ABSTRACT

Title of Dissertation: PROGRAM SYNTHESIS FOR QUANTUM APPLICATIONS

Haowei Deng
Doctor of Philosophy, 2025

Dissertation Directed by: Professor Xiaodi Wu
Department of Computer Science

Quantum computing has the potential to revolutionize various fields by solving problems intractable for classical computers. However, developing efficient quantum programs remains challenging due to the unique constraints of quantum systems, including noise, limited qubit connectivity, and hardware variability. Unlike classical programming, where high-level abstractions and optimized compilers ease development, quantum programming still relies heavily on low-level circuit representations, making manual implementation complex and error-prone. Program synthesis, an approach that automatically generates programs satisfying given specifications, offers a promising solution by optimizing quantum circuits while minimizing human effort. However, applying classical program synthesis techniques to quantum computing presents unique challenges across different abstraction levels. The development of novel synthesis and verification applications specifically tailored for quantum programming is highly desired.

In this thesis, we introduce three novel quantum program synthesis frameworks addressing key challenges across different levels of quantum computing. First, we present QSynth, the first

framework for synthesizing unitary quantum programs with recursive structures, enabling efficient automated verification. Second, we introduce MQCC, a quantum meta-programming framework that balances trade-offs among multiple constraints specific to targeted applications and hardware. Finally, we propose NuQes, a neuro-symbolic quantum error correction (QEC) code synthesis framework that leverages heuristic functions generated by large language models (LLMs) to optimize QEC code design. Together, these frameworks advance quantum program synthesis by improving efficiency, reducing errors, and enhancing scalability.

PROGRAM SYNTHESIS FOR QUANTUM APPLICATIONS

by

Haowei Deng

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2025

Advisory Committee:
Professor Xiaodi Wu, Chair/Advisor
Dr. Alexander Barg, Dean's Representative
Professor Milijana Surbatovich
Professor Leonidas Lampropoulos
Professor Runzhou Tao

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost, I would like to express my sincere gratitude to my Ph.D. advisor, Xiaodi Wu. His unique insights have deepened my understanding of computing technology development and guided us toward exploring quantum computing from important yet underexamined perspectives. His mentorship has sharpened my technical skills and strategic thinking, empowering me to develop innovative solutions to complex challenges while shaping my approach to scientific research. Beyond academia, our five years of collaboration have left a lasting impact on me, teaching me to focus on what truly matters despite countless distractions. These experiences have profoundly influenced my outlook on life and will continue to guide me in the future.

I want to thank the dissertation committee overseeing this work. Committee chair Xiaodi Wu, dean's representative Alexander Barg, and members Milijana Surbatovich, Leonidas Lampropoulos, and Runzhou Tao offered invaluable feedback and insights

The Ph.D. program at the University of Maryland provided me with access to numerous high-quality courses, which generated opportunities to learn from outstanding faculty members and engage with bright classmates. I sincerely thank them because the valuable knowledge gained from these well-structured courses was instrumental in my research and contributed significantly to the composition of this dissertation.

Over the past few years, I have had the privilege of collaborating with numerous talented and amiable researchers. These collaborations have enabled me to carry out research that has culminated in this dissertation. I want to express my gratitude to all these coauthors: Runzhou Tao, Yuxiang Peng, Suying Liu, Yingkang Cao, Charles Cao, Micheal Hicks, Yuxin Wang, and Zihan Xia. Not only have they made significant research contributions, but they have also brought immense joy to my life. I would like to extend my gratitude beyond my direct collaborators to the members of many related organizations. These include Xiaodi Wu's research group, our institute Joint Center for Quantum Information and Computer Science, my research fields (mainly programming language community and quantum computing community), and the University of Maryland. The discussions with these great people have sparked a multitude of exciting research ideas and boundless happiness.

Finally, I want to express my profound gratitude to my family, whose endless support has been crucial in my exploration pursuing academic aspirations. Their continuous guidance and nurturing have shaped the person I am today.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Program Synthesis For Quantum Computing Applications

Quantum computing has gained significant attention due to its potential to provide asymptotically faster solutions to certain computing problems compared to the best-known classical algorithms [8]. A scalable, functioning quantum computer is believed to hold promise for solving complex computational challenges in fields such as scientific discovery, materials research, chemistry, and drug design, among others [9, 10, 11, 12]. Quantum programming is a key step in implementing these quantum computing applications on the near-term quantum hardware. However, developing efficient quantum programs remains a challenge due to the unique constraints and complexities of quantum systems.

Quantum computing introduces a fundamentally different programming paradigm from classical computing, requiring programmers to be familiar with unintuitive quantum knowledge. Writing efficient quantum programs is complex since it requires programmers to deeply understand quantum mechanics and carefully optimize various hardware-dependent factors, including gate fidelity, decoherence time, and qubit connectivity constraints. Unlike classical software development, where high-level abstractions and optimized compilers ease the programming burden, most existing quantum programming languages are still using the low-level circuit abstraction[13, 14, 15, 16], making it difficult to write efficient and error-free programs manu-

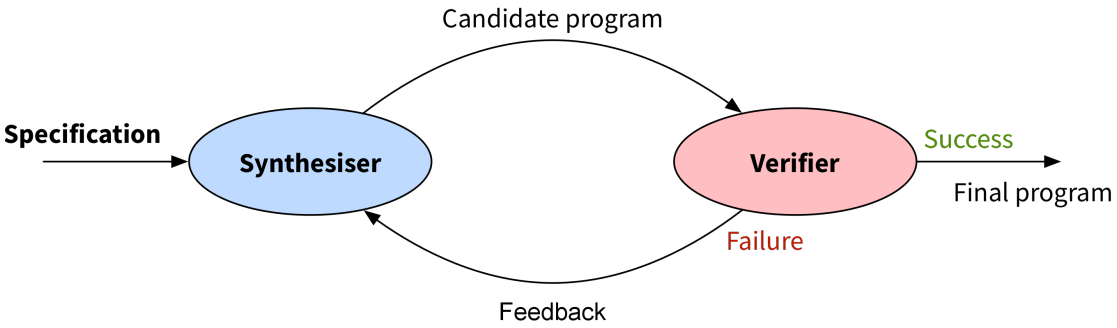


Figure 1.1: General idea of inductive program synthesis

ally. Furthermore, quantum hardware is noisy and limited in resources, making it essential to find program implementations that minimize errors, reduce execution time, and optimize qubit utilization.

Program synthesis, which automatically generates programs that satisfy a given specification, presents a compelling solution to these challenges by enabling the generation of quantum circuits that are both correct and efficient without requiring programmers to handle every low-level detail manually. Fig. 1.1 shows the overview of classical inductive program synthesis, which consists of three core components: a specification, a synthesizer, and a verifier. The specification provided by the users defines the constraints or requirements of the target program. The synthesizer generates candidate programs iteratively using techniques such as syntax-guided search, large language models (LLMs), or stochastic sampling. These candidates are then evaluated by the verifier, which checks their correctness against the specification using methods like symbolic execution, constraint solving, automated testing, or manual inspection. If a candidate program fails verification, feedback is provided to refine and improve subsequent program generations. By automating program construction and validation, classical program synthesis significantly accelerates the software development process and enhances program correctness. This approach is

particularly valuable in domains like quantum computing, where designing optimized programs manually is highly complex and error-prone.

By leveraging synthesis techniques, quantum developers can focus on high-level algorithm design rather than tedious circuit optimization, significantly improving quantum software developers' productivity. Moreover, program synthesis can lower the entry barrier for newcomers, allowing those who do not have quantum expertise to develop and experiment with quantum applications. Additionally, program synthesis can explore alternative implementations that might be overlooked by human programmers, potentially leading to more resource-efficient solutions that require fewer gates, fewer qubits or other resources that are limited in the current quantum hardware. This can bridge the gap between theoretical algorithms and their efficient realization on real-world quantum hardware.

Given that program synthesis is highly beneficial for quantum programming, several quantum unique challenges arise when building the three core components (specification, synthesizer and verifier) to directly apply traditional program synthesis techniques to this domain. When synthesizing high-level quantum programs with recursion or loop, the main challenge arises from automated verification. Quantum programs operate on quantum states, which are represented as complex-valued vectors in a high-dimensional Hilbert space. Verifying recursive quantum programs requires reasoning about matrices, complex numbers, and formulas with a non-fixed number of terms, all of which have very limited support in existing verification tools. At the quantum circuit level, which can be regarded as the assembly level of quantum programs, a synthesis framework must allow users to specify and balance different trade-offs, as current quantum hardware has limited resources and must meet various hardware constraints. The synthesizer should explore multiple solutions, carefully balancing efficiency, accuracy, and hardware limita-

tions. Lastly, at the hardware level, synthesizing solutions like Quantum Error Correction (QEC) codes is particularly difficult due to the exponentially growing search space as the size of the system increases. Additionally, the unintuitive nature of quantum mechanics makes it challenging to manually design heuristic functions, which are commonly used to improve the efficiency of synthesis frameworks. Overcoming these challenges requires novel approaches that integrate heuristic learning and search strategies tailored for quantum computing. These fundamental differences between classical and quantum computation necessitate the development of novel synthesis and verification applications tailored specifically for quantum programming.

1.2 Dissertation Overview

In this thesis, we study synthesis frameworks designed to automatically generate efficient and practical solutions for synthesizing quantum programs at the high-level language, assembly language, and hardware control level language. Each framework addresses unique quantum-specific challenges within a particular application domain, reducing the effort required by human programmers to develop effective solutions.

1.2.1 Inductive Unitary Quantum Program Synthesis

To help ease the programming of quantum computers, circuit synthesis techniques have been proposed to automatically generate quantum circuits [17, 18, 19, 20, 21, 22, 23].

Unfortunately, these synthesis frameworks do not support any loop or recursive structure in the desired program and underperform when the number of qubits of the quantum circuit to synthesize is as large as 5. Moreover, synthesizing a quantum circuit will even fail at the

start because it is impossible to write the exponential-sized matrix specifying the synthesis goal. Additionally, the circuit generated by the synthesizer is hard to understand by humans, which prevents potential human customization to the circuit after synthesis.

In Chapter 2, we propose QSynth, the first synthesis framework for recursive unitary quantum programs. In contrast to previous frameworks focusing on fixed-size circuits, the synthesis target of QSynth is *inductively-defined families of quantum circuits* without mid-circuit measurements (i.e. unitaries). QSynth can exploit the inductive structure of quantum programs and, compared to previous circuit synthesis methods, 1) generate quantum circuits with an arbitrary number of qubits, 2) allow user to use a single input-output style specification for synthesis, and 3) produces more readable and structured quantum program code that are easy to customize by human.

To enable the synthesis of inductive quantum programs, there are three quantum unique challenges. First, the synthesis framework requires a representation of the specification of quantum programs. Previous methods such as quantum Hoare triple [24], path-sum [25], and tree automaton [26] are all defined on fixed-dimension quantum systems and thus cannot be directly applied. Naive extensions of these representations with a variable qubit size do not work because symbolically representing matrices and automata is hard, limiting their usage for synthesis and verification. We introduce a specification language in QSynth, which enables intuitive input-output style specification for quantum programs and supports all path-sum quantum states with an arbitrary number of qubits. We also propose the *hypothesis-amplitude* ($h - \alpha$) specification that uses two functions to specify quantum programs. The specification written in QSynth-spec language will be compiled into $h - \alpha$ specification to use in the verification step, which avoids SMT-unfriendly matrices (or graphical representations like automata).

The second challenge is to find a subset of quantum programs that is both expressive enough

and efficient to synthesize. Previous programming languages for verifying quantum programs are either low-level circuit languages that do not support inductive structure (e.g., SQIR [27]) or high-level languages that do not have a detailed structure of unitaries (e.g., Quantum-while language used in Quantum Hoare Logic [24]). We design the Inductive-SQIR (ISQIR) language, an extension of SQIR that supports inductively defined quantum programs. We also define a Hoare-type verification logic that verifies an ISQIR program with respect to an $h - \alpha$ specification.

Finally, QSynth needs an automated verifier for inductive quantum programs. This requires reasoning about matrices, complex numbers, and formulas with a non-fixed number of terms, all of which have very limited support in SMT solvers. To solve this challenge, we define *parameterized path-sum amplitudes*, which, together with a sparsity constraint, can be efficiently encoded in SMT solvers. We further show that this set of expressions is expressive enough to specify and synthesize many quantum programs.

1.2.2 Automating NISQ application design

Near-term, intermediate-scale quantum computing (NISQ) devices have few quantum bits (qubits), and these are prone to errors from several sources. General-purpose error correction techniques [28, 29, 30, 31] consume a substantial number of qubits, so they are not a practical remedy. As a result, the design of NISQ applications needs to explore ways to optimize the efficiency and/or reliability of quantum circuits by balancing competing tradeoffs.

Many optimization frameworks have been proposed, including syndrome extraction optimization [32, 33], approximate circuits of important quantum subroutines [34], program rescheduling for best leverage target architectural constraints and so on. Unfortunately, each only offers

one-off improvements. The works either lack algorithmic/automated support entirely, or when they have it, this support cannot be composed or combined easily, due to conflicting tradeoffs that themselves would need balancing.

In Chapter 3, we propose **Meta Quantum Circuits** with **Constraints** (MQCC), the first general-purpose approximate computing framework for quantum programs. MQCC is a framework that makes it easy to design, implement, and experiment with optimizations, and to support *programming* their customized composition while leveraging automated reasoning. Crucially, MQCC allows users to express and optimize a *variety* of metrics and combine several constraints together. Because we are in a stage of rapid development for quantum hardware, application designers need to balance trade-offs among many emerging or even unknown factors. MQCC allows programmers to write (or reuse) routines to compute various **object attributes** of quantum programs, making it easy and flexible to give specifications for the target programs. Object attributes can be reused or shared for future application design, which eases the effort of designing NISQ applications. MQCC allows programmers to express their application as a succinct collection of normal quantum circuits stitched together by a set of (manually or automatically) added meta-level choice variables, whose values are constrained according to a programmable set of quantitative optimization criteria. MQCC's verifier generates the appropriate constraints and solves them via an SMT solver to produce a solution program, and then MQCC's compiler will generate the final optimized and runnable program following the goals and constraints given by the programmers.

1.2.3 Efficient Quantum Error Correction Code Synthesis

Quantum Error Correction [28] (QEC) is essential for the reliable operation of quantum computing systems, where physical qubits are prone to noise and decoherence. Large-size QEC codes with high encoding rates are particularly desirable, as they enhance the fault-tolerance and efficiency of quantum computations. However, the search for effective QEC codes has been limited to a small subspace of symbolic representations, leaving vast regions of the possible code space unexplored. This limitation arises due to the inherent complexity of generating and evaluating QEC codes, making the discovery of high-performance codes a significant challenge.

Generating large-size QEC codes is non-trivial due to two primary challenges. First, a code family with a symbolic representation implies an exponential search space. As the number of qubits increases, the number of possible QEC codes grows exponentially, making exhaustive searches impractical. Second, evaluating a large-size QEC code requires substantial computational resources and time. Each potential code must be verified to ensure it meets the necessary error correction criteria, adding to the overall difficulty of code synthesis. These factors contribute to the bottleneck in the development of advanced QEC codes and necessitate novel approaches for efficient code generation.

Heuristic search is a powerful technique for finding good solutions in exponentially large search spaces. However, designing an effective heuristic function for QEC code search is particularly challenging due to the unintuitive nature of quantum mechanics. The intricate and non-classical behaviors of quantum systems make it difficult to manually define heuristics that guide the search effectively. This is where Large Language Models (LLMs) can play a crucial role. Recent advancements in artificial intelligence, particularly in Large Language Models (LLMs),

have demonstrated remarkable capabilities in synthesizing symbolic representations.

In Chapter 4, we propose NeUrosymbolic Quantum Error correction code Search framework (NuQES) for synthesizing QEC codes with a low logical error rate and high encoding rate. The key idea of NuQES to address the challenges of QEC code synthesis is the adoption of a two-step synthesis approach. First, FunSearch, a LLM-based function synthesis framework [35], is utilized to synthesize a heuristic function based on a small QEC code database that can be explored using an enumerative search. Second, these heuristics are applied to explore the exponentially large QEC code space, significantly improving the feasibility of discovering high-quality large-size QEC codes. This approach leverages both symbolic search strategies and heuristic-driven exploration, offering a scalable solution to the QEC code synthesis problem. We utilize NuQES to search (1) Bivariate Bicycle (BB) LDPC Code[3]; (2) QLego codes[36]. NuQES successfully found two $[[170, 16, 10]]$ and $[[288, 12, 22]]$ BB codes, which have higher encoding rates and error thresholds than the best codes found by Bravyi et al. [3].

1.3 Preliminaries

1.3.1 Quantum States

A quantum state consists of one or more *quantum bits*. A quantum bit (or *qubit*) can be expressed as a two dimensional vector $\binom{\alpha}{\beta}$ such that $|\alpha|^2 + |\beta|^2 = 1$. The α and β are called *amplitudes*. We frequently write this vector as $\alpha|0\rangle + \beta|1\rangle$ where $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ are *basis states*. A state written $|\phi\rangle$ is called a ket, following Dirac's notation. When both α and β are non-zero, we can think of the qubit as being "both 0 and 1 at once," a.k.a. a *superposition*. For example, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is an equal superposition of $|0\rangle$ and $|1\rangle$. A qubit is only in superposition

until it is *measured*, at which point the outcome will be 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$.

A quantum state with n qubits is represented as vector of length 2^n. We can join multiple qubits together by means of the *tensor product* ($\otimes$) from linear algebra. For convenience, we write $|x_0\rangle \otimes |x_1\rangle \otimes \cdots \otimes |x_m\rangle$ as $|x_0 x_1 ... x_m\rangle$ for $x_i \in \{0, 1\}, 0 \leq i \leq m$; we may also write $|k\rangle$ where $k \in \mathbb{N}$ is the decimal interpretation of bits $|x_0 x_1 ... x_m\rangle$. For example, a 2-qubit state is represented as a $2^2 = 4$ length vector where each component corresponds to (the square root of) the probability of measuring $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, respectively. We may also write these four kets as $|0\rangle, |1\rangle, |2\rangle, |3\rangle$. Sometimes a multi-qubit state cannot be expressed as the tensor product of individual qubits; such states are called *entangled*. One example is the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, known as a *Bell pair*.

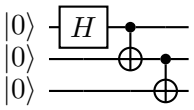
1.3.2 Quantum Programs

Quantum programs are composed of a series of *quantum operations*, each of which acts on a subset of qubits in the quantum state. Quantum operations can be expressed as matrices, and their application to a state is expressed as matrix multiplication. For example, the *Hadamard* operator H on one qubit is expressed as a matrix $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. Applying H to state $|0\rangle$ yields state $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. n-qubit operators are represented as $2^n \times 2^n$ matrices. For example, the $CNOT$ operator over two qubits is expressed as the $2^2 \times 2^2$ matrix shown at the right.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

In the standard presentation, quantum programs are expressed as *circuits*, as shown in

Fig 1.2(a). In these circuits, each horizontal wire represents a qubit and boxes on these wires indicate quantum operations, or *gates*. The circuit in Fig 1.2(a) has three qubits and three gates: the *Hadamard* (`H`) gate and two *controlled-not* (`CNOT`) gates. The semantics of a gate is a *unitary matrix*. Applying a gate to a state is tantamount to multiplying the state vector by the gate's matrix. The matrix corresponding to the circuit in Fig 1.2(a) is shown in Fig 1.2(c), where I is the 2×2 identity matrix.



(a) Quantum circuit

```
H 0;
CNOT 0 1;
CNOT 1 2;
```

(b) Unitary SQIR

$$(I \otimes CNOT) \times (CNOT \otimes I) \times (H \otimes I \otimes I)$$

(c) Matrix expression

Figure 1.2: Example quantum program: 3-qubit GHZ state preparation.

1.3.3 Quantum Assembly Language (QASM)

A quantum circuit can be specified using the "quantum assembly language" (QASM) [37, 38]. QASM is a simple text language that describes quantum circuits as a sequence of gate operations on numbered qubits. OpenQASM [39] provides a bit more high-level structure, while still being compatible with modern hardware [40]. Fig 1.3(a) is an example. The only storage

```
qreg q[2];  creg c[2];
gate cz a,b { h b; cx a,b; h b; }
x q[0];     cz q[0],q[1];
measure q[1] -> c[1];
if (c==2) x q[0];
```

(a)

```
CX r[0],r[1];
h q[0];
barrier r,q[0];
h r[1];
```

(b)

Figure 1.3: OpenQASM Examples.

types of OpenQASM (version 2.0) are classical and quantum registers, which are one-dimensional arrays of bits and qubits, respectively. The statement `qreg name[size];` declares an array of

qubits while the statement `creg name[size];` declares a `size`-bit classical register. The qubits are initialized as $|0\rangle$ and the classical bits are initialized to 0.

OpenQASM supports a built-in set of arbitrary single-qubit gates with `CNOT` (written `cx`) as the sole two-qubit gate. A programmer can define different gates using a subroutine-like mechanism with keyword `gate`; the example code defines a new `cz` gate which consists of two Hadamard gates and one CNOT gate. The `measure` statement measures a qubit and stores the result in a classical bit. The `if` statement conditionally executes a quantum operation based on the value of a classical register. This register is interpreted as an integer, using the bit at index zero as the low order bit.

OpenQASM allows gate sequential control through a special instruction `barrier`, which prevents reordering gates across its source line. Consider the example in Fig 1.3(b). The instruction `h r[1]` has to wait until all gates on `r[0],r[1],q[0]` before line 3 are finished. In particular, it cannot be executed with `h q[0]` in parallel.

1.3.4 Unitary SQIR

SQIR [27] is a simple quantum language embedded in the Coq proof assistant. SQIR's *unitary fragment* is a sub-language for expressing programs consisting of unitary gates.

Syntax A unitary SQIR program P is a sequence of applications of gates G to qubits q:

$$P := P_1; P_2 \mid G\ q \mid G\ q_1\ q_2 \mid G\ q_1\ q_2\ q_3.$$

$$[\![P_1;\ P_2]\!]_d = [\![P_2]\!]_d \times [\![P_1]\!]_d;$$

$$[\![G\ q_1 \cdots q_i]\!]_d = \begin{cases} apply_i(G,\ q_1, ...,\ q_i,\ d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases},\ i = 1, 2, 3.$$

Figure 1.4: Semantics of unitary SQIR programs, assuming a global register of dimension d. The $apply_k$ function maps a gate name to its corresponding unitary matrix and extends the intended operation to the given dimension by applying an identity operation on every other qubit in the system.

Qubits are referred to by natural numbers that index into a global register. A SQIR program is parameterized by a unitary gate set g (from which G is drawn) and the size n of the global register (i.e., the number of available qubits). In Coq, a unitary SQIR program `U` hence has type `ucom g n`. U has type `ucom` g n, where g identifies the gate set and n is the size of the global register.

```
1 Fixpoint ghz (n : nat) : ucom g (n + 1) :=
2   match n with
3     | 0 ⇒ H 0
4     | S n' ⇒ ghz n'; CNOT n' n
5   end.
```

The Coq function `ghz` on the left recursively constructs a SQIR program, which prepares the $n+1$-qubit GHZ state. When $n = 0$, the program applies the Hadamard gate H to qubit 0. Otherwise, `ghz` calls itself recursively with input $n - 1$ and then applies $CNOT$ to qubits q_{n-1}, q_n. For example, `ghz 2` generates the circuit in Fig 1.2(a).

Semantics The semantics of unitary SQIR is shown in Fig 1.4. A program P is well-typed if every gate's index arguments are within the bounds of the global register and no index is repeated. The program's semantics follows from the composition of the matrices that correspond to each of the applications of its unitary gates. A gate application's matrix needs to apply the identity operation to the qubits not being operated on. This is the purpose of using $apply_1, apply_2$ and

$apply_3$ For example, $apply_1(G_u,\ q_1,\ d) = I_{2^q} \otimes u \otimes I_{2^{(d-q-1)}}$ where u is the matrix interpretation of the gate G_u and I_k is the $k \times k$ identity matrix.

Suppose that M_1 and M_2 are the matrices corresponding to unitary gates P_1 and P_2, which we want to apply to a quantum state vector $|\psi\rangle$. Matrix multiplication is associative, so $M_2(M_1 |\psi\rangle)$ is equivalent to $(M_2 M_1) |\psi\rangle$. Moreover, multiplying two unitary matrices yields a unitary matrix. As such, the semantics of SQIR program $P_1;\ P_2$ is naturally described by the unitary matrix $M_2 M_1$. Fig 1.2(b) shows an example unitary SQIR program for the circuit in Fig 1.2(a).

1.3.5 Path-sum Representation

Path-sum, proposed by recent works on quantum program verification [25, 41], is a representation for describing quantum states based on Feynman's *path integral* formalism of quantum mechanics, which is widely applied to circuit simulation [42, 43] and optimization [44, 45, 46]. The idea of this formalism is to describe a quantum state's amplitude by an integral over all paths leading to that state. In practice, a discrete sum-over-path technique rather than integral is typically used [25, 41, 42, 43, 45, 46, 47, 48, 49]. We can describe a sum-over-path abstractly as a discrete set of paths $T \in \mathbb{Z}_2^m$, together with an amplitude function ψ and a state transformation f, both depending on specific path y, to represent a unitary U:

$$U : |x\rangle \mapsto \sum_{y \in T} \psi(x, y) \, |f(x, y)\rangle \, .$$

All representations based on the sum-over-path in these previous works[25, 41, 42, 47, 48, 49] share in common that the amplitude $\psi(x, y)$ of all possible paths have the same *magnitude* and only the *phase*s are different. These forms of the same *magnitude* but different *phase* are so typical

in many quantum algorithms that they can be used as a succinct representation in the verification, which makes them useful for synthesis purposes.

1.3.6 Tensors, tensor networks, and tensor network diagrams

For our purposes, a *tensor* is a multidimensional array of complex numbers. The *rank* of a tensor is the number of indices. Thus, a rank-0 tensor is scalar (x), a rank-1 tensor is a vector (v_α), and a rank-2 tensor is a matrix ($A_{\alpha\beta}$).

An *index contraction* is the sum over all the possible values of the repeated indices of a set of tensors. For instance, the matrix product

$$C_{\alpha\gamma} = \sum_{\beta=1}^{D} A_{\alpha\beta} B_{\beta\gamma} \tag{1.1}$$

is the contraction of index β, which amounts to the sum over its D possible values. One can also have more complicated contractions, such as this one:

$$F_{\gamma\omega\rho\sigma} = \sum_{\alpha,\beta,\delta,\nu,\mu=1}^{D} A_{\alpha\beta\delta\sigma} B_{\beta\gamma\mu} C_{\delta\nu\mu\omega} E_{\nu\rho\alpha} \ , \tag{1.2}$$

where for simplicity we assumed that contracted indices $\alpha, \beta, \delta, \nu$ and μ can take D different values. As seen in these examples, the contraction of indices produces new tensors, in the same way that e.g. the product of two matrices produces a new matrix. Indices that are not contracted are called *open indices*.

A *Tensor Network* (TN) is a set of tensors where some, or all, of its indices are contracted according to some pattern. Contracting the indices of a TN is called, for simplicity, *contracting*

the TN. The above two equations are examples of TN. In Eq.(1.1), the TN is equivalent to a matrix product, and produces a new matrix with two open indices. In Eq.(1.2), the TN corresponds to contracting indices $\alpha, \beta, \delta, \nu$ and μ in tensors A, B, C and E to produce a new rank-4 tensor F with open indices γ, ω, ρ and σ. In general, the contraction of a TN with some open indices gives as a result another tensor, and in the case of not having any open indices the result is a scalar. This is the case of e.g. the scalar product of two vectors,

$$C = \sum_{\alpha=1}^{D} A_\alpha B_\alpha \ , \tag{1.3}$$

where C is a complex number (rank-0 tensor). A more intrincate example could be

$$F = \sum_{\alpha,\beta,\gamma,\delta,\omega,\nu,\mu=1}^{D} A_{\alpha\beta\delta\gamma} B_{\beta\gamma\mu} C_{\delta\nu\mu\omega} E_{\nu\omega\alpha} \ , \tag{1.4}$$

where all indices are contracted and the result is again a complex number F.

Once this point is reached, it is convenient to introduce a diagrammatic notation for tensors and TNs in terms of *tensor network diagrams*, see Fig.(1.5). In these diagrams tensors are represented by shapes, and indices in the tensors are represented by lines emerging from the shapes. A TN is thus represented by a set of shapes interconnected by lines. The lines connecting tensors between each other correspond to contracted indices, whereas lines that do not go from one tensor to another correspond to open indices in the TN.

