Received 20 February 2023, accepted 11 March 2023, date of publication 14 March 2023, date of current version 17 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3257192

RESEARCH ARTICLE

Quantum Program Synthesis Through Operator Learning and Selection

SIHYUNG LEE[1] AND SEUNG YEOB NAM[2], (Senior Member, IEEE)

[1]School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, South Korea
[2]Department of Information and Communication Engineering, Yeungnam University, Gyeongsan 38541, South Korea

Corresponding author: Sihyung Lee (sihyunglee@knu.ac.kr)

ABSTRACT Programming for quantum computers is complicated and time-consuming, because quantum operations are counterintuitive and their combined effects are difficult to understand. Existing tools allow automatic synthesis of quantum programs, which releases the burden of handwriting. However, many existing systems arrange predetermined operators in successive manner to gradually reduce the gap with requirements; these methods are quick but often produce lengthy programs, and they are difficult to adopt for new operators. Other systems depend on stochastic or heuristic search; they identify near-optimal programs for certain cases, but it is not easy to tune the algorithms for a wide range of cases. We propose a system that produces compact programs for most cases and easily evolves with new operators. The system automatically learns the roles of available operators by composing various possible programs. Based on the knowledge, it selects a subset of operators most appropriate for requirements and uses them to compose a program. The learning is geared toward concise programs; thus, the system tends to produce programs with the fewest operators possible. We implemented the system and evaluated it by synthesizing over 400 programs. In comparison with a state-of-the-art system, the proposed system produced programs with approximately 40-times fewer operators at the cost of increased synthesis time from seconds to minutes. We also observed that the system successfully adopted new operators by learning their differences from existing operators and utilizing them in right places. We believe that the system provides a basis of utilizing machine learning for quantum program synthesis.

INDEX TERMS Machine learning, neural networks, program synthesis, quantum computing, quantum program, supervised learning.

I. INTRODUCTION

Quantum computers have been shown to perform certain operations faster than classical computers, including factorization, database search, and simulation [1]. As such, the two types of computers are expected to work together, enabling speedy processing of huge data [2]. The bit of a quantum computer, also known as a qubit, possesses a unique characteristic of being able to exist in multiple states simultaneously [3]. Therefore, we can perform operations on multiple possibilities at once, thus saving time to explore one possibility at a time. For example, to search for a specific item using Grover's algorithm, the amplitudes of all items are flipped around their mean, causing the amplitude of the target item to become more pronounced [1].

Many studies recognize the capabilities of quantum computers and propose algorithms that leverage them. However, implementing and optimizing these algorithms is generally a difficult task that requires time and effort [4]. The operators available on quantum computers, such as Hadamard and Controlled Not [2], manipulate qubits as per quantum physics, and its principles are often considered counterintuitive. Therefore, it is not easy to understand their roles and to anticipate the combined effects when a sequence of operators is jointly used. Hence, hand crafting and optimizing a quantum program becomes less feasible as more complex operations are required [5], [6].

Due to the complexity of quantum programming, research has focused on automatic methods to synthesize quantum

The associate editor coordinating the review of this manuscript and approving it for publication was Siddhartha Bhattacharyya.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

programs, given requirements (or targets) by a user. Many methods begin with a small set of operators and continue to place them according to predefined rules (i.e., in predefined orders and locations) until the program satisfies the requirements [7], [8]. They demonstrate that the operator set can implement most programs. Moreover, the predefined rules allow quick synthesis in general. However, the rules are not always optimal for various targets and often lead to lengthy programs with many operators [9], which in turn increase error rates and execution time. Furthermore, the fixed rules make it difficult to adopt new operators and test how well these operators can be used to build programs.

Other synthesis methods adopt a different approach by searching through a range of potential programs to identify the one that employs the smallest number of operators, instead of relying on a predetermined set of operators and arrangement rules. Due to the vastness of the search space, these methods strive to minimize the search time. In particular, the methods prioritize the order in which they discover different possibilities, such as exploring programs with the fewest predicted operators first. However, most methods select the order based on stochastic algorithms [4], [10] or heuristic functions [11], which produces near-optimal results for small samples but not for a diverse range of targets.

Considering the existing methods, we aim to devise a system that synthesizes programs with as few operators as possible, quickly produces appropriate results for a wide range of targets, and easily incorporates new operators. The proposed system utilizes machine learning. It begins by receiving a set of available operators, and then it learns what the operators can implement by composing and observing various programs. After learning, a user submits synthesis requests. For each request, the system refers to the learned knowledge and selects a subset of operators necessary to fulfill the request with a low operator count. Using the selected operators, the system explores possible programs and identifies one that matches the request. Table 1 summarizes our objectives and how we fulfill them. It also lists potential use cases. A demonstration of the proposed system is posted at https://youtu.be/WLZ-VjLcEh4.

We summarize our contributions as follows.

- We propose a system that synthesizes quantum programs with fewer operators than existing systems (e.g., 10 operators vs. 100 operators on average) and without excessive delay in synthesis time (i.e., synthesis completes in the order of minutes). The system uses knowledge regarding the role of each operator and the best utilization methods. This knowledge is learned by composing a diverse range of programs and observing their characteristics, which does not require human effort and hand optimization.
- We propose a machine-learning model appropriate for distinguishing the roles of different quantum operators with a low learning cost. We also describe details regarding model training and output utilization, such that synthesis outcomes favor programs with fewer operators.
- We evaluated the proposed system by requesting the synthesis of 400 random programs and multiple benchmarks that require 2–5 qubits. Our observation revealed that the system generated programs that used approximately 40-times fewer operators compared to a state-of-the-art system, albeit with an increase in the average synthesis time from seconds to minutes. We also demonstrated that when provided with new operators, the system could promptly adopt and use these operators to compose more compact programs.

TABLE 1. Summary of objectives and use cases.

ID	Objective	How We Fulfill the Objective
1	Synthesize with as few operators as possible	• Use search-based method (not rule-based) • Prefer to learn implementations with fewer operators if multiples exist for the same target
2	Quickly produce decent results for diverse targets	• Compose and learn diverse programs • Select a small subset of operators for synthesis, thereby exploring fewer programs • Skip duplicates when exploring programs
3	Easily incorporate new operators	• Learn operators and discover the best ways to utilize them independently without human intervention

ID	Use Case
1	**Synthesize** part of a program that is performance-critical
2	**Optimize** a known implementation of an algorithm, which was either handmade or synthesized by existing works
3	**Adopt new operators** into a user-level or native operator set, after synthesizing various programs with the operators and observing how well and widely they are utilized
4	**Learn and study quantum operations** by synthesizing many small programs with a pool of well-known operators, thereby effectively understanding their meanings and various usage

The remainder of this paper is organized as follows. Section II describes preliminaries on quantum programs that are used throughout the paper and presents existing methods related to the proposed system. Section III presents the proposed system in detail, including how to learn the roles of quantum operators and use the knowledge to synthesize quantum programs. Section IV evaluates the proposed system compared with a state-of-the-art system. Finally, Section V concludes the study and presents future work.

II. BACKGROUND AND RELATED WORK

A. BACKGROUND ON QUANTUM PROGRAMS

In this subsection, we explain background information on quantum programs. In particular, we explain the notations in Table 2, which are used throughout the paper. Each notation has an ID from 1 to 5 and examples on the right.

In a classical computer, a bit is the smallest unit of data, and its value can be either 0 or 1. In a quantum computer, the smallest unit is called a qubit; it can simultaneously have two values with different probabilities and phases [3]. When we measure the qubit, its value becomes 0 or 1, according

to the probabilities. As such, a qubit is represented with a vector of two complex numbers (α, β), as in notation 1 in Table 2; their amplitudes indicate the probabilities of the qubit measured as 0 and 1, respectively, such that $|\alpha|^2 + |\beta|^2 = 1$; the ratio between real and imaginary parts determines the phases. For example, (1, 0) indicates that the qubit is 0 for sure, and (0, 1) indicates that it is definitely 1, both demonstrating zero phases. Another example ($1/\sqrt{2}$, $-1/\sqrt{2}$) indicates equal probabilities of being 0 and 1 with phases 0 and π, respectively.

