based approach. The key idea is, in each iteration, the problem size is reduced such that *one bit is completely ruled out* from the remaining process. In doing so, our primary concern is to minimize the number of Toffoli gates which are treated as the universal gate in quantum computing community in a similar sense to what NAND gate means in classical computing. Compared with existing algorithms, we have observed that the new algorithm shows good performance in the number of Toffoli gates or QC cost for the functions that are believed to be hard to synthesize, but it is not efficient when the function is structured as expected.

About the time complexity, the algorithm runs in time $O(n2^{2n})$ which is obviously flawed at least by a factor of 2^n even compared with an elementary algorithm such as transforming sequences $(*, *, *, \dots) \mapsto (1, *, *, \dots) \mapsto (1, 2, *, \dots) \mapsto \cdots$ by consecutive transpositions. Although the efficiency in time has not been our concern, it would be a fair question to ask if the same algorithm can run in time $O(n2^n)$. In fact in our analysis, the culprit

[9] The reason for the manual optimization is that the output circuit of the proposed algorithm consists of MCT gates, whereas all the other entries are written in terms of smaller gates such as T. Therefore, we first manually transform MCT gates into NCT gates, and then Q# is applied.

for the gap is the action of a gate which takes $O(2^n)$ time in our current implementation. Once a subroutine such as CONS gives a gate sequence, we do not know the resulting permutation until the *gates are actually applied*. In comparison with the elementary algorithm, applying transposition immediately gives the resulting permutation, allowing only to count the number of required transpositions to estimate the complexity. In fact, since the considered elementary gates act only locally, in principle it is possible to bring down its complexity to $\tilde{O}(n2^n)$. If we are to improve the time complexity, an efficient gate action should be considered such as using tensor network techniques [64].

References

[1] 2021. Algorithm implementation. (2021). https://github.com/ReversibleLogicCircuit/SizeReduction

[2] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Phys. Rev. A* 70 (Nov 2004), 052328. Issue 5.

[3] A. Agrawal and N.K. Jha. 2004. Synthesis of reversible logic. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Vol. 2. 1384–1385 Vol.2.

[4] Mishal Almazrooie, Azman Samsudin, Rosni Abdullah, and Kussay N. Mutter. 2018. Quantum Reversible Circuit of AES-128. *Quantum Information Processing* 17, 5 (28 Mar 2018), 112.

[5] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 2013. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 6 (2013), 818–830.

[6] Mona Arabzadeh and Mehdi Saeedi. 2013. RCViewer+, version 2.5, 2013. (2013).

[7] Pascal Aubry, Sergiu Carpov, and Renaud Sirdey. 2020. Faster Homomorphic Encryption is not Enough: Improved Heuristic for Multiplicative Depth Minimization of Boolean Circuits. In *Topics in Cryptology – CT-RSA 2020*, Stanislaw Jarecki (Ed.). Springer International Publishing, Cham, 345–363.

[8] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Phys. Rev. A* 52 (1995), 3457–3467. Issue 5.

[9] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Phys. Rev. A* 52 (Nov 1995), 3457–3467. Issue 5.

[10] PSLM Barreto and Vincent Rijmen. 2000. The khazad legacy-level block cipher. *Primitive submitted to NESSIE* 97, 106 (2000).

[11] Daniel J Bernstein. 2009. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete. *Workshop Record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic systems* 9 (2009), 105.

[12] Subodh Bijwe, Amit Kumar Chauhan, and Somitra Kumar Sanadhya. 2020. Quantum Search for Lightweight Block Ciphers: GIFT, SKINNY, SATURNIN. Cryptology ePrint Archive, Report 2020/1485. https://eprint.iacr.org/2020/1485.

[13] Xavier Bonnetain, María Naya-Plasencia, and André Schrottenloher. [n. d.]. Quantum Security Analysis of AES. ([n. d.]).

[14] Sergey Bravyi, Joseph A. Latone, and Dmitri Maslov. 2022. 6-qubit optimal Clifford circuits. *npj Quantum Information* 8, 1 (05 Jul 2022), 79.