Using TN diagrams it is much easier to handle calculations with TN. For instance, the contractions in Eqs.(1.1, 1.2, 1.3, 1.4) can be represented by the diagrams in Fig.(1.6). Also tricky calculations, like the trace of the product of 6 matrices, can be represented by diagrams as in

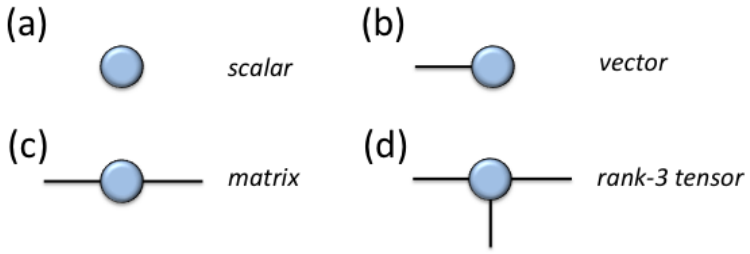


Figure 1.5: (color online) Tensor network diagrams: (a) scalar, (b) vector, (c) matrix and (d) rank-3 tensor

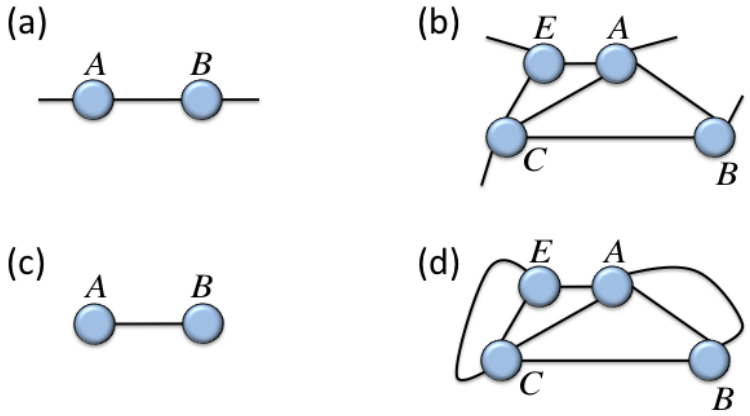


Figure 1.6: (color online) Tensor network diagrams for Eqs.(1.1, 1.2, 1.3, 1.4): (a) matrix product, (b) contraction of 4 tensors with 4 open indices, (c) scalar product of vectors, and (d) contraction of 4 tensors without open indices.

Fig.(1.7). From the TN diagram the cyclic property of the trace becomes evident. This is a simple example of why TN diagrams are really useful: unlike plain equations, TN diagrams allow to handle with complicated expressions in a visual way. In this manner many properties become apparent, such as the cyclic property of the trace of a matrix product. In fact, you could compare the language of TN diagrams to that of Feynman diagrams in quantum field theory. Surely it is much more intuitive and visual to think in terms of drawings instead of long equations. Hence, from now on we shall only use diagrams to represent tensors and TNs.

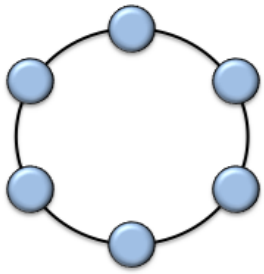


Figure 1.7: (color online) Trace of the product of 6 matrices.

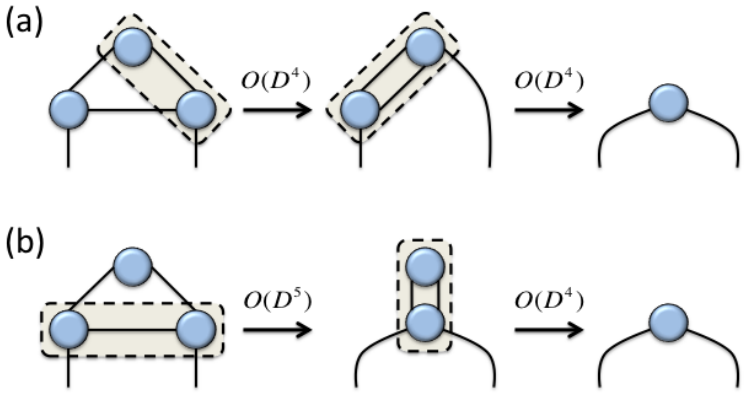


Figure 1.8: (color online) (a) Contraction of 3 tensors in $O(D^4)$ time; (b) contraction of the same 3 tensors in $O(D^5)$ time.

Chapter 2: Quantum Unitary Programs Synthesis with QSynth

2.1 Introduction

Quantum programming is a key step in enabling the various application of quantum computing such as factorization, simulation of physics and optimization. However, programming quantum computers is hard due to unintuitive quantum mechanics. To help ease the programming of quantum computers, circuit synthesis techniques have been proposed to automatically generate quantum circuits [17, 18, 19, 20, 21, 22, 23].

Unfortunately, these synthesis frameworks underperform when the number of qubits of the quantum circuit to synthesize is as large as 5. For example, QFAST [22], a recent quantum circuit synthesis framework, can only synthesize QFT and adder circuit up to 5 qubits. And QSyn [23], a quantum circuit synthesis method based on user-supplied components, needs an average time of 687.5 seconds for solving 4-qubit problems and fails to synthesize a 6-qubit circuit within one hour. These methods cannot scale with the rapid development of qubit numbers in hardware, with more than 1000 qubits by the end of 2023 estimated by IBM [50]. Moreover, synthesizing a quantum circuit will even fail at the start because it is impossible to write the exponential-sized matrix specifying the synthesis goal. For example, QSyn requires 16 input-output state vector pairs as the specification for a 4-qubit Toffoli circuit. Additionally, the circuit generated by the synthesizer is hard to understand by humans, which prevents potential human customization to the

circuit after synthesis.

In this work, we propose QSynth, the first synthesis framework for inductive quantum programs. In contrast to previous frameworks focusing on circuits, the synthesis target of QSynth is *inductively-defined families of quantum circuits* without mid-circuit measurements (i.e. unitaries). QSynth can exploit the inductive structure of quantum programs and, compared to previous circuit synthesis methods, 1) generate quantum circuits with an arbitrary number of qubits, 2) allow user to use a single input-output style specification for synthesis, and 3) produces more readable and structured quantum program code that are easy to customize by human, as illustrated by Figure 2.1.

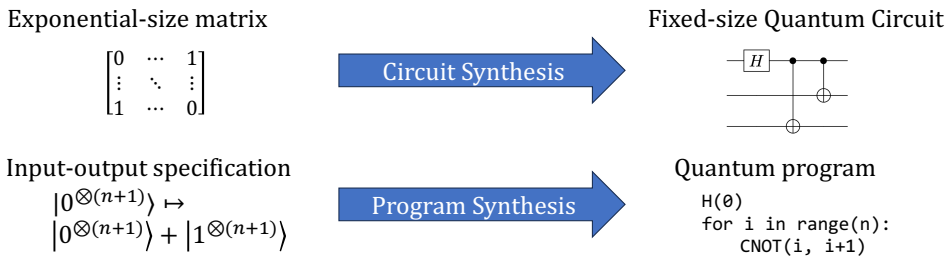


Figure 2.1: Circuit synthesis vs. program synthesis. Circuit synthesis takes an exponential-sized matrix or state vector pairs as input and generates a fixed-size circuit, while program synthesis takes in an input-output specification and generates a program denoting a family of circuits for any input size.

To enable the synthesis of inductive quantum programs, there are three challenges. First, the synthesis framework requires a representation of the specification of quantum programs. Previous methods such as quantum Hoare triple [24], path-sum [25], and tree automaton [26] are all defined on fixed-dimension quantum systems and thus cannot be directly applied. Naive extensions of these representations with a variable qubit size do not work because symbolically representing matrices and automata is hard, limiting their usage for synthesis and verification. We introduce a specification language in QSynth, which enables intuitive input-output style specification for

quantum programs and supports all path-sum quantum states with an arbitrary number of qubits. We also propose the *hypothesis-amplitude* ($h - \alpha$) specification that uses two functions to specify quantum programs. The specification written in QSynth-spec language will be compiled into $h - \alpha$ specification to use in the verification step, which avoids SMT-unfriendly matrices (or graphical representations like automata).

The second challenge is to find a subset of quantum programs that is both expressive enough and efficient to synthesize. Previous programming languages for verifying quantum programs are either low-level circuit languages that do not support inductive structure (e.g., SQIR [27]) or high-level languages that do not have a detailed structure of unitaries (e.g., Quantum-while language used in Quantum Hoare Logic [24]). We design the Inductive-SQIR (ISQIR) language, an extension of SQIR that supports inductively defined quantum programs. We also define a Hoare-type verification logic that verifies an ISQIR program with respect to an $h - \alpha$ specification.

Finally, the synthesizer needs automated verification of inductive quantum programs. This requires reasoning about matrices, complex numbers, and formulas with a non-fixed number of terms, all of which have very limited support in SMT solvers. To solve this challenge, we define *parameterized path-sum amplitudes*, which, together with a sparsity constraint, can be efficiently encoded in SMT solvers. We further show that this set of expressions is expressive enough to specify and synthesize many quantum programs.

We evaluate QSynth on 10 inductive quantum programs including state preparation, arithmetic and textbook quantum algorithm procedures. We showcase that QSynth can synthesize practical quantum programs including quantum adders [51, 52], a quantum subtractor, the eigenvalue inversion for HHL [53] algorithm, quantum teleportation [54] and Quantum Fourier Transform [55]. All synthesis processes succeed in 5 minutes, while many of the programs cannot be

synthesized by previous methods when the number of qubits is larger than 6. A further investigation of synthesized programs shows that QSynth successfully captures the inductive structure of the targeting problem and produces better programs than human-written programs in Qiskit.

Contributions The key contributions in this chaptor are multi-folded.

- We propose QSynth, the first synthesis framework for inductively-defined quantum unitary circuit family.
- We introduce the QSynth-spec language that enables input-output style specification for inductive quantum programs, and the hypothesis-amplitude ($h-\alpha$) specification for scalable verification of quantum programs.
- We develop the syntax and the semantics of the inductive SQIR (ISQIR) language that supports recursively defined families of quantum unitary circuits, and a Hoare-type logic for proving the correctness of ISQIR program, with an $h-\alpha$ specification as predicates.
- We design Parameterized path-sum amplitude (PPSA) function, which leads to the efficient encoding of the verification process into SMT instances.
- We evaluate QSynth with a benchmark of 10 quantum programs, and show that QSynth is able to synthesize practical quantum programs.

2.2 Overview

The workflow of QSynth is shown in Fig 2.2. First, the user writes an input-output style specification of the program to be synthesized using the QSynth-spec language. Then, QSynth will

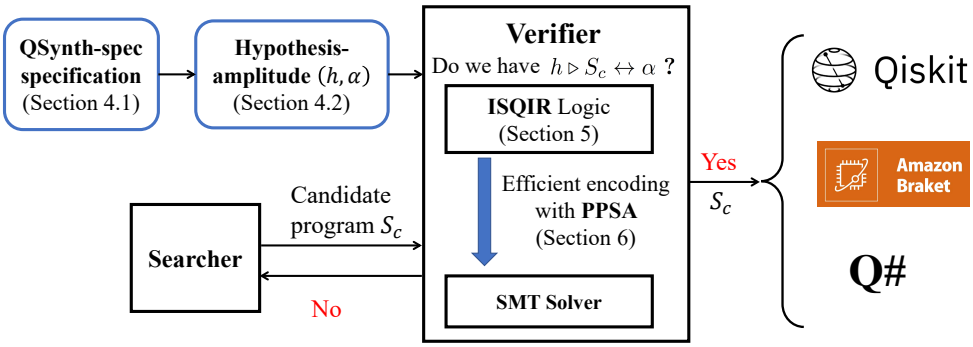


Figure 2.2: QSynth Overview.

translate the specification into a hypothesis-amplitude (h-α) pair for later verification. Meanwhile, QSynth will invoke a syntax-guided program searcher [56] to generate all possible candidate programs written in the ISQIR language within a given search space. For each candidate ISQIR program, QSynth will use the unitary ISQIR logic to verify whether the program satisfies the (h-α) specification. The verification process will be done in an SMT solver in which all the numbers are encoded in the form of parameterized path-sum amplitudes (PPSA). If the verification process succeeds, a correct program is synthesized and the program can be translated by QSynth's ISQIR compiler into commercial quantum programming languages including Qiskit, Q# and Braket, as shown in Figure 2.3. If the verification fails, the searcher will try the next candidate program.

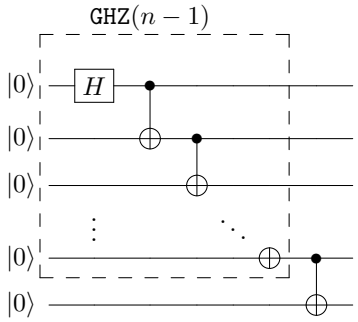
Next, we will walk through QSynth's components using the synthesis of the GHZ state preparation program as an example.

Target Program An N-qubit ($N \in Z^+$) Greenberger-Horne-Zeilinger state (GHZ state) is an entangled quantum state given by

$$|GHZ\rangle_N = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes N} + |1\rangle^{\otimes N}),$$

$S_{GHZ} :=$ **fix**$_1$ Id
const `H 0`
const `ID`
const `CNOT n-1 n`

(a) ISQIR program



(b) $(n+1)$-qubit GHZ circuit

```
def GHZ(N):
    circuit=
    QuantumCircuit(N+1)
    def S(circ, n):
        if n==0:
            circ.h(0)
        else:
            S(circ, n-1)
            circ.cx(n-1,
    n)
    S(circuit,N)
    return circuit
```

(c) Qiskit program

Figure 2.3: $(n+1)$-qubit GHZ state preparation programs in different programming languages. (a) ISQIR program S_{GHZ}. (b) The quantum circuit represented by S_{GHZ}. (c) Qiskit function compiled from S_{GHZ}. Statement `circ.cx` in Qiskit means appending a `CNOT` gate to the circuit. Qiskit's semantic requires the program always use class `QuantumCircuit(n)` to initialize a circuit. So QSynth compiler will wrap program S with an outside function `GHZ` to initialize the circuit.

which was first studied by Greenberger et al. [57] and is widely used in quantum information, e.g., [58, 59, 60, 61]. Preparing the N-qubit GHZ state from $|0\rangle^{\otimes N}$ for any N is a natural task for program synthesis.

Namely, we hope to synthesize an ISQIR program S such that $[\![\{\!\{S\}\!\}(n)]\!]$, the instantiation of S with $n \in \mathbb{N}$ where $\mathbb{N}$ denotes the natural number in the following paper, is a unitary that transfers state $|0\rangle^{n+1}$ to state $|GHZ\rangle_{n+1}$ for any $n \geq 0$.

Target Specification Our first challenge is how to specify our target program, which generates $|GHZ\rangle_{n+1}$ from $|0\rangle^{n+1}$. The difficulty is two-folded: (1) we want a specification for any input n, which excludes any existing specification methods for fixed dimensions (e.g., Quantum Hoare triples); (2) the specification should be as succinct as possible. Then, a direct matrix representation that might require $2^{n+1} \times 2^{n+1}$ is less desirable.

In QSynth, the specification is given in an input-output manner using the QSynth-spec language in the form of GHZ : $|0_{n+1}\rangle \mapsto |0_n\rangle \uplus |(2^{n+1}-1)_n\rangle$, where $|a\rangle \uplus |b\rangle$ means the normalized sum $(|a\rangle + |b\rangle)/\sqrt{2}$. Then, the input-output style specification is compiled into the following *hypothesis-amplitude* $(h - \alpha)$ *specification* (Definition 2.3.1) for later verification. An instantiation of $(h - \alpha)$ to the GHZ target program is given as follows:

$$h := \{(n, x, y) \mid x = 0\}, \qquad \alpha_{GHZ}(n, x, y) := \frac{1}{\sqrt{2}}\delta(y = 0 \vee y = 2^{n+1} - 1), \tag{2.1}$$

where the term $\delta(b)$ in α_{GHZ} is a {0,1}-valued function that returns 1 if the Boolean expression b is True and 0 otherwise.

Intuitively, the hypothesis function h specifies the interesting input to the program, the

desired program's behaviour which is specified by the amplitude function α. Precisely, given input x, the amplitude $\langle y| [\![\{\!\{S\}\!\}(n)]\!] |x\rangle$ of basis y on the input x for a desired program S is given by $\alpha(n, x, y)$, where x, y are bit strings.

In the GHZ example, we are only interested in input $|0\rangle^{n+1}$, which leads to a trivial h containing only $x = 0$ in (2.1). The output state, which is $\frac{1}{\sqrt{2}}(|0\rangle^{n+1} + |1\rangle^{n+1})$, corresponds to an amplitude function $\alpha(n, x, y)$ with only non-zero value $\frac{1}{\sqrt{2}}$ on either $y = 0$ (referring to $|0\rangle^{n+1}$) or $y = 2^{n+1} - 1$ (referring to $|1\rangle^{n+1}$), which explains (2.1).

Our hypothesis-amplitude specification is arguably as natural as the common classical specifications that describe the desired input-output relationship, except that one could have many such input-output pairs (i.e., *superposition*) with potential complex amplitudes, in the quantum setting, which requires an explicit use of our amplitude function $\alpha(n, x, y)$.

Verification of Candidate Programs Assume the QSynth searcher

has identified a candidate S_{GHZ}, same as Fig 2.3 (a), on the left of Fig 2.4. The program S_{GHZ} is constructed by a FIX syntax with subprograms $S_0 =$ `H 0`, $S_L =$ `ID` and $S_R =$ `CNOT n-1 n`, which is a recursive program with the base case $S_{GHZ}(0) := S_0$ and the inductive case, $S_{GHZ}(n) := S_L(n); S_{GHZ}(n-1); S_R(n)$. This ISQIR program is equivalent to the recursive Qiskit program in Figure 2.3c. The FIX syntax, similar to the fixpoint in Coq, enables inductive structures in ISQIR programs.

QSynth verifier leverages the newly developed unitary ISQIR logic (Section 2.4.2) to verify the goal judgement $h \triangleright S_{GHZ} \leftrightarrow \alpha_{GHZ}$, which basically states that candidate program S_{GHZ} satisfies the (h, α) specification.

QSynth verifier recursively applies the logic rules to split the judgement of h, α for larger

programs into that of smaller programs. The side conditions are checked by SMT solvers, and the h, α judgement for constant SQIR program for quantum gates that are independent of n are directly computed.

In the GHZ example, QSynth verifier first uses the FIX rule to split the goal judgement into two parts. The first part is two formulas (right bottom of Fig 2.4) to be checked by the SMT solver, with details elaborated on later.

The second part is three judgements for S_L, S_R, S_0 (right hand of the verification goal box in Fig 2.4). After applying the FIX rule QSynth uses the WEAKEN rule to adjust these judgements into an appropriate format. Since S_0, S_L, S_R are simple programs (formally called const SQIR programs), their corresponding hypothesis-amplitude specifications (i.e. $\alpha_I, \alpha_H, \alpha_{CNOT}$ on the upper right corner of Fig 2.4) are predefined in QSynth and can be verified directly using the CONST rule. α_I returns 1 if $x = y$ else 0, indicating the identity matrix. α_H represents the matrix of the program **const** H 0, i.e., $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes I^n$. The expression $e^{2\pi i \cdot \frac{x[0]*y[0]}{2}}$ in α_H returns -1 if $x[0] = 1, y[0] = 1$ and returns 1 otherwise, indicating the matrix $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$; the expression $\delta(x\backslash 2 = y\backslash 2)$, where $x\backslash 2$ and $y\backslash 2$ are integer division (e.g. $10\backslash 3 = 3$), returns 1 if x, y only have the lowest bit difference and 0 otherwise, indicating the matrix I^n on qubits $q_1, ..., q_n$. α_{CNOT} will be discussed later. After this verification step, the synthesis terminates and QSynth compiles S_{GHZ} into programs as shown in Fig 2.3.

SMT solving with PPSA We continue with the two formulas in the bottom right corner box in Fig 2.4 generated by the FIX rule and aim to verify them by SMT solvers.

The first formula indicates the *base* case is correct. The relation $\equiv_n$ indicates the equivalence between two hypothesis-amplitude specifications given specified n. In this GHZ example, the first

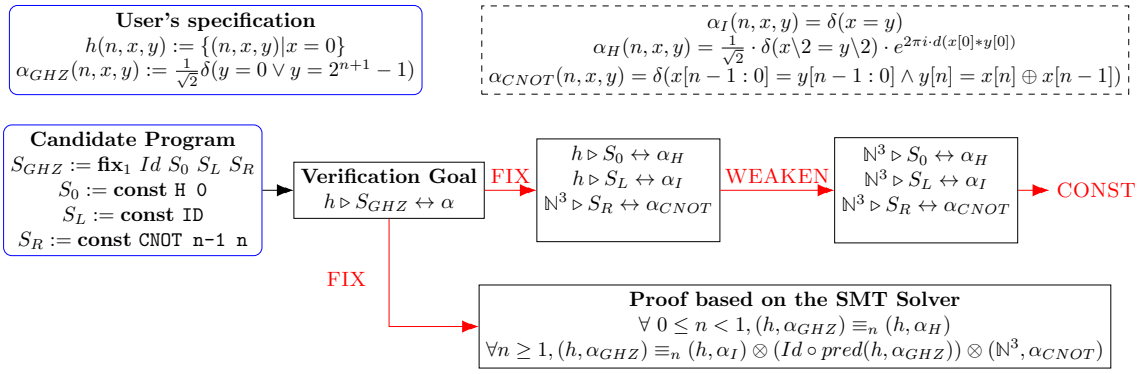


Figure 2.4: An example of synthesizing $n+1$-qubit GHZ state preparation program.

formula becomes:

$$\forall x\ y \in \mathbb{N}, x = 0 \rightarrow \alpha_{GHZ}(0,x,y) = \alpha_H(0,x,y).$$

Note that the amplitude functions α_{GHZ} or α_H are generally complex-valued, the automatic equivalence check of which are not generally supported by any existing tool.

Inspired by recent work on quantum program verification [25, 41], QSynth restricts α into a succinct path-sum representation yet with rich enough expressiveness (elaborated on in Section 2.5.1), called the parameterized path-sum amplitude (PPSA) in Definition 2.5.1. In PPSA representation, the non-zero amplitudes over x, y share the same magnitude, which depends on n, but can have different phases. Thus, instead of representing a general amplitude function $\alpha(n,x,y)$, it suffices to represent $\alpha(n,x,y)$ by components. For instance, α_H, α_{GHZ} defined in Fig 2.4, can be described with three components:

- A fraction expression indicating the amplitude: $\frac{1}{\sqrt{2}}$ for both α_{GHZ}, α_H.
- A Boolean expression indicating the non-zero value: $\delta(y = 0 \vee y = 2^n - 1)$ for α_{GHZ} and $\delta(x \backslash 2 = y \backslash 2)$ for α_H.

- An expression indicating the phase: $e^{2\pi i \cdot 0}$ for α_{GHZ} and $e^{2\pi i \cdot \frac{x[0]*y[0]}{2}}$ for α_H, where $x[0]$ means the lowest (i.e. 0th) bit of x's binary representation (e.g. $6[0] = (110)_b = 0$).

A direct substitution of α_H, α_{GHZ} would require QSynth, or SMT solvers, to verify

$$\forall x\ y \in \mathbb{N}, x = 0 \to \frac{1}{\sqrt{2}}\delta(y = 0 \vee y = 1) = \frac{1}{\sqrt{2}}\delta(x \backslash 2 = y \backslash 2) \cdot e^{2\pi i \cdot \frac{x[0]*y[0]}{2}},$$

which is infeasible. Using PPSA, one can equivalently verify the following by SMT solvers:

$$\forall x\ y \in \mathbb{N}, x = 0 \to \frac{1}{\sqrt{2}} = \frac{1}{\sqrt{2}} \wedge \delta(y = 0 \vee y = 1) = \delta(x \backslash 2 = y \backslash 2) \wedge \frac{x[0] * y[0]}{2} = 0.$$

The second formula concerns the correctness of the *induction* case. The function $pred(h, \alpha_{GHZ})$ generates the hypothesis-amplitude specification for the recursive call (i.e. $S_{GHZ}(n-1)$), and $(h_1, \alpha_1) \otimes (h_2, \alpha_2)$ calculates the one when composing two ISQIR programs together, both formally in Definition 2.4.4. In this example, the composition with (h, α_I) is trivial since α_I represents an identity matrix, and the second formula becomes

$$\begin{gathered}
\forall n \geq 1, \forall x\ y \in \mathbb{N}, (h, \alpha_{GHZ}) \equiv_n (h, \alpha'), \\
\text{where } \alpha'(n, x, y) = \sum_{z \in \mathbb{N}} \alpha_{GHZ}(n-1, x, z)\alpha_{CNOT}(n, z, y), \qquad (2.2) \\
\alpha_{CNOT}(n, x, y) = \delta(x[n-1:0] = y[n-1:0] \wedge y[n] = x[n] \oplus x[n-1]).
\end{gathered}$$

α_{CNOT} is designed to represents S_R: the term $x[n-1:0] = y[n-1:0]$ indicates the identity map $|q_0 ... q_{n-1}\rangle \mapsto |q_0 ... q_{n-1}\rangle$; the term $y[n] = x[n] \oplus x[n-1]$ indicates the map $|q_{n-1}\rangle |q_n\rangle \mapsto |q_{n-1}\rangle |q_{n-1} \oplus q_n\rangle$. Their combination indicates S_R's map, i.e., $|q_0 ... q_{n-1}\rangle |q_n\rangle \mapsto$

$|q_0 ... q_{n-1}\rangle\,|q_{n-1} \oplus q_n\rangle$.

There is another major challenge to verify (2.2) by SMT solvers due to the infinite summation over $z \in \mathbb{N}$, which comes from the composition of two amplitude specifications (details in Section 2.4). As a result, α' could have an unbounded number of terms, which makes it infeasible for any SMT solver.

To circumvent this general difficulty, we introduce a *sparsity* constraint, which restricts the number of non-zero points with any fixed x or y to be constant (Definition 2.5.2), and prove that the composition of two amplitude functions will have finite terms if *one of* the composed function is sparse. We also prove that all quantum gates applied on a constant number of qubits (e.g. all SQIR programs) have a sparse amplitude function. We only apply this sparsity constraint to SQIR programs predefined in QSynth, and *non-sparse* functions can be generated as synthesis specifications. However, the use of the FIX statement could generate non-sparse amplitude functions.

As a result, we only allow one use of the FIX statement in our synthesis, as otherwise we could risk composing two non-sparse amplitude functions that would lead to an infinite sum. The specification for the target program, however, could be non-sparse, as we won't need to compose the target programs with others.

In the GHZ example, α_{CNOT} is sparse and we have

$$\forall n, y \in \mathbb{N}, \quad \alpha_{CNOT}(n, z, y) \neq 0 \leftrightarrow z = y \oplus (y[n-1] \ll n).$$

Here the expression $y \oplus (y[n-1] \ll n)$ sets the $(n+1)$th bit of y (i.e. $y[n]$) to $y[n] \oplus y[n-1]$ and keeps $y[n-1:0]$ unchanged. Let $z = y \oplus (y[n-1] \ll n)$. With this sparsity of α_{CNOT}, we can

simplify the formula (2.2) to

$$\forall n \geq 1, \forall x\ y \in \mathbb{N}, (h, \alpha_{GHZ}) \equiv_n (h, \alpha'),$$

$$\text{where } \alpha'(n, x, y) = \alpha_{GHZ}(n-1, x, z)\alpha_{CNOT}(n, z, y) = \frac{1}{\sqrt{2}}\delta(z = 0 \vee z = 2^n - 1).$$

With the PPSA representation, it suffices to verify the following SMT instance:

$$\forall n \geq 1, \forall x\ y \in \mathbb{N}, x = 0 \rightarrow \frac{1}{\sqrt{2}} = \frac{1}{\sqrt{2}} \ \wedge\ \delta(y = 0 \vee y = 2^{n+1} - 1) = \delta(z = 0 \vee z = 2^n - 1).$$

Organization In Section 2.3 we describe the QSynth's specification language and hypothesis-amplitude specification. In Section 2.4 we introduce the ISQIR programming language for the inductive quantum circuit family, and its associated Hoare-type logic, In Section 2.5 we introduce the PPSA encoding. In Section 2.6 we discuss the implementation and the evaluation of QSynth on a benchmark of 10 programs. In Section 2.7 we discuss the related work. In Section 2.8 we discuss the limitation of QSynth and future work. In Section **??** we give the conclusion.

2.3 Specification

In this section, we explain how specifications are provided and processed in QSynth. Users provide the input-output specification in the QSynth-spec language (defined in Section 2.3.1). Then, the specification will be compiled into a hypothesis-amplitude pair (defined in Section 2.3.2) as the predicate for later verification. We describe the compilation process in Section 2.3.3

2.3.1 QSynth-spec Interface

To allow users to give the synthesis specification more intuitively, we design language QSynth-spec shown in Fig 2.5. Programmers provide the synthesis specification in QSynth-spec, and QSynth will compile it to the corresponding hypothesis-amplitude specification.

(Spec)	$\mathcal{S}$	::=	$\mathcal{I} \mapsto \mathcal{O}$
(Input)	$\mathcal{I}$	::=	$\|c_l\rangle \mid \|v[\ell]\rangle \mid \mathcal{I}_1 \otimes \mathcal{I}_2$
(Output)	$\mathcal{O}$	::=	$\|E\rangle \mid e^{i\cdot E} \cdot O' \mid \mathcal{O}_1 \otimes \mathcal{O}_2 \mid e^{i\cdot E_1} \|c_1\rangle \uplus e^{i\cdot E_2} \|c_2\rangle \mid \biguplus_{z\in\{0,1\}^l} \delta(B) \cdot e^{-2\pi i\cdot E} \|z\rangle$
(VarExp)	E	::=	$c_\ell \mid v \mid \textbf{uop}\ E' \mid E_1\ \textbf{bop}\ E_2$
(BoolExp)	B	::=	$E_1\ \textbf{rop}\ E_2 \mid B_1 \wedge B_2 \mid B_1 \vee B_2$
(Length)	ℓ	::=	$c \mid \mathtt{n} \mid \textbf{uop}\ \ell' \mid \ell_1\ \textbf{bop}\ \ell_2$
(Variable)	v	$\in$	Variables *(Constant)* $c \in \mathbb{N}$

Figure 2.5: QSynth-spec syntax. **uop** and **bop** are common unary operators and binary operators (e.g. + - * /)
. **rop** are common relation operators (e.g. ==, !=, >, <). $\delta(B)$ is a function that returns 1 if B is True and returns 0 otherwise. $\uplus$ means normalized summation.