Multiple qubits are represented with the Kronecker product $\otimes$ of the qubits' states. Notation 2 in Table 2 shows the state of two qubits. The first and second qubits are represented with (α_1, β_1) and (α_2, β_2), respectively. In the result of the product, the four complex numbers ($\alpha_1\alpha_2, \alpha_1\beta_2, \beta_1\alpha_2, \beta_1\beta_2$) show the likelihood and phase of four possible values, namely 00, 01, 10, and 11, respectively. For example, two qubits (1, 0) and (0, 1) are represented together with (0, 1, 0, 0), indicating that the two qubits are definitely 0 and 1, respectively. Similarly, n qubits are represented with a vector of 2^n complex numbers.

TABLE 2. Notations related to quantum programs.

ID	Notation	Example
1	$\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$	$\begin{bmatrix}1\\0\end{bmatrix}$ $\begin{bmatrix}0\\1\end{bmatrix}$ $\begin{bmatrix}1/\sqrt{2}\\-1/\sqrt{2}\end{bmatrix}$
2	$\begin{bmatrix}\alpha_1\\ \beta_1\end{bmatrix} \otimes \begin{bmatrix}\alpha_2\\ \beta_2\end{bmatrix} = \begin{bmatrix}\alpha_1\alpha_2\\ \alpha_1\beta_2\\ \beta_1\alpha_2\\ \beta_1\beta_2\end{bmatrix}$	$\begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$
3	$\begin{bmatrix}m_{11} & m_{12}\\ m_{21} & m_{22}\end{bmatrix} \times \begin{bmatrix}\alpha\\ \beta\end{bmatrix} = \begin{bmatrix}\alpha'\\ \beta'\end{bmatrix}$	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \times \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1/\sqrt{2}\\-1/\sqrt{2}\end{bmatrix}$
4	q[i] –[M_1]– q[i+1] –[M_2]– $M_1 \otimes M_2$	q[0] –[X]–; q[1] –[H]–; $X \otimes H$ q[0] ——; q[1] –[X]–; $I \otimes X$
5	stage: 1 2 q[0], q[1], ⋮, q[n] –[M_1]–[M_2]– $M_2 \times M_1$	stage: 1 2 q[0] –[X]—— q[1] –[H]–[X]– $(I \otimes X) \times (X \otimes H)$

We now explain quantum operations performed on qubits. An operation on n qubits can be represented as a $2^n \times 2^n$ unitary matrix [12]. By multiplying this matrix on the qubits' state, we obtain the state resulting from the operation. Notation 3 in Table 2 demonstrates an operation performed on *1* qubit. The operation is represented as $2^1 \times 2^1$ matrix and transforms the qubit's state from (α, β) to (α', β'). The example on the right shows an operation called Hadamard [2]. It transforms the initial state (0, 1) into ($1/\sqrt{2}$, $-1/\sqrt{2}$), so that the qubit can possibly have two values 0 and 1 with an equal probability.

Operations on different qubits are represented with the Kronecker product of the corresponding matrices, similarly to the manner in which multiple qubits are represented. Notation 4 in Table 2 demonstrates operations M_1 and M_2 performed on qubit i and qubit $i+1$, respectively. Their combined effect is represented as $M_1 \otimes M_2$. The first example on the right shows operations X and H on qubits *0* and *1*, respectively; thus, the resulting operation is $X \otimes H$. The second example does not apply any operation on qubit 0 and applies X on qubit *1*; no operation is considered the identity operation I, as the qubit's state remains the same, and thus the resulting operation is $I \otimes X$.

A quantum program performs a sequence of operations on qubits to transform them into intended states. In this context, a stage refers the operations that are applied simultaneously on different qubits. The effect of two consecutive stages is represented as their product. Notation 5 in Table 2 demonstrates a subsequent application of two operations, M_1 and M_2. Their combined effect is represented as $M_2 \times M_1$. For example, if stages 1 and 2 perform $X \otimes H$ and $I \otimes X$, respectively, then the program as a whole is considered to perform $(I \otimes X) \times (X \otimes H)$. Although we do not show in the example, the final stage typically involves observation, which measures the qubits to obtain the computation results.

Finally, the objective of synthesis is to determine how to arrange operators to achieve a specific function given by a user. In particular, we allow the user to describe the target function as a matrix M_{target}. Given this matrix, the proposed method discovers an arrangement that implements the target (e.g., $(I \otimes X) \times (X \otimes H)$ if this is equal to M_{target}). In general, various implementations exist that implement the same target, and we aim to find the one with the fewest possible stages and operators, because it has low error rates and quick execution. Note that we consider two implementations M_1 and M_2 to be equal if they differ only by a global phase (i.e., $M_1 = e^{i\theta} \times M_2$), as in many previous studies [6], [13]. This is because algorithms often utilize the relative phase differences among qubits, rather than the global phase. Moreover, the global phase can be eliminated during measurement without affecting the computation results.

B. RELATED WORK

A universal set of quantum operators refers to those that can implement any unitary target. Early studies discover universal sets and suggest methods to arrange them to implement a target. A universal set is typically composed of one *2*-qubit operator and several *1*-qubit operators. For example, Barenco et al. [7] show that successive application of a *2*-qubit operator {*CNOT*} and *1*-qubit operators {R_x, R_y, R_z} can approximate any n-qubit target. Boykin et al. [8] propose different *1*-qubit operators {H, T} and demonstrate that the operators can be implemented with greater fault tolerance compared to previously proposed operators. It is also shown that the 2-qubit operator {*CNOT*} can be replaced with either of {*CZ*, *iSWAP*} while the set remains universal [5].

Given a universal set, various methods exist to implement a target. Many of the methods are based on predefined rules—they continue to arrange operators in particular locations, such that each placement reduces the distance from the target. For example, Shende et al. [14], as shown in Fig. 1, decompose an *n*-qubit target into *(n-1)*-qubit unitaries and controlled operators. The decomposition continues until the unitaries become *1* or *2* qubits, which are then mapped to the universal set of operators [15]. Other studies decompose a target into basic tasks and implement one task at a time. For example, Niemann et al. [16], [17] synthesize a target by implementing superpositions, followed by permutations, and then by incorporating phase shifts. To summarize, the rule-based methods use predefined operators and rules for their arrangement. These rules quickly synthesize most of the targets, but the resulting programs tend to use numerous stages and operators, exponentially proportional to the number of qubits [9]. Furthermore, to explore a different operator set, the rules must to be adjusted considering the peculiarities of the new operators; such adjustment takes time and effort even by experts. We aim to devise methods that produce more compact programs and that can be easily adjusted to new operators.

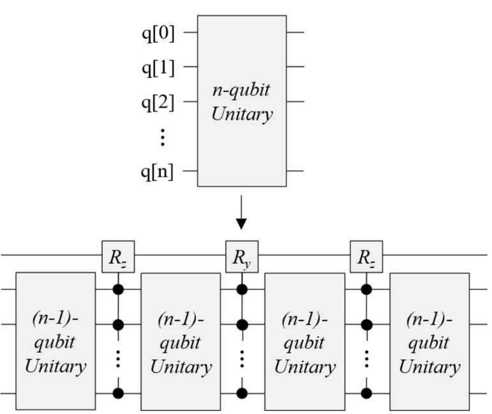


FIGURE 1. Example of rule-based synthesis: Decomposition of n-qubit unitary into (n-1)-qubit unitaries.

A few studies take different approaches from the rule-based methods. The studies are based on search—they explore various possible programs until they find one that matches the target [4], [6], [10], [11]. This approach can generally identify more compact implementations than the rule-based strategy, because it examines more arrangements than those that are predefined in rules.

Some search-based studies employ stochastic search based on ant colony optimization [4] and genetic algorithm [10]. One method develops a program by appending operators up to a certain depth, where the operators are randomly selected in accordance with a probability distribution [4]. Considering how well the program fits the target, the method updates the probability distribution, which then is used to develop the next program. This process is repeated until the program matches the target. Another method continues to revise an initial program by randomly performing one of multiple revisions, such as adding and removing an operator, replacing one operator with another, and switching the locations of two operators [10]. This method also considers how well the resulting program fits the target and accordingly updates the possibility of the revisions in the next round. Overall, the stochastic methods depend on random selections. They may find the target in a few iterations if the selections happen to be on the right track. However, the methods may also take many iterations, and the resulting program can be lengthy and far from the optimal. The proposed method does not depend on randomization, and it uses machine learning to select the most appropriate operators for each different target.