[15] S. Bravyi, T. Yoder, and D. Maslov. 2021. Efficient ancilla-free reversible and quantum circuits for the Hidden Weighted Bit function. *IEEE Trans. Comput.* 71, 01 (apr 2021), 1170–1180.

[16] R.E. Bryant. 1991. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Comput.* 40, 2 (1991), 205–213.

[17] Amit Kumar Chauhan and Somitra Kumar Sanadhya. 2020. Quantum resource estimates of Grover's key search on ARIA. In *Security, Privacy, and Applied Cryptography Engineering*, Lejla Batina, Stjepan Picek, and Mainack Mondal (Eds.). Springer International Publishing, Cham, 238–258.

[18] Matthew Chun, Anubhab Baksi, and Anupam Chattopadhyay. 2023. DORCIS: Depth Optimized Quantum Implementation of Substitution Boxes. *IACR Cryptol. ePrint Arch.* (2023), 286. https://eprint.iacr.org/2023/286

[19] Edinelço Dalcumune, Luis Antonio Brasil Kowada, André da Cunha Ribeiro, Celina Miraglia Herrera de Figueiredo, and Franklin de Lima Marquezino. 2021. A reversible circuit synthesis algorithm with progressive increase of controls in generalized Toffoli gates. *J. Univers. Comput. Sci.* 27, 6 (2021), 544–563.

[20] William James Dally and Brian Patrick Towles. 2004. *Principles and practices of interconnection networks.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[21] Vishnu Asutosh Dasu, Anubhab Baksi, Sumanta Sarkar, and Anupam Chattopadhyay. 2019. LIGHTER-R: Optimized Reversible Circuit Implementation For SBoxes. In *32nd IEEE International System-on-Chip Conference, SOCC 2019, Singapore, September 3-6, 2019.* IEEE, 260–265. https://doi.org/10.1109/SOCC46988.2019.1570548320

[22] Maslov Dmitri. 2009. Reversible Logic Synthesis Benchmarks Page. (2009). https://reversiblebenchmarks.github.io/

[23] Thomas G. Draper, Samuel A. Kutin, Eric M. Rains, and Krysta M. Svore. 2006. A logarithmic-depth quantum carry-lookahead adder. *Quantum Info. Comput.* 6, 4 (July 2006), 351–369.

[24] K. Fazel, M. A. Thornton, and J. E. Rice. 2007. ESOP-based Toffoli Gate Cascade Generation. In *2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*. 206–209.

[25] O. Golubitsky, S. M. Falconer, and D. Maslov. 2010. Synthesis of the optimal 4-bit reversible circuits. In *Design Automation Conference*. 653–656.

[26] Oleg Golubitsky and Dmitri Maslov. 2012. A Study of Optimal 4-Bit Reversible Toffoli Circuits and Their Synthesis. *IEEE Trans. Comput.* 61, 9 (2012), 1341–1353.

[27] Markus Grassl, Brandon Langenberg, Martin Roetteler, and Rainer Steinwandt. 2016. Applying Grover's algorithm to AES: quantum resource estimates. In *PQCrypto 2016*. 29–43.

[28] Lov K. Grover. 1997. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.* 79 (1997), 325–328. Issue 2.

[29] P. Gupta, A. Agrawal, and N.K. Jha. 2006. An Algorithm for Synthesis of Reversible Logic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 11 (2006), 2317–2330.

[30] Yong He, Ming-Xing Luo, E. Zhang, Hong-Ke Wang, and Xiao-Feng Wang. 2017. Decompositions of n-qubit Toffoli gates with linear circuit complexity. *International Journal of Theoretical Physics* 56, 7 (01 Jul 2017), 2350–2361.

[31] Kyungbae Jang, Anubhab Baksi, Jakub Breier, Hwajeong Seo, and Anupam Chattopadhyay. 2023. Quantum Implementation and Analysis of DEFAULT. *Cryptography and Communications* (2023), 1–17.