A QSynth-spec specification is given in the form $Input \mapsto Output$. $Input$ indicates the input quantum state to the desired unitary. It is a ket expression constructed by l-bit constant numbers c_l, a $l-$bit variable $v[l]$, or the tensor product of two quantum states. Programmers can arbitrarily declare variables in the $Input$. For example, consider the $Input$ specification for a n-bit quantum adder: $|A\rangle |B\rangle |0\rangle^n \mapsto |A\rangle |B\rangle |A+B\rangle$.

$$|0\rangle |A[n]\rangle |B[n]\rangle |0_n\rangle \mapsto |c_0\rangle |A\rangle |B\rangle |A+B\rangle \tag{2.3}$$

where c_0 is the carry bit of $A+B$. We simplify $|\phi\rangle \otimes |\psi\rangle$ as $|\phi\rangle |\psi\rangle$ and omit the length specification for the one-bit state $|0\rangle, |1\rangle$. This specification indicates the qubit q_0 is initialized as $|0\rangle$ to hold the carry bit; qubits $q_1 \sim q_n$ and qubits $q_{n+1} \sim q_{2n}$ store two n-bit numbers $A[n], B[n]$; qubit

$q_{2n+1} \sim q_{3n}$ are initialized to $|0\rangle^n$ to store the sum of $A[n], B[n]$.

$Output$ suggests the state transformed from $Input$ by the target unitary program. $Output$ can be constructed by a variable expression E, a state shifted by the phase e^{iE}, or the tensor product of two states. The variables that appear in E are bound: they can only be a variable declared in $Input$ or the sum variable y when it is in the sum scope. Output can also be constructed as the superposition state with the same magnitude but different phase by $e^{i\cdot E_1}|c_1\rangle \uplus e^{i\cdot E_2}|c_2\rangle$ and $\biguplus_{z\in\{0,1\}^l} \delta(B)\cdot e^{-2\pi i\cdot E}|z\rangle$. For example, consider the specifications for n-qubit GHZ state program and QFT program.

$$\text{GHZ}_n : |0_n\rangle \mapsto |0_n\rangle \uplus |(2^n-1)_n\rangle \qquad \text{QFT}_n : |x[n]\rangle \mapsto \biguplus_{y\in\{0,1\}^n} e^{-2\pi i\cdot\frac{xy}{2^n}} \tag{2.4}$$

Programmers can omit the amplitude normalization term, which will be calculated by QSynth during the compilation from QSynth-spec to the hypothesis-amplitude specification.

2.3.2 Hypothesis-amplitude Specification for Verification

A critical component of program synthesis is the ability of expressing desired properties of the target programs, usually called specifications. The pre and post conditions of programs in typical Hoare triples provide a natural approach to express these input-output specifications. Quantum Hoare triples [24] are hence a natural candidate for describing input-output specifications for quantum programs. However, contrary to the classical setting where pre/post conditions have a lot of flexibility in description, the conventional pre/post conditions in quantum Hoare triples are described by quantum predicates which are Hermitian matrices of exponential dimensions in terms of the system size. The exponential dimension of quantum predicates incurs both the scalability

issue and technical inconvenience in automating the reasoning directly based on quantum Hoare triples.

Moreover, for ISQIR programs, one needs to express the specifications for a family of programs for different sizes, which is like classical program synthesis with a varying-length array of variables. However, existing Hoare triples can only be used to provide specifications for quantum systems of a fixed dimension.

To that end, we develop the so-called *hypothesis-amplitude* specifications for quantum circuit families, where the hypothesis component (denoted h) of the specification describes a certain subset of input states x in the computational basis, and the amplitude component (denoted α) describes the output state of the program on the given input x. Both h, α are functions of the index n so that they can describe a family of quantum circuits.

Definition 2.3.1. A *hypothesis-amplitude triple* contains h, S and α. Here h is a set of tuples $(n, x, y) \in \mathbb{N}^3$ (we abuse the notation h to also represent its indicator function of type $\mathbb{N}^3 \to \mathbb{B}$) that specifies the interested entries of S' semantics, S is a quantum circuit family parameterized with a natural number n, and $\alpha(n, x, y)$ is a complex function with natural number inputs.

A hypothesis-amplitude triple is a valid judgement, denoted $h \triangleright S \leftrightarrow \alpha$, when

$$h \triangleright S \leftrightarrow \alpha \iff \forall (n, x, y) \in h,\ \langle y| \left[\!\left[\{\!\{S\}\!\}(n)\right]\!\right] |x\rangle = \alpha(n, x, y). \tag{2.5}$$

Following the above definition, the hypothesis h is like classical pre-conditions and specifies the set of inputs where the post-conditions are provided. For any such input x, the output state of the program $S(n)$ is given by $\left[\!\left[\{\!\{S\}\!\}(n)\right]\!\right] |x\rangle = \sum_y \alpha(n, x, y) |y\rangle$, which explains why α is called the amplitude. We do not always need to specify the program's semantics for all inputs, so we

use set h to filter those unnecessary information. By the linearity of unitary, the input-output specification on a set of inputs in the computational basis can be extended to a specification in the linear space spanned by the given input set.

Compared with quantum Hoare triples, our hypothesis-amplitude specification provides a more flexible and arguably more intuitive way to formalize the desired properties on the target functions. For instance, for state preparation, our specification is almost straightforward to use and avoids the extra efforts of converting specifications into exponential-size quantum predicates. [1] Moreover, for all unitary programs, our hypothesis-amplitude specification could provide the same expressive power as general quantum Hoare triples at the cost of using potentially complicated h, α. Nevertheless, efficiently encoding into SMT instances are only known in restricted cases of h, α as discussed in Section 2.5.

2.3.3 From QSynth-spec to Hypothesis-amplitude Specification

Given a QSynth-spec specification $I \mapsto O$, QSynth compiles it to the hypothesis-amplitude specification in two steps: (1) generates the corresponding hypothesis h and a variable map Π based on I, where Π maps variables claimed in I to the qubits; (2) generates the corresponding amplitude function α based on O and Π.

Generate Π and h from I QSynth first calculates the total number of input qubits by adding up the length specifications of constants and variables in the input. This number depends on the parameter n. Then, QSynth assigns indexes from low to high for each variable and constant in I, according to the order they appear in I. The assignment of the variables is included in the variable

[1] Quantum Hoare triples, however, need a post-predicate where the target state spans the subspace with eigenvalue 1, and the rest space has eigenvalue 0. These complication comes from the generality of quantum Hoare triple.

map Π. For each constant number c represented by qubits $q_a \sim q_b$, QSynth adds expression $x[b:a] = c$ into the hypothesis h. For example, consider the specification for the quantum adder circuit $I : |0\rangle\,|A[n]\rangle\,|B[n]\rangle\,|0_n\rangle$, the variable map Π and the hypothesis h generated by QSynth are:

$$\Pi = \{A \mapsto q_1 \sim q_n, B \mapsto q_{n+1} \sim q_{2n}\}, \qquad h = \{x[0] = 0, x[3n : 2n+1] = 0\} \tag{2.6}$$

Generate α from O and Π QSynth will then compute the value $\alpha(n, x, y)$ from the output O and the variable map Π. QSynth first calculates the number of qubits. Then, it calculates the normalization factor k which is the total number of qubits in the summed variable plus the number of additions. Then, QSynth evaluates the output into a basis state $eval_{\Pi,x,y}(O)$ by replacing input variables with slices of x according to Π and summed variables with corresponding slices of y together with a phase factor $e^{i\phi(O,\Pi,x,y)}$. Finally, the value of $\alpha(n, x, y)$ will be $1/\sqrt{2^k}e^{i\phi(O,\Pi,x,y)} \cdot \delta(eval_{\Pi,x,y}(O) = y)$. If the output contains additions, there will be a basis state and a phase factor for each term and the α value will be the sum of their α. For example, the α's of GHZ and QFT in Equation 2.4 are

$$\alpha_{GHZ}(n, x, y) = \frac{1}{\sqrt{2}}\delta(y = 0 \vee y = 2^{n+1} - 1), \qquad \alpha_{QFT}(n, x, y) = \frac{1}{\sqrt{2^n}}e^{2\pi i x \cdot y/2^n} \tag{2.7}$$

2.4 Inductive SQIR and Its Logic

QSynth's goal is to synthesize programs with inductive structures.

To that end, we extend the existing intermediate representation SQIR [27] into a language called Inductive SQIR (ISQIR) that defines a family of quantum circuits inductively in Sec-

tion 2.4.1. In Section 2.4.2, we also develop a logic for reasoning about an ISQIR program with respect to an $h - \alpha$ specification.

2.4.1 Inductive SQIR (ISQIR)

We extend SQIR with an inductive structure, similar to `fixpoint` in Coq, to equip the language with the ability to describe a family of quantum circuits for general input n.

Definition 2.4.1 (Inductive SQIR- Syntax)**.** An ISQIR program is defined inductively by:

$$S ::= \textbf{const P} \mid \textbf{seq}\ S_1\ S_2 \mid \textbf{relabel}\ \pi\ S \mid \textbf{fix}_k\ \pi\ P_0\ P_1\ \cdots\ P_{k-1}\ S_L\ S_R.$$

Here, $P, P_0, ..., P_{k-1}$ are SQIR programs, π is a series of injective natural number mappings.

At a high level, any ISQIR program is a succinct way to describe a series of SQIR programs indexed by an integer (or input size) $n = 0, 1, 2, \cdots$. Intuitively, **const** P represents a repeating series of SQIR programs where every entry in the series is the same SQIR program P. **seq** S_1 S_2 concatenates two series S_1 and S_2 by concatenating SQIR programs of each entry. We also use $S_1; S_2$ and **seq** S_1 S_2 interchangeably for notation convenience. **relabel** π S permutes the qubit labels for the nth entry with permutation $\pi(n)$.

fix$_k$ is the new *inductive* structure introduced to ISQIR. Specifically, **fix**$_k$ constructs a series of SQIR programs by recursion, with k base cases $P_0, ..., P_{k-1}$, and the recursive call for the n-th entry is sandwiched by the n-th entries of ISQIR programs S_L and S_R. The choice of **fix**$_k$ is inspired by commonly seen quantum programs and serves as a good syntax guide for synthesis purposes for all the case studies in this paper.

We formulate the *semantics* of ISQIR programs as functions from a natural number to a SQIR program, i.e., $\mathbb{N} \to$ SQIR.

Definition 2.4.2 (ISQIR- Semantics)**.** An ISQIR program represents a series of unitary SQIR programs $\{P_0, P_1, ...\}$. We define the IR semantics $\{\!\{S\}\!\}$ of ISQIR program inductively by:

$$\{\!\{\textbf{const } P\}\!\}(n) = P; \qquad \{\!\{\textbf{seq } S1\ S2\}\!\}(n) = \{\!\{S1\}\!\}(n); \{\!\{S2\}\!\}(n);$$

$$\{\!\{\textbf{relabel } \pi\ S\}\!\}(n) = \texttt{map\_qb}(\pi(n), \{\!\{S\}\!\}(n));$$

$$\{\!\{\textbf{fix}_k\}\!\}(n) = \begin{cases} P_n, & n < k \\ \{\!\{S_L\}\!\}(n); \texttt{map\_qb}(\pi(n), \{\!\{\textbf{fix}_k\}\!\}(n-1)); \{\!\{S_R\}\!\}(n); & \text{else} \end{cases}$$

Here `;` is the sequential construct in SQIR, `map_qb` is a function relabeling the indices of qubits in a SQIR program according to injective function $\pi(n)$, and $\textbf{fix}_k$ is an abbreviation of $\textbf{fix}_k\ \pi\ P_0\ ...\ P_{k-1}\ S_L\ S_R$. We use a slightly changed denotational semantics of any SQIR program P, denoted $[\![P]\!]$, which is an **infinite-dimensional** matrix (i.e., $\mathbb{N}^2 \to \mathbb{C}$), that returns entries of P's original semantics within P's dimension and 0 otherwise.

Example 2.4.1. As an example, recall the GHZ program written in ISQIR syntax:

$$\textbf{fix}_1\ Id\ (\textbf{const } \text{H } 0)\ (\textbf{const } \text{ID})\ (\textbf{relabel } \pi\ (\textbf{const } \text{CNOT } 0\ 1))$$

where Id is an identity map, (**const** H 0) is a Hadamard gate on qubit q_0 which is the P_0, (**const** ID) is an identity unitary (served as S_L), and $\pi(n) = \lambda x.x + n - 1$ for $n \geq 1$. $\pi(n)$ maps the CNOT gate (served as S_R) on qubits q_0, q_1 to a CNOT gate on qubit q_{n-1}, q_n.

For notation convenience, when relabeling a SQIR program with a map π (i.e. **relabel** π **const** P),

we usually omit the **relabel** key word. Thus, S_R can also be denoted as **const** CNOT n-1 n.

We also develop the following syntax for general permutation π used in ISQIR programs that is also part of the candidate program search space.

Definition 2.4.3 (Permutation Syntax)**.**

$$\pi ::= Id \mid w[e_1 \rightleftharpoons e_2] \mid \textbf{shift}\ e_1\ e_2\ m \mid \pi_1 \cdot \pi_2$$

$$e ::= n \mid m \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \qquad m \in \mathbb{N}$$

Id is an identity permutation; $w[e_1 \rightleftharpoons e_2]$ swaps the mapping e_1 and e_2. **shift** $e_1\ e_2\ m$ maps any $x \in [e_1, e_2)$ to $[(x - e_1 + m) \bmod (e_2 - e_1)] + e_1)$.

For example, permutation $w[1 \rightleftharpoons n](n)$ maps 1 to n and maps n to 1. Permutation (**shift** 2 5 1)(n) maps 4 to 2, 2 to 3, and 3 to 4.

2.4.2 Unitary ISQIR Logic

We develop *unitary* ISQIR *logic* shown in Fig 2.6 to reason about ISQIR program's semantics with respect to the hypothesis-amplitude specification (h, α) with helper functions in Def 2.4.4. The soundness is formally proven in Theorem 2.4.5, whose proof is postponed to Appendix 6.1.1 in the supplementary material.

One can view (h, α) as a series of incomplete matrices: only those entries in the set h are known, whose values can be looked up in α. This gives the intuition behind our rules. The WEAKEN rule states that any subset of h can also be observed by α. The CONST rule lifts SQIR semantics to ISQIR semantics. The REPLACE rule states that if the observed entries are the

same for two amplitude functions, then one can substitute the other one with hypothesis h. The RELABEL rule relabels the entries of matrices for both h and α. The SEQ rule calculates matrix multiplication for each term in the series.

The FIX rule checks observed entries for terms with index $i < k$ (base cases) and computes matrix multiplication for $i \geq k$ (inductive cases).

Definition 2.4.4. For a hypothesis set h and an amplitude function α,

the **relabeling function** and the **predecessor functions** of h and α are defined by

$$\pi \circ (h, \alpha) := (\pi \circ h, \pi \circ \alpha) \qquad \text{pred}\,(h, \alpha) := (\text{pred}\,h, \text{pred}\,\alpha)$$

$$\pi \circ h := \{(n, \pi(n, x), \pi(n, y)) \mid (n, x, y) \in h\} \quad (\pi \circ \alpha)(n, x, y) := \alpha(n, \pi(n, x), \pi(n, y))$$

$$(\text{pred}\,\alpha)(n, x, y) := \alpha(n - 1, x, y) \qquad \text{pred}\,h := \{(n - 1, x, y) \mid (n, x, y) \in h\}$$

The entries of h and α are relabeled according to a series of injective function π. Predecessor functions move the series by one index, and are used for recursive calls in fixed-point programs.

The **composition function** of h and α are defined by:

$$\begin{aligned}
\text{comp}(h_1, \alpha_1, h_2, \alpha_2) := \Big\{ & (n, x, y) : \forall z, ((n, x, z) \in h_1 \wedge (n, z, y) \in h_2) \vee \\
& ((n, x, z) \in h_1 \wedge \alpha_1(n, x, z) = 0) \vee ((n, z, y) \in h_2 \wedge \alpha_2(n, z, y) = 0) \Big\}, \\
(\alpha_1 * \alpha_2)(n, x, y) := & \sum_{z \in \mathbb{N}} \alpha_1(n, x, z) \alpha_2(n, z, y), \\
(h_1, \alpha_1) \otimes (h_2, \alpha_2) := & (\text{comp}(h_1, \alpha_1, h_2, \alpha_2), \alpha_1 * \alpha_2).
\end{aligned}$$

$$\frac{h \triangleright S \leftrightarrow \alpha, \quad h' \subseteq h}{h' \triangleright S \leftrightarrow \alpha}\ \textsc{Weaken} \qquad \frac{\alpha \equiv^{\mathbb{N}^3} \lambda n.[P]}{\mathbb{N}^3 \triangleright (\mathbf{const}\ P) \leftrightarrow \alpha}\ \textsc{Const}$$

$$\frac{h \triangleright S \leftrightarrow \alpha, \quad \alpha \equiv^{h} \alpha'}{h \triangleright S \leftrightarrow \alpha'}\ \textsc{Replace} \qquad \frac{h \triangleright S \leftrightarrow \alpha \quad \alpha' \equiv^{\pi \circ h} \pi \circ \alpha}{\pi \circ h \triangleright \mathbf{relabel}\ \pi\ S \leftrightarrow \alpha'}\ \textsc{Relabel}$$

$$\frac{\begin{smallmatrix} h_i \triangleright S_i \leftrightarrow \alpha_i \quad \forall i{=}1{,}2 \\ (h,\alpha){=}(h_1,\alpha_1) \otimes (h_2,\alpha_2) \end{smallmatrix}}{h \triangleright \mathbf{seq}\ S_1\ S_2 \leftrightarrow \alpha}\ \textsc{Seq}$$

$$\frac{\begin{smallmatrix} h_L \triangleright S_L \leftrightarrow \alpha_L, \quad h_R \triangleright S_R \leftrightarrow \alpha_R \\ \forall i{<}k, \quad h_i \triangleright \mathbf{const}\ P_i \leftrightarrow \alpha_i, \quad (h,\alpha) \equiv_i (h_i,\alpha_i) \\ \forall i{\geq}k, \quad (h,\alpha) \equiv_i (h_L,\alpha_L) \otimes (\pi \circ \mathrm{pred}\ (h,\alpha)) \otimes (h_R,\alpha_R) \end{smallmatrix}}{h \triangleright \mathbf{fix}_k\ P_0\ \cdots\ P_{k-1}\ S_L\ S_R \leftrightarrow \alpha}\ \textsc{Fix}$$

Figure 2.6: The unitary ISQIR logic.

We also define several restricted **equivalence relations**:

$$h_1 \equiv_n h_2 \Leftrightarrow (\forall x, \forall y, (n,x,y) \in h_1 \leftrightarrow (n,x,y) \in h_2)$$

$$\alpha_1 \equiv_n^h \alpha_2 \Leftrightarrow \forall x, \forall y, (n,x,y) \in h \rightarrow \alpha_1(n,x,y) = \alpha_2(n,x,y)$$

$$\alpha_1 \equiv^h \alpha_2 \Leftrightarrow \forall n, \alpha_1 \equiv_n^h \alpha_2, \qquad (h_1,\alpha_1) \equiv_n (h_2,\alpha_2) \Leftrightarrow h_1 \equiv_n h_2 \ \wedge\ \alpha_1 \equiv_n^{h_1} \alpha_2$$

Theorem 2.4.5. *The rules of unitary* ISQIR *logic in Fig 2.6 are sound.*

2.5 Efficient Encoding by Parameterized Path-sum Amplitude

In this section we explain how to encode the hypothesis-amplitude triple introduced In Definition 2.3.1 and the functions and relations in Definition 2.4.4 into the SMT instance.

2.5.1 Encoding The Hypothesis-amplitude Triple

The hypothesis-amplitude triple in Definition 2.3.1 includes a hypothesis set h and an amplitude function α, which will be encoded separately.

Encoding the hypothesis set The hypothesis set h refers to a set of natural numbers. Intuitively, we encode the hypothesis h with a Boolean expression B constructed by n, x, y, whose value is true if and only if (n, x, y) is inside the hypothesis set. Namely,

$$(n, x, y) \in h \iff B(n, x, y) = \textbf{True}.$$

Encoding the amplitude function Encoding the complex function α is challenging since there is currently no automated program verification tool that supports complex numbers. We solve this by restricting the function α in a limited form that can be encoded into SMT instances. This is a trade-off between the expressiveness of our specification and the feasibility of automated verification.

Parameterized path-sum amplitude (PPSA) function We restrict an amplitude function α to be a *Parameterized Path-Sum Amplitude* function:

Definition 2.5.1. A Parameterized Path-Sum Amplitude (PPSA) function $\alpha_p : \mathbb{N}^3 \to \mathbb{C}$ is defined as

$$\alpha_p(n, x, y) := \frac{1}{\sqrt{\beta(n)}} \sum_{i=0}^{m} \delta(B_i(n, x, y)) \cdot e^{2\pi i \cdot d(V_i(n,x,y))}$$

- β is a natural number expression of n and it decides the magnitude of all paths.
- $m \in \mathbb{N}$ is a constant number. $\{B_i\}_m$ is a group of boolean expressions constructed by (n, x, y) and satisfies $\forall n\ x\ y \in \mathbb{N}, \sum_{i=0}^{m} \delta(B_i(n, x, y)) \leq 1$, where $\delta(B) : \text{Bool} \to \{0, 1\}$ is a function that returns 1 if B is True and returns 0 otherwise.

n, x, y : Variables $\in \mathbb{N}$ $\quad k$: Fixed number $\in \mathbb{Z}$

$$
\begin{aligned}
\text{Boolean Expression } B ::= &\quad \textbf{True} \mid \textbf{False} \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \mid \neg B' \mid V_1 \textbf{ rop } V_2 \\
\text{Binary-}\mathbb{N}\ V ::= &\quad x \mid y \mid n \mid k \mid \delta(B) \mid V_1[V_2] \mid V'[V_1 : V_2] \mid \textbf{uop } V' \mid V_1 \textbf{ bop } V_2 \\
\text{Magnitude } \beta ::= &\quad k \mid n \mid \beta_1 \textbf{ bop } \beta_2 \mid 2^n
\end{aligned}
$$

Figure 2.7: Syntax of the PPSA function.

- $\{V_i\}_m$ is a group of natural number expressions constructed by (n, x, y).
- For $x \in \mathbb{N}$, suppose x's binary representation is $x_q ... x_1 x_0$ where $x_i \in \{0, 1\}$, we use $d(x)$ to denote the fractional binary notation of x.

$$d(x) = [0.x_0 x_1 ... x_q]_2 = \sum_{i=0}^{q} x_i \cdot 2^{-(i+1)}.$$

Fig. 2.7 shows the syntax we allow to construct the expressions β, B, V in a PPSA function. "V_1 **rop** V_2" is a set of common relational operators bewteen V_1, V_2(e.g. $= \neq > < \geq \leq$). "**uop** V" is a set of common unary operator on V(i.e. - ! & | $\oplus$). "V_1 **bop** V_2" is a set of binary operators between V_1 and V_2, including arithmetic operators (i.e. + - * \ %) and bit-wise operators (i.e. & | $\oplus$ « »). All these operators have the same meaning as they have in C language. $V_1[V_2]$ means the V_2-th bit of V_1's binary representation (in the order from low to high). $v'[V_1 : V_2]$ is the natural number represented by the binary representation truncated from high bit V'_{V_1} to low bit V'_{V_2}. For example, let $V_1 = (6)_{10} = (110)_2$, we have $V_1[0] = 0, V_1[2 : 1] = (11)_2 = 3$. Z3 SMT solver supports all these syntaxes.

By restricting amplitude function α to a PPSA function, we disassembled the complex number function α into the combination of several integer or boolean expressions, which allows us to represent α with a set of SMT expressions that enable us to encode the calculation in

Definition 2.4.4 into the SMT solver. This will be discussed in Section 2.5.2.

Our design for the PPSA function is inspired by Feynman's *sum-over-path* formalism described in Section 1.3.5 which has inspired many quantum state representations. However, all of these representations can only express constant size unitary operators and fail to work for any input size. PPSA inherits the expressibility of the existing sum-over-path representations, which can express most famous quantum algorithms (e.g., [25, 41]), and works for a general input size. Hence, we believe the restriction to PPSA is mild and serves as a good balance between expressiveness and feasibility. Some common unitary operators that can be represented by h-α triple while restricting the amplitude function to PPSA are listed in Table 2.1. More examples are provided in Section 2.6.

Table 2.1: Examples of Unitary Operators represented by the H-α Specification.

Name	**Unitary Operator**	**H-α Specification**
Uniform_{n+1}	$\lvert 0\rangle^{n+1} \mapsto \frac{1}{\sqrt{2^{n+1}}} \sum_{0\le y<2^{n+1}} \lvert y\rangle$	$h = \{(n,x,y) \mid x = 0\}$ $\alpha(n,x,y) = \frac{1}{\sqrt{2^{n+1}}}\delta(y < 2^{n+1})$
Toffoli_{n+1}	$\lvert q_0 q_1 \cdots q_n\rangle \mapsto$ $\lvert q_0 q_1 \cdots (q_n \oplus \prod_{i=0}^{n-1} q_i)\rangle$	$h = \{(n,x,y) \mid x < 2^{n+1} \wedge y < 2^{n+1}\}$ $\alpha(n,x,y) = \delta(x[n-1:0] = y[n-1:0] \wedge$ $y[n] = x[n] \oplus (\&x[n-1:0]))$
QFT_{n+1}	$\lvert x\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{\frac{2\pi i \cdot xy}{2^n}} \lvert y\rangle$	$h = \{x < 2^{n+1} \wedge y < 2^{n+1}\}$ $\alpha(n,x,y) = \frac{1}{\sqrt{2}^{n+1}} \cdot e^{2\pi i \cdot d((x\cdot y) \gg (n+1))}$

2.5.2 Encoding Reasoning Based on ISQIR Logic

To enable SMT-based automation in reasoning,

one needs to encode the functions and the equivalence relations of h, α defined in Definition 2.4.4 into SMT instances. In the cases of the relabeling functions $(\pi \circ h), (\pi \circ \alpha)$, the predecessor function pred (h, α) and the composition function $\text{comp}(h_1, \alpha_1, h_2, \alpha_2)$, all used operations are supported by SMT solvers directly and the encoding is trivial. We hence focus our

discussison on the non-trivial encoding of the composition function $\alpha_1 * \alpha_2$ and the equivalence relations.

Encoding the composition function Recall the $\alpha_1 * \alpha_2$ function from Definition 2.4.4:

$$(\alpha_1 * \alpha_2)(n, x, y) = \sum_{z \in \mathbb{N}} \alpha_1(n, x, z)\alpha_2(n, z, y).$$

Since the summation is over $z \in \mathbb{N}$, by definition, the $\alpha_1 * \alpha_2$ function is a composition of two infinite-dimension unitaries, and hence cannot be calculated directly.

All existing symbolic matrix multiplication methods can only deal with a fixed dimension or a fixed number of terms (e.g., [25]) and hence are not applicable in our case.

Fortunately, we observe that in many cases, non-zero values of the function α are sparse, making the composition possible. *In particular, we show the possibility of computing the function $\alpha_1 * \alpha_2$ when one of α_1 or α_2 is sparse*. The sparsity of α is precisely defined as

Definition 2.5.2. We say a function $\alpha : \mathbb{N}^3 \to \mathbb{C}$ is **sparse** iff: there exist two functions $\mathcal{X}, \mathcal{Y} : \mathbb{N}^2 \to \{\mathbb{N}\}$ and for any inputs, the sets returned by $\mathcal{X}, \mathcal{Y}$ always have constant sizes (i.e., independent of inputs n, x, y), and further satisfy

$$\forall n\ x\ y \in \mathbb{N}, \quad \alpha(n, x, y) \neq 0 \to x \in \mathcal{X}(n, y) \wedge y \in \mathcal{Y}(n, x).$$

We denote such sparsity by $\alpha \trianglelefteq (\mathcal{X}, \mathcal{Y})$.

Intuitively, when $\alpha \trianglelefteq (\mathcal{X}, \mathcal{Y})$ holds, for any given $n_0, x_0 \in \mathbb{N}$, $\alpha(n_0, x_0, y)$ has non-zero values only on a finite of y points, the set of which is $\mathcal{Y}(n_0, x_0)$. The same intuition holds for $\mathcal{X}$

except for the case when n_0, y_0 are given.

Example 2.5.1. We show the amplitude function that can represent the ISQIR program **const** H 0 and its sparsity tuple $\mathcal{X}, \mathcal{Y}$ as an example.

$$\mathbb{N}^3 \triangleright \textbf{const H } 0 \leftrightarrow \alpha_H, \qquad \alpha_H(n,x,y) = \frac{1}{\sqrt{2}}\delta(x\backslash 2 = y\backslash 2) \cdot e^{2\pi i \cdot \frac{x[0]*y[0]}{2}}$$

$$\alpha_H \trianglelefteq (\mathcal{X}, \mathcal{Y}), \qquad \mathcal{X}(n,y) = \{y, y \oplus 1\}, \qquad \mathcal{Y}(n,x) = \{x, x \oplus 1\}.$$

The operation $\oplus$ is a bit-wise operation and the expression $x \oplus 1$ flips the 0th bit of x (e.g. $(101)_2 \oplus 1 = (100)_2 = 4$). The expression $\delta(x\backslash 2 = y\backslash 2)$ in α_H indicates that

$$\forall n\ x\ y \in \mathbb{N}, \quad \alpha_H(n,x,y) \neq 0 \rightarrow x \in \mathcal{X}(n,y) \wedge y \in \mathcal{Y}(n,x)$$

Intuitively, $\mathcal{X}, \mathcal{Y}$ are constructed in this way since **const** H 0 only modifies the 0th qubit.