Other search-based studies propose methods to limit the search time. QSearch [11] utilizes A* search algorithm to explore programs with the fewest predicted operator counts first. The prediction is based on a heuristic function, and its accuracy determines the synthesis time and the conciseness of resultant programs. It is not easy to devise a heuristic function that produces accurate prediction over a wide range of targets. The Meet-in-the-Middle algorithm [6] saves synthesis time by exploring programs with up to s stages to synthesize a program that requires $2s$ stages. In particular, if the algorithm discovers two programs P_1 and P_2, such that $P_1^\dagger \times M_{target} = P_2$ (M_{target} represents the target matrix, and $P^\dagger$ is the conjugate transpose of the unitary P), then they synthesize the target by concatenating the two programs (i.e., $M_{target} = P_1 \times P_2$). Similar to the existing studies, the proposed method is based on search, but introduces a different strategy to narrow down the search. The method begins with a pool of operators and selects a subset necessary to implement the target with low cost. Since this subset is relatively smaller than the initial pool, we can explore fewer programs, thereby reducing search time. The proposed system can be used along with the existing methods and improve efficiency.

In practice, a combination of rule-based and search-based methods are used. For example, Qiskit [18], the software development kit for IBM's quantum computers, uses rule-based method to decompose a target into 1–2 qubit unitaries [19], and then performs stochastic search to implement the unitaries with a minimum number of operators. Furthermore, Qiskit has a final optimization stage that merges multiple operators into one and thus reduces operator count [20]. For example, two consecutive operator S's are replaced with one operator Z^{-1}, because $S \times S = Z^{-1}$. The proposed system also uses such relationships, not only to reduce operator count but also to explore unique programs. In Section IV, we evaluated the proposed method compared to Qiskit, because it is the most up-to-date, widely used, and produces decent results.

III. LEARNING AND SYNTHESIS METHODS

A. OVERVIEW OF SYNTHESIS METHOD

We outline the proposed learning and synthesis methods in this subsection. Fig. 2 presents an overview of the methods, which proceed in two phases.

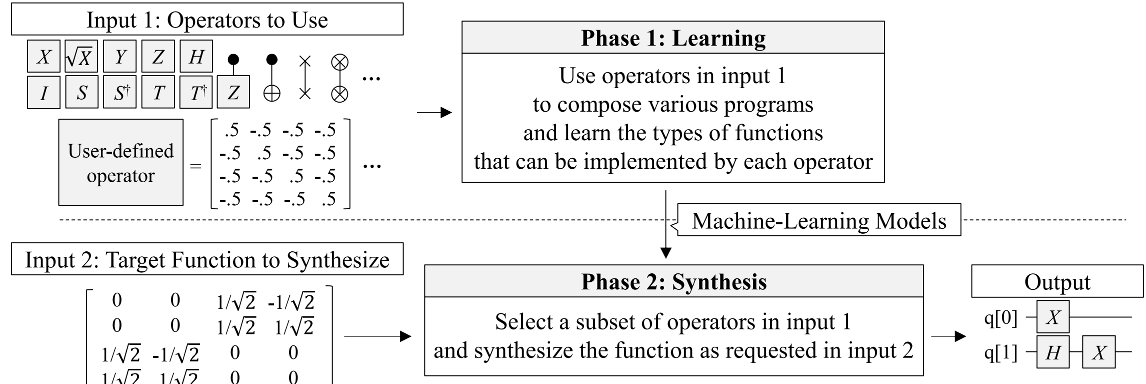


FIGURE 2. Overview of proposed learning and synthesis methods.

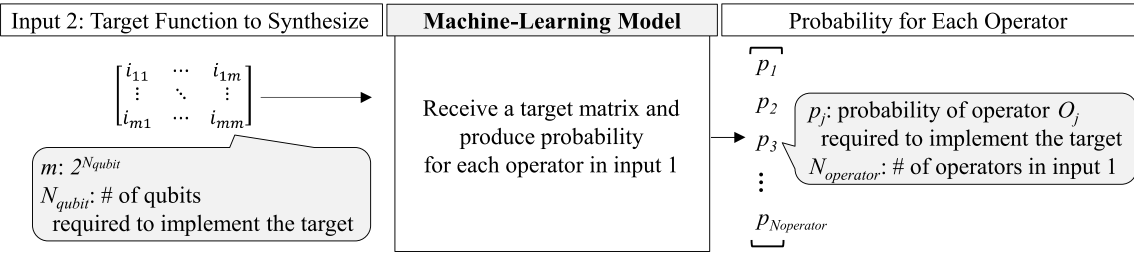


FIGURE 3. Overview of utilization of machine learning to differentiate operators in input 1.

In phase 1, the user specifies input 1 of the system, a set of operators to use for synthesis. This set contains native operators directly supported in the hardware, as well as those composed of multiple such operators. The user can also define a previously-built program as a custom operator, so that it can be used as a building block for synthesis. Then, the system learns to distinguish the operators—it composes various programs with the operators and characterizes the functions that each operator can implement. For example, in Fig. 2, the user specifies operators available at a quantum computer (e.g., X and H) and also defines a custom operator (the one defined with a matrix). Then, the system learns their characteristics, such as X inverts a qubit, H creates a uniform superposition, and the custom operator flips amplitude about the mean.

In phase 2, the user specifies input 2 of the system, the target function to synthesize as a matrix format. The system then selects a subset of operators likely to implement the target, based on the knowledge learned in phase 1. Following the selection, the system examines different arrangements of the operators and finds one that correctly implements the target. For example, in Fig. 2, the user specifies a target that exhibits inversion and superposition; thus, the system selects $\{X, H\}$. By arranging the selected operators, the system discovers the implementation, as shown in the output in far right. When the synthesis is complete, the system repeats phase 2 as the user specifies different targets.

We consider two objectives when designing the learning methods. The first objective is to synthesize a target without excessive delay. For this purpose, the system must select a subset of operators essential to implement the target; otherwise, the system needs to explore a large number of arrangements with unnecessary operators. An accurate selection of operators depends on how well the system learns their characteristics in phase 1. The second objective is to discover an implementation with a low cost (i.e., we prefer an implementation with fewer operators and stages). This goal is also related to the learning in phase 1. If multiple arrangements implement the same target, we choose to learn those with minimal costs. The following subsection (Section III-C) describes details on how we realize the two objectives when preparing training data and learning operator characteristics.

B. PHASE 1: LEARNING OPERATOR ROLES

In phase 1, the proposed system learns the characteristics of operators in input 1 using a machine-learning model. Fig. 3 shows the input/output of this model and its utilization.[1] We aim to train the model, such that when it receives a target function as a matrix in phase 2, it produces a vector of probabilities p_1 to $p_{Noperator}$, one for each operator. Probability p_j is the chance of operator O_j required to implement the target, and $N_{operator}$ is the number of operators in input 1. If the

[1]The machine-learning model can be various models, and we demonstrate the use of neural networks in the evaluation (Section IV-A).

predicted probability is greater than a threshold, the system selects O_j as a building block for implementing the target. To train the model in this manner, we need to prepare various target functions as matrices (i.e., input to the model) and label them with probability vectors (i.e., output to the model).

We prepare the training data as follows. When the user specifies the set of operators in input 1, they specify two additional parameters N_{qubit} and N_{stage_train}, the numbers of qubits and stages to use for training, respectively. The system then composes various programs by arranging the operators to fill N_{qubit} qubits and up to N_{stage_train} stages. For each such program, the system computes the corresponding matrix representation and labels it with a probability vector $[p_1, p_2, \ldots, p_{Noperator}]$; $p_j = 1$ if operator O_j is used in the program and $p_j = 0$ otherwise. Using these labeled matrices, we train the machine-learning model. As a result, when the trained model receives a target matrix in phase 2, the predicted probability p_j is likely to be near 1 if O_j must be used, and thus O_j is selected; otherwise, p_j is likely to be near 0, and thus O_j is not selected.