[32] Kyoungbae Jang, Hyunjun Kim, Siwoo Eum, and Hwajeong Seo. 2020. Grover on GIFT. Cryptology ePrint Archive, Report 2020/1405. https://eprint.iacr.org/2020/1405.

[33] Kyungbae Jang, Gyeongju Song, Hyeokdong Kwon, Siwoo Uhm, Hyunji Kim, Wai-Kong Lee, and Hwajeong Seo. 2021. Grover on PIPO. *Electronics* 10, 10 (2021), 1194.

[34] Samuel Jaques, Michael Naehrig, Martin Roetteler, and Fernando Virdia. 2020. Implementing Grover oracles for quantum key search on AES and LowMC. In *Advances in Cryptology – EUROCRYPT 2020*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 280–310.

[35] Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. 2016. Breaking Symmetric Cryptosystems Using Quantum Period Finding. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9815)*, Matthew Robshaw and Jonathan Katz (Eds.). Springer, 207–237. https://doi.org/10.1007/978-3-662-53008-5_8

[36] Pawel Kerntopf. 2004. A New Heuristic Algorithm for Reversible Logic Synthesis. In *Proceedings of the 41st Annual Design Automation Conference* (San Diego, CA, USA) *(DAC '04)*. Association for Computing Machinery, New York, NY, USA, 834–837.

[37] Panjin Kim, Daewan Han, and Kyung Chul Jeong. 2018. Time-space complexity of quantum search algorithms in symmetric cryptanalysis: applying to AES and SHA-2. *Quantum Information Processing* 17, 12 (Oct 2018), 339.

[38] Vadym Kliuchnikov and Dmitri Maslov. 2013. Optimization of Clifford circuits. *Phys. Rev. A* 88 (Nov 2013), 052307. Issue 5.

[39] B. Langenberg, H. Pham, and R. Steinwandt. 2020. Reducing the cost of implementing the advanced encryption standard as a quantum circuit. *IEEE Transactions on Quantum Engineering* 1 (2020), 1–12.

[40] Z. Li, H. Chen, B. Xu, X. Song, and X. Xue. 2008. An algorithm for synthesis of optimal 3-qubit reversible circuits based on bit operation. In *2008 Second International Conference on Genetic and Evolutionary Computing*. 455–458.

[41] Da Lin, Zejun Xiang, Runqing Xu, Xiangyong Zeng, and Shasha Zhang. 2023. Quantum circuit implementations of SM4 block cipher based on different gate sets. *Quantum Inf. Process.* 22, 7 (2023), 282. https://doi.org/10.1007/S11128-023-04002-4

[42] Dmitri Maslov, Gerhard W. Dueck, and D. Michael Miller. 2007. Techniques for the synthesis of reversible Toffoli networks. *ACM Trans. Design Autom. Electr. Syst.* 12, 4 (2007), 42.

[43] D.M. Miller and M.A. Thornton. 2006. QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits. In *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*. 30–30.

[44] D Michael Miller, Gerhard W Dueck, and Dmitri Maslov. 2004. A synthesis method for MVL reversible logic [multiple value logic]. In *Proceedings. 34th international symposium on multiple-valued logic*. IEEE, 74–80.

[45] D. Michael Miller, Dmitri Maslov, and Gerhard W. Dueck. 2003. A transformation based algorithm for reversible logic synthesis. In *Proceedings of the 40th Annual Design Automation Conference* (Anaheim, CA, USA) *(DAC '03)*. Association for Computing Machinery, New York, NY, USA, 318–323.

[46] D. Michael Miller and Zahra Sasanian. 2012. Recent Developments on Mapping Reversible Circuits to Quantum Gate Libraries. In *International Symposium on Electronic System Design, ISEDs 2012, Kolkata, India, December 19-22, 2012*. IEEE, 17–22.

[47] Mikko Möttönen, Juha J. Vartiainen, Ville Bergholm, and Martti M. Salomaa. 2004. Quantum circuits for general multiqubit gates. *Phys. Rev. Lett.* 93 (Sep 2004), 130502. Issue 13.