Now we explain how to encode α function $\alpha_1 * \alpha_2$ when one of α_1, α_2 is sparse. Suppose α_2 is sparse and we have $\alpha_2 \trianglelefteq (\mathcal{X}, \mathcal{Y})$, we know that $\alpha_2(n,z,y) \neq 0$ only when $z \in \mathcal{X}(n,y)$. So $\alpha_1 * \alpha_2$ can be calculated by

$$(\alpha_1 * \alpha_2)(n,x,y) = \sum_{z \in \mathbb{N}} \alpha_1(n,x,z)\alpha_2(n,z,y) = \sum_{z \in \mathcal{X}(n,y)} \alpha_1(n,x,z)\alpha_2(n,z,y).$$

The summation on the right hand has only a fixed number of terms by sparsity which allows encoding into SMT instances. Similarly, when α_1 is sparse and $\alpha_1 \trianglelefteq (\mathcal{X}, \mathcal{Y})$, we have

$$(\alpha_1 * \alpha_2)(n,x,y) = \sum_{z \in \mathbb{N}} \alpha_1(n,x,z)\alpha_2(n,z,y) = \sum_{z \in \mathcal{Y}(n,x)} \alpha_1(n,x,z)\alpha_2(n,z,y).$$

Moreover, sparsity of α can be established in many cases. (Proof in Appendix 6.1.1).

Theorem 2.5.3 (α-sparsity)**.** *Suppose $\alpha, \alpha_1, \alpha_2$ are amplitude functions:*

- *Let P be a unitary* SQIR *program and $\mathbb{N}^3 \triangleright$ **const** $P \leftrightarrow \alpha$, then α is sparse.*
- *If α is sparse and π is a series of injective natural number mappings, then $\pi \circ \alpha$ is sparse.*
- *If both α_1, α_2 are sparse, so is $\alpha_1 * \alpha_2$.*

The above theorem shows that the α functions for all SQIR programs, and for relabeling a SQIR program or composing two SQIR programs are sparse. So non-sparse αs only appear in the fixpoint syntax. The candidate program from our searcher has at most one fixpoint due to the challenge discussed in Section 2.8. So when composing two amplitude functions α_1, α_2, there is always at least one sparse function and our composition strategy can work.

Encoding the equivalence relations Given a hypothesis h and two complex functions α, α', suppose the functions α, α' are in the form:

$$\alpha(n,x,y) = \frac{1}{\sqrt{\beta(n)}} \sum_{i=0}^{m} \delta(B_i(n,x,y)) \cdot e^{2\pi i \cdot d(V_i(n,x,y))}$$
$$\alpha'(n,x,y) = \frac{1}{\sqrt{\beta'(n)}} \sum_{i=0}^{m'} \delta(B'_i(n,x,y)) \cdot e^{2\pi i \cdot d(V'_i(n,x,y))}$$

QSynth verifier checks the equivalence $\alpha \equiv^h \alpha'$ by the checking following SMT instance and rejects the equivalence when the SMT solver gives a negative result.

$$\forall n\ x\ y \in \mathbb{N},\ \ h(n,x,y) \rightarrow \beta(n) = \beta'(n) \wedge \sum_{i=0}^{m} B_i(n,x,y) = \sum_{i=0}^{m'} B'_i(n,x,y)$$
$$\wedge \sum_{i=0}^{m} B_i(n,x,y) * V_i(n,x,y) = \sum_{i=0}^{m'} B'_i(n,x,y) * V'_i(n,x,y).$$

2.6 Experimental Case Studies

We demonstrate six additional case studies and provide the output Qiskit programs compiled from synthesized ISQIR programs for better illustration. Then in Section 2.6.7, we compare the performance of QSynth against the previous quantum circuit synthesis frameworks, QFAST [22] and Qsyn [23].

2.6.1 Quantum Adder

Motivation and Background Quantum circuits for arithmetic operations are required for quantum algorithms. One important example is the adder circuit. Feynman [51] first proposes the quantum *full adder* circuit to implement $|0\rangle |A\rangle |B\rangle |0\rangle^{\otimes n} \rightarrow |c_0\rangle |A\rangle |B\rangle |A+B\rangle$ where A, B are n-bit natural number. The first $|0\rangle$ is the carry bit and it is changed to carry value $|c_0\rangle$ after the addition. This design unfortunately needs n more qubits to store the sum of $A + B$. To reduce the qubit usage, Cuccaro et al. [52] proposed a new *ripple-carry adder* that

uses n fewer qubits than the full adder design. When given different specifications, QSynth can synthesize both adder circuits.

Full Adder Synthesis We let QSynth synthesize a program S_a that $S_a(n)$ provides a n-qubit full quantum adder (i.e. $S_a(0)$ is an identity unitary) with the specification in Equation 2.3. QSynth generates a program S_a as shown in Fig 2.8(a)(c) (i.e. Qiskit function `full_adder`). When $n = 0$, S_a does nothing since the circuit only contains the carry bit. When $n \geq 1$, S_a first call $S_a(n-1)$ recursively to get a $n-1$-bit full adder to calculate $|A_{n-1} + B_{n-1}\rangle$ and the carry bit is stored in qubit 0. Then S_a uses the one-bit adder circuit S_R to sum the highest bit in A_n and B_n.

Cuccaro's Adder Synthesis To reduce the number of qubits in the circuit, we want to synthesize an in-place adder. The specification is given as:

$$|0\rangle |A[n]\rangle |B[n]\rangle \mapsto |c_0\rangle |A\rangle |A + B\rangle \tag{2.8}$$

With this specification, QSynth generates program S_c shown in Fig 2.8(b)(d). We use Cuccaro's MAJ and UMA circuit structures as predefined modules in the synthesis. QSynth flattens these two modules in the compilation process.

2.6.2 Quantum Subtractor

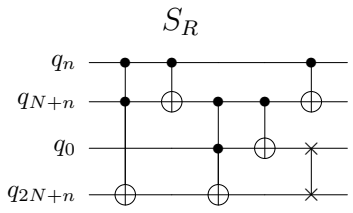
Another important quantum arithmetic operation is the quantum subtractor. A classical n-bit subtractor is usually implemented based on the two's complement theory (i.e. $B - A = B + \bar{A} + 1$ where $\bar{A}$ flips each bit in A), which is also used by many existing quantum libraries (e.g. QLib [62],QPanda [63]). However, this method requires additional ancilla qubits to build the "+1" operation. To reduce the qubit usage, we let QSynth synthesize a n-bit subtractor using the same number of qubits in the n-bit ripple adder.

```python
def full_adder(N):
    circuit = QuantumCircuit(3N+1)
    def Sa(circ, n):
        if n==0:
            return
        else:
            Sa(circ,n-1)
            # See the circuit SR below
            circ.ccx(n, N+n, 2N+n)
            circ.cx(n, N+n)
            circ.ccx(N+n, 0, 2N+n)
            circ.cx(N+n, 0)
            circ.cx(n, N+n)
            circ.swap(0, 2N+n)
    Sa(circuit,N)
    return circuit
```

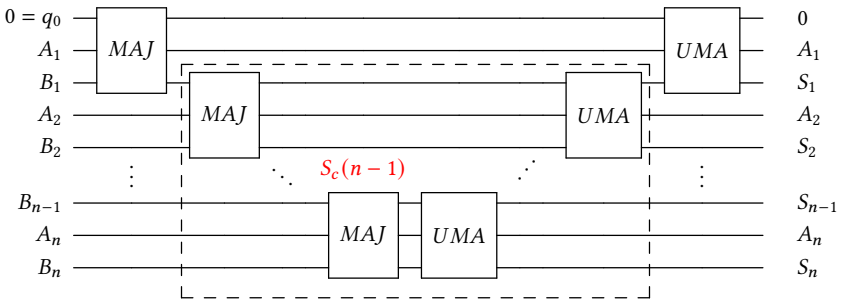
(a)

```python
def Cuccaro_adder(N):
    circuit = QuantumCircuit(2*N+1)
    def S(circ, n):
        if n==0:
            return
        else:
            # MAJ
            circ.cx(N-n+1,2*N-n+1)
            circ.cx(N-n+1,N-n)
            circ.ccx(N-n,2*N-n+1,N-n+1)
            S(circ,n-1)
            # UMA
            circ.ccx(N-n,2*N-n+1,N-n+1)
            circ.cx(N-n+1,N-n)
            circ.cx(N-n,2*N-n+1)
    S(circuit,N)
    return circuit
```

(b)



(c)



(d)

Figure 2.8: (a)(c) Full quantum adder program S_a written in Qiskit. `circ.ccx(a,b,c)` means appending a Toffoli gate that controlled by qubit q_a, q_b on qubit q_c to the circuit `circ`. The circuit S_R is exactly a one-bit quantum full adder circuit. (b)(d) Cuccaro's quantum ripple-carry adder program written in Qiskit language.

```
def RippleSubtractor(N):
    circuit=QuantumCircuit(2*N+1)
    def S(circ, n):
        if(n==0):
            circ.id(0)
        else:
            circ.x(2*N-n+1)
            circ.cx(N-n+1,2*N-n+1)
            circ.cx(N-n+1,0)
            circ.ccx(0,2*N-n+1,N-n
    +1)
            S(circ,n-1)
            circ.ccx(0,2*N-n+1,N-n
    +1)
            circ.cx(N-n+1,0)
            circ.cx(0,2*N-n+1)
            circ.x(2*N-n+1)
    S(circuit,N)
    return circuit
```

Figure 2.9: N-bit ripple subtractor program written in Qiskit language.

```
def ConditionalAdder(N):
    circ=QuantumCircuit(2*N+2)
    def S(circ, n):
        if(n==0):
            circ.id(0)
        else:
            circ.ccx(0,N-n+2,2*N-n
    +2)
            circ.ccx(0, N-n+2,N-n
    +1)
            circ.ccx(N-n+1,2*N-n
    +2,N-n+2)
            S(circ,n-1)
            circ.ccx(N-n+1,2*N-n
    +2,N-n+2)
            circ.ccx(0,N-n+2,N-n
    +1)
            circ.ccx(0,N-n+2,2*N-n
    +2)
    S(circuit,N)
    return circuit
```

Figure 2.10: N-bit conditional adder program written in Qiskit language.

Synthesis with QSynth We let QSynth synthesize a program with the specification below.

$$|0\rangle |A[n]\rangle |B[n]\rangle \mapsto |c_0\rangle |A\rangle |B - A\rangle$$

This specification is similar to the specification for Cuccaro's Adder (Equation 2.8) except for changing $B + A$ to $B - A$. With this specification, QSynth generates the program shown in Fig 2.9. This program equals to the circuit shown in Fig 2.11, indicating the logical expression $B - A = \overline{\bar{B} + A}$, which is different from the traditional subtractor implementation. This subtractor generated by QSynth uses no ancilla qubits and saves quantum resources.

2.6.3 Quantum Conditional Adder

Motivation and Background Quantum Conditional Adder is a necessary arithmetic operation for many known quantum algorithms, including quantum multiplier and Thapliyal et al. [64]'s quantum long division algorithm. A n-bit Quantum conditional adder circuit implements the transformation $|ctrl\rangle\,|0\rangle\,|A\rangle\,|B\rangle \mapsto |ctrl\rangle\,|ctrl * c0\rangle\,|A\rangle\,|ctrl * A + B\rangle$. It sums A_n and B_n when the control qubit $|ctrl\rangle$ is in state $|1\rangle$ and keeps the state unchanged when $|ctrl\rangle$ is in state $|0\rangle$.

One way to construct such a program is by replacing each gate in Cuccaro's adder program (i.e., the program in Fig 2.8(b)) with its conditional version, which is also the circuit generated by Qiskit. However, this method needs four-qubit Toffoli gates, which needs 14 CNOT gates to implement, increasing the total count of CNOT gates in the decomposed circuit. We let QSynth synthesize a target program using only X gate, CNOT gate, and Toffoli gate to find a better solution.

Synthesis with QSynth We let QSynth synthesize a program with the specification below

$$|f[1]\rangle\,|0\rangle\,|A[n]\rangle\,|B[n]\rangle \mapsto |f\rangle\,|f * c_0\rangle\,|A\rangle\,|f * A + B\rangle$$

The term $f * A + B$ indicates qubit q_0 is the flag qubit. With this specification, QSynth generates program S shown in Fig 2.10. We compare the resource count between the conditional adder circuits generated by QSynth and Qiskit, which is shown in Fig 2.12. All circuits are decomposed with $\{u_3, CNOT\}$ gateset by Qiskit's decomposition pass for comparison, where u_3 is a generic single-qubit rotation gate. The conditional adder programs generated by QSynth always use fewer quantum resources compared to the one from Qiskit.

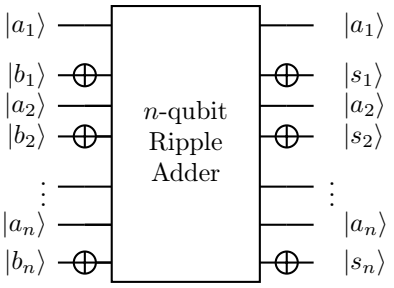


Figure 2.11: N-bit ripple subtractor circuit

Qubit	QSynth		Qiskit	
	u_3	cnot	u_3	cnot
4	182	144	244	208
5	228	180	305	260
6	274	216	366	312
7	320	252	427	364

Figure 2.12: Comparison of resource count between the conditional adder circuit generated by QSynth and Qiskit.

2.6.4 Eigenvalue Inversion

Background and Motivation Eigenvalue inversion is a necessary arithmetic step in HHL[53], a quantum algorithm for linear systems of equations. Given a c-qubit eigenvalue state $|\lambda\rangle$, the eigenvalue inversion circuit needs to calculate $|1/\lambda\rangle$. In practice, only the first n decimal places of the $1/\lambda$, denoted as d_n, and the remainder r_0 are kept. The precision n depends on the accuracy requirement of the algorithm.

Synthesis with QSynth We let QSynth synthesize a program that can calculate the first n decimal places of $1/\lambda$ and keep the remainder r_0 for further use.

Since QSynth's specification syntax only supports binary integers, we use the fact $2^n = \lambda * d_n + r_0$ where d_n is the quotient we want to give the specification. The example below shows our intuition.

Base 10: $1/7 = 0.14285714... \Leftrightarrow 10^6/7 = 142857.14... \quad \Leftrightarrow 10^6 = 142857 * 7 + r_0$

Base 2: $1/7 = 0.001001001... \Leftrightarrow 2^6/7 = 001001.001... \quad \Leftrightarrow 2^6 = (001001)_2 * 7 + r_0$

```
def inversion(N):
  circ = QuantumCircuit(N+2*c
   +1)
  def S(circ, n):
    if n<1:
      pass
    else:
      Append(circ,SUB(c),0,1,
   c+n+1)
      Append(circ,C_ADD(c-1),
        2*c+n,c,c+n)
      circ.x(2*c+n)
      circ=S(circ,n-1)
    return circ
  circ=S(circ,N)
  return circ
```

Figure 2.13: N-bit precision eigenvalue inversion program.

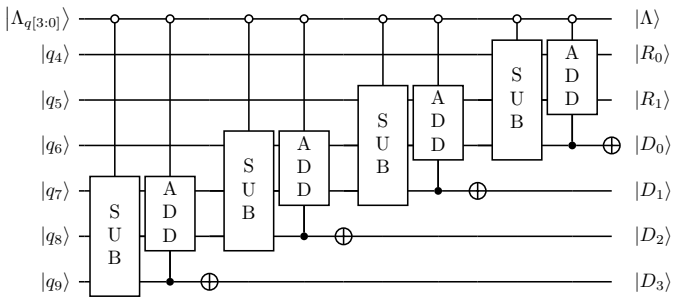


Figure 2.14: Eigenvalue inversion circuit for $n = 3, c = 3$.

The specification we send to QSynth for the n-bit precision eigenvalue inversion program is

$$|0\rangle |\Lambda[c]\rangle |0_n\rangle |1\rangle |0_{c-1}\rangle \mapsto |0\rangle |\Lambda\rangle |2^n \% \Lambda\rangle_c |2^n / \Lambda\rangle$$

The specification uses $\Lambda[c]$ to represent the signed input c-bit eigenvalue λ. It suggests qubit $q_{c+1} \sim q_{2c}$ store the remainder (i.e. $|2^n \% \Lambda\rangle$) and qubit $q_{2c+1} \sim q_{2c+n}$ store the quotient (i.e. $|2^n / \Lambda\rangle$).

With this specification, QSynth generates the program shown in Fig 2.13. Fig 2.14 shows the corresponding circuit for $n = 3, c = 3$. This program is a variant of Thapliyal [64]'s general quantum division circuit. Compared to calculating $|1/\lambda\rangle$ with Thapliyal's division circuit, which is constructed by $max(n, c)$-qubit subtractor and conditional adder, this program uses c-qubit one. This significantly reduces the number of qubits required when n is large.

2.6.5 Quantum Fourier Transform

Motivation and Background Quantum Fourier Transform (QFT) [55] is the classical discrete Fourier Transform applied to the vector of amplitudes of a quantum state. QFT is a part of many quantum algorithms, notably Shor's algorithm [65], QPE algorithm [66], and algorithms for the hidden subgroup problem [67]. A QFT over $\mathbb{Z}_{2^n}$ can be expressed as a map in two equivalent forms:

$$\text{QFT: } |x\rangle \mapsto \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{\frac{2\pi i \cdot xy}{2^n}} |y\rangle \qquad \text{OR} \qquad \text{QFT: } |x\rangle \mapsto \bigotimes_{k=0}^{n-1} |z_k\rangle \tag{2.9}$$

$$|z_k\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i \cdot [0.x_k x_{k-1} \ldots x_0]} |1\rangle), \qquad [0.x_m \ldots x_1 x_0] = \sum_{k=0}^{m} \frac{x_k}{2^{m-k+1}}. \tag{2.10}$$

Synthesis with QSynth We use the specification in Equation 2.4 to synthesize a program S_Q that $S_Q(n)$ returns the $n+1$-qubit QFT circuit.

QSynth fails to synthesize a simple fixpoint structure QFT circuit. Therefore we synthesize it in two steps to help QSynth synthesize a nested structure circuit. First, we let the synthesizer generate a program S_z that transforms the state of qubit n into state $|z_n\rangle$ and keep the state of qubits $q_0 \sim q_{n-1}$ unchanged. The specification is:

$$|X[n+1]\rangle \mapsto |X_0 X_1 \ldots X_{n-1}\rangle \otimes (|0\rangle \uplus e^{\frac{2\pi i \cdot X}{2^n}} \cdot |1\rangle)$$

With this specification, the synthesizer generates the program as shown in Fig 2.15. Statement `circ.cp(pi/2**n, N-n, N)` in Qiskit means a controlled phase rotation gate R_n[2] on

[2]Precisely, R_n is a single-qubit unitary $\begin{pmatrix} 1 & 0 \\ 0 & \omega_n \end{pmatrix}$ where $\omega_n = exp(2\pi i/2^n)$.

```
def Zn(N):
    circ = QuantumCircuit(N+1)
    def S(circ,n):
        if n == 0:
            circ.h(N)
        else:
            S(circ,n-1)
            circ.cp(pi/2**n , N-n, N)
    S(circ,N)
    return circ
```

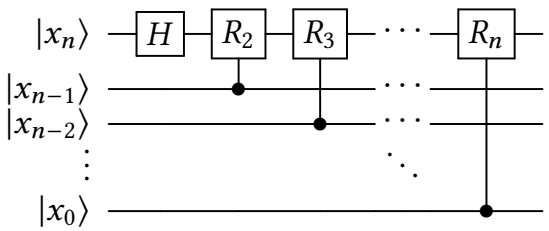


Figure 2.15: Qiskit program `Zn` that transforms qubit q_n to state $|z_n\rangle$

```
def QFT(N):
    circuit = QuantumCircuit(N+1)
    def S(circ,n):
        if n == 0:
            circ.h(n)
        else:
            circ.append(Zn(n))
            S(circ, n-1)
    S(circuit, N)
    return circuit
```

Figure 2.16: $N+1$-bit QFT program written in Qiskit language.

```
def teleport(N):
    circuit=QuantumCircuit(3*N)
    def S(circ, n):
        if(n<0):
            return
        else:
            circ.h(N+n)
            circ.cx(N+n, 2*N+n)
            circ.cx(n, N+n)
            circ.h(n)
            S(circ,n-1)
    S(circuit,N)
    return circuit
```

Figure 2.17: N-qubit quantum teleportation program.

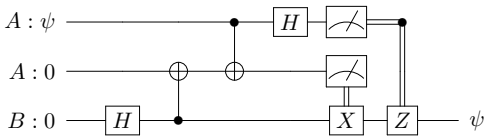
qubit q_N controlled by qubit q_{N-n}.

Then we insert this ISQIR program S_z (i.e. Qiskit program `Zn`) into the database so QSynth can use it for further synthesis, which leads to the QFT program in Fig 2.16.

2.6.6 n-qubit Quantum Teleportation

Motivation and Background Quantum teleportation[54] is one of the most famous quantum applications that can be implemented in the near term. It is a technique for transferring quantum information from a sender at one location to a receiver some distance away. The sender does not

have to know the particular quantum state being transferred. Moreover, the recipient's location can be unknown, but to complete the quantum teleportation, classical information needs to be sent from sender to receiver. The right-hand side circuit shows the process for sending one-qubit state $|\phi\rangle$ from Alice to Bob.



Synthesis with QSynth We let QSynth synthesize the unitary circuit part (before the measurement) of the n-qubit quantum teleportation process. Suggest Alice wants to send a state $|\phi_n\rangle$ stored in n data qubits to Bob, and they each have n more ancilla qubits which are initialized to state $|0\rangle^n$ for the teleportation process. We follow the same measurement strategy as the one-qubit teleportation: Alice will measure $|\phi_n\rangle$ and Alice's n aniclla qubits after the unitary circuit, then Bob apply bit-wise CZ and CX gate to Bob's n ancilla qubits based on the result. Assume the data qubits, Alice's ancilla qubits and Bob's ancilla qubits are in state $|a_n\rangle, |b_n\rangle, |c_n\rangle$ respectively after the measurement, Bob can use bit-wise CZ and CX gate to reproduce the state $|\phi_n\rangle$ if $(-1)^{\oplus a_n}(|b_n\rangle \oplus |c_n\rangle) = |\phi_n\rangle$. Here $|b_n\rangle \oplus |c_n\rangle$ is bit-wise XOR operation (e.g. $|101\rangle \oplus (|100\rangle + |110\rangle) = |001\rangle + |011\rangle$) and $\oplus a_n$ is a reduction XOR operation on a_n (e.g. $\oplus 101 = 1 \oplus 0 \oplus 1 = 0$). With this intuition, we let QSynth use the specification below to synthesize the unitary circuit part of a n-qubit quantum teleportation program,

$$|\phi[n]\rangle\,|0_n\rangle\,|0_n\rangle \mapsto \biguplus_{z\in\{0,1\}^{3n}} \delta(\phi = b_n \oplus c_n)\cdot e^{-\pi i\cdot(\oplus a_n)}\,|z\rangle$$

where $a_n = z[n-1:0], b_n = z[2n-1:n], c_n = z[3n-1, 2n]$. Fig 2.17 shows the program generated by QSynth.

2.6.7 Performance Evaluation

In this section, we compare the performance of QSynth and previous circuit synthesis methods.

Implementation In the experiment, we use the Syntax-Guided Top-down Tree Search [56] as the searcher with the following bounds on the search space: (1) when searching a candidate program under ISQIR syntax in Definition 2.4.1, we set the maximum program length to 10 and enumerate the value of k in the FIX syntax in $\{1, 2, 3\}$; (2) shorter candidate programs are sent to the verifier first; (3) when searching a permutation π under the syntax in Definition 2.4.3, we set the maximum syntax derivation depth to 4; (4) when deriving the syntax rule $e ::= m, m \in \mathbb{N}$, we enumerate $m \in [0, 3]$. All benchmarks in this paper can be synthesized under these bounds.

The implementation of QSynth uses 1k lines of Python. All the experiments are run with Z3 solver version 4.8.9 and Python 3.8.

Benchmarks Table 2.2 summarizes all 10 benchmarks. They are in three categories: arithmetic circuits, state preparation and sub-programs widely used in quantum algorithms. Many benchmarks are collected from textbooks [68]. Arithmetic circuits are frequently used in quantum oracle designs which is necessary for most famous quantum algorithms (e.g. Simons, Shor's algorithms, Grover search algorithm). State preparation is necessary for the setup of many quantum applications (e.g. quantum teleportation). We also collect the necessary quantum sub-program used in

quantum algorithms from their paper (e.g. HHL algorithm).

Table 2.2: Summary of all benchmarks used in the evaluation.

Benchmark Type	Benchmark Name	Description	Gate Set
State Preparation	n-GHZ	Greenberger–Horne–Zeilinger state [57]	H,CX
	n-Uniform	n-qubit uniform distribution state	H, X, Y, Z
Arithmetic	n-full-Add	n-qubit full adder	QFT, CX, SWAP, CCX, X
	n-Add	n-qubit in place adder	MAJ, UMA, QFT, CS, CZ, X
	n-Sub	n-qubit in place subtractor	MAJ, UMA, QFT, CS, CZ, X
	Cond n-Add	n-qubit conditional in place adder	Toffoli, QFT, CX, CS, CZ, X
Algorithm Module	n-QFT	n-qubit Quantum Fourier Transform	H,CS,CT, SW AP
	Inversion	n-qubit precision eigenvalue inversion for HHL algorithm [53]	c-adder, subtractor,X
	n-Toff	n-qubit Toffoli gate	Toffoli, CX, X
	n-Teleport	n-qubit quantum teleportation	H, CX

We compare the performance of QSynth against QFAST [22] and Qsyn [23]. For QSynth, we use input-output style specification written in QSynth-spec as input. For the other two frameworks, we use their specification interfaces and try to synthesize circuits with $n = 3, 4, 5, 6$. We use the same gate set when comparing QSynth and Qsyn in each case, while for QFast we use its hard-coded gate set.

We stop the synthesis and regard the synthesis as a failure if the running time is over 1 hour. All runtimes are a median of three runs.

Table 2.3 shows the running time of all the experiments. We can see that QSynth successfully synthesizes programs in all 10 benchmarks in at most 5 minutes, while QFAST and Qsyn fail to synthesize circuits for $n = 6$. From the result, we can see that when the size increases, the time of QFAST and Qsyn indeed grow exponentially, while QSynth only pays a fixed cost for all sizes.

The synthesis time of QSynth is comparable to the time to synthesize a corresponding circuit

Table 2.3: Running time of all benchmarks used in the evaluation.

Benchmark Name	**QSynth time (s)**	**QFAST time (s)**				**Qsyn time (s)**			
		3	4	5	6	3	4	5	6
n-GHZ	1.793	0.122	2.48	59.1	-*	0.091	1.33	49.2	-*
n-Uniform	0.415	0.027	1.25	32.7	-*	0.096	1.40	39.8	-*
n-Full-Add	288.1	617	-*	-*	-*	812.4	-*	-*	-*
n-Add	170.4	942	3511	-*	-*	132	2818	-*	-*
n-Sub	168.6	667	-*	-*	-*	287	3240	-*	-*
Cond n-Add	185.2	851	-*	-*	-*	192	1804	-*	-*
n-Toff	66.7	<0.01	21.8	679	-*	<0.01	16.5	354	-*
n-QFT	145.49	21.4	397	3598	-*	67.2	473	-*	-*
Inversion	93.5	1622	-*	-*	-*	22.5	740	-*	-*
n-Teleport	185.1	897	-*	-*	-*	614	-*	-*	-*

* Time out after 1 hour.

with size 3, with an exception of the n-Toff benchmark. We note that this is because 3-Toffoli is exactly a single Toffoli gate and 4-Toffoli can be done using two Toffoli gates, which are straightforward for QFAST and Qsyn to search. In contrast, QSynth's search is longer because it needs to consider the inductive structure and corner cases of $n = 1$ and 2. Nevertheless, QSynth quickly outperforms other frameworks on n-Toff at $n = 5$.

2.7 Related Work

Synthesis of quantum circuits Many methods have been proposed to synthesis quantum circuits of a fixed size [17, 18, 19, 20, 21, 22, 23, 69, 70]. These methods do not consider the inductive structure of quantum programs and do not scale due to their exponential blowup with the number of qubits.

Synthesis of classical programs The tasks of synthesizing classical programs are intensively-studied in the recent decades [56, 71]. The problem definitions of program synthesis are diverse and orienting, including syntax-guided synthesis [72, 73, 74, 75], example-guided synthesis

[76, 77, 78], semantics-guided synthesis [79, 80], and resource-guided synthesis [81, 82]. The modern approaches to solve these problems make use of sophisticated search algorithms, such as enumerative search with pruning [83, 84], constraint solver like satisfiability modulo theory (SMT) solver [72, 85, 86], and machine learning [87, 88]. We refer curious readers to surveys [56, 71] for a comprehensive picture on the development of classical program synthesis techniques. Many synthesis frameworks are developed into productive tools. For example, the SKETCH [86] framework completes programs with holes by specifications. ROSETTE [89] builds solvers into the language to automatically fill in holes when programming. However, QSynth needs to deal with unique challenges from quantum programs.