Table 3 lists two training examples generated by the proposed system. We assume that $N_{qubit} = 2$, $N_{stage_train} = 2$, and input 1 includes four operators $\{I, X, \sqrt{X}, H\}$. Example 1 is a one-stage arrangement, and its matrix represents $X \otimes \sqrt{X}$. This arrangement uses X and $\sqrt{X}$, and thus is labeled with probabilities [0, 1, 1, 0]. Example 2 is a two-stage arrangement, and its matrix represents $(I \otimes X) \times (X \otimes H)$. This arrangement includes I, X, and H, and thus is labeled with [1, 1, 0, 1].

TABLE 3. Examples of training data.

We assume that $N_{qubit} = 2$, $N_{stage_train} = 2$, and input 1 = $\{I, X, \sqrt{X}, H\}$.

ID	Operator Arrangement	Matrix Representation	Probability Vector
1	q[0] –[X]– q[1] –[√X]–	$\begin{bmatrix} 0 & 0 & .5{+}.5i & .5{-}.5i \\ 0 & 0 & .5{-}.5i & .5{+}.5i \\ .5{+}.5i & .5{-}.5i & 0 & 0 \\ .5{-}.5i & .5{+}.5i & 0 & 0 \end{bmatrix}$	[0, 1, 1, 0]
2	q[0] –[X]––––– q[1] –[H]–[X]–	$\begin{bmatrix} 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} \\ 0 & 0 & 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 & 0 \end{bmatrix}$	[1, 1, 0, 1]

When the system generates training data by arranging operators, different arrangements often result in the same function. In this case, we include one arrangement for training data and exclude the others to improve implementation cost and learning accuracy. Table 4 presents four different sample arrangements that lead to the same function. Arrangements 1–3 result in the same matrix, and arrangement 4 differs from the others only by a global phase. Among such equivalent arrangements, we select the one with the lowest cost of implementation—we prefer the one with fewer stages and operators.[2] For example, Table 4 shows that arrangements 1–2 have fewer stages than arrangement 4 and fewer operators than arrangement 3; thus, we include either of the two for training data. As a result, the system learns implementations with lower costs and thus tends to prefer these implementations when synthesizing targets in phase 2. This way of pruning training data also improves learning accuracy; this is because different operators have fewer overlapping examples in training data, and thus the system can better distinguish the operators.

TABLE 4. Examples of equivalent programs.

We assume that every operator has the same cost of 1, except for operator I (i.e., empty place), which has the cost of 0.

ID	Operator Arrangement	Matrix Representation	(# of Stages, Operators)
1	stage: 1 2 q[0] –[X]––––– q[1] –[H]–[X]–	$\begin{bmatrix} 0 & 0 & 1/\sqrt{2} & -1/\sqrt{2} \\ 0 & 0 & 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 \\ 1/\sqrt{2} & 1/\sqrt{2} & 0 & 0 \end{bmatrix} = M$	(2, 3)
2	q[0] –––––[X]– q[1] –[H]–[X]–	$= M$	(2, 3)
3	q[0] –[√X]–[√X]– q[1] –[H]–[X]–	$= M$	(2, 4)
4	stage: 1 2 3 q[0] –[S†]–[Y]–[S]– q[1] –[H]–[X]–––––	$\begin{bmatrix} 0 & 0 & -1/\sqrt{2} & 1/\sqrt{2} \\ 0 & 0 & -1/\sqrt{2} & -1/\sqrt{2} \\ -1/\sqrt{2} & 1/\sqrt{2} & 0 & 0 \\ -1/\sqrt{2} & -1/\sqrt{2} & 0 & 0 \end{bmatrix} = e^{i\pi} \times M$	(3, 5)

In Table 5, we put together the methods for generating and pruning training data. The code begins from the function find_unique_programs It uses a while loop to explore N_{qubit}-qubit and s-stage arrangements, where s ranges from 1 to N_{stage_train} (lines 12–23). We compare each arrangement P_{new} with previously reported arrangements (lines 16–18) to retain only unique programs with the lowest costs (lines 19–20). If the function of P_{new} emerges the first time, we store it for comparison with later arrangements (lines 21–22). We compute and compare the matrix representations of two arrangements to confirm their equivalence (line 16). In particular, we transform the matrices, such that they appear identical if they differ only by a global phase, as follows (lines 17, 28–30). If the first non-zero element of a matrix is $(a + bi)$ (i.e., the non-zero element with the smallest row and column numbers), we multiply the matrix with a scalar $e^{i\theta} = (a - bi)/\sqrt{a^2 + b^2}$, so that the element becomes $\sqrt{a^2 + b^2}$ and thus has a zero phase. For example, in arrangements 1 and 4 in Table 4, the first non-zero elements are $1/\sqrt{2}$ and $-1/\sqrt{2}$ at row 1 and column 3, respectively. Therefore,

[2] We can fine-tune the cost assignments to better reflect the peculiarities of hardware (e.g., by assigning different weights to operators).

TABLE 5. Algorithm for exploring unique programs with low costs.

```
01  # Amber: Keyword, Green: Variable
02  #
03  # N_qubit, N_stage_train: Numbers of qubits and stages to use for training
04  # P: Arrangement of operators in S_operator (i.e., a quantum program)
05  # P.cost: Cost of implementing P
06  # S_operator: Set of operators {O_1, O_2, ..., O_Nop} in input 1
07  # S_program: Symbol table with unique matrices as keys
08  #    and the corresponding operator arrangements as values
09
10  # Find unique arrangements of operators with low costs
11  function find_unique_programs(N_qubit, N_stage_train, S_operator)
12    s = 1
13    while s ≤ N_stage_train
14      for each N_qubit, s-stage arrangement P_new with operators in S_operator
15        if is_duplicate(P_new), then continue
16        M = matrix representation of P_new
17        M = convert_first_nonzero_element_to_zero_phase(M)
18        if M is in S_program, then
19          P_old = S_program[M]
20          if P_new.cost < P_old.cost, then S_program[M] = P_new
21        else
22          S_program[M] = P_new
23      s = s+1
24    return S_program
25
26  # Multiply the matrix M by a scalar,
27  #    so that the first non-zero element has a zero phase
28  function convert_first_nonzero_element_to_zero_phase(M)
29    a + bi = first non-zero element of M
30    return (a−bi)/√(a²+b²) × M
31
32  # Check the program P to see if it has frequent patterns of duplicates
33  function is_duplicate(P)
34    if an I in stage s-1 precedes a non-I in stage s, then return true
35    if two successive operators in stages s-1 and s are equivalent to
36      another operator, then return true
37    return false
```

the corresponding matrices are multiplied by the scalars 1 and -1, respectively, making them appear identical.

Before we compute matrix representations and test their equivalence, we skip two types of arrangements that definitely lead to duplicate programs (lines 15, 33–37). We found that these two types form a majority of duplicate programs; thus, skipping these arrangements reduces the cost of matrix computation and comparison. The first type occurs when an operator I in stage $s-1$ precedes a non-I operator in stage s (line 34); this type is equivalent to having the non-I and I in stages $s-1$ and s, respectively. For example, in arrangement 2 in Table 4, the I in stage 1 precedes the X in stage 2 on q[0]; this is equivalent to having X and I in stages 1 and 2, respectively, as in arrangement 1; thus, we skip arrangement 2. The second type occurs when two successive operators in stages $s-1$ and s are equivalent to another operator O (lines 35–36); replacing the two with one O leads to an equivalent arrangement with one fewer operators. For example, in arrangement 3 in Table 4, one $\sqrt{X}$ precedes the other on q[0]; replacing the two $\sqrt{X}$'s with one X leads to an equivalent arrangement with fewer operators, as in arrangement 1; thus, we skip arrangement 3. We identify all such equivalent relationships before we begin find_unique_programs For example, when input 1 is $\{I, X, \sqrt{X}, H\}$, we first discover three equivalent relationships $X \times X = I$, $\sqrt{X} \times \sqrt{X} = X$, and $H \times H = I$, and then use them to skip duplicate arrangements in find_unique_programs To summarize, when we explore programs and generate training data, we perform multiple checks to eliminate duplicates at early stages, thereby retaining only unique programs with low costs.