[48] National Bureau of Standards. 1977. Data Encryption Standard. *Fips publication* 46 (1977).

[49] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum computation and quantum information: 10th anniversary edition*. Cambridge University Press. https://doi.org/10.1017/CBO9780511976667

[50] NIST. 1998. Skipjack and KEA algorithm specifications, version 2.0. (29 May 1998).

[51] NIST. 2001. *Advanced Encryption Standard*. FIPS PUB 197.

[52] Yujin Oh, Kyungbae Jang, Anubhab Baksi, and Hwajeong Seo. 2023. Depth-Optimized Implementation of ASCON Quantum Circuit. Cryptology ePrint Archive, Paper 2023/1030. https://eprint.iacr.org/2023/1030.

[53] Ketan N. Patel, Igor L. Markov, and John P. Hayes. 2008. Optimal synthesis of linear reversible circuits. *Quantum Info. Comput.* 8, 3 (mar 2008), 282–294.

[54] Luca Phab, Stéphane Louise, and Renaud Sirdey. 2022. A First Attempt at Cryptanalyzing a (Toy) Block Cipher by Means of QAOA. In *Computational Science - ICCS 2022 - 22nd International Conference, London, UK, June 21-23, 2022, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 13353)*, Derek Groen, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer, 218–232. https://doi.org/10.1007/978-3-031-08760-8_19

[55] Aditya K. Prasad, Vivek V. Shende, Igor L. Markov, John P. Hayes, and Ketan N. Patel. 2006. Data Structures and Algorithms for Simplifying Reversible Circuits. *J. Emerg. Technol. Comput. Syst.* 2, 4 (oct 2006), 277–293.

[56] Mostafizar Rahman and Goutam Paul. 2021. Grover on Present: Quantum Resource Estimation. Cryptology ePrint Archive, Report 2021/1655. https://eprint.iacr.org/2021/1655.

[57] RevLib. 2008. An Online Resources for Reversible Functions and Circuits. (2008).

[58] M. Saeedi. 2008. QDA reversible benchmarks. http://ceit.aut.ac.ir/qda/benchmarks.htm.

[59] Mehdi Saeedi, Mona Arabzadeh, Morteza Saheb Zamani, and Mehdi Sedighi. 2011. Block-based quantum-logic synthesis. *Quantum Info. Comput.* 11, 3 (March 2011), 262–277.

[60] Mehdi Saeedi and Igor L. Markov. 2013. Synthesis and Optimization of Reversible Circuits—a Survey, Vol. 45. Association for Computing Machinery, New York, NY, USA, Article 21, 34 pages.

[61] Mehdi Saeedi, Morteza Saheb Zamani, Mehdi Sedighi, and Zahra Sasanian. 2010. Reversible circuit synthesis using a cycle-based approach. *J. Emerg. Technol. Comput. Syst.* 6, 4, Article 13 (Dec. 2010), 26 pages.

[62] Saeedi, Mehdi and Zamani, Morteza Saheb and Sedighi, Mehdi and Sasanian, Zahra. 2010. Synthesis of reversible circuit using cycle-based approach. *J. Emerg. Technol. Comput. Syst* 6, 4 (2010), 13.

[63] Tsutomu Sasao. 1993. *Logic synthesis and optimization*. Vol. 2. Springer.

[64] Ulrich Schollwöck. 2011. The density-matrix renormalization group in the age of matrix product states. *Annals of Physics* 326, 1 (2011), 96 – 192. January 2011 Special Issue.

[65] A. Shafaei, M. Saeedi, and M. Pedram. 2013. Reversible logic synthesis of k-input, m-output lookup tables. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1235–1240.

[66] V.V. Shende, A.K. Prasad, I.L. Markov, and J.P. Hayes. 2003. Synthesis of reversible logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22, 6 (2003), 710–722.

[67] Gyeongju Song, Kyungbae Jang, Hyunjun Kim, Siwoo Eum, Minjoo Sim, Hyunji Kim, Wai-Kong Lee, and Hwajeong Seo. 2021. Grover on SPEEDY. Cryptology ePrint Archive, Report 2021/1211. https://eprint.iacr.org/2021/1211.