Verification of quantum programs An important procedure in syntax-guided synthesis is to verify any candidate program. Various logic and verification tools for quantum programs are developed in the last decade. QWIRE [90] embedded the formal verification of quantum programs manually in the Coq proof assistant. QBricks [41] do formal verification of quantum programs semi-manually using Why3. Quantum abstract interpretation [91] provides efficient tools to test the properties of quantum programs. In particular, the path-sum representation [25] of quantum program semantics inspired our representation. Chen et al. [26] uses tree automaton to verify fixed-size quantum circuits. It is hard to generalize to general-size cases because graphical structures like automaton are hard to symbolically model in SMT solvers.

Quantum Hoare logic [24] uses quantum predicates and Hoare triples to express and derive properties of quantum programs. Its language, quantum while language does not have the detailed structure of unitary executions. Its specifications are quantum predicate matrices. Therefore, it cannot be applied to synthesize quantum unitary circuit families.

The use of SMT solvers to automate the reasoning has also appeared in Giallar [92], Quartz [93] and symQV [94], although they only work on quantum circuit compilation passes, quantum circuit optimizations or fixed-size quantum circuit verification.

2.8 Discussion and Future Work

QSynth comes with several limitations. At a high level, QSynth sacrifices the expressiveness of the specification due to the limitation in efficient SMT encodings. For example, the lack of the equivalence verification of general complex functions forces us to consider the special form of amplitude in Definition 2.5.1. The sparsity requirement of α is another such restriction. Any relief of such restrictions would enlarge the space of programs that can be synthesized by QSynth.

Another limitation is that QSynth cannot directly synthesize programs involving nested fixed-point structures. Synthesizing a program nested loop or fixed-point structure is also a challenging problem in the classical domain. This is because the loop invariant of the program in nested loop structures is unknown and usually non-trivial to figure out. So we make the current QSynth only expand the FIX syntax at most once when generating the candidate program and avoid directly sending a program in nested fixed-point structure to the QSynth verifier. A natural next step is to include the search for the loop invariant in nested loop structures as part of QSynth, and to verify the candidate program against both the specification and the generated loop invariant.

Many quantum algorithms such as Bernstein-Vazirani [95] and Deustch-Jozsa [96] have quantum oracle as part of the program. However, it is hard to synthesize and verify quantum programs with oracles because it requires higher-order logic to quantify over arbitrary oracles. It is also an interesting next step to extend QSynth to support quantum oracles.

The design of ISQIR inherits concrete qubit indices from SQIR. This design introduces complications in the inductive variant that have to be addressed with explicit permutations and relabelings. We will explore the possibility of using a different representation of variables to make synthesis less complicated in the future.

Besides extending the expressiveness of specification and the support of more complicated quantum programs, it is also interesting to improve the synthesis performance by integrating classical program synthesis techniques into the quantum domain, including counterexample-guided synthesis, and various search heuristics.

Chapter 3: Automating NISQ Application Design with Meta Quantum Circuits with Constraints (MQCC)

3.1 Introduction

Quantum computers offer potentially significant performance advantages over classical ones for important classes of problems [97, 98]. Hardware advances have been bringing this potential ever closer to reality, but we are not there yet. Near-term, intermediate-scale quantum computing (NISQ) devices have few quantum bits (qubits), and these are prone to errors from several sources. General-purpose error correction techniques [28, 29, 30, 31] consume a substantial number of qubits, so they are not a practical remedy.

As a result, the design of NISQ applications needs to explore ways to optimize the efficiency and/or reliability of programs by balancing competing tradeoffs. Consider the following few examples.

Syndrome extraction for fault-tolerant quantum error correction [32, 33] is an example where one wants to minimize the fidelity loss in extracting quantum error syndrome information by measurements. Sequentially extracting syndromes, one by one, would incur many measurements, which each introduces noise. Chao and Reichardt [4] propose a scheme to extract multiple syndromes at once, *reducing* the total number of measurements; however, this scheme requires

more (noisy) qubits. Chao and Reichardt present an analytical tradeoff strategy for when all qubits have the same quality. However, such theoretical analysis becomes impossible when qubits are heterogeneous, which is common on NISQ machines.

Another example is the generation of **approximate circuits of important quantum subroutines**, like Quantum Fourier Transformation (QFT) and Quantum Phase Estimation (QPE) [34]. The goal is to save *resources* by reducing *accuracy*. The approach is to identify and prune gates in the fully accurate QFT and QPE circuits, where the pruned gates won't significantly change the generated unitary under some distance measure. A specific gate pruning strategy was suggested by Barenco et al. which is however non-optimal for specific input sizes and gate selection.

A final category of example is **scheduling programs to best leverage target architectural constraints**. By doing *multi-programming* (MP), we can *increase* overall computer utilization by running multiple programs at once (in "parallel"). But doing so may *decrease* reliability: particular gates and qubits may be more error-prone than others, so multi-programming tightens scheduling options. Das et al. [1] propose an algorithm to balance the tradeoff. *Crosstalk mitigation* (CM) is another such example, where nearby gate/qubit pairs scheduled in parallel may experience noise due to crosstalk. Placing them in sequence *decreases* crosstalk noise, but *increases* the chances of error due to decoherence. Murali et al. [99] propose a layout algorithm to balance the tradeoff.

All of these works offer mechanisms to relax a program's output fidelity so as to optimize some other attribute of its performance. In this sense, they support *approximate computing* [100, 101, 102], which aims to make the best use of imperfect hardware. Unfortunately, each only offers one-off improvements. The works either lack algorithmic/automated support entirely, or when they have it, this support cannot be composed or combined easily, due to conflicting tradeoffs that themselves would need balancing.

For example, removing "parallelism" to mitigate crosstalk reduces the utilization MP hopes to gain, but also adds a new source of error MP should consider.

Contribution

In this chaptor, we present **Meta Quantum Circuits** with **Constraints** (MQCC), the first general-purpose approximate computing framework for quantum programs. MQCC is a framework that makes it easy to design, implement, and experiment with optimizations, and to support *programming* their customized composition while leveraging automated reasoning. Crucially, MQCC allows users to express and optimize a *variety* of metrics. Because we are in a stage of rapid development for quantum hardware, application designers need to balance trade-offs among many emerging or even unknown factors.

To use MQCC, an optimization designer starts by writing (or reusing) routines to compute various **attributes** of quantum programs; these are the basis of optimization. We have implemented seven attributes so far: *qubitcount*, *gatecount*, *crosstalk* [7], *fidelity*, *accuracy*, *circuit depth*, and *quantum circuit space-time volume* [103, 104]. Next, the designer writes a *transpiler* that takes the optimization's input program(s) and introduces meta-level **choice variables** into them.[1] In essence, a program with choice variables is a *family* of programs, and a valuation of those variables identifies one member of the family. Finally, the designer states an optimization goal in terms of the attributes of interest, e.g., to maximize one attribute while keeping another below a threshold.

Now, given a transpiled input, MQCC selects the values of the choice variables that satisfy the goal. It analyzes the program with respect to the attributes of interest and generates symbolic *cost expressions* over the choice variables which express the program's attributes' values with

[1]Transpilation is not strictly needed—users of an optimization could insert choice variables manually.

respect to the goal. MQCC encodes these as Satisfiability Modulo Theories (SMT) formulae and solves for the choice variables, thereby selecting a final program to run on an actual platform. In the worst case, formula sizes are exponential in the number of choice variables due to the essential hardness associated with the worst-case optimization problem. But for NISQ-era programs, formula sizes are often not large so running times are often reasonable. The scalability study in Section 3.4.6 shows that most middle-size QASMbench programs, which have up to 10^4 gates, can still be handled by MQCC efficiently. We have also identified a special case of *additive* attributes whose formulae are linear in the number of choice variables, and thus scale better.

We demonstrate MQCC's generality by using it to implement several case studies. We employ MQCC to automate the selection of a syndrome extraction scheme of Chao and Reichardt [4], where we easily handle the heterogeneous qubit case in MQCC. We also implemented an automated procedure to trade accuracy for savings of circuit volume in implementations of Quantum Fourier Transformation (QFT) and Quantum Phase Estimation (QPE). Lastly, we implemented both the MP and CM optimizations listed above (each involves a tradeoff of two attributes) and also implemented a novel composition of MP and CM which balances the tradeoff among the *three* attributes from MP and CM combined—both use *depth*, but individually they use *noise* and *crosstalk* attributes. MQCC makes this composition simple to express.

We compared our syndrome extraction strategy with Chao and Reichardt [4]'s original extraction scheme as well as an alternative, more parallel scheme on quantum systems that have of qubits with heterogeneous random errors, developed by Reichardt [5]. Our experiments suggest that the MQCC-based solution can always achieve the minimum logical error rate for all kinds of random errors. In the experiment, MQCC generates the solution in less than 0.1 seconds.

For gate-pruning QFT and QPE, MQCC's automation is able to identify more efficient

strategies than existing ones [34] for the entire parameter range. MQCC can generate the solution for QFT/QPE instances with 50 qubits and 11k space-time volume [103] in 40 seconds.

For the remaining case studies, we applied the optimizations to a benchmark of quantum programs and demonstrated the benefits by running on actual NISQ machines. For CM and MP, we match or improve previously reported results. Our new optimization, which combines both MP and CM attributes, allows one to take the crosstalk-induced noise into the consideration in MP tasks. As a result, we generate multi-programming schedules with both high success probability and small circuit depth, compared with the one generated with MP attributes alone, on actual NISQ machines. In all these cases, MQCC's solver performs well, taking less than 0.1 seconds for each program.

Because NISQ-ready programs are small, we also ran a separate experiment on a benchmark of larger programs (too big to run on today's hardware) to see how well MQCC scales.

In particular, we test MQCC's performance of MP, CM, and MP-CM tasks on a collection of middle-size circuits (10$\sim$20 qubits with 100$\sim$1000 gates) from representative quantum applications in QASMBench [2]. MQCC is able to generate the solution for most test cases within a few seconds, with some exceptions in a few minutes.

As a final remark, MQCC can be easily integrated into the existing ecosystem of quantum computing tool-chains. MQCC builds on top of OpenQASM [39] and can produce executable programs in Qiskit and AWS Braket. All code is freely available.

Related Work. Fast but error-prone chips in classical computation inspired the development of frameworks to trade correctness for performance. Carbin et al. [101] proposed Rely to handle reliability specifications and analysis. Users of Rely can specify the quantitative reliability of

each component, and the compiler automatically reasons whether the program is reliable enough. Misailovic et al. [100] made one step further with Chisel, automatically optimizing the tradeoff between reliability and accuracy via integer linear programs. MQCC is inspired by Chisel's approach: both set up a constraint problem whose solution selects instructions based on an optimized-for objective. However, MQCC is more general: With MQCC, users can easily define their own attributes and objective to optimize, whereas with Chisel both the attributes and objective are fixed. Moreover, in actual manifestation, we have used MQCC on many more, and various, applications than Chisel did in the classical realm.

Hung et al. [102] and Tao et al. [105] define logics of *quantum robustness* to assess the potential noise accumulation in quantum programs. These logics permit reasoning about noise, but provide no means to automatically compensate for it.

The optimized quantities in these works [102, 105] —reliability, noise, resources, etc.—are *additive attributes*, in our terminology. For our applications, we also crucially rely on the flexibility of general attributes provided by MQCC.

SMT solvers are widely used in programming language and architecture designs, e.g., as the basis for automation in program verification [106, 107], and specification-based program synthesis [108, 109, 110]. The solver-aided host language Rosette [89, 111] has been designed to ease the construction of solver-aided domain-specific languages. SMT solvers have also been employed to design NISQ applications. In addition to the crosstalk example [99], one can also model the qubit mapping and gate scheduling problems as SMT instances [7, 112]. MQCC provides a flexible framework that leverages SMT solvers to automate NISQ designs.

3.2 Meta Quantum Circuits with Constraints

This section describes MQCC, using the problem of fault-tolerant quantum error correction (FQEC) as an example [1].

3.2.1 MQCC Overview

MQCC's architecture is shown in Fig 3.1(a). The core of MQCC is the MQCC solver, which takes three inputs: a quantum *meta-program*; the definitions of relevant *object attributes*; and an optimization *goal*.

MQCC meta-programs. The syntax of the MQCC meta-program is essentially standard OpenQASM, but is extended to include *choice variables*. The valuation of the choice variables determines the actual program that will run on the quantum computer—different choice-variable valuations will yield different programs. Consider the example to the left. After declaring a quantum register, the program defines a *free choice variable* `c1` whose value can be either 0 or 1. `c1` is used in the subsequent `choice` statement. When the MQCC solver produces a solution to the choice variables, it replaces each `choice` statement with the branch corresponding the solution. So, if `c1` solved to 0, lines 3-6 would be replaced by `x(q[0]);` if it solved to 1, they would be replaced by `h(q[0]);`. After replacement, the program is normal OpenQASM and can run on quantum hardware.

```
1 qreg q[1];
2 fcho c1 = {0, 1};
3 choice (c1) {
4     0: x(q[0]);
5     1: h(q[0]);
6 }
```

The user of an optimization need not write the meta-program directly; as shown in Fig 3.1(a),

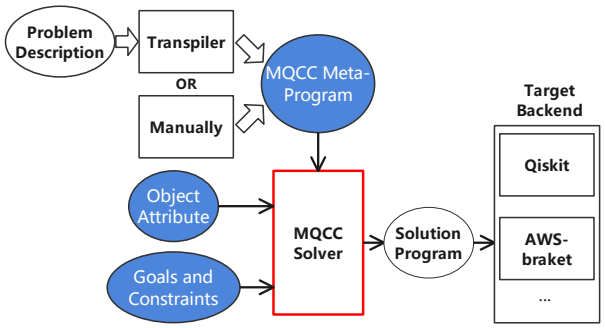
an optimization-specific transpiler can produce it from a higher-level problem description. The insertion of choice variables for all applications in our paper is handled automatically by transpilers, whose design, however, requires domain knowledge of the corresponding application. A new transpiler is likely required for each new application, and its complexity will depend on the application. MQCC's transpilers often leverage Qiskit [113] code for generating meta-programs.

Object attributes. The second MQCC solver input is a set of relevant *object attributes*. An attribute is essentially a function from a quantum circuit to a numeric value (e.g., the count of qubits used by the circuit, the count of gates in the circuit).

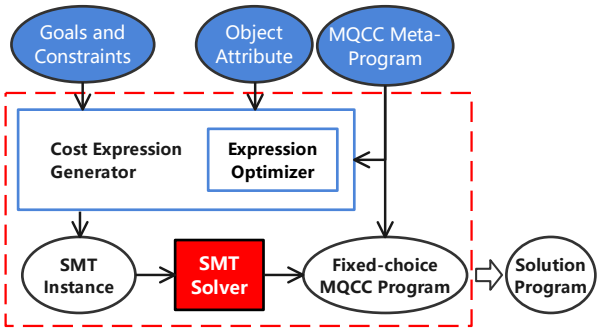
Goal and constraints. The third MQCC solver input is the optimization goal and constraints that must be satisfied. The MQCC solver applies the provided attributes to the meta-program and thus generates a formula that expresses the attribute in terms of the program's choice variables. The solver then comes up with a solution for the choice variables such that these formulae satisfy the given constraints while meeting the stated goal (e.g., minimize the count of gates in the circuit while keeping the count of qubits used by the circuit under the given threshold).

3.2.2 Example: Fault-tolerant Quantum Error Correction

As an example of MQCC usage, we present how it can maximize fidelity when using fault-tolerant quantum error correction (FQEC) schemes, whose efficacy can depend on the target program and architectural constraints.



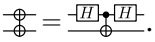
(a) Overview of MQCC



(b) MQCC Solver

Figure 3.1: Overview of MQCC

3.2.2.1 Background

Fault-tolerant quantum error correction protects quantum information from noise. Classical error correction based on error-correcting codes employs redundancy and extracts a *syndrome* to diagnose the error that corrupts an encoded state. Quantum error correction also employs syndrome extraction [33]. Each syndrome is extracted by applying a specific circuit to the data qubits to extract their information into the ancilla qubits. Then these ancilla qubits are measured to retrieve the syndrome information and can be reused for the next syndrome extraction. Fig 3.2(a)(b) shows two examples where **Z** indicates a $|0\rangle, |1\rangle$ measurement, **X** indicates $|+\rangle, |-\rangle$ measurement and  . In each circuit, the top five qubits are data qubits and others are ancilla qubits. The ancilla qubits measured with **Z** measurement are called syndrome qubits and their measurement results are used to decide whether the data qubits are corrupted. The data qubits are not corrupted only when all syndrome qubits are measured as $|0\rangle$. Otherwise, correction circuits will be applied to the data qubits based on the measurement results. The ancilla qubits measured with **X** measurement are called "flag" qubits and their measurement results are used to detect the error in the syndrome extraction circuits. The syndrome extraction circuits are correct

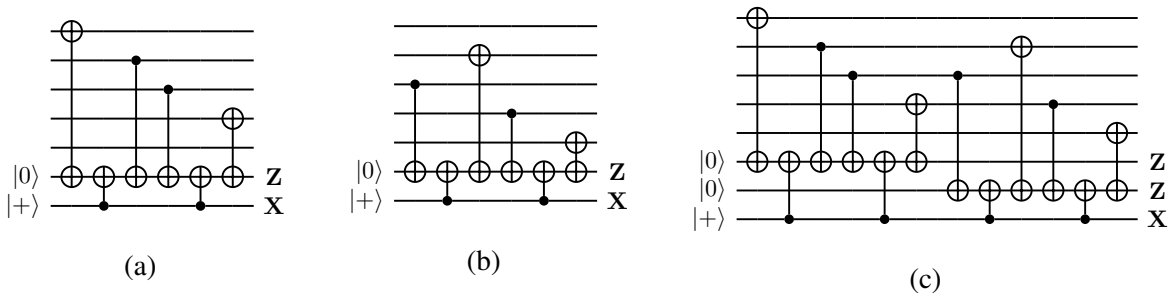


Figure 3.2: Syndrome extraction circuit for the $[\![5, 1, 3]\!]$ code [4, 5]. (a) Circuit to extract the $XZZXI$ syndrome. (b) Circuit to extract the $IXZZX$ syndrome. (c) Extract syndrome $XZZXI$ and $IXZZX$ in parallel.

only if all flag qubits are measured as $|+\rangle$.

As with classical error correction, many efficient FQEC codes are known [30, 103, 114]. One is the perfect $[\![5, 1, 3]\!]$ code [115]. For this code, there are four syndromes, named $XZZXI$, $IXZZX$, $XIXZZ$, and $ZXIXZ$. Fig 3.2(a)(b) respectively show the circuits that extract syndromes $XZZXI$ and $IXZZX$.

There are also many existing syndrome extraction strategies. For example, Shor-style syndrome extraction [116] requires $w + 1$ or w ancilla qubits, where w is the largest weight of a stabilizer generator. Steane [117, 118] uses at least a full code block of extra qubits, while Knill [119] uses an encoded EPR state and thus at least two ancilla code blocks. Chao and Reichardt [4] and Yoder and Kim [120] propose syndrome extraction strategies based on flag qubits that use only two ancilla qubits.

For a large code with many syndromes, it can be inefficient to extract the syndromes one after the other. Reichardt [5] introduce the method to extract multiple syndromes in parallel. The circuit in Fig 3.2(c) extracts $[\![5, 1, 3]\!]$ code's $XZZXI$ and $IXZZX$ syndromes at once. Compared to extracting two syndromes sequentially, extracting syndromes in parallel requires one more qubit (eight vs. seven) but one fewer qubit measurement (**ZZX** vs. **ZX**, **ZX**). There is a trade-off between the count of ancillary qubits and the number of measurements, so the strategy for extracting syndrome should be chosen carefully. Quantum error correction aims to protect quantum information from quantum noise and the trade-off should aim to increase the procedure's fidelity. Since error may occur during the qubit measurement, fewer qubit measurements can improve fidelity. On the other hand, using more ancilla qubits can harm it. In practice, different physical qubits have different error rates and the application will use the qubits with the highest fidelity first; more ancillary qubits mean that qubits with higher error rates might be used, decreasing the

fidelity of the whole procedure.

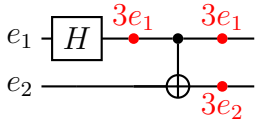
3.2.2.2 Expressing the FQEC tradeoff with MQCC

We can use MQCC to maximize **Fidelity**—an attribute modeled by the probability that no error occurs in the error correction procedure—while satisfying the constraint **QubitCount** $< \theta$ where **QubitCount** is an attribute that estimates the count of ancilla qubits, and θ is a provided threshold. We give the definition of **Fidelity** and **QubitCount** below.

We model the probability of error using the standard depolarizing noise model [6, 121]. The depolarizing noise model assumes that a quantum operation's error consists of random, independent applications of products of Pauli operators after the gate with probabilities determined by the gate. Suppose the "depolarizing" error for a qubit e_p: the probability that $|0\rangle$ ($|+\rangle$) state preparation erroneously produces $|1\rangle$ ($|-\rangle$). A binary (such as **Z** or **X**) measurement results in the wrong outcome with probability $e_m = e_p$; for a quantum gate, each qubit the gate is applied is modified by one of the three possible Pauli operators, each with probability e_p. With this noise model, given the e_p for each qubit, the probability P that no error occurs in a quantum circuit S can be estimated as the product of the probability of no error for each operation:

$$P = \prod_{op \in S} (1 - E_{op}) \iff \log(P) = \sum_{op \in S} \log(1 - E_{op}), \tag{3.1}$$

where E_{op} denotes the probability that the operation op is modified by Pauli operators. The corresponding logarithm term can change the probability expression into a linear expression. For example, consider the Bell state preparation circuit to the right. Given e_1, e_2 as the error rate for two qubits,



depolarizing error may occur at the red points with the corresponding probability. So the probability P that no error occurs in the circuit is the product $(1-3e_1)^2(1-3e_2)$.

Now we can define our two attributes, **Fidelity** and **QubitCount**. The **Fidelity** attribute applied to a syndrome extraction circuit S estimates $log(P)$ where P is the probability that no error occurs in S based on the standard depolarizing noise model [6] described in Section 3.2.2.1. The **QubitCount** attribute applied to S computes the total qubit count used by the circuit. Programmers can easily define their own attributes, and these can be reused for different programs and problems. Formal definitions of **Fidelity** and **QubitCount** are given in Section 3.3.

Suppose we want to optimize the program in Fig 3.3(b), which extracts all syndromes from the five-qubit array `data` in which the information has been pre-encoded. The program invokes syndrome extraction circuits `Extract_...`, defined in Fig 3.3(a) (and corresponding to those in Fig 3.2), one after the other. We want to allow for the possibility that some could be extracted in parallel depending on our optimization tradeoff. In MQCC, this possibility is expressed with choice variables. Fig 3.3(c) shows the meta program that corresponds to Fig 3.3(b). It differs in that it has introduced two choice variables whose solution may allow calling `Extract_both_12` and/or `Extract_both_34`, respectively, which invoke two extraction circuits in parallel, rather than the alternative invocation in sequence. We produce this meta-program automatically by running a transpiler on Fig 3.3(b). Transpilation is often easy to build, which will be discussed in Section 3.2.4.

Let's look more closely at the meta-program in Fig 3.3(c).

Line 2-5 define quantum and classical registers used in the program as usual, using the **qreg** and **creg** syntax.

Line 6 declares the program's choice variables. We use keyword **fcho** to define two *free*

choice variables c_1, c_2 that choose value in {0,1}. A choice variable's value can be any integer within an enumeration $\{a_1, a_2, ..., a_n\}$ or an interval $[a_1, a_2]$ with $a_1 < a_2$. In this program, these two choice variables are used to decide the strategy of extracting four syndromes.

Line 7 is a placeholder for the syndrome extraction circuits given in Fig 3.3(a). A module in MQCC can be viewed as macro over its parameters.

Lines 10-25 contains the part of the meta-program that expresses the possible schedules. Two **choice** statements decide to extract syndromes sequentially or in parallel, based on the value of choice variables c_1, c_2. The **choice** statement on lines 10-16 says that extracting syndrome $XZZXI$ and $IXZZX$ sequentially with only two ancilla qubits if $c1 = 0$ or in parallel with three ancilla qubits if $c1 = 1$. The **choice** statement on lines 19-25 does similarly for syndrome $XIXZZ$ and $ZXIXZ$.

3.2.3 MQCC Solver

Now let us see how the MQCC solver works. Its operation is shown in Fig 3.1(b).

Cost Expression Generator The MQCC solver's *Cost Expression Generator* (CEG) computes each input attribute for the meta-program to produce a formula that expresses that attribute's value in terms of the meta-program's choice variables. For the example in Fig 3.3, the CEG would produce the following formula for the **QubitCount** attribute:

$$\textbf{QubitCount}: \quad 7\delta_{c_1}^0\delta_{c_2}^0 + 8\delta_{c_1}^0\delta_{c_2}^1 + 8\delta_{c_1}^1\delta_{c_2}^0 + 8\delta_{c_1}^1\delta_{c_2}^1. \tag{3.2}$$

Here, the term δ_c^i evaluates to 1 if the value of c equals i, and evaluates to 0 otherwise. Referring to Fig 3.3, we can see that if $c_1 = c_2 = 0$ then all syndromes are extracted sequentially and only need seven qubits in total (five data qubits + two ancilla qubits); otherwise, syndromes are extracted in parallel in at least one choice statement, so eight qubits are needed in total.

In the general case, the number of terms in a CEG-produced formula relates to the number of valuations of the choice variables. We see this in the formula for **QubitCount**, which has four terms. However, we identified an optimized cost generation algorithm for attributes we call *additive*, which means that the attribute value of a program S can be computed from a linear combination of its sub-programs. The resulting expression will be linear in the number of choice variables. As an example, the **Fidelity** attribute is additive. The probability that no error occurs after two operations is the product of the individual operations' probabilities; the **Fidelity** attribute evaluates the logarithm of the probability, translating the product to a sum. A programmer may specify when an attribute is additive, which will prompt the CEG to optimize the generation of its cost expression (indicated as *Expression Optimizer* in the figure).

Consider our example in Fig 3.3. Suppose the **Fidelity** when extracting syndromes $XZZXI$ and $IXZZX$ sequentially or in parallel is -0.011 and -0.012, respectively. Then the fidelity of the first **choice** statement (lines 10-16) is calculated as $-0.011\delta_{c_1}^0 + (-0.012)\delta_{c_1}^1$. Similarly suppose the error of the second **choice** statement (lines 19-25) is calculated as $-0.013\delta_{c_2}^0 + (-0.012)\delta_{c_2}^1$. Since the **Fidelity** attribute is additive, the CEG sums these two:

$$\textbf{Fidelity}: \quad -0.011\delta_{c_1}^0 + (-0.012)\delta_{c_1}^1 + (-0.013)\delta_{c_2}^0 + (-0.012)\delta_{c_2}^1. \tag{3.3}$$

How this formula was computed is explained in Section 3.3.3.

Solution by SMT Encoding With the cost expressions generated for each object, MQCC encodes them as SMT instances based on the user's goal and constraints. Then MQCC uses an SMT solver to assign values to choice variables. In FQEC case, if the **QubitCount** threshold is 7, MQCC will choose $c_1 = 0, c_2 = 0$, and extract all syndromes sequentially. If there is more allowed qubits, MQCC will choose $c_1 = 0, c_2 = 1$ to maximize the Equation 3.3.

Scalability This example application demonstrates that some attributes are exponential in the choice space, but the choice space tends to be rather small in many applications. Equation 3.2 shows the formula of attribute QubitCount as an example. QubitCount is not an additive attribute. When composing several subprograms whose QubitCount is determined by a choice variable, evaluating the total QubitCount of the composed program has exponential complexity since each subprogram may share some qubits. But in FQEC applications, the number of choice variables depends on the count of syndromes to extract. The syndrome extraction program for the most commonly used quantum error correction code needs no more than ten syndromes (e.g., five in the perfect code, six in the Steane code, and eight in the Shor code). So MQCC has good scalability in this application.

3.2.4 Construction of MQCC Meta-programs

Ideally, an optimization designer will write a transpiler to automatically construct an appropriate MQCC meta-program given a normal target program. This allows normal programmers to use MQCC in a push-button fashion: They specify the tradeoff they want to optimize, provide their input program to that optimization's custom transpiler, and then invoke MQCC on the result, which produces the optimized program. Without a transpiler, e.g., perhaps when experimenting

with a new optimization of different tradeoffs, a programmer can manually construct a meta program.

In our experience, writing transpilers is straightforward: We have done so for every optimization presented in this paper. For the FQEC example just presented, the transpiler works by first detecting the substitutable usage of FQEC modules, e.g., `Extract_IXZZX` immediately followed by `Extract_XIXZZ` on the same registers. When it finds one, it generates a corresponding `choice` variable (as on line 6 in Fig 3.3(c)) and introduces a `choice` statement which selects between the original usage and one that happens in parallel (as on lines 10-25 in Fig 3.3(c)). Building a transpiler requires modeling the design space of the target problem, which will then determine the use of choice variables: their locations and granularity, and pieces of alternative QASM programs that will be stitched together. In our experience, the modeling step is natural given the problem, and the engineering overhead of building a transpiler is minimal; it took us 1 or 2 hours on average, for each optimization.

3.2.5 Implementation of MQCC

We implement MQCC in Python using `PLY` [122], which provides lex and yacc parsing tools. We choose Python for its popularity, accessibility, and flexibility. In particular, it allows the developers to easily define various attributes with Python classes. The SMT optimization for MQCC uses the Z3 SMT solver [123] version 4.8.9.