C. PHASE 2: SYNTHESIS WITH LEARNED KNOWLEDGE

In phase 2, the proposed system synthesizes the target function specified in input 2. Among the operators in input 1, the system predicts a subset of operators required to implement the target, based on the knowledge learned in phase 1. Using this subset, the system explores unique programs and identifies one that matches the target.

Table 6 shows a pseudo code for phase 2. The code begins from the function synthesize(), and this function takes four arguments. The first argument M_{target} is the target function as a matrix, and we assume that it requires N_{qubit} qubits (i.e., M_{target} has 2^{Nqubit} rows and columns). The second argument $S_{operator}$ is the set of operators to use for synthesis, as specified in input 1. The third argument $T_{operator}$ is the threshold for selecting a subset of operators[3]; the operators with predicted probabilities greater than this threshold will be used as building blocks. The last argument $N_{stage_synthesis}$ is the maximum number of stages to search; the system explores arrangements with up to $N_{stage_synthesis}$ stages until the target is found.

TABLE 6. Algorithm for synthesizing target function.

```
01  # Amber: Keyword, Green: Variable
02  #
03  # N_qubit, N_stage_synthesis: Numbers of qubits and stages to use for synthesis
04  # P: Arrangement of operators in S_selected (i.e., a quantum program)
05  # S_operator: Set of operators {O_1, O_2, ..., O_Nop} in input 1
06  # S_selected: Set of operators currently selected among those in S_operator
07  # S_remaining: Set of operators not currently selected
08  # T_operator: Threshold for selecting an initial set of operators. We select
09  #    those with predicted probabilities greater than this threshold
10
11  # Synthesize M_target, the target function in matrix format
12  function synthesize(M_target, S_operator, T_operator, N_stage_synthesis)
13    predict probabilities by feeding M_target to the machine-learning model
14    S_selected = operators in S_operator whose probabilities ≥ T_operator
15    S_remaining = S_operator - S_selected
16    while S_remaining is not {}
17      s = 1
18      while s ≤ N_stage_synthesis
19        for each unique N_qubit, s-stage arrangement P with S_selected
20          if P's matrix representation equals M_target, then return P
21        s = s+1
22      move into S_selected the operator with greatest probability in S_remaining
23    return none
```

The function synthesize() works as follows. We first feed M_{target} to the machine-learning model, which then outputs

[3] In the evaluation (Section IV-B), we used the threshold $T_{operator} = 0.5$. In practice, the range 0.45–0.55 produced similar results, while a threshold outside this range decreased prediction accuracy.

a vector of probabilities, one for each operator (line 13). Among the operators, we select those with probabilities larger than the threshold (line 14). Using these operators, we compose various programs with up to $N_{stage_synthesis}$ stages (lines 17–21); if we identify a match with the target, we return the program (line 20). If the machine-learning model fails to predict the probabilities accurately or if a proper threshold is not chosen, then we may not identify any match. In this case, we include one additional operator with the highest probability among those that have not been included (line 22). Using this new operator, we repeat the same process of exploring possible programs. Thus, we end up adding operators in descending order of their probabilities (lines 16–22). If we utilize all operators in input 1 and fail to identify any match, we return none (line 23), meaning that the target cannot be implemented with the operators and $N_{stage_synthesis}$ stages.[4]

When we compose programs with a subset of operators (lines 17–21), we explore unique programs (line 19) and disregard those with the same function again. For this purpose, we apply the same methods of duplicate removal as described in Section III-B (lines 15–17 in Table 5); (i) we skip the two types of arrangements that must lead to duplicates with possibly large costs, and (ii) we compare matrix representations after multiplying them with scalars, such that those that differ by a global phase appear identical.

TABLE 7. Application of algorithm in table 6 on target in fig. 2.

Input 1={I, X, $\sqrt{X}$, H} with predicted probabilities [0.92, 0.84, 0.002, 0.46]
$T_{operator}$=0.5 and $N_{stage_synthesis}$=2

$S_{selected}$	# of Stages	Unique Programs Explored
{I, X}	1	(q[0]: X; q[1]: —), (q[0]: —; q[1]: X), (q[0]: X; q[1]: X)
{I, X}	2	No unique program (e.g., (q[0]: X–X; q[1]: X) = (q[0]: —; q[1]: X))
{I, X, H}	1	(q[0]: H; q[1]: —), (q[0]: —; q[1]: H), (q[0]: X; q[1]: H), (q[0]: H; q[1]: X), (q[0]: H; q[1]: H)
{I, X, H}	2	(q[0]: X–H; q[1]: —), (q[0]: X; q[1]: X–H), …, (q[0]: X; q[1]: H–X) (Target found)

Table 7 illustrates the programs explored by synthesize(), during the synthesis of the target in Fig. 2. We assume that input 1 includes four operators $\{I, X, \sqrt{X}, H\}$ and their predicted probabilities are [0.92, 0.84, 0.002, 0.46]. We also assume that $T_{operator} = 0.5$ and $N_{stage_synthesis} = 2$. The system initially selects $\{I, X\}$, since their probabilities are greater than $T_{operator}$. With the two operators, the system explores three unique programs with one stage. The system skips all programs with two stages, because they are duplicates of one-stage programs. After exploring programs with $\{I, X\}$, the system cannot find the target. Thus, the system adds H, the remaining operator with the largest probability. The system then continues to explore unique programs that have not been considered in the previous rounds, and finally discovers the target among those with two stages. Note that the system does not explore possibilities with operator $\sqrt{X}$, because it has a low probability and the target was found before this operator was added. If the system used all four operators from the beginning, it must have explored 3–4 times more programs till the target was identified. With more operators in input 1, an accurate selection of its subset will make even greater differences.

[4]If the system confirms that the target cannot be implemented, the user can increase $N_{stage_synthesis}$ or add more operators to input 1, and retry synthesis.

IV. EVALUATION

We implemented and demonstrated the proposed system, based on the learning and synthesis methods in Section III. Section IV-A describes the details of experimental setup, including data preparation and learning operator patterns. Section IV-B evaluates the synthesis results of the proposed system, in comparison with an existing work.

A. PHASE 1: DATA PREPARATION AND LEARNING

We prepared data for machine learning (i.e., programs) and trained the machine-learning model, as described in Section III-B. Table 8 summarizes the parameters and algorithms used for the learning. We explain them throughout this section.

TABLE 8. Parameters and algorithms used for learning in phase 1.

ID	Description
1	Set of operators in input 1 = {I, X, $\sqrt{X}$, H, Y, Z, S, $S^\dagger$, T, $T^\dagger$, $CNOT$, CZ, $SWAP$, $iSWAP$, $CCNOT$, $CSWAP$, CS, CT, $QFT2$, $QFT3$}
2	Number of qubits to use for training, N_{qubit} = 2,3,4,5
3	Number of stages to use for training, N_{stage_train} = 4
4	# of epochs used for training = 500
5	Adam optimization with learning rate α = 0.001
6	Batch normalization with momentum = 0.99 The momentum was used to compute moving means and variances.
7	L2 regularization with regularization constant λ = 0.1 No dropout was used.

Input 1 includes 20 operators as building blocks. We chose the first 16 operators (i.e., $\{I, X, \ldots, CSWAP\}$), because they are supported in many quantum computers [21], [22]. The four other operators are controlled-S (CS), controlled-T (CT), and quantum Fourier transforms for 2 and 3 qubits ($QFT2$, $QFT3$). These operators are typically implemented with multiples of the first 16 operators [1]. We assumed that the four operators were recently added to the user operator set, because they had been frequently used as building blocks for large programs. The operators work on a range of qubits, including one qubit (I, X, $\sqrt{X}$, H, Y, Z, S, $S^\dagger$, T, $T^\dagger$), two qubits ($CNOT$, CZ, $SWAP$, $iSWAP$, CS, CT, $QFT2$), and three qubits ($CCNOT$, $CSWAP$, $QFT3$). Using the operators,

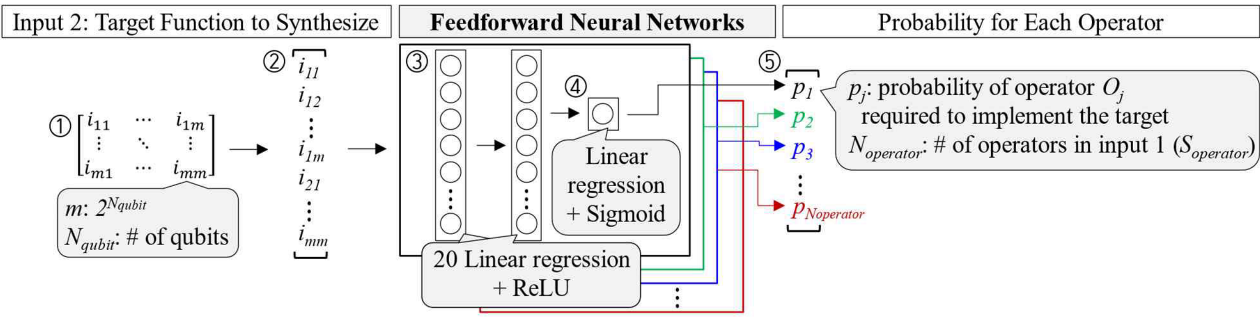


FIGURE 4. Neural-network model for learning to differentiate operators in input 1.