[68] Tommaso Toffoli. 1980. Reversible computing. In *Automata, Languages and Programming*, Jaco de Bakker and Jan van Leeuwen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 632–644.

[69] George F. Viamontes, Igor L. Markov, and John P. Hayes. 2009. *Quantum Circuit Simulation*. Springer Publishing Company. https://doi.org/10.1017/978-90-481-3065-8

[70] Alexis De Vos and Yvan Van Rentergem. 2008. Young subgroups for reversible computers. *Adv. Math. Commun.* 2, 2 (2008), 183–200.

[71] Robert Wille, Mathias Soeken, Nils Przigoda, and Rolf Drechsler. 2012. Exact Synthesis of Toffoli Gate Circuits with Negative Control Lines. In *42nd IEEE International Symposium on Multiple-Valued Logic, ISMVL 2012, Victoria, BC, Canada, May 14-16, 2012*, D. Michael Miller and Vincent C. Gaudet (Eds.). IEEE Computer Society, 69–74.

[72] Guowu Yang, Xiaoyu Song, William N. N. Hung, and Marek A. Perkowski. 2007. Bi-directional synthesis of 4-bit reversible circuits. *Comput. J.* 51, 2 (07 2007), 207–215. arXiv:https://academic.oup.com/comjnl/article-pdf/51/2/207/995876/bxm042.pdf

[73] Dmitry V. Zakablukov. 2016. Application of Permutation Group Theory in Reversible Logic Synthesis. In *Reversible Computation*, Simon Devitt and Ivan Lanese (Eds.). Springer International Publishing, Cham, 223–238.

[74] Christof Zalka. 2000. Using Grover's quantum algorithm for searching actual databases. *Phys. Rev. A* 62 (Oct 2000), 052305. Issue 5.

[75] Jun Zhang, Jiri Vala, Shankar Sastry, and K. Birgitta Whaley. 2004. Optimal quantum circuit synthesis from controlled-unitary gates. *Phys. Rev. A* 69 (Apr 2004), 042309. Issue 4.

[76] Jian Zou, Zihao Wei, Siwei Sun, Ximeng Liu, and Wenling Wu. 2020. Quantum circuit implementations of AES with fewer qubits. In *Advances in Cryptology – ASIACRYPT 2020*, Shiho Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 697–726.

A An Example for PICK, CONS and ALLOC

Below we show a 4-bit example of how the subroutines work. To make it simple, the permutation consists only of row numbers in normal positions. Consider the following permutation:

$$(0, 1, 2, 11, 12, 3, 10, 5, 4, 15, 14, \boxed{7}, \boxed{6}, 9, 8, 13).$$

First, PICK picks an eligible row pair according to Proposition 3.4. According to the proposition, there exists at least one pair of relevant row numbers located in the right half. In this example, 6 and 7 are the numbers (not the only eligible pair, but the pair with the smallest row numbers is picked for simplicity). Notice that 6 and 7 do not form a block yet, as the two numbers occupy different block-wise positions. Next, the pair is passed on to CONS for construction. The construction rule follows the proof of Lemma 3.5. The number of block already built is 1 thus we have $l = 1$ (Fig. 2). We want to put a new block at $i = 1$ block-wise position, and thus we have $m = 2$ (Eq. (5)). The proof of Lemma 3.5 describes how to construct a block with only one C^mX gate which is in this case C^2X gate. Indeed, CX_{23}, X_4, CX_{142}, X_4 gates are applied in order leading to,

$$(0, 1, 2, 11, 10, 5, 12, 3, 8, 15, \boxed{6,7}, 4, 13, 14, 9).$$

Now, the block is passed on to ALLOC for the left-allocation. The rule is as described in the proof of Lemma 3.6. As mentioned earlier, we want to put a block at $i = 1$ and we have $m = 2$. Since $i' = i - h_n(m-1)/2 = 1$ in Lemma 3.6, i' as a bit string is 0001, and therefore its Hamming weight is 1, telling us that the allocation of the constructed block does not require Toffoli gate (since $C^{HW(i')}X$ gate is only a CNOT gate now). Indeed, CX_{13} gate is applied leading to,