3.3 Formalization of MQCC

This section presents MQCC formally, including its meta-language, attribute definitions, and symbolic cost expressions (but not its app-specific transpilers). We prove that additive attributes' optimized cost procedure is correct.

3.3.1 Language Syntax

The formal syntax of MQCC meta-programs is shown in Figure 3.4. A meta-program P consists of a sequence of declarations D and a statement S. There are two kinds of declarations:[2] $RegDecl$ declares classical and quantum registers, and $VarDecl$ declares MQCC choice variables. A statement S can be empty, an operation O, a $case$, a $choice$, or a sequence of semicolon-separated statements.

- Operations O are primitive operations op over a list of registers $\overrightarrow{reg}$, according to a list of (optional) parameters $\overrightarrow{r}$. An operation could be a quantum gate, in which case op is the name of the gate and $\overrightarrow{reg}$ identifies input/output quantum registers, e.g., `cnot(q1,q2)`. An operation could also be a measurement, e.g., `measure(q1,c1)`, which measures `q1`'s contents and stores the result in `c1`. Operations could also be purely classical, e.g., `add(c1,c2)` to add `c1` to `c2` and store the result back in `c1`.

- A $case$ statement is a classical conditional. It chooses a branch based on the value of the classical register $creg$. Similar to OpenQASM, $creg$ is interpreted as an integer, using the bit at index zero as the low order bit.

[2] We omit **module** definitions and register arrays (e.g., **qreg** $q1[10]$) from the formal definition, which can be easily encoded.

- A *choice* statement chooses a candidate statement based on the valuation of choice variable var; a value i denotes statement S_i.

MQCC is a meta-language, in the sense that a meta-program P's semantics is determined by the quantum program that remains once its choice variables are decided. Let σ be a map from choice variables var to their values i. We can reduce P to a normal program by replacing each **choice**$(var)\{\overline{i : S_i}\}$ statement with (recursively reduced) branch S_k when $\sigma(var) = k$. The reduced program is trivially compiled to an equivalent OpenQASM program.

3.3.2 Attribute Semantics

An attribute A is particular characterization of a quantum program's execution. An attribute is defined according to a tuple $(T, \texttt{empty}, \texttt{op}, \texttt{case}, \texttt{value})$. Here, T is the type of the *state* of attribute A, and we can view a program statement S as an *attribute state transformer*: Given an initial state s and a valuation of choice variables to values σ, we say program statement S will produce attribute state s' when $[\![S]\!](\sigma, s) = s'$. Rules for computing the attribute state are given in Figure 3.5.

In the rules, we write $A.\texttt{x}$ to refer to the $\texttt{x}$ element of the attribute A's tuple. Each of these elements we define as follows:

- T is the type of an attribute's state used to compute the cost.
- $\texttt{empty} : T$ is the initial (empty) state.
- $\texttt{op} : T \times (OpID \times \overrightarrow{\mathbb{R}} \times \overrightarrow{reg}) \to T$ takes a state and an operation (its name and arguments), and produces a new state. It is used in the first rule of Figure 3.5.

- `case` : $T \times \mathit{reg} \times \overrightarrow{T} \to T$ takes a state, the guard choice register, and a list of states corresponding to each case branch, and generates the new state. It is used in the third rule of Figure 3.5.

- `value` : $T \to \mathbb{R}$ computes the cost of this attribute from the information stored in a state.

We define the tuples of two example attributes, **Fidelity** and **QubitCount**, in Section 3.3.4.

An attribute A's cost for a particular valuation of choice variables σ is simply $A.\texttt{value}([\![S]\!](\sigma, A.\texttt{empty}))$. We want to generate a formula that expresses all possible costs, so the SMT solver can decide what choice-variable valuation to use. We do so as follows. Let $\Sigma \subset (\mathit{Vars} \to \mathbb{Z})$ be the variables' possible valuations, then cost_A of attribute A is a function that maps an MQCC program into an expression over $Vars$:

$$\text{cost}_A(S) = \sum_{\sigma \in \Sigma} \delta_{\mathit{Vars},\sigma} \cdot A.\texttt{value}([\![S]\!](\sigma, A.\texttt{empty})).$$

Here δ is a variant of the Kronecker delta function: $\delta_{\mathit{Vars},\sigma} = \prod_{\mathit{var} \in \mathit{Vars}} \delta_{\mathit{var}}^{\sigma[\mathit{var}]}$, and $\delta_{\mathit{var}}^{i}$ is a unit expression that contains variable var, which equals 1 if var's value is i, and 0 otherwise. An example formula was given in Section 3.2.3, for attribute **QubitCount**. Each term in the formula is the depth for a different possible choice of σ—only one term will be non-zero for a given σ.

3.3.3 Additive Attributes

In general, the generated cost expression $\text{cost}_A(S)$ has a size exponential in the number of choice variables in S. We can use *additive attributes* to reduce this size.

Let σ_ϕ denote an arbitrary valuation of choice variables. Then an attribute A is additive if it

satisfies two conditions:

1. for any $s : T$, *op*, and valid *exps* and *regs*, we have

$$A.\texttt{value}(A.\texttt{op}(s, \textit{op}, \textit{exps}, \textit{regs})) = A.\texttt{value}(s) + A.\texttt{value}(A.\texttt{op}(A.\texttt{empty}, \textit{op}, \textit{exps}, \textit{regs}));$$

2. for any $s : T$ and **choice**-free statements S_i, we have

$$\begin{aligned} & A.\texttt{value}(A.\texttt{case}(s, \textit{creg}, [\![S_i]\!](\sigma_\phi, s)]_i)) \\ = \; & A.\texttt{value}(s) + A.\texttt{value}(A.\texttt{case}(A.\texttt{empty}, \textit{creg}, [\![S_i]\!](\sigma_\phi, A.\texttt{empty})]_i)) \end{aligned}$$

We directly derive the cost expression $\text{cost}_A^+(S)$ from the rules in Figure 3.6 for S that contain no **choice** statements inside branches of **case** (so as to meet the second condition). Let V be the maximal number of possibilities of a variables' values. Notice that $\text{cost}_A(S)$ has $O(V^d)$ terms where d is the number of choice variables, and $\text{cost}_A^+(S)$ has at most $O(|S| \cdot V)$ terms where $|S|$ is the number of constructs of S.

The following theorem shows the correctness of cost_A^+. Its proof is based on induction on S and provided in Appendix 6.2.1.

Theorem 3.3.1. *For a statement S such that there is no **choice** nested in **case**, we have $\textit{cost}_A^+(S) = \textit{cost}_A(S)$ for any valid valuation $\sigma \in Vars$.*

3.3.4 Examples of Attributes

Here we present two attributes, **Fidelity** and **QubitCount**, used in the FQEC problem in Section 3.2. We have developed five additional attributes in the case studies in Section 3.4. Here we use mathematical notation; in our implementation(Section 3.2.5), programmers use Python classes.

Fidelity Here we define the **Fidelity** attribute to characterize a circuit program's chances of not producing an error.

```
T = ℝ
empty = 0.0
value(s:T) = s
op(s:T, Op:OpID, exps:⃗ℝ, regs:⃗reg) = s + log(1 - calNoise(Op,exps,regs))
case(s:T, creg:reg, sbs:⃗T) = min sbs
```

The type T of **Fidelity**'s state is $\mathbb{R}$, i.e., the state is a real number, representing the $log(1-P)$ where P is the probability any error occurs. The `empty` state is 0.0—an empty circuit program will never introduce error so $P = 0$ and $log(1 - P) = 0.0$. The **Fidelity** attribute's cost is the fidelity itself, so the `value` function simply returns its argument `s`. The remaining two elements, `op` and `case`, define the fidelity of the program's basic building blocks:

- The `op` function increases the program's total fidelity by the given operation's fidelity (which depends on the operation and the qubits it uses). This fidelity is calculated by the function `calNoise`, which can be implemented variously based on the target machine.
- For the `case` function, the parameter `sbs` refers to the fidelity computed for each branch of

the **case**. Since we do not know which branch will be chosen in run-time, we conservatively use the min of these.

Fidelity is an additive attribute so MQCC can generate an optimized cost expression cost_A^+. We can see that for the example in Section 3.2.3.

QubitCount The second attribute used in Section 3.2 was **QubitCount**, which characterizes the maximum count of operations applied to any qubit in a circuit program. Here is its formal definition:

```
T = Set[Qubit]
empty = ∅
value(s:T) = |s|
op(s:T, Op:OpID exps:ℝ⃗, regs:reg⃗) = s∪{q | q ∈ regs, q is a qubit}
case(s:T, creg:reg, sbs:T⃗) = ⋃_{a∈sbs} a
```

The type T of **QubitCount**'s state is a set of qubits. The `empty` element of **QubitCount** is an empty set. The cost of **QubitCount** is the cardinality of the qubit set. The elements `op` and `case` are defined thus:

- The `op` function unions the original qubit set s with the set that contains the qubits used in the input operation Op.
- The `case` function unions the qubit sets of all branches; we do not know which branch will be chosen at run-time, so we must conservatively remember them all.

QubitCount is not an additive attribute, so we must enumerate all possible valuations when computing the cost; an example formula is shown in Section 3.2.3.

3.3.5 Limitations

MQCC is limited in the optimization problems it can express. In particular, MQCC restricts choice variables to a finite number of predefined options. Thus, it cannot represent tradeoffs that consider an infinite number of choices; e.g., it cannot decide the real-valued rotation angle of a parameterized gate. Moreover, non-additive attributes cannot be used when a large number of choice variables is involved, for scalability reasons. Despite these limitations, MQCC can express a variety of interesting problems useful for near-term architectures, as the next section shows. Moreover, programming attributes in Python afford a fair degree of flexibility; we implemented the eight attributes in this paper without any difficulty.

3.4 Case Studies

We evaluate MQCC's utility by evaluating its use in five case studies. The first considers MQCC's performance on the FQEC problem. The second develops a new optimization that trades off accuracy for circuit volume in QFT and QPE implementations; we show that traditional by-hand approaches fare worse than MQCC's approach. The third and the fourth encode previously proposed optimizations for multi-programming and crosstalk mitigation [1, 99], while the fifth is a novel combination of the third and the fourth cases, showcasing how MQCC facilitates composition.

We apply these optimizations to a benchmark of NISQ-ready programs and find that MQCC runs quickly, and the optimized programs demonstrate the intended effects when run on real quantum hardware (matching or bettering previously reported results). We apply MQCC's optimizations to the middle-scale benchmark suite collected from QASMBench [2], and a large scale

QFT circuit; these programs are too big to run on today's quantum hardware. We find that even with these larger programs, MQCC scales well. Finally, we apply a sensitivity study for MQCC to analyze how the input noise parameters affect the programs generated by MQCC for CM and MP tasks.

3.4.1 Fault-tolerant Quantum Error Correction

We describe the fault-tolerant quantum error correction (FQEC) problem and MQCC's programmed solution in Section 3.2. The definitions of its relevant attributes **Fidelity** and **QubitCount** are given in Section 3.3.4.

Evaluation Existing quantum hardware does not support a complete quantum error correction procedure and the common evaluation for quantum error correction is based on hypothetical machine errors and simulators. We follow the same evaluation methodology as Chao and Reichardt [4], Reichardt [5], which simulate error correction using a standard depolarizing noise error model and collect the logical failure rate of different syndrome extraction circuits. The standard depolarizing noise model [6] is described in Section 3.2.2.2. In the evaluation, we assume the "depolarizing" error e_p of all qubits is in Gaussian Distribution $G(\mu, \sigma)$. We use the Qiskit noise simulator to simulate the error correction procedure using the $[\![5, 1, 3]\!]$ code for 10^5 rounds based on three syndrome extraction strategies: (1) extracting syndromes by Chao and Reichardt [4]'s circuit (Fig 3.2(c)) sequentially; (2) extracting syndromes in parallel by Reichardt [5]'s circuits (Fig 3.2(d)) and (3) extracting syndromes by circuits produced by MQCC.The error correction procedure of $[\![5, 1, 3]\!]$ code is a small-size circuit ($7 \sim 8$ qubits, < 100 gates) and MQCC solver gives its solution instantly.

Fig 3.7 shows the logical error rate of different strategies with various σ/μ. The result is intuitive: under the same μ, larger σ means that some qubits have very bad fidelity; the sequential strategy performs better since it uses fewer qubits and can avoid using them. MQCC also tends to generate programs close to the sequential strategy in this case. When σ is small, parallel strategy becomes better since it uses fewer qubit measurements, and MQCC tends to generate programs close to the parallel strategy. The evaluation shows that MQCC can always achieve the minimum logical error rate given various qubit error rates.

3.4.2 Trade-off between Accuracy and Resources

The accuracy of a numeric computation is limited by the resources devoted to computing it. We can use MQCC to balance the accuracy/resource tradeoff.

3.4.2.1 Approximate QFT

Quantum Fourier Transform is a crucial part of many quantum information processing algorithms. Consider the n-qubit Quantum Fourier Transform circuit shown in Fig 3.8. This circuit has $O(n^2)$ gates. However, there are many rotations by small angles that do not affect the final result very much. The standard way to implement an Approximate Quantum Fourier Transform (AQFT) is by pruning these small-angle rotation gates [34]. Given a unitary U and its approximate one V, we use $\|U - V\|$ to estimate the standard distance between the exact circuit and the approximate one, where $\|\cdot\|$ is the spectral norm. We apply the union bound to upper bound the distance between circuits by the sum of the distance between corresponding gates. For each qubit q_j, if we prune the gates R_k, where $k > h_j$ for a threshold h_j, the approximation error

ϵ_j on q_j can be estimated by $\epsilon_j < \frac{1}{2^{h_j}}$. We aim to keep the total approximation error less than a threshold ϵ_{QFT}; i.e., $\sum_{i=1}^{n} \epsilon_i < \epsilon_{QFT}$.

Our MQCC goal is to minimize the gate count in the AQFT circuit while satisfying the accuracy bound ϵ_{QFT}. To do this, we allocate one choice variable for each qubit in AQFT to decide its threshold h_j. The MQCC program generated by the transpiler is

```
\\n-bit AQFT
\\controlled phase rotation gates for q[1]
choice({0,1,2,...}){
    0 : CRZN(h1_0, q[1]);
    1 : CRZN(h1_1, q[1]);
    2 : CRZN(h1_2, q[1]);
    ...
}
\\controlled phase rotation gates for q[2]
choice({0,1,2,...}){
    0 : CRZN(h2_0, q[2]);
    1 : CRZN(h2_1, q[2]);
    2 : CRZN(h2_2, q[2]);
    ...
} ...
```

Term `{0,1,2,...}` in the `choice` statement indicates an *anonymous* (undeclared) choice variable whose value ranges in `{0,1,2,...}`. Term `CRZN(h, q[j])` is a module representing the various controlled phase rotation gates on qubit `q[j]`, controlled by qubits `q[j+1],...,q[h]` (so small angle gates controlled by `q[h+1],...,q[n]` are pruned). The parameters `ha_b` (i.e., `h1_0, h1_1,...`) in the `CRZN` specify the threshold. We also define two new attributes,

Approximation and **GateCount**, for MQCC to estimate program's approximation error and the total gate count, respectively. (Both attributes are additive, and similar to **Fidelity**.)

For evaluation, we use MQCC to minimize a 50-qubit AQFT's gate count, given various ϵ_{QFT}. We compare against a baseline result is produced by pruning a fixed number—call it h_s—of small angle gates for each qubit [34]. We chose h_s by enumerating all possible values and choosing the one minimizes the circuit's gate count. Fig 3.9 graphs the result, which shows that MQCC cuts down more gates than the naive pruning strategy since MQCC adjusts the pruning threshold for each qubit independently.

3.4.2.2 Quantum Phase Estimation

Quantum Phase Estimation (QPE) [121] is a direct application of QFT. It estimates the eigenphases of an oracle unitary transformation. Consider the implementation in Fig 3.11. The top k qubits yield a k-bit approximation error to the phase. The value of k to choose for QPE depends on the desired accuracy [124]. To make the success probability of QPE reach 50%, the desired accuracy bound ϵ_{QPE} can be bounded by k as $\epsilon_{QPE} \leq 16\pi/(2^k - 1)$. QPE requires an AQFT in the final step. Therefore, to achieve an overall target accuracy bound ϵ_{total}, we need to keep $\epsilon_{QPE} + \epsilon_{QFT} < \epsilon_{total}$.

Our MQCC goal is to minimize the circuit's volume but ensure that the circuit's approximation error does not exceed the desired accuracy bound. To achieve this, a choice variable is used to decide the value of k, and the choice of various k can be encoded by the following MQCC program

```
1 choice({0,1,...}){
2     0:  Control_U(k0);
```

```
3         AQFT(k0);
4     1:  Control_U(k1);
5         AQFT(k1);
6     ... }
```

Here, `Control_U(k)` represents the list of controlled oracle gates in the QPE (controlled-$U, U^2, ..., U^{2^k}$ in Fig 3.11). The input parameter `k` is an integer and represents the number of controlled qubits used in the QPE circuit. `k0,k1,...` are specified choice for k. This program reuses the code in the AQFT examples as the `AQFT` module so that MQCC can figure out how many qubits are required and how many small angle rotation gates will be removed from the AQFT simultaneously. We reuse the **Approximation** attribute defined in the AQFT example to calculate the circuit's approximation error. We also define a **Volume** attribute for MQCC to calculate the circuit's volume.

For evaluation, we use MQCC to minimize a QPE circuit's volume given various ϵ_{total}. The number of qubits in the QPE ranges in $15 \sim 30$, and Fig 3.12 shows the result. The baseline result is produced by the natural optimization idea for a circuit with multiple parts: divide ϵ_{total} into $\gamma\epsilon_{total} + (1-\gamma)\epsilon_{total}$ with some appropriate ratio $\gamma \in (0, 1)$, then use $\gamma\epsilon_{total}$ and $(1-\gamma)\epsilon_{total}$ as thresholds to optimize the controlled unitary part and the AQFT part in QPE separately. We decide γ by enumerating $\gamma \in (0, 1)$ and the one that minimizes the circuit's volume is chosen. The experiment shows that MQCC can cut down more circuit volume, especially in small ϵ_{total} cases. We also measure MQCC's running time on AQFT circuits with large volume. A circuit's space-time volume is defined as the multiplication of its depth and qubit count [103]. Fig 3.10 shows the result.

3.4.3 Multi-Programming Quantum Computers

Quantum computer multi programming was proposed by Das et al. [1]. The idea is simple: Given two quantum circuits A and B, instead of running A to completion and then B, we can create a combined circuit $A + B$ that allocates A and B to distinct qubits so they can be run in parallel. Doing so better utilizes the computer but may decrease the quality of the result. This is because different qubits of a NISQ computer have different error rates; running A and B serially on the highest-fidelity qubits will reduce overall error.

The goal of the *multi-programming* (MP) problem is to maximize utilization while keeping the fidelity above a stated threshold θ. Whether to run in serial or in parallel depends on the programs A and B, the noise characteristics of the hardware, and θ. Das et al. develop a custom solver for this problem; we can program it using the MQCC framework.

Solution of MQCC In this case, we define the relevant attribute **Depth** and reuse the attribute **Fidelity** defined in Section 3.3.4. The **Depth** attribute applied to S computes, for each qubit, the length of the sequence of S's gates operating on that qubit, and then returns the maximum over all qubits. In the multi-programming problem, minimizing a circuit's depth will minimize the time that the quantum chip needs to finish executing all programs. In this problem, given a specific group of programs that need to run, the workload of the quantum chip is fixed and minimizing depth is equivalent to maximizing utilization of the chip. MQCC's goal is to minimize **Depth** and the constraint is keeping **Fidelity** above threshold θ.

We assign a choice variable for each application to decide which qubits are allocated to this application. Different solutions for these choice variables trade off noise for utilization. For

example, Fig 3.13 shows the code of two quantum applications to multi-program. `Bell1` prepares the Bell state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, while `Bell2` prepares $\frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$. Both applications need two qubits; we suppose our target computer can schedule them on either qubits {q[1],q[2]} or qubits {q[7],q[8]}. With this information the MQCC transpiler (which uses Qiskit's qubit allocation and mapping library to find low-error areas and routing paths) will produce the meta-program in Fig 3.3. Then, taking this meta-program, the **Depth** and **Fidelity** attributes, and the optimization goal and constraint ("minimize depth with bounded fidelity"), MQCC will solve for the choice variables to produce a final program that schedules `Bell1` and `Bell2`. Suppose the MQCC solver's *Cost Expression Generator* produce the following formula for the **Depth** and **Fidelity** attribute.

$$\textbf{Depth}: \quad 7\delta^0_{c_1}\delta^0_{c_2} + 4\delta^0_{c_1}\delta^1_{c_2} + 4\delta^1_{c_1}\delta^0_{c_2} + 7\delta^1_{c_1}\delta^1_{c_2}$$

$$\textbf{Fidelity}: \quad (-0.045)\delta^0_{c_1} + (-0.066)\delta^1_{c_1} + (-0.027)\delta^0_{c_2} + (-0.043)\delta^1_{c_2}$$

Here, the term δ^i_c evaluates to 1 if the value of c equals i, and evaluates to 0 otherwise. Referring to Fig 3.13, we can see that if $c_1 = c_2$ then the two applications will run in sequence, yielding a total depth of 7; otherwise they can run in parallel, and the longest sequence is the max of the two, which is 4. When $c_1 = 0$ and $c_2 = 1$ (running in parallel), **Depth** is 3 while **Fidelity** $= (-0.045) + (-0.043) = -0.088$. When $c_1 = 0$ and $c_2 = 0$ (running in sequence), **Depth** is 5 while **Fidelity** is higher: $(-0.045) + (-0.027) = -0.072$. The result is intuitive: compared to running two applications sequentially on the low error-rate area {q[1],q[2]}, running them in parallel yields a higher error rate but a faster execution time.

Evaluation We use the same applications (Table 3.1) and follow the same evaluation methodology as Das et al. [1]. We package multiple applications as a group and generate several groups. Applications in each group are then executed in three ways: (1) in parallel, as the baseline; (2) in sequential, as the baseline; and (3) multi-programmed by MQCC. Das et al. [1] does not provide their source code so we do not compare with them directly. The error probability data used by MQCC is collected from the target machine's daily calibration. Prior work [1] multi-programs exactly two applications but MQCC can naturally accept any number; we demonstrate several three-application cases.

For each group, we run 8192 trials on IBMQ Boelingen (20 qubits) and Rigetti Aspen-9 (32 qubits). A trial in which all applications in the group give the correct result is regarded as *successful*. The rate of success - the *Probability of a Successful Trial* (PST) - is used to evaluate the reliability. As Figure 3.14 demonstrates, since we set a small error threshold in the evaluation, solutions based on MQCC successfully maintain higher reliability of all groups compared to running applications in parallel and are closed to running application in sequential. If we use a larger error threshold, MQCC will produce a solution with PST close to running applications in parallel but with less circuit depth compare to in sequential.

Application	**Description**	**Qubits**	**Gates**	**CNOTs**
bv3	Bernstein-Vazirani [125]	3	8	2
bv4	Bernstein-Vazirani [125]	4	11	3
h3	Hamiltonian Simulation	3	11	4
h4	Hamiltonian Simulation	4	15	6
Toff3	Toffoli gate	3	15	6
Fred3	Fredkin gate	3	17	8
Pere3	Peres gate	3	16	7

Table 3.1: Applications used by Das et al. [1].

3.4.4 Circuit Reschedule for Crosstalk Mitigation

Machine-dependent *crosstalk* arises when certain gates are executed in parallel. It is a major source of noise in NISQ systems [99, 126]. Figure 3.15 shows the layout of the IBMQ Boelingen machine with links between *high crosstalk* gate pairs. Sarovar et al. [127] propose a protocol for detecting and localizing the crosstalk in hardware. Niu and Todri-Sanial [128] report several protocols for characterizing crosstalk in NISQ hardware, and discuss different crosstalk mitigation methods in both hardware and software. In software, crosstalk can be avoided by running problematic gates in sequence, but doing so increases circuit depth, which increases the chance of decoherence errors. Murali et al. [99] propose a software-based method to balance this tradeoff. They employ an SMT-based scheduler that judiciously decides whether gates should be executed in parallel or serially. The schedule of each gate is encoded by two variables—the gate's start time and its duration—and these in turn are included in an SMT instance that encodes the crosstalk along with other constraints based on features of each gate. Ding et al. [129] proposes a software solution to alleviate crosstalk by systematically tuning qubit frequencies according to input programs. However, current quantum hardware often does not allow dynamical qubit frequency adjustment; e.g., IBM's current quantum platforms are built with fixed qubit frequencies.

Solution of MQCC With MQCC We can *program* a solution like Murali et al. [99]. As with multiprogramming, a transpiler encodes different gate schedules via choice variables. It makes use of the OpenQASM `barrier` operation described in Section 1.3.3, which is also used in Murali et al..

For example, consider the following module for a `CNOT` gate:

```
module cnotb(c,q1,q2){
    choice (c){
        0:  cnot(q1,q2);
        1:  barrier(q);  cnot(q1,q2);
}}
```

When $c = 0$, the gate is applied normally (maximum parallel); Otherwise, suppose the program declares quantum registers with **qreg** `q[n]`. The `barrier(q)` forces the `CNOT` to be executed sequentially only after all gates on `q` are finished. We encode all those `CNOT`s that use different qubits from the their precursor `CNOT` into this form.

Our goal is to minimize both decoherence error and crosstalk. We define attributes **Crosstalk** and **Decoherence** which take into account the expected appearance of the `barrier` operations in the meta-program.

Evaluation We follow the evaluation methodology of Murali et al. In particular, we use the same meet-in-the-middle SWAP sequences as their benchmarks. The reason this is a sensible benchmark is that in superconducting QC systems, CNOTs are permitted only between adjacent qubits. To apply a CNOT between two far-away qubits, compilers usually insert a sequence of SWAP operations that move two qubits into adjacent locations through exchanges. For example, in IBMQ Boeblingen, `CNOT 15 8` can be implemented as `SWAP 15 16; SWAP 16 11; SWAP 8 7; SWAP 7 12; CNOT 11 12`.

The IBMQ Poughkeepsie and Johannesburg machines used in the evaluation by Murali et al. are currently unavailable, so we use similar SWAP sequences based on IBMQ Boeblingen. We run 8192 trials for each SWAP sequence and consider those with desired outputs to be successful. We

compare the PST of four scheduling strategies: running all instructions serially (`Seq`); running all instructions maximally in parallel (`Par`) which is the default strategy used by Qiskit; the strategy produced by Murali et al. (`Xtalk`); and the strategy produced by MQCC. Figure 3.16 shows the result. Circuits generated by MQCC always have higher PST than `Seq` and `Par`. They have performance similar to `Xtalk`.

3.4.5 Multi-programming with Crosstalk Mitigation

In this section, we combine problems from Section 3.4.3 and Section 3.4.4 to show MQCC's ability to handle multiple optimization tradeoffs simultaneously.

Motivation Crosstalk can happen within a single application, but also across different applications that are multi-programmed to be in parallel. MQCC can be used to directly combine the previous crosstalk mitigation and multiprogramming applications (thus informing scheduling decisions by crosstalk-induced noise, but can also include other methods for crosstalk mitigation as well: e.g., transforming circuits into crosstalk-resistant forms.

Figure 3.17(a) shows two equivalent circuits. In the presence of another CNOT gate, as shown in Figure 3.17(b), the first structure may introduce much higher crosstalk than the second one since the crosstalk between two CNOT gates is much greater than the crosstalk between a CNOT and a single-qubit gate. One can encode choices between these equivalent forms by MQCC choice variables, to automatically determine the best one to use.

Solution of MQCC Our goal is to minimize **Depth** while keeping for **Crosstalk** under an upper bound and the **Fidelity** above a lower bound[3], instead of **Crosstalk** only.

[3]We do not add the **Decoherence** attribute because is effectively represented by **Depth**, which we are minimizing.

Consider equivalent circuit structures as shown in Figure 3.18(a), the choice of which can be encoded as Fig 3.18(b):

This circuit is a common part of many quantum applications such as Bernstein-Vazirani algorithm (BV) (Figure 3.19(a)) and Hamiltonian Simulation (HS). For example, the 3-qubit BV circuit in Fig 3.19(a) can be coded as Fig 3.19(b). We similarly encode Hamiltonian Simulation applications involving structure in Figure 3.18(a) into MQCC.

We then reuse the MQCC setup in Section 3.4.3 to multi-program these applications. MQCC will determine which form to use for each `twoCnot` in addition to the choice variables for multi-programming.

Evaluation We use the instances of BV and HS shown in Table 3.1 for evaluation. This size-restriction is due to the size limit of IBMQ Boebligen.

We compare the PST and circuit depth among three scheduling strategies: running benchmarks in serial (`seq`), multi-programmed by MQCC without considering crosstalk (`multi-p`), and when considering crosstalk (`multi-c`). As shown in Fig. 3.20(b), workloads scheduled by `multi-p` and `multi-c` have lower circuit depth than `seq`. However, as shown in Fig. 3.20(a), the PST of `multi-p` is lower than both `seq` and `multi-c` because of high crosstalk,[4] where the difference is more significant when multi-programming more applications. In contrast, `multi-c` could always maintain a comparable PST to `seq` while reducing the circuit depth significantly.

[4]There is one exception with the case of bv3-bv4 and it might be caused by the fluctuation of the quantum machine.