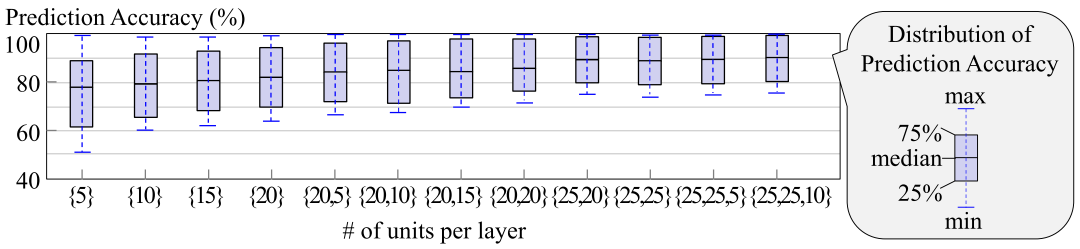


FIGURE 5. Prediction accuracy of neural-network model for various numbers of units and hidden layers.

we aimed to synthesize programs that require 2–5 qubits (N_{qubit}). To this end, we produced various programs that utilize 2–5 qubits and learned the relationships between the programs and their operator usage. For each qubit number in the range 2–5, we performed the same steps in the following paragraphs.

We generated programs that use up to 4 stages (N_{stage_train}). In particular, for each number of stages, we randomly chose 3,000 programs. We then split the programs into the following three sets: (i) S_{train} with 2400 programs for training machine-learning models, (ii) S_{params} with 300 programs for estimating parameters (i.e., parameters in Table 8, machine-learning model, and thresholds), and (iii) S_{test} with the 300 remaining programs for testing the models.[5] We annotated the programs with their matrix representations and probability vectors, as shown in Table 3, so that the machine-learning model could learn their relationships.

Using the data in S_{train}, we trained the machine-learning model (Fig. 3), so that it learns the characteristics of operators. Fig. 4 depicts the model, and it consists of a set of feedforward neural networks [23]. Each network focuses on one operator O_j in input 1 and learns to differentiate whether O_j is required or not for various programs in S_{train}. After training, the network receives a target function as input and generates a probability p_j, indicating the possibility of requiring O_j for synthesizing the target. Hence, the entire set of networks learns the characteristics of all operators and produces a vector of probabilities, one for each operator.

Each of the neural networks performs the following functions; it receives ① a target matrix as input and flattens the matrix into ② a one dimensional array. This flattening preserves all matrix elements and does not lose information. The network then passes the array through ③ two hidden layers and ④ an output layer, producing ⑤ a probability p_j for operator O_j. Each of the hidden layers in ③ includes 20 units, where one unit performs linear regression and ReLU [24]. The output layer in ④ contains one unit that performs linear regression and sigmoid, producing a probability between 0 and 1 in ⑤.

The number of hidden units in the neural networks affects the prediction accuracy, and the optimal number can change as we alter the operators in input 1. Therefore, we describe guidelines on determining the number. We incrementally incorporated more units, trained the networks with S_{train}, and measured the prediction accuracy with S_{params}. We measured the accuracy as the average proportion of operators that are correctly classified when the threshold $T_{operator} = 0.5$. An operator is correctly classified (i) when it is required and its predicted probability $\geq T_{operator}$ or (ii) when it is not required and its probability $< T_{operator}$. Fig. 5 presents the effect of different unit numbers. On the horizontal axis, $\{N_1, N_2, \ldots, N_L\}$ indicates the numbers of units over different layers, where N_j is the number of units at layer j, and L is the number of hidden layers. For example, {20,5} indicates two hidden layers with 20 and 5 units, respectively. The vertical axis shows the distribution of prediction accuracy over all programs in S_{params}. The accuracy generally increased with

[5] There can be fewer than 3,000 programs. For example, 104 unique programs exist that utilize 2 qubits and 1 stage. In this case, we used all of the programs and split them into three sets, S_{train}, S_{params}, and S_{test}, according to the ratio 8:1:1.

(a) Stage count for N_{qubit}=2 (b) Operator count for N_{qubit}=2 (c) Synthesis time for N_{qubit}=2

(d) Stage count for N_{qubit}=3 (e) Operator count for N_{qubit}=3 (f) Synthesis time for N_{qubit}=3

(g) Stage count for N_{qubit}=4 (h) Operator count for N_{qubit}=4 (i) Synthesis time for N_{qubit}=4

(j) Stage count for N_{qubit}=5 (k) Operator count for N_{qubit}=5 (l) Synthesis time for N_{qubit}=5

FIGURE 6. **Stage/operator counts in synthesized programs and time required to synthesize them for proposed and existing systems.**

the number of units. When the unit numbers reached {20,20}, the median accuracy was nearly 85% with the minimum above 70%. We selected {20,20} for two reasons. Firstly, beyond the selected numbers, the accuracy increased only slightly, whereas the time required to train the models rapidly increased. Secondly, the proposed system selected a subset of the required operators when the prediction was not perfect; however, if the minimum accuracy was above 70%, the system soon added the rest of required operators in the next few iterations (lines 16–22 in Table 6) and thus synthesized the target without excessive delay (i.e., within 10 minutes on average). We demonstrate that this is the case in Section IV-B.

We trained the models on a machine with a 3.4GHz CPU (Intel Core i7-6700) and a 16GB RAM. We used the machine-learning algorithms and parameters summarized in Table 8 (IDs 4–7), and implemented the algorithms using Python, TensorFlow [25], and NumPy [26]. Training the neural networks took approximately 20 minutes to 6 hours for N_{qubits} in the range 2–5.

B. PHASE 2: SYNTHESIS AND EVALUATION

Using the trained model, we synthesized various target functions and evaluated the results.[6] In particular, we compared the proposed methods with the synthesis functionality of

[6]At GitHub (https://github.com/sihyunglee26/Quantum-Program-Synthesis-23), we post our experiment data, including target programs that we synthesized and the synthesis results, so that they can be used as benchmarks in later work

Qiskit [18]; it is one of the most widely used libraries for quantum programming, under active maintenance and updates, and its codes are publicly available.

We synthesized the targets in S_{test}, the programs generated in phase 1 and set aside for testing. In particular, we randomly selected 100 programs from S_{test} for each $N_{qubit} = 2$–5. We also synthesized known functions that are used as benchmarks in previous studies [6], [13]; these functions include controlled gates (i.e., controlled-X, $\sqrt{X}$, Y, Z, H, P, and T), W, *Toffoli*, *Fredkin*, *Peres*, quantum *OR* gates, the Grover's algorithm, quantum Fourier transform, and a reversible full adder. Many of these functions are meant to be used as building blocks for large programs.

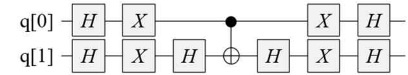
The proposed system used the 20 operators learned in phase 1. We also configured the system, such that it begins with the operators with probabilities $\geq T_{operator} = 0.5$ and explores programs up to $S_{stage_synthesis} = 7$ stages. Qiskit used $\{I, R_X, R_Y, R_Z, CNOT\}$ as building blocks, as required by its synthesis rules. We set the optimization level of Qiskit to the highest possible value of 3. This setting enabled Qiskit to merge consecutive operators using the most of predefined rules, resulting in the smallest possible number of operators.