$$(\boxed{0,1}, \boxed{6,7}, 10, 5, 14, 9, 8, 15, 2, 11, 4, 13, 12, 3).$$

Repeating the procedure, the others blocks are constructed and allocated as follows:

$$\begin{aligned}
&(\boxed{0,1}, \boxed{6,7}, \boxed{10,11}, 14, 15, 12, 3, 4, 13, 2, 5, 8, 9)\\
\mapsto&(\boxed{0,1}, \boxed{6,7}, \boxed{10,11}, \boxed{14,15}, 12, 3, 4, 13, 2, 5, 8, 9)\\
\mapsto&(\boxed{10,11}, \boxed{14,15}, \boxed{0,1}, \boxed{6,7}, \boxed{2,3}, 8, 13, 12, 5, 4, 9)\\
\mapsto&(\boxed{10,11}, \boxed{0,1}, \boxed{6,7}, \boxed{14,15}, \boxed{2,3}, \boxed{4,5}, 12, 9, 8, 13)\\
\mapsto&(\boxed{0,1}, \boxed{10,11}, \boxed{14,15}, \boxed{6,7}, \boxed{4,5}, \boxed{2,3}, \boxed{8,9}, \boxed{12,13})
\end{aligned}$$

The resulting permutation can be understood as a three-bit permutation (0,5,7,3,2,1,4,6). The algorithms $\mathcal{A}_{\text{mix}}$ and $\mathcal{A}_{\text{pre}}$ transform it into (3,2,1,7,4,5,6,0) and repeating the procedures,

$$\begin{aligned}
&(\boxed{4,5}, 6, 0, 3, 2, 1, 7)\\
\mapsto &(\boxed{4,5}, \boxed{6,7}, 3, 2, 1, 0)\\
\mapsto &(\boxed{6,7}, \boxed{4,5}, \boxed{1,0}, \boxed{3,2})\\
\overset{CX_{12}}{\mapsto} &(\boxed{6,7}, \boxed{4,5}, \boxed{0,1}, \boxed{2,3}).
\end{aligned}$$

Again it can be read as a smaller permutation $(3, 2, 0, 1)$. Applying X_1 followed by CX_{12} completes the synthesis.

B Application to cryptography

Various aspects of the proposed algorithm applied to quantum cryptography is discussed in this section.

B.1 Discussion

In quantum cryptography, especially in cryptanalysis, one aspect in recent trend is that researchers produce their result in terms of the exact circuit, not in the asymptotic form. In many cases this entails generating a quantum circuit for the target cipher itself, for example a number of papers that optimize the quantum implementation of AES algorithm can be found in the literature [4, 13, 27, 34, 39, 76]. The proposed algorithm can be an option for such tasks, especially when the nonlinear part of the cipher is to be synthesized since the linear part is known to be buildable without needs for non-Clifford gates nor extra qubits.

There could be various ways to make use of the proposed algorithm, but for now we can think of two main use cases and one promising direction. The first one is that the attack cost should be minimized for the space. One caveat is that this case may not be the usual scenario in Grover where reducing the space requirement does not 'proportionally' increase the time. See, [11] for the details. Therefore this usage is better be avoided in Grover attacks unless other options are limited. Instead, one may look for the cryptanalysis by using Simon or quantum approximate optimization algorithms [35, 54]. As such algorithms have studied relatively less than Grover attacks, a study of cryptanalysis by using the proposed algorithm may require an independent project.

The other case the algorithm can be useful is where the nonlinear part of the cipher is hard to quantumly implement. In less scientific terms, we believe if a nonlinear part is over 5-bit size 'and' if the part does not have efficient algebraic structures (one does not know how to simply simulate the function by additions or multiplications), the proposed algorithm can be a good option. We have indirect clues. At the time of writing, there exist a number of studies on the quantum implementations of 5-bit or smaller S-boxes [12, 18, 21, 31–33, 52, 56], but not many results are out there for larger than 5-bit ones when the algebraic construction is hindered but not entirely impossible [67]. Related with the issue, it is worth noting that for 5-bit or smaller functions, near optimal algorithms exist which use exhaustive or meet-in-the-middle techniques. DORCIS algorithm introduced in Section B.2 is a good example.