3.4.6 Scalability Study of QSynth for MP and CM tasks

The running time of QSynth on the small-scale benchmarks in Table 3.1 is less than 0.01s for all problems. To demonstrate QSynth's scalability, we measure QSynth's running time on some larger benchmarks, which cannot run on current real machines due to hardware constraints. These benchmarks are collected QASMBench [2], an open-source OpenQASM benchmark suite. QASMBench [2] collects commonly seen quantum algorithms and routines from a variety of domains with distinct properties. These experiments are carried on Intel Core i7-5960X in Ubuntu 20.10 environment. The running time of each benchmark is the average of three trials.

Table 3.2 compares QSynth's running time with Xtalk's [113] running time for mitigating the crosstalk in all QASMBench middle-scale benchmarks. Both running times are measured on the same hardware. MQCC's running time depends more on the circuit's structure rather than its size. For example, the "vqe_uccsd8" is the longest circuit. But most CNOTs in it have to be executed in serial due to topological constraints and cannot cause any crosstalk. So QSynth can find a solution for "vqe_uccsd8" in a reasonable time. On the contrary, Xtalk assigns SMT variables and generates SMT instances for every gate in the circuit, even though some of the gates can never cause crosstalk due to topological constraints. So Xtalk's running time highly depends on the number of gates in the circuit, and it times out when the circuit grows large.

We use QSynth to multi-program QASMBench's middle-scale benchmarks and the left tabular in Table 3.3 shows QSynth's running time. The complexity of solving MP with QSynth depends on the number of applications to multi-program. Since these benchmarks are too big to run on today's quantum hardware, we use the architecture information from qiskit's noise simulator. Then we choose those benchmarks for which QSynth needs more than 0.1s to mitigate their

crosstalk (i.e., benchmark 1,3,4,5,6,7,12,13,14,16,17) and group them in pairs to test QSynth's MP-CM running time. We discard benchmark 9 since QSynth already times out(> 60min) for mitigating its crosstalk. The right tabular in Table 3.3 shows this result.

ID	Benchmarks	#Qubits	#Gates	#CNOT	Time (s): MQCC	Time (s): Xtalk
1	adder	10	142	65	**0.1120**	0.2645
2	bv	14	41	13	0.0251	**0.0143**
3	seca	11	216	84	**1.983**	3.567
4	ising	10	480	90	**1.932**	237.8
5	multiply	13	98	40	0.2670	**0.1121**
6	qf21	15	311	115	**0.1097**	22.18
7	qft	15	540	210	**34.98**	12 min
8	qpe	9	123	43	**0.0422**	0.3512
9	sat	11	679	252	Timeout	Timeout
10	cc	12	22	11	0.0231	**0.00312**
11	simons	6	44	14	0.0341	**0.00512**
12	vqe_uccsd6	6	2282	1052	**1.847**	Timeout
13	vqe_uccsd8	8	10808	5488	**15.58**	Timeout
14	qaoa	6	270	54	**0.202**	13.21
15	bb84	8	27	0	0.0092	**0.0021**
16	dnn	8	1008	192	**5.674**	Timeout
17	multiplier	15	574	246	**16 min**	46 min

Table 3.2: Running time of solving CM by QSynth for all middle-scale circuits in QASMBench. The benchmarks are described in Li et al. [2]. Timeout is 60 minutes, shortest times in **bold**.

ID	MP Running time (s)
1-5	0.212
1-8	1.89
1-11	12.6
1-14	101.2
1-17	15min

ID	MP-CM Running time (s)
1,3	2.61
4,5	21.27
6,7	65.12
12,13	36.92
14,16	16.59
16,17	Timeout

Table 3.3: **Left table:** Running time of multi-programming the benchmarks from a to b in Table 3.2 by QSynth. **Right table:** Running time of QSynth for handling multi-programming and crosstalk mitigation simultaneously.

3.4.7 Sensitivity Study of QSynth for MP and CM tasks

In this section, we discuss how the input parameters for MP and CM tasks affect the structure of output circuits from MQCC.

Fig 3.21 shows the *Depth Growth Rate* of output circuits from MQCC for benchmarks *vqe_uccsd8, vqe_uccsd6* and *dnn* (the largest three in middle QASMBench, see Table 3.2) for CM task under different error proportions. In CM task, MQCC receives an input circuit from the user and uses `barrier` operations to serialize gates to mitigate crosstalk; these operations increase the circuit's depth. So we define depth growth rate of the circuit C generated by MQCC as $(d_C - d_i)/d_i$, where d_C is C's depth with barriers added and d_i is the depth of the circuit MQCC receives as input. Thus, higher circuit depth growth rate indicates MQCC serializes more gates. $w_{decohere}/w_{cross}$ indicates the proportion of decoherence to crosstalk in the final error: When $w_{decohere}/w_{cross}$ is 0, crosstalk is the only error and MQCC tries to avoid any crosstalk by serializing some gates, which increases the circuit's depth. When $w_{decohere}/w_{cross}$ is 20, decoherence is the dominant error and MQCC decides to run the circuit in maximum parallel to minimize the decoherence, which makes the depth growth rate 0.

Fig 3.22 shows the depth of circuits generated by MQCC for MP task under different error thresholds. The benchmark is described in Section 3.4.3. We suppose measurement and qubit reset operations add one unit to the depth. When the input error threshold is 10^{-3} (used for the experiments in Fig 3.14), MQCC uses qubits with the highest fidelity to run all applications sequentially, which leads to the deepest circuit. When the error threshold becomes larger, MQCC utilizes qubits with lower fidelity to run applications in parallel, leading to circuits with less depth.

```
module dualCZ(q1, q2){
    h(q1);
    cnot(q1,q2);
    h(q1);
}

module Extract_XZZXI(data,anc,r){
    \\Extract sydrome XZZXI
    \\The same circuit as in Fig.1(a)
    reset(anc[0],anc[1]);
    h(anc[1]);
    dualCZ(data[0], anc[0]);
    cnot(anc[1], anc[0]);
    cnot(data[2], anc[0]);
    cnot(data[3], anc[0]);
    cnot(anc[1], anc[0]);
    dualCZ(data[3], anc[0]);
    measureZ(anc[0],r[0]);
    measureX(anc[1], r[1]);
}
module Extract_IXZZX(data,anc,r) {...}
    \\The same circuit as in Fig.1(b)
\\Similar circuits to extract other syndromes
module Extract_XIXZZ(data,anc,r) {...}
module Extract_ZXIXZ(data,anc,r) {...}

module Extract_both_12(data,anc,r){
    \\Extract XZZXI and IXZZX in parallel
    \\The same circuit as in Fig.1(c)
    ...
}
module Extract_both_34(data,anc,r){
    \\Extract XIXZZ and ZXIXZ in parallel
    ...
}
```

(a)

```
qreg data[5];
qreg anc1[2];
creg r1[2], r2[2], r3[2], r4[2];
... \\Module declaration from part (a)
Extract_XZZXI(data, anc1, r1);
Extract_IXZZX(data, anc1, r2);
Extract_XIXZZ(data, anc1, r3);
Extract_IXZZX(data, anc1, r4);
```

(b)

```
\\Register and variable declarations
qreg data[5];
qreg anc1[2], anc2[3];
creg r1[2], r2[2], r3[2], r4[2];
creg r5[3], r6[3];
fcho c1,c2 = {0, 1};
... \\Module declaration from part (a)
\\Main part of the meta-program
\\Extract syndrome XZZXI and IXZZX
choice (c1){
    0:
    Extract_XZZXI(data, anc1, r1);
    Extract_IXZZX(data, anc1, r2);
    1:
    Extract_both_12(data, anc2, r5);
};

\\Extract syndrome XIXZZ and ZXIXZ
choice (c2){
    0:
    Extract_XIXZZ(data, anc1, r3);
    Extract_IXZZX(data, anc1, r4);
    1:
    Extract_both_34(data, anc2, r6);
};
```

(c)

Figure 3.3: (a) Predefined modules of the $[\![5,1,3]\!]$ code syndrome extraction circuits. (b) Fault-tolerant syndrome extraction for $[\![5,1,3]\!]$ code given by programmer. (c) MQCC meta-program produced by transpiler running on (b).

$n \in \mathbb{N} \quad i \in \mathbb{Z} \quad r \in \mathbb{R} \quad var \in Vars \quad op \in OpID$
$qreg \in Quantum\,reg. \quad creg \in Classical\,reg.$

$$
\begin{aligned}
reg &::= qreg \mid creg \\
P \in Program &::= \overrightarrow{D}\ S \\
D \in Declaration &::= RegDecl \mid VarDecl \\
RegDecl &::= \textbf{qreg}\ qreg; \mid \textbf{creg}\ creg; \\
VarDecl &::= \textbf{fcho}\ var = \{\overrightarrow{i}\}; \mid \textbf{fcho}\ var = [i_1, i_2]; \\
S \in Stmt &::= \epsilon \mid O \mid case \mid choice \mid S; S \\
O \in Operation &::= op(\overrightarrow{r}, \overrightarrow{reg}) \\
case &::= \textbf{case}(creg)\{\overline{i:\ S_i}\} \\
choice &::= \textbf{choice}(var)\{\overline{i:\ S_i}\}
\end{aligned}
$$

Figure 3.4: Formal syntax of MQCC meta-programs.

$$\frac{S = \textit{op(exps, regs)}}{[\![S]\!](\sigma, s) = A.\texttt{op}(s, \textit{op, exps, regs})} \qquad \frac{}{[\![S_1; S_2]\!](\sigma, s) = [\![S_2]\!](\sigma, [\![S_1]\!](\sigma, s))}$$

$$\frac{S = \textbf{case}(\textit{creg})\{\overline{i:\ S_i}\}}{[\![S]\!](\sigma, s) = A.\texttt{case}(s, \textit{creg}, [[\![S_i]\!](\sigma, s)]_i)} \qquad \frac{S = \textbf{choice}(\textit{var})\{\overline{i:\ S_i}\} \qquad k = \sigma[\textit{var}]}{[\![S]\!](\sigma, s) = [\![S_k]\!](\sigma, s)}$$

Figure 3.5: The semantics of MQCC program as an attribute-transformer over the program's statement S, using attribute A. $\sigma[var]$ is the valuation of choice variable var.

$$\frac{S = \textit{op(exps, regs)}}{\text{cost}^+_A(S) = A.\texttt{value}(A.\texttt{op}(A.\texttt{empty}, \textit{op, exps, regs}))} \qquad \frac{}{\text{cost}^+_A(S_1; S_2) = \text{cost}^+_A(S_1) + \text{cost}^+_A(S_2)}$$

$$\frac{S = \textbf{case}(\textit{creg})\{\overline{i:\ S_i}\} \quad S_i \text{ is } \textbf{choice}\text{-free}}{\begin{array}{c}\text{cost}^+_A(S) = \\ A.\texttt{value}(A.\texttt{case}(A.\texttt{empty}, \textit{creg}, [[\![S_i]\!](\sigma_\phi, A.\texttt{empty})]_i))\end{array}} \qquad \frac{S = \textbf{choice}(\textit{var})\{\overline{i:\ S_i}\}}{\text{cost}^+_A(S) = \sum_{i \in \bar{i}} \delta^i_{\textit{var}}\, \text{cost}^+_A(S_i)}$$

Figure 3.6: The cost expression of choice-in-case-free S for *additive* attributes. Here $\bar{i}$ is the set of enumerated values that variable *var* can take.

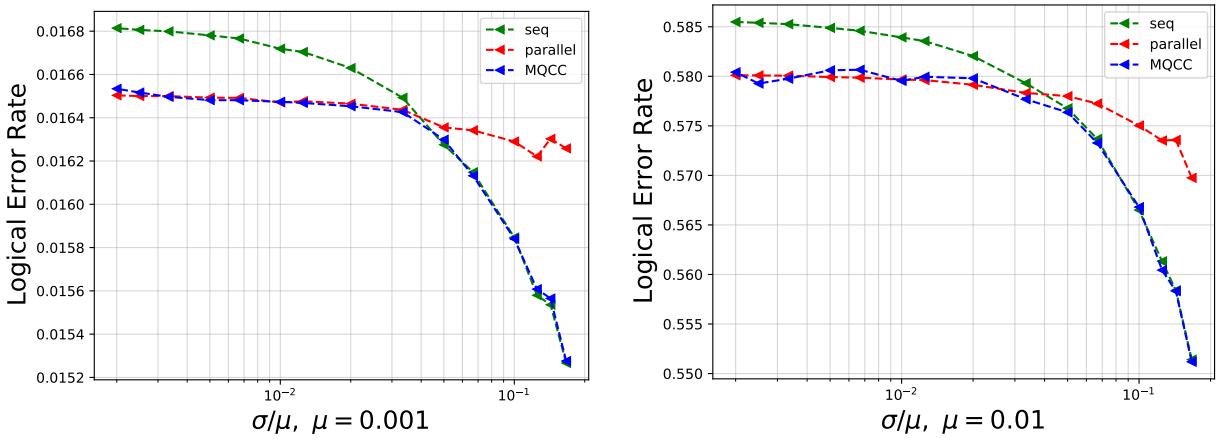


Figure 3.7: Logical error rates for simulated error correction compared with previous strategies (Lower is better); using the parallel syndrome-extraction circuit of Fig 3.2(d), in red, the sequential syndrome-extraction circuit of Fig. 3.2(c), in green, and the syndrome-extraction produced by MQCC, in blue. Errors are from a standard depolarizing noise model [6], with the depolarizing error of all qubits in Gaussian distribution $G(\mu, \sigma)$.

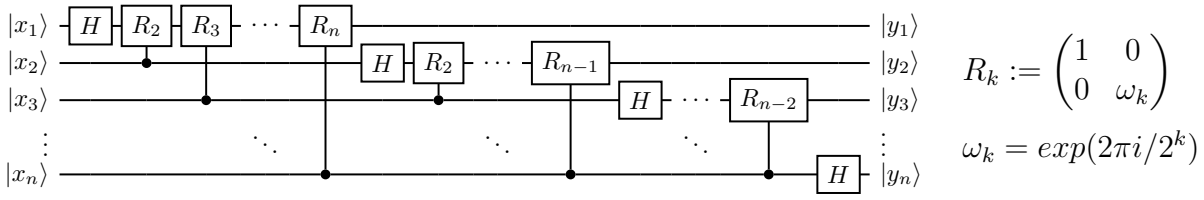


Figure 3.8: Quantum circuit for QFT algorithm

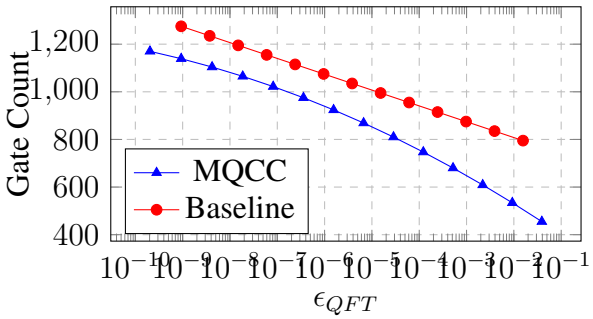


Figure 3.9: Trade-off between the gate count and the accuracy for 50-qubit AQFT circuit. Lower gate count is better.

Qubits	Space-time Volume	Running time
20	1.84k	2.28s
30	4.08k	8.85s
40	7.04k	12.93s
50	11k	39.33s

Figure 3.10: MQCC's running time on various size AQFT circuits.

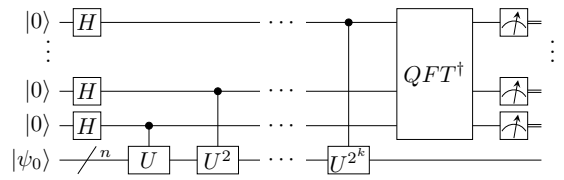


Figure 3.11: Quantum circuit performing Quantum Phase Estimation on an n-qubit system with an accuracy of $k+1$ bits. U is the given oracle unitary.

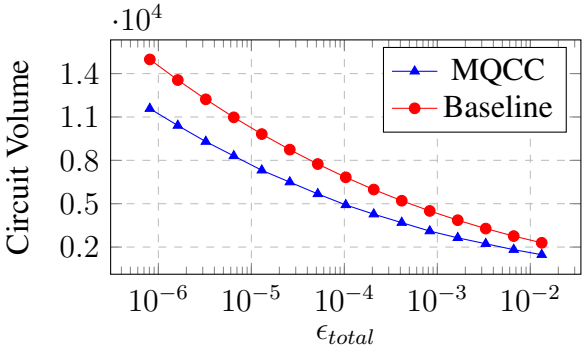


Figure 3.12: Trade-off between circuit volume and the precision for QPE circuit. Here lower circuit volume is better.

```
module Bell1(q1,q2, r){
    reset(q1,q2);
    h(q1);
    cnot(q1, q2);
    measure(q1,r[0]);
    measure(q2,r[1]);
}

module Bell2(q1, q2, r){
    reset(q1,q2);
    x(q1);
    h(q1);
    cnot(q1, q2);
    measure(q1,r[0]);
    measure(q2,r[1]);
}
```

```
\\Register and variable declarations
qreg q[10];
creg r1[2], r2[2];
fcho c1 = {0, 1}, c2 = [0, 1];
module Bell1(q1,q2){ ... } \\See Left
module Bell2(q1,q2){ ... } \\See Left
\\Main part of the meta-program
choice (c1){
    0:  Bell1(q[1], q[2],r1);
    1:  Bell1(q[7], q[8],r1);
};
choice (c2){
    0:  Bell2(q[1], q[2],r2);
    1:  Bell2(q[7], q[8],r2);
};
```

Figure 3.13: MQCC meta-program for multi-programming two Bell state quantum applications.

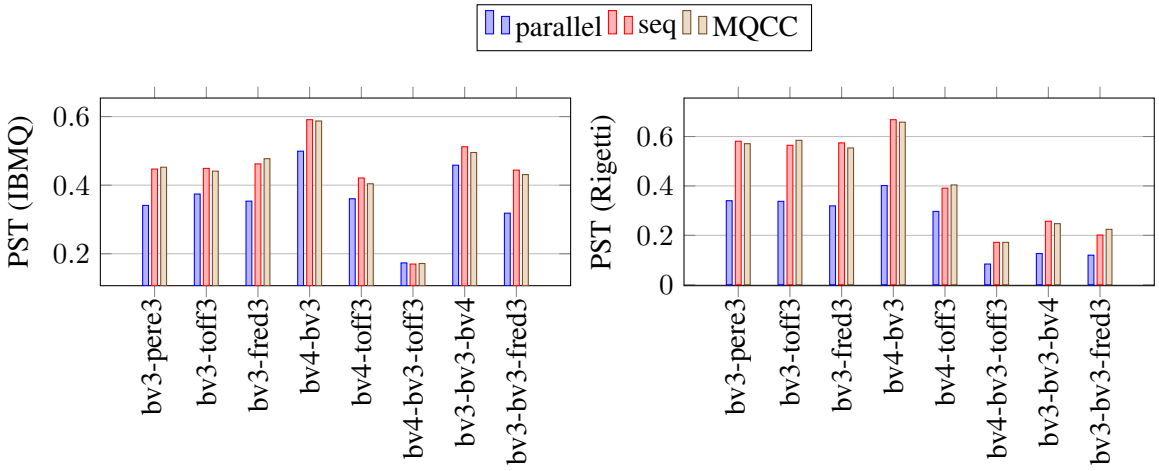


Figure 3.14: PST under isolated or multi-programmed execution for each group on IBMQ (left) and Rigetti (right) quantum machines. Group name *A-B* means the group contains two applications *A* and *B*. Similarly for the name *A-B-C*.

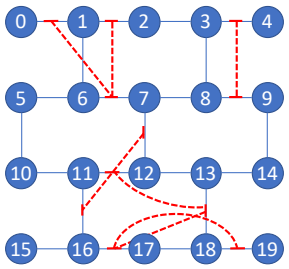


Figure 3.15: Layout of IBMQ Boeblingen [7]. Red dashed edges indicate *high crosstalk* gate pairs (e.g., the pair of `CNOT 0 1` and `CNOT 6 7`), where the error caused by their simultaneous execution is much higher than their independent gate error.

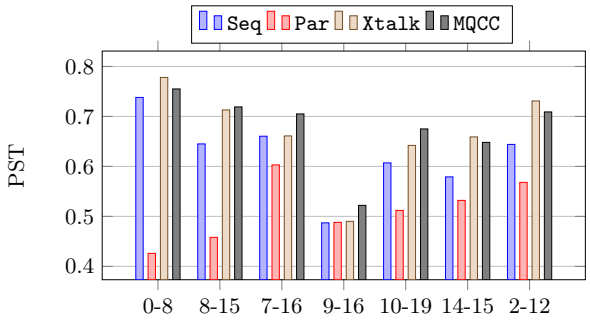
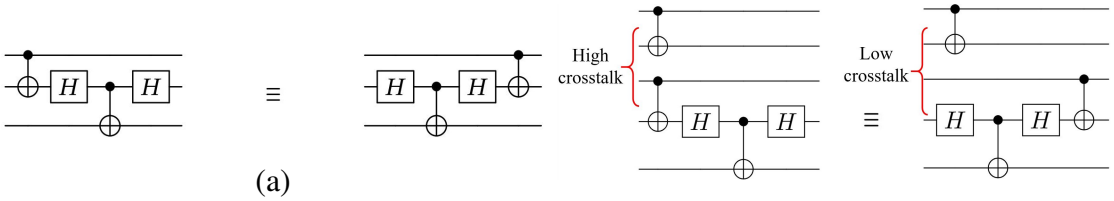
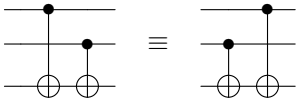


Figure 3.16: The measured PST for SWAP circuits on IBMQ Boeblingen using the four schedulers. Higher PST is better. a-b refers a SWAP circuit connecting qubit a and b.



(a) (b)

Figure 3.17: (a) Two equivalent circuits. (b) The choice of equivalent circuit affects crosstalk with nearby gates.

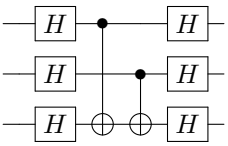


(a)

```
module twoCnot(c,q1,q2,q3){
    choice (c){
        0: cnot(q1,q3); cnot(q2,q3);
        1: cnot(q2,q3); cnot(q1,q3);
    }}
```

(b)

Figure 3.18: (a) Equivalent circuit. (b) Encoding as an MQCC program.

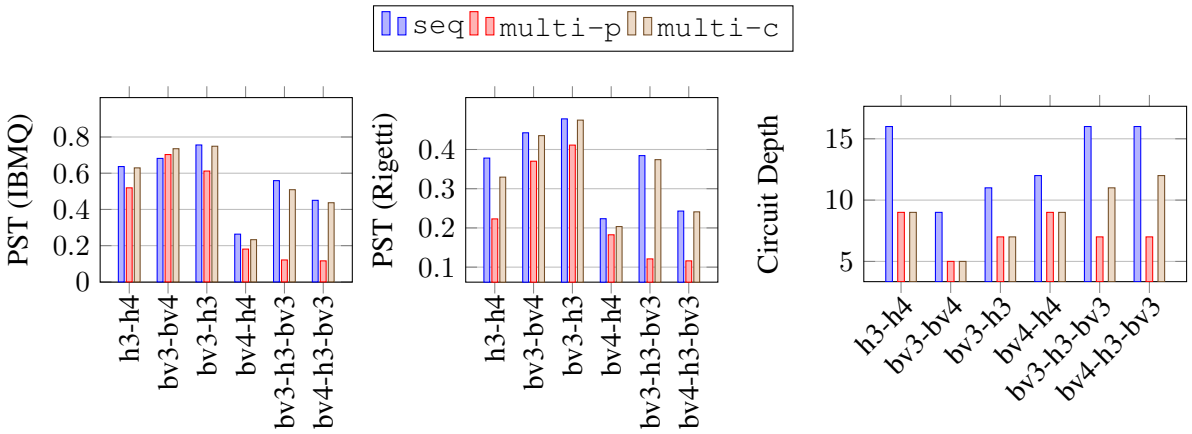


(a)

```
module BV3(c,q1,q2,q3){
    h(q1,q2,q3);
    twoCnot(c,q1,q2,q3);
    h(q1,q2,q3);
}
```

(b)

Figure 3.19: (a) 3-qubit BV circuit. (b) MQCC meta-program for the 3-qubit BV circuit.



(a) Probability of Successful Trial (PST) IBMQ Boebligen and Rigetti Aspen-9 . Here higher PST is better.

(b) Circuit Depth. Here lower circuit depth is better.

Figure 3.20: Multi-Programming with Crosstalk Mitigation.

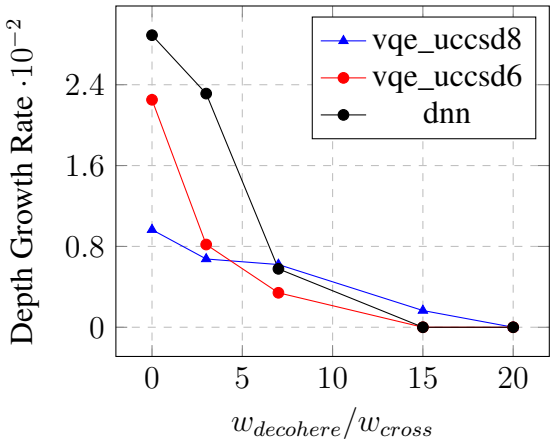


Figure 3.21: Depth growth rate of the circuit generated by MQCC for CM task under different error proportion.

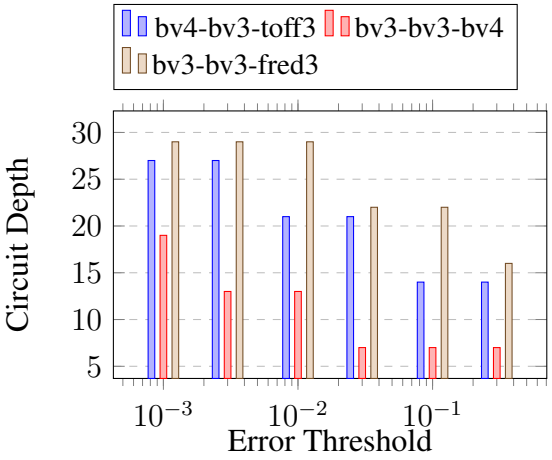


Figure 3.22: Circuit Depth under different error threshold for MP task.

Chapter 4: Neurosymbolic Search for Quantum Error Correcting Codes enhanced by Large Language Models

Quantum computing has gained significant attention due to its potential to provide asymptotically faster solutions to certain computational problems compared to the best-known classical algorithms [8]. A scalable, functioning quantum computer is believed to hold promise for solving complex computational challenges in fields such as scientific discovery, materials research, chemistry, and drug design, among others [9, 10, 11, 12].

However, one of the main challenges in building a quantum computer is the fragility of quantum information, which is susceptible to various sources of noise. This fragility arises because isolating a quantum computer from external influences while controlling it to perform desired computations are inherently conflicting tasks, making noise inevitable. The sources of noise include imperfections in qubits, materials used, control apparatus, State Preparation and Measurement (SPAM) errors, and a range of external factors, from man-made sources like stray electromagnetic fields to universal ones such as cosmic rays. See Ref. [130] for a summary. While some noise sources can be mitigated through improved control [131], better materials [132], and shielding [133, 134], others are much harder to eliminate. These more challenging noise sources include spontaneous and stimulated emission in trapped ions [135, 136] and interaction with the bath (Purcell Effect) [137] in superconducting circuits, which are critical to leading quantum

technologies. Therefore, error correction is essential for developing a scalable and functional quantum computer.

The possibility of quantum fault tolerance was established earlier [28]. By redundantly encoding a logical qubit into multiple physical qubits, it is possible to diagnose and correct errors through repeated measurements of parity check operator syndromes. However, error correction is effective only if the hardware error rate is below a certain threshold, which depends on the specific error correction protocol used. Early quantum error correction proposals, such as concatenated codes [21, 138, 139], focused on demonstrating the theoretical feasibility of error suppression. As quantum error correction theory and quantum technology capabilities advanced, attention shifted to developing practical error correction protocols. This led to the development of the surface code [140, 141, 142, 143], which offers a high error threshold close to 1

Unfortunately, finding good and large-size LDPC codes is non-trivial since an LDPC code class with a symbolic definition usually implies an exponentially large search space. Recently, Bravyi et al. [3] and Lin et al. [144] proposed a class of high encoding rate and error threshold LDPC codes based on group algebra. Their LDPC codes show excellent performance in the near-threshold regime and offer more than a 10× reduction in encoding overhead compared to the surface code [103], which has remained an uncontested leader in terms of its high error threshold for nearly 20 years. Due to the exponentially large cost of computing the distance of a large-size QECC and the large potential search space, Bravyi et al. [3] and Lin et al. [144] only applied a numerical search over a small search space. Su et al. [145] searched for QLego codes with an optimal logical error rate via reinforcement learning but only focused on CSS codes, a special type of stabilizer code constructed from classical codes with specific properties. Fortunately, most practical QEC codes, including the examples above, are equipped with a kind of symbolic

representation, which allows them to be studied apart from their native check matrices. The group algebra codes [3, 144] are defined by two symbolic polynomials over an abelian or non-abelian group, while a Quantum Lego code [36] is defined by a symbolic tensor network. Such features allow us to utilize *Neuro-symbolic AI* search techniques to help find good QEC codes.

Neuro-symbolic AI is a technique that merges knowledge-based symbolic approaches with neural network methods to address each of their weaknesses, enabling them to reason about abstract concepts, extrapolate from limited data, and generate explainable results [146].