To compare the quality of synthesis results, we used two measures that have been used in the previous studies [6], [13]: the numbers of (i) stages and (ii) operators. For (ii), the number of operators, we counted an n-qubit operator as n. For example, if a program contains one 1-qubit operator X and one 2-qubit operator *CNOT*, then we consider that the program uses $1+2=3$ operators. This is because an operator with a larger n is considered more expensive in general. We counted the operator I as 0, since it corresponds to an empty place. In addition to the two quality measures, we recorded the time required to synthesize the programs. When the time exceeds 3 hours, we terminated the synthesis process and marked the case as a failure.

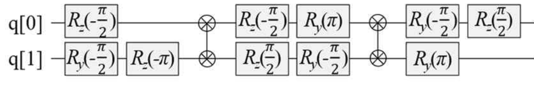
Fig. 6 shows the cumulative distribution of the three measures for the proposed system and Qiskit. Each row has three subfigures that collectively correspond to a distinct N_{qubit}, ranging from 2 to 5. The three subfigures present stage count, operator count, and synthesis time. The horizontal axes list the three measures in log scale and the vertical axes indicate the cumulative percentage of programs. On the horizontal axes, the F's in the far right indicate failures.

Overall, the proposed system used fewer stages and operators than Qiskit at the expense of synthesis time. This trend became more evident as N_{qubit} increased. The proposed system used 1.9-, 20.0-, 85.5-, and 398.7-times fewer stages and 1.8-, 12.0-, 33.1-, and 116.1-times fewer operators than Qiskit as N_{qubit} progressed from 2, 3, 4, and 5, respectively. In the meantime, the average synthesis times of the proposed system grew from 39.3 sec, 54.5 sec, 334.5 sec (5.58 min), and 779.2 sec (13.0 min) for $N_{qubit} = 2$, 3, 4, and 5, respectively, whereas those of Qiskit remained within 10.0 sec.

We identified the primary factor behind the reduced number of stages and operators in the proposed system, which is outlined below. Qiskit uses the same set of operators and continues to stack them in the predefine manner that gradually reduces the distance from the target. Depending on the target, its distance varies greatly as well as the number of operators required. In contrast, the proposed system inspects the target and accordingly selects from a spectrum of operators the most suitable subset. Often, there exist several different subsets that implement the same target; among the subsets, the system is trained to prefer those with low costs. We illustrate the differences in Fig. 7 and Fig. 8. Each figure contains three subfigures (a)–(c) showing different implementations of the same target, and the targets are Grover's algorithm with 2 qubits (Fig. 7) and Peres gate with 3 qubits (Fig. 8). The subfigures (a) present reference implementations in the literature, where Fig. 7(a) is hand optimized [1] and Fig. 8(a) is synthesized by the method presented in [6]. The subfigures (b) and (c) show synthesis results of the proposed system and Qiskit, respectively. The proposed system generally found implementations with fewer stages and operators than the others. This is because it selects a different subset of operators that are necessary and cost-efficient for each particular target (i.e., $\{CNOT, H, Y, Z\}$ for Grover's and



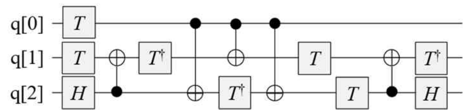
(a) Reference in literature [1] (7 stages, 12 operators)



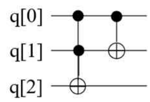
(b) Proposed system (3 stages, 6 operators)

(c) Qiskit (8 stages, 14 operators)

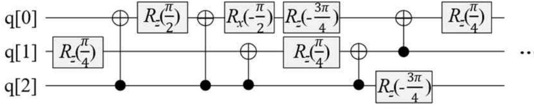
FIGURE 7. Synthesis results for Grover's algorithm with proposed and existing systems.



(a) Reference in literature [6] (10 stages, 19 operators)



(b) Proposed system (2 stages, 5 operators)

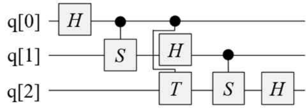


(c) Qiskit (16 stages, 30 operators – last 7 stages omitted)

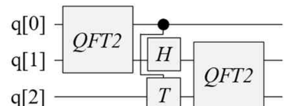
FIGURE 8. Synthesis results for Peres gate with proposed and existing systems.

{*CCNOT*, *CNOT*} for Peres). Note that it is not easy for humans to identify an optimal subset of operators, since there exist many different targets and operators; thus, the proposed system utilizes machine learning. Qiskit tended to use more stages and operators. It utilizes the same set of operators and places them in a predetermined manner, which is not always optimal for a diverse range of targets. In fact, Qiskit uses an additional stage that finds a more optimal implementation through stochastic search and merging equivalent operators into one; however, it is applied after the target is decomposed into $1-2$ qubit operators, rather than at an earlier stage, and thus is not sufficient to find a solution more concise than that of the proposed system.

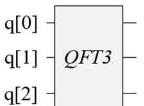
The operator selection in the proposed system has an extra positive outcome that it adapts to changes in the operator pool (input 1). This is illustrated in Fig. 9. The three subfigures show changes in the synthesis results with the inclusion of additional operators. The target remained the same as quantum Fourier transform with 3 qubits. When we used the first 16 operators in input 1 plus {*CS*, *CT*}, the system selected {*I*, *H*, *CS*, *CT*} and synthesized a program with 5 stages and 9 operators (Fig. 9(a)). When we added *QFT2* to the operator pool, the system utilized this new operator to create a program with fewer stages and operators (Fig. 9(b)). Finally, when we added *QFT3*, the system recognized that this operator is an exact match of the target and thus generated the simplest implementation (Fig. 9(c)). In all of the three cases, the operator selections were based on the machine-learned knowledge about the operators.



(a) First 16 operators in input 1+{*CS*,*CT*} (5 stages, 9 operators)



(b) First 16 operators+{*CS*,*CT*,*QFT2*} (3 stages, 7 operators)



(c) First 16 operators+{*CS*,*CT*,*QFT2*,*QFT3*} (1 stage, 3 operators)

FIGURE 9. Synthesis results of proposed system for quantum Fourier transform with different operator pools.

We now explain two main reasons why the synthesis times grew more quickly in the proposed system than in Qiskit. We also suggest ways to suppress the growth rate. The first reason is that the operator selection in the proposed system becomes inaccurate as more qubits are used, and thus the system must go through more iterations until all required operators are selected. With more qubits, target matrices become larger and more diverse. Therefore, accurately learning these matrices requires more complicated machine-learning models and larger training data, although these solutions come at the cost of increased learning time. The second reason is that the proposed method needs to explore various arrangements, even when all necessary operators are correctly selected. With more qubits, more diverse arrangements exist to explore, and thus more time is required to discover a target implementation. We were able to reduce the time from hours to minutes by disregarding duplicate arrangements, as described in Sections III-B and C. One way to further reduce the time is to learn and predict likely locations of selected operators. For example, when the system selects operators {H, X}, it can also predict that X on q[0] and H on q [1] is more likely to produce the target than other arrangements. We plan to explore this method in future work.

As for failure rates, the proposed system failed to synthesize 16 targets out of 421 (3.80%), and Qiskit failed to synthesize 11 targets (2.61%). In the proposed system, most failures occurred when the synthesis time exceeded the 3-hour limit. In particular, the system tended to produce more failures as N_{qubit} increased (i.e., we observed 2, 4, 3, and 7 failures for $N_{qubit} = 2, 3, 4,$ and 5, respectively). This was because the number of arrangements to explore increases with N_{qubit}, as well as the time required to identify a target. In Qiskit, most failures occurred due to an internal error, 'diagonalization failure.' We reported these cases to the developer community and hope that the issues are resolved in later versions.

To summarize, the proposed method can be used as an alternative to the existing system, when synthesizing a more compact program is a priority and an increase in synthesis time can be tolerated. In other words, one can choose between the proposed and existing methods, considering acceptable time and program size. The proposed system can also be applied to assess the feasibility of introducing a new operator, either at the user or hardware level. By synthesizing different programs with the new operator, we can determine whether it contributes to building a diverse range of programs and if it is worthwhile to incorporate the new operator.