Lastly, it has been pointed out by an anonymous referee that synthesis algorithms can be applied to improve the performance of homomorphic encryptions by optimizing the multiplicative depth of Boolean circuits[7]. This also deserves an independent study which may potentially find a real-world application.

Right directions for applying the proposed algorithm to cryptography are discussed above, but in this section we only examine the limited applicability and provide the data to the readers mainly for comparisons. Below we make comparisons with various S-boxes that have been and have not been studied before.

B.2 Previously studied S-boxes

Table 3. Comparisons of previously studied S-boxes in quantum cryptanalysis. For ARIA, the result is given by Toffoli-depth. For ASCON, the number in the square bracket means the number of garbage qubits generated.

Target	Previous work		This work	
	#qubits	depth	#qubits	depth
AES[34]	137	6 (T)	15	579 (T)
ARIA[17]	40	196 (Tof)	11	749 (Tof)
SM4[41]	13	72 (T)	15	594 (T)
ASCON[52]	15 [5]	0 (T)	10	24 (T)
DEFAULT$_{\text{Core}}$[18]	8	4 (T)	8	6 (T)
DEFAULT$_{\text{Layer}}$[18]	8	1 (T)	8	5 (T)

Three 8-bit, one 5-bit, and two 4-bit S-boxes are examined and summarized in Table 3. All 8-bit S-boxes (AES, ARIA, SM4) utilize the inverse of a field element in $\mathbb{F}_{2^8}$, enabling an efficient construction. Also, those three ciphers happen to be the only ones with relatively large (over 5-bit) S-boxes that we can find in the literature so far.[10]

About 4-bit S-boxes, there exist a handful of ciphers previously studied, but we only examine the two in the table. Note that for these S-boxes the algorithm designed by[18] should work better than most heuristic algorithms as it should be near optimal for small S-boxes.

B.3 Unstructured S-boxes

Relatively larger S-boxes (> 5) without algebraic structures have attracted less attention from the community, possibly due to the factor discussed in Section B.1. We choose DES, Skipjack, and KHAZAD for the application. There are eight different 6-bit input 4-bit output DES S-boxes. In Table 4, we summarize the results of all DES S-boxes together with Skipjack and KHAZAD. (In Table 1, only the averaged values of QC and the number of Toffoli gates are presented.)

Table 4. Costs of reversible circuits for various unstructured S-boxes, obtained by the algorithm with depth-3 partial search. DES has eight different S-boxes, where each takes as input a 6-bit string and outputs a 4-bit string. Two bits of garbage are unavoidable in the reversible implementation of the DES S-box.

				$d = 0$		$d = 1$		$d = 2$		$d = 3$	
S-box	#in	#out	#grb	QC	#TOF	QC	#TOF	QC	#TOF	QC	#TOF
DES-1	6	4	2	946	143	679	93	714	97	721	95
DES-2	6	4	2	874	121	763	103	760	101	672	92
DES-3	6	4	2	845	123	723	104	723	104	731	104
DES-4	6	4	2	874	129	698	97	684	94	684	94
DES-5	6	4	2	904	128	721	101	756	102	739	101
DES-6	6	4	2	880	128	714	99	718	102	794	112
DES-7	6	4	2	879	125	791	109	791	109	718	101
DES-8	6	4	2	946	135	776	104	805	112	744	100
Skipjack	8	8	0	7553	1100	5575	803	5562	791	5440	771
KHAZAD	8	8	0	7578	1075	5568	819	5411	794	5126	742

Received 8 February 2024; revised 8 February 2024; accepted 21 May 2024

[10] There exists a 6-bit S-box studied in[67], but it does not provide a suitable data format.