V. CONCLUSION

We propose a method to synthesize programs for quantum computers, given a set of arbitrary operators. The method explores various programs with the operators and learns to distinguish the types of programs that each operator can implement. Based on the knowledge, the system selects a subset of operators necessary to implement a target and arranges the selected operators to match the target. We evaluated the system by requesting 400 randomly selected programs and several benchmarks that require 2–5 qubits. The system successfully synthesized most of the programs. Although the synthesis times increased from seconds to minutes as the number of qubits increased, the resulting programs were more compact than those of an existing system, using 0.79% and 2.45% of stages and operators on average, respectively.

We plan to apply the system to synthesizing programs with more than 5 qubits. The major obstacle is that synthesis time

grows quickly as more qubits are used, since the system needs to explore more diverse programs. We have been trying to learn and utilize additional information beyond operator characteristics to decrease synthesis time. One such candidate is to learn proper positions to place operators. Using this knowledge, the system can predict how to best position selected operators and does not need to explore many unnecessary arrangements. We observed encouraging results and intend to perform more extensive experiments. We also plan to apply different machine-learning algorithms, such as the extreme learning machine, to further reduce training time and increase learning accuracy.

REFERENCES

[1] A. Adedoyin, J. Ambrosiano, P. Anisimov, A. Bartschi, W. Casper, G. Chennupati, C. Coffrin, H. Djidjev, D. Gunter, S. Karra, and N. Lemons, ''Quantum algorithm implementations for beginners,'' *ACM Trans. Quantum Comput.*, vol. 3, no. 4, pp. 1–92, Jul. 2022, doi: 10.1145/3517340.

[2] J. Vos, *Quantum Computing in Action*. Shelter Island, NY, USA: Manning, 2022.

[3] C. Bernhardt, *Quantum Computing for Everyone*. Cambridge, MA, USA: MIT Press, 2020.

[4] T. Atkinson, A. Karsa, J. Drake, and J. Swan, ''Quantum program synthesis: Swarm algorithms and benchmarks,'' in *Proc. Eur. Conf. Genetic Program.*, Leipzig, Germany, 2019, pp. 19–34.

[5] A. Zulehner and R. Wille, *Introducing Design Automation for Quantum Computing*. Berlin, Germany: Springer, 2020.

[6] M. Amy, D. Maslov, M. Mosca, and M. Roetteler, ''A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits,'' *IEEE Trans. Comput.-Aided Design Integr.*, vol. 32, no. 6, pp. 818–830, Jun. 2013, doi: 10.1109/TCAD.2013.2244643.

[7] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, ''Elementary gates for quantum computation,'' *Phys. Rev. A, Gen. Phys.*, vol. 52, pp. 3457–3467, Nov. 1995, doi: 10.1103/PhysRevA.52.3457.

[8] P. O. Boykin, T. Mor, M. Pulver, V. Roychowdhury, and F. Vatan, ''A new universal and fault-tolerant quantum basis,'' *Elsevier Inf. Process. Lett.*, vol. 75, no. 3, pp. 101–107, Aug. 2000, doi: 10.1016/S0020-0190(00)00084-3.

[9] J. J. Vartiainen, M. Möttönen, and M. M. Salomaa, ''Efficient decomposition of quantum gates,'' *Phys. Rev. Lett.*, vol. 92, no. 17, Apr. 2004, Art. no. 177902, doi: 10.1103/PhysRevLett.92.177902.

[10] Y. Xiao, S. Nazarian, and P. Bogdan, ''A stochastic quantum program synthesis framework based on Bayesian optimization,'' *Sci. Rep.*, vol. 11, no. 1, Jun. 2021, Art. no. 13138, doi: 10.1038/s41598-021-91035-3.

[11] M. G. Davis, E. Smith, A. Tudor, K. Sen, I. Siddiqi, and C. Iancu, ''Towards optimal topology aware quantum circuit synthesis,'' in *Proc. IEEE Int. Conf. Quantum Comput. Eng. (QCE)*, Denver, CO, USA, Oct. 2020, pp. 223–234.

[12] W. Scherer, *Mathematics of Quantum Computing*. Berlin, Germany: Springer, 2019.

[13] E. Younis, K. Sen, K. Yelick, and C. Iancu, ''QFAST: Conflating search and numerical optimization for scalable quantum circuit synthesis,'' in *Proc. IEEE Int. Conf. Quantum Comput. Eng. (QCE)*, Broomfield, CO, USA, Oct. 2021, pp. 232–243.

[14] V. V. Shende, S. S. Bullock, and I. L. Markov, ''Synthesis of quantum-logic circuits,'' *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 25, no. 6, pp. 1000–1010, Jun. 2006, doi: 10.1109/TCAD.2005.855930.

[15] M. Saeedi, M. Arabzadeh, M. S. Zamani, and M. Sedighi, ''Block-based quantum-logic synthesis,'' *Quantum Inf. Comput.*, vol. 11, nos. 3–4, pp. 262–277, Mar. 2011, doi: 10.26421/QIC11.3-4-6.

[16] P. Niemann, R. Wille, and R. Drechsler, ''Efficient synthesis of quantum circuits implementing Clifford group operations,'' in *Proc. 19th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Singapore, Jan. 2014, pp. 483–488.

[17] P. Niemann, R. Wille, and R. Drechsler, ''Improved synthesis of Clifford+T quantum functionality,'' in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Dresden, Germany, Mar. 2018, pp. 597–600.

[18] *Qiskit: Open-Source Quantum Development*. Accessed: Feb. 8, 2023. [Online]. Available: https://qiskit.org/documentation/apidoc/transpiler.html

[19] R. Iten, O. Reardon-Smith, E. Malvetti, L. Mondada, G. Pauvert, E. Redmond, R. S. Kohli, and R. Colbeck, ''Introduction to UniversalQ-Compiler,'' 2019, *arXiv:1904.01072*.

[20] S. Bravyi, R. Shaydulin, S. Hu, and D. Maslov, ''Clifford circuit optimization with templates and symbolic Pauli gates,'' *Quantum*, vol. 5, no. 1, pp. 580–595, Nov. 2021, doi: 10.22331/q-2021-11-16-580.

[21] R. Loredo, *Learn Quantum Computing With Python and IBM Quantum Experience*. Birmingham, U.K.: Packt, 2020.

[22] E. Johnston and N. Harrigan, *Programming Quantum Computers: Essential Algorithms and Code Samples*. Sebastopol, CA, USA: O'Reilly, 2019.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.

[24] K. Hara, D. Saito, and H. Shouno, ''Analysis of function of rectified linear unit used in deep learning,'' in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Killarney, Ireland, Jul. 2015, pp. 1–8.

[25] *TensorFlow: An Open-Source Machine Learning Library for Research and Production*. Accessed: Feb. 8, 2023. [Online]. Available: https://www.tensorflow.org/

[26] *NumPy: The Fundamental Package for Scientific Computing With Python*. Accessed: Feb. 8, 2023. [Online]. Available: https://numpy.org/

SIHYUNG LEE received the B.S. (summa cum laude) and M.S. degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), South Korea, in 2000 and 2004, respectively, and the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University (CMU), USA, in 2010. From 2010 to 2011, he was a Postdoctoral Researcher with the Network Management Group, IBM Thomas J. Watson Research Center, USA. From 2011 to 2019, he was a Professor with the Department of Information Security, Seoul Women's University, South Korea. Since 2019, he has been a Professor with the School of Computer Science and Engineering, Kyungpook National University. His research interests include pattern mining from social network traffic and program synthesis for classical and quantum computers.

SEUNG YEOB NAM (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology (KAIST), in 1997, 1999, and 2004, respectively. From 2004 to 2006, he was a Postdoctoral Research Fellow with the CyLab, Carnegie Mellon University. From 2006 to 2007, he was a Postdoctoral Researcher with the Department of Electrical Engineering and Computer Science, KAIST. In 2007, he joined the Department of Information and Communication Engineering, Yeungnam University, Gyeongsan, South Korea, where he is currently a Professor. His research interests include network security, blockchain, network management, and wireless networks.