Neuro-symbolic AI has demonstrated its power in many practical problems, including program synthesis [147, 148, 149, 150], semantic detection [151, 152, 153, 154], and rule induction [155, 156]. It is natural to apply Neuro-symbolic AI to QEC code search for two reasons: (1) the data collected from enumerative searches over limited-size QEC codes can be used as training data for the neuro-symbolic AI to extrapolate abstract concepts; (2) Neuro-symbolic AI can generate explainable results, which is highly desirable for QEC code search. However, all the previous neuro-symbolic AI works mentioned above deal with human-readable contexts (e.g., programs, videos, human language text), and Neuro-symbolic AI tools like LLMs can directly help with the search. However, many experiments show that LLMs perform poorly when directly dealing with complex mathematical symbolic representations, which are common in quantum cases. The semantics of symbolic QEC codes are usually large-size matrices, making it harder to directly incorporate LLMs into symbolic QEC code search.

Recently, Romera-Paredes et al. proposed FunSearch [35], a neuro-symbolic function search framework based on LLMs for mathematical discoveries. To tackle LLMs' weaknesses in mathematical problems, FunSearch builds the solution-finding process within a greedy or heuristic search structure, enabling LLMs to learn a good heuristic function. FunSearch has demonstrated

its ability to make mathematical discoveries for the capset problem [157] and the bin packing problem [158].

Inspired by FunSearch, we incorporate LLMs into symbolic QEC code search by utilizing them to generate good heuristic functions that guide the search for large-size QEC codes. In this chaptor, we present a Neuro-symbolic Quantum Error Correction Code Search framework (NuQES). The core idea behind NuQES is as follows:

1. Generate a set of codes below a certain size by brute-force search and characterize their properties, such as distance, error threshold, and logical error rate.

2. Ask LLMs to create a sift function to pick out codes with good properties from this set.

3. Apply the functions to the set of codes, evaluate their performance, and retain the best ones.

4. Feed the best-performing functions back into the LLM, asking it to generate improved sift functions based on existing results.

5. Repeat steps 2–4 until NuQES reaches the maximum iteration limit.

6. Use the best sift function generated in steps 2–5 to guide the code search for larger sizes.

We utilize NuQES to search for (1) Bivariate Bicycle (BB) LDPC Codes [3] and (2) QLego codes [36]. NuQES successfully found two $[[170, 16, 10]]$ and $[[288, 12, 22]]$ BB codes, which have higher encoding rates and error thresholds than the best codes found by Bravyi et al. [3]. In the QLego formalism, we also produce stabilizer codes that outperform their counterparts of similar length, rate, and distance under biased noise, some by as much as X orders of magnitude. In particular, the QLego codes are no longer restricted to $k = 1$ CSS codes as in Su et al. [145].

4.1 FunSearch Review

Many mathematical problems in computer science are 'easy to evaluate', despite being typically 'hard to solve'. For example, the famous NP-complete *Cap Set* problem refers to the task of finding the largest possible subset of vectors in $\mathbb{Z}_3^n$ such that no three vectors sum to zero. Geometrically, finding the largest possible subset such that no three points of a cap set are in a line. The cap set problem admits a polynomial-time evaluation procedure (counting the size of the solution subset), despite the widespread belief that no polynomial-time algorithms to solve this problem exist when n is large. This problem has drawn much interest since it serves as a model for the many real-life problems involving 'three-way interactions'

Based on this motivation, Romera-Paredes et al. [35] proposed FunSearch, an evolutionary procedure based on the Large Language Model (LLM). Here LLM is a machine learning model that can comprehend a human language prompt as the input and generate human language text. Example LLMs include LLaMA, BERT, and chatGPT. Given an efficient 'evaluate' function and the skeleton of the 'solve' program, FunSearch will try to generate the missing part of the 'solve' program such that its outputs receive high scores from the 'evaluate' function. FunSearch has demonstrated its ability to discover solutions better than the best existing one for several practical problems, not only the cap set problem [157], but also bin packing problem [158], corners problem [159], Shannon capacity of cycle graphs [160] and so on.

FunSearch combines a pretrained (frozen) LLM, whose goal is to provide creative solutions, with an evaluator, which guards against confabulations and incorrect ideas. FunSearch iterates over these two components, evolving initial low-scoring programs into high-scoring ones to discover new knowledge. Key to the success of this simple procedure is a combination of several essential

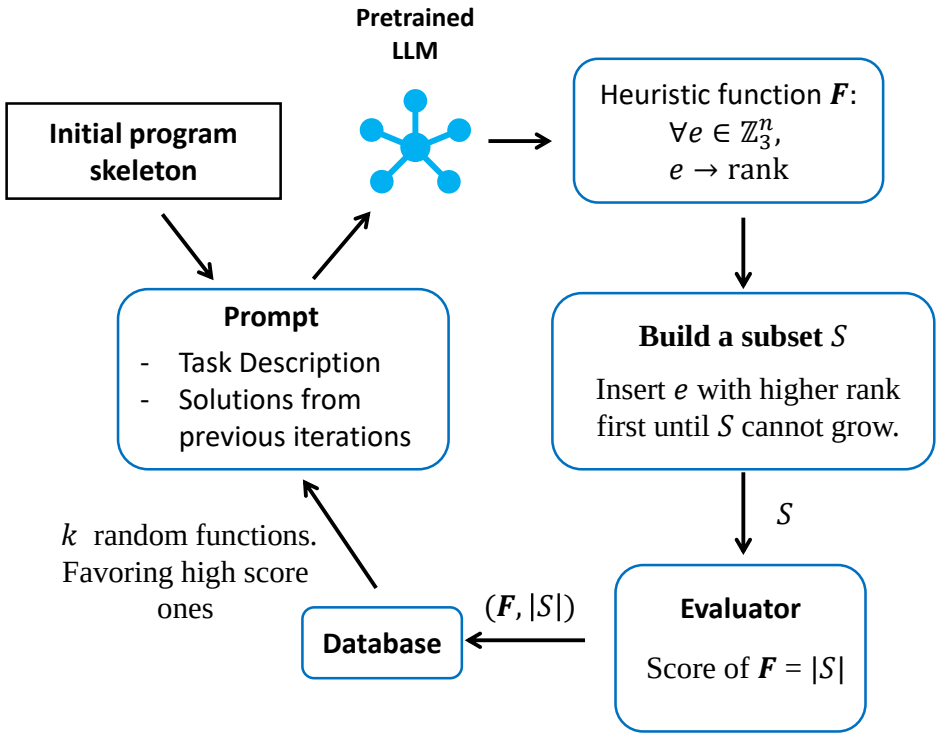


Figure 4.1: Overview of how Funsearch solves the capset problem.

ingredients. First, Funsearch samples best-performing programs and feeds them back into prompts for the LLM to improve on. Second, Funsearch starts with a program in the form of a skeleton (containing boilerplate code and potentially known structure about the problem), and only evolve the part governing the critical program logic. Third, Funsearch maintains a large pool of diverse programs by using an island-based evolutionary method that encourages exploration and avoids local optima.

We use the cap set problem as an example to explain how FunSearch works and Fig 4.1 shows the overview. FunSearch works similarly in other cases discussed in the main part and see Ref[35] for more implementation details. In the cap set problem case, the ‘evaluate‘ function sent to FunSearch is trivial (counting the size of the solution subset). The skeleton of the ‘solve‘ program given to FunSearch is a greedy search algorithm and the missing part is its heuristic

program which need to give a rank to all elements in $\mathbb{Z}_3^n$. This greedy algorithm starts with an empty set S and iteratively adds outside elements with the highest rank based on the heuristic program F into S, until no elements can be added (all outside elements are in a line with at least two elements in S). Then it returns this subset as the result.

FunSearch iteratively asks the equipped pre-trained LLM to generate a good heuristic function $\mathbf{F}$ for this greedy algorithm. Each heuristic function $\mathbf{F}$ is scored by the size of the subset S generated by the greedy algorithm. The subsets constructed based on the heuristic programs generated by LLM at the starting point usually get low scores from the 'evaluate' program. FunSearch stores these heuristic functions with their scores in a program database. In each iteration Funsearch asks LLM to generate a new heuristic function, with a prompt built by combining k heuristic functions sampled from the program database, favouring high-scoring ones, and let LLM try to generate an improved one based on these k sample ones. Newly created functions are then scored and stored in the database (if correct), thus closing the loop. Funsearch terminates after a specified number of iterations and retrieves the highest-scoring heuristic functions in the database discovered so far. This heuristic function will be filled into the greedy search algorithm and generate a subset as the final result.

4.2 Neurosymbolic QECC Search

4.2.1 NuQES Overview

QEC Code Synthesis with FunSearch Heuristic search is a method developed to accelerate the search process when traditional techniques are too slow to find an exact or approximate solution in a vast search space. It works by sacrificing optimality, completeness, accuracy, or precision

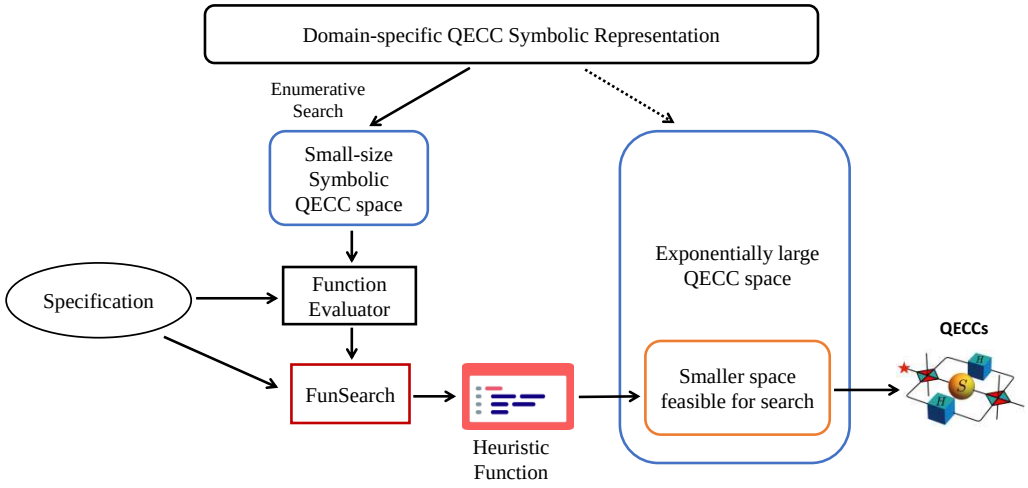


Figure 4.2: Overview of NuQES.

in exchange for faster results, essentially acting as a shortcut. Heuristic search has long been an effective approach to solving complex problems. However, creating an effective heuristic function for quantum code search is often challenging due to the complexity of calculating code distances. To deal with this challenge, NuQES utilizes FunSearch to generate good heuristic functions based on the small-size QEC codes database generated by the enumerative search, and then applies the best heuristic fucntions to the search of large-scale QEC codes. Fig 4.2 shows the overview of QEC codes synthesis through NuQES, which is split into two steps: a heuristic function synthesis through FunSearch and then a QEC code synthesis guided by the generated heuristic function. In the first step, NuQES applies a simple enumerative search over small-size LDPC codes. All codes found in this step will be stored and used as a part of the ‘evaluate’ function when calling the FunSearch subprogress. Then we ask FunSearch to generate a heuristic function which can sift out good codes from the large potential search space. The ’evaluate’ function is given as: for each ’solve’ program, the ’evaluate’ function uses it to sift out codes from the small-size LDPC codes found in the first step and compute some necessary properties of those codes (e.g. logical error rate) as the score. The ’solve’ program with the highest score will be used as the heuristic

function to guide the search over large-scale codes which truncate the exponentially large search space in a smaller space feasible for synthesis.

We use NuQES to synthesize two kinds of quantum error correction codes, Bivariate Bicyle quantum LDPC code and Quantum Lego code. NuQES generates codes better than the best existing one in both cases. In the following section, we explain how to build the evaluator sent to FunSearch and how to use the heuristic function generated from FunSearch to guide the QEC code search for these two applications.

4.3 Bivariate Bicycle Quantum LDPC Codes Synthesis

4.3.1 Code Definition

Bivariate Bicycle (BB) quantum LDPC codes[3] are high-rate LDPC codes with a few hundred physical qubits equipped with a low-depth syndrome measurement circuit, an efficient decoding algorithm and a fault-tolerant protocol for addressing individual logical qubits. These codes show an error threshold p_0 close to 0.7%, show excellent performance in the near-threshold regime, and offer a 10 times reduction of the encoding overhead compared with the surface code.

Let I_ℓ and S_ℓ be the identity matrix and the cyclic shift matrix of size $\ell \times \ell$ respectively. The i-th row of S_ℓ has a single nonzero entry equal to one at the column $i+1 \pmod{\ell}$. For example,

$$S_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \text{and} \quad S_3 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

Consider matrices

$$x = S_\ell \otimes I_m \quad \text{and} \quad y = I_\ell \otimes S_m.$$

Note that $xy = yx$ and $x^\ell = y^m = I_{\ell m}$. A BB code is defined by a pair of matrices

$$A = A_1 + A_2 + A_3 \quad \text{and} \quad B = B_1 + B_2 + B_3 \tag{4.1}$$

where each matrix A_i and B_j is a power of x or y. Here and below the addition and multiplication of binary matrices is performed modulo two, unless stated otherwise. Thus, we also assume the A_i are distinct and the B_j are distinct to avoid cancellation of terms. For example, one could choose $A = x^3 + y + y^2$ and $B = y^3 + x + x^2$. Note that A and B have exactly three non-zero entries in each row and each column. Furthermore, $AB = BA$ since $xy = yx$. The above data defines a BB LDPC code denoted $\mathsf{QC}(A, B)$ with length $n = 2\ell m$ and check matrices

$$H^X = [A|B] \quad \text{and} \quad H^Z = \left[B^T|A^T\right]. \tag{4.2}$$

Here the vertical bar indicates stacking matrices horizontally and T stands for the matrix transposition. Both matrices H^X and H^Z have size $(n/2)\times n$. Each row $v \in \mathbb{F}_2^n$ of H^X defines an X-type check operator $X(v) = \prod_{j=1}^n X_j^{v_j}$. Each row $v \in \mathbb{F}_2^n$ of H^Z defines a Z-type check operator $Z(v) = \prod_{j=1}^n Z_j^{v_j}$. Any X-check and Z-check commute since they overlap on even number of qubits (note that $H^X(H^Z)^T = AB + BA = 0 \pmod 2$). To describe the code parameters we use certain linear subspaces associated with the check matrices, see Table 1 for our notations. Then

Notation	Definition
$\mathsf{rs}(H)$	Linear span of rows of H
$\mathsf{cs}(H)$	Linear span of columns of H
$\mathsf{ker}(H)$	Vectors orthogonal to each row of H
$\mathsf{rk}(H)$	$\mathsf{rk}(H) = \dim(\mathsf{rs}(H)) = \dim(\mathsf{cs}(H))$

Table 4.1: Notations for linear spaces associated with a binary matrix H. Here the linear span, orthogonality, and dimension are computed over the binary field $\mathbb{F}_2 = \{0, 1\}$. If H has size $s \times n$ then $\mathsf{rs}(H) \subseteq \mathbb{F}_2^n$, $\mathsf{cs}(H) \subseteq \mathbb{F}_2^s$, and $\mathsf{ker}(H) \subseteq \mathbb{F}_2^n$.

$[[n, k, d]]$	Net Encoding Rate r	ℓ, m	A	B
$[[72, 12, 6]]$	1/12	$6, 6$	$x^3 + y + y^2$	$y^3 + x + x^2$
$[[90, 8, 10]]$	$1/23$	$15, 3$	$x^9 + y + y^2$	$1 + x^2 + x^7$
$[[108, 8, 10]]$	1/27	$9, 6$	$x^3 + y + y^2$	$y^3 + x + x^2$
$[[144, 12, 12]]$	$1/24$	$12, 6$	$x^3 + y + y^2$	$y^3 + x + x^2$
$[[288, 12, 18]]$	$1/48$	$12, 12$	$x^3 + y^2 + y^7$	$y^3 + x + x^2$
$[[170, 16, 10]]$	$1/21$	$17, 5$	$x + x^{16} + y$	$x^4 + x^{13} + y$
$[[288, 12, 22]]$	$1/48$	$12, 12$	$x^1 + x^2 + y^9$	$x^3 + y^5 + y^{10}$

Table 4.2: Small examples of Bivariate Bicycle LDPC codes and their parameters. All codes have weight-6 checks, thickness-2 Tanner graph, and a depth-7 syndrome measurement circuit. We round r down to the nearest inverse integer. The codes have check matrices $H^X = [A|B]$ and $H^Z = [B^T|A^T]$ with A and B defined in the last two columns. The matrices x, y obey $x^\ell = y^m = 1$ and $xy = yx$.

the code $\mathsf{QC}(A, B)$ has parameters $[[n, k, d]]$ with

$$n = 2\ell m, \quad k = 2 \cdot \dim\left(\mathsf{ker}(A) \cap \mathsf{ker}(B)\right) \tag{4.3}$$

$$d = \min\left\{|v|\colon v \in \mathsf{ker}(H^X) \backslash \mathsf{rs}(H^Z)\right\} \tag{4.4}$$

Here $|v| = \sum_{i=1}^n v_i$ is the Hamming weight of a vector $v \in \mathbb{F}_2^n$. We note that the code $\mathsf{QC}(A, B)$ can be viewed as a special case of the Lifted Product construction [161] based on the abelian group $\mathbb{Z}_\ell \times \mathbb{Z}_m$. Here $\mathbb{Z}_j$ denotes the cyclic group of order j.

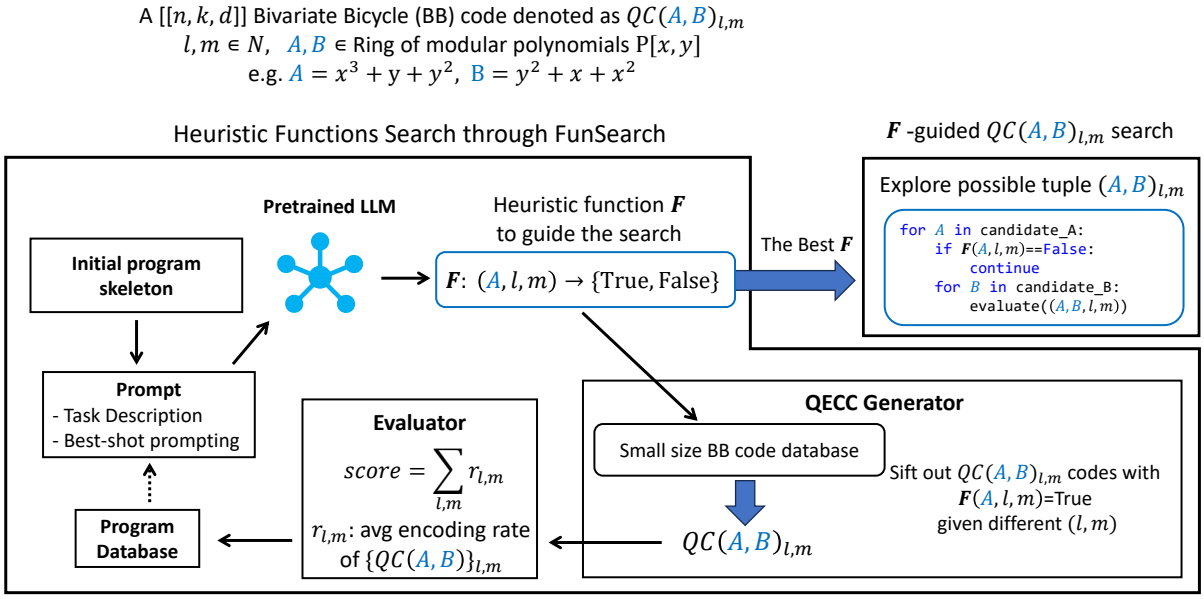


Figure 4.3: Bivariate Bicycle LDPC codes synthesis through NuQES.

4.3.2 Bivariate Bicycle Codes Synthesis

Code Criteria BB codes are designed for realizing a fault-tolerant quantum memory with a small qubit overhead. Consider encoding $k \gg 1$ logical qubits into n data qubits and use c ancillary check qubits to measure the error syndrome. In total, the code relies on $n + c$ physical qubits. The net encoding rate is therefore $r = \frac{k}{n+c}$. We followed the same BB code selection criteria as Bravyi et al.[3].

1. We desire a code with a large distance d and a high encoding rate $r \gg 1/d^2$,

2. offers a pseudo-threshold close to 1% (or higher) for the circuit-based noise model,

A pseudo-threshold p_0 of an error correction protocol is defined as a solution of the break-even equation $p_L(p) = kp$. Here kp is an estimate of the probability that at least one of k unencoded qubits suffers from an error.

$[[n,k,d]]$	Encoding Rate r	d_{circ}	$p_0 \cdot 10^{-3}$	$p_L(0.001)$	$p_L(0.0001)$
$[[72,12,6]]$	1/12	≤ 6	4.76	7×10^{-5}	7×10^{-8}
$[[90,8,10]]$	1/23	≤ 8	5.33	5×10^{-6}	4×10^{-10}
$[[108,8,10]]$	1/27	≤ 8	5.79	3×10^{-6}	1×10^{-10}
$[[144,12,12]]$	1/24	≤ 10	6.52	2×10^{-7}	8×10^{-13}
$[[288,12,18]]$	1/48	≤ 18	6.91	2×10^{-12}	1×10^{-22}
$[[170,16,10]]$	**1/21**	≤ 10	**7.11**	1.4×10^{-7}	4.1×10^{-7}
$[[288,12,22]]$	1/48	≤ 18	**7.59**	$\mathbf{1.2 \times 10^{-12}}$	$\mathbf{4.3 \times 10^{-23}}$

Table 4.3: Performance comparison between Bivariate Bicycle LDPC codes found by NuQES (codes in blue) and found by Bravyi et al. [3] (codes in black). All codes have weight-6 checks, thickness-2 Tanner graph, and a depth-7 syndrome measurement circuit. A code with parameters $[[n,k,d]]$ requires $2n$ physical qubits in total and achieves the net encoding rate $r = k/2n$ (we round r down to the nearest inverse integer). Circuit-level distance d_{circ} The pseudo-threshold p_0 is a solution of the break-even equation $p_L(p) = kp$, where p and p_L are the physical and logical error rates respectively. The logical error rate p_L was computed numerically for $p \geq 10^{-3}$ and extrapolated to lower error rates. See Section 4.3.3 for details of computing p_L.

Synthesis Result Table 4.3 shows the best existing BB codes (black) and the better BB code (blue) synthesized by NuQES. The circuit distance d_{circ}, error threshold p_0, and the logical error rate $p_L(p)$ are computed with the same simulation program from Bravyi et al.[3] which is based on Belief Propagation with an Ordered Statistics postprocessing step Decoder (BP-OSD) proposed by Panteleev and Kalachev [162]. We use GAP[163] package QDistRnd[164] to compute the code distances. See section 4.3.3 for the numerical simulation details. Bravyi et al.[3] claim that the $[[144,12,12]]$ code is the most promising one among the codes they found for near-term demonstrations, as it combines large distance and high net encoding rate $r = 1/24$. The $[[170,16,10]]$ code found by NuQES outperforms it with a higher error threshold p_0 and larger encoding rate $r = 1/21$. Additionally, NuQES found a new $[[288,12,22]]$ code, which has the same encoding rate as Bravyi et al.[3]'s $[[288,12,18]]$ code but a larger distance and higher error threshold. We want to point out that NuQES also found the $[[144,12,12]]$ and $[[288,12,22]]$ code proposed by Bravyi et al.[3] during the synthesis.

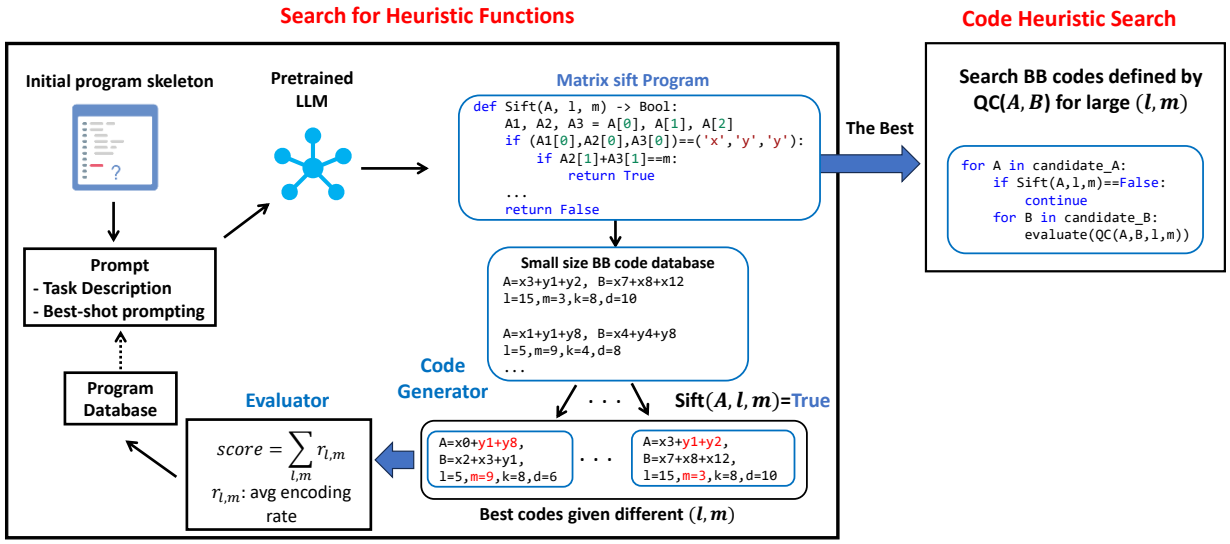


Figure 4.4: Bivariate Bicycle LDPC codes synthesis with NuQES

Two step synthesis with NuQES Fig 4.3 shows the overview of synthesizing BB LDPC codes with NuQES. Each symbolic BB code denoted as $QC(A,B)_{l,m}$ is defined by two integers l, m and two polynomials A, B over the ring of modular polynomials $P[x, y]$. NuQES first applies an enumerative search for small-size BB codes to build a code database. Then NuQES utilizes FunSearch to find a heuristic function $\mathbf{F}$. At each iteration of the heuristic function synthesis step, NuQES builds a prompt by combining k programs sampled from the program database (favoring high-scoring ones). We choose $k = 2$ in this case. The prompt is then fed to the pre-trained LLM, and new heuristic functions are created and fed to the code generator. Each function $\mathbf{F}$ maps the integers l, m and a polynomial A to a boolean variable. The 'evaluate' function sent to FunSearch for giving a score to each candidate $\mathbf{F}$ works as follows: (1) For different (l, m), pick code set $\{QC(A,B)\}_{l,m}$ with `sift(A,l,m)=True` in the BB code database and select the code with the highest $r_{best} = \frac{kd}{n}$ and $d \geq 3$ from $\{QC(A,B)\}_{l,m}$. (2) compute $r_{l,m} = r_{best}/ln(|\{QC(A,B)\}_{l,m}|)$ and score $\mathbf{F}$ by the sum of these codes' $\frac{kd}{n}$. After the heuristic function synthesis step, NuQES uses the best F to guide the search of possible (A, B) pair of the

```
def Sift(A, l, m) -> Bool:
    A1, A2, A3 = A[0], A[1], A[2]
    if (A1[0],A2[0],A3[0])==('x','y','y'):
        if A2[1]+A3[1]==m:
            return True
    if (A1[0],A2[0],A3[0])==('x','x','y'):
        if A1[1]+A2[1]==l:
            return True
    if A1[1]+A2[1]+A3[1]==l:
        return True
    if A1[1]+A2[1]+A3[1]==m:
        return True
    return False
```

Figure 4.5: Heuristic Function $\mathbf{F}$ generated by FunSearch to synthesize large size BB codes.

large-size BB codes. As shown in the pseudo-code at the right-hand side, the code searcher only explores codes $QC(A, B)$ with `F(A,l,m)=True` under different (l, m) and skip the iteration otherwise.

Fig 4.4 shows a detailed example of synthesizing BB codes with NuQES. Consider the `Sift` Python function in Fig 4.4 as a candidate heuristic function generated by LLM during the heuristic function synthesis step. The input variable `A`, which means the matrix A of a BB code, contains three tuples, each representing the power of x or y. For example, the matrix $A = x^3 + y + y^2$ is represented by a Python variable `[('x',3), ('y',1), ('y',2)]`. The `Sift` function in Fig 4.4 returns `True` if the matrix A is in the format $x^a + y^b + y^c$ where $b + c = m$. Given $l = 5, m = 9$, the best code sifted out by the code generator is constructed by $A = x^0 + y^1 + y^8$ where $1 + 8 = 9$. After sifting out the best codes for different (l, m), the candidate heuristic function is scored and stored in the program database and may be used to construct the prompt for future iterations. Fig 4.5 shows the result heuristic function we utilized FunSearch to generate and it is used to guide the synthesis for the QEC code synthesis step.

4.3.3 Numerical Simulation Details

We follow the same syndrome measurement circuit and decoder used by Bravyi et al.[3] to compute the pseudo-threshold p_0 of the codes we found. The full cycle of syndrome measurement for a length-n BB code requires n ancillary check qubits to store the measured syndromes. Accordingly, the net encoding rate is $r = k/(2n)$. Check qubits are coupled with the data qubits by applying a sequence of `CNOT` gates. The full cycle of syndrome measurement requires only 7 layers of `CNOT`s regardless of the code length. The check qubits are initialized and measured at the beginning and at the end of the syndrome cycle respectively.

The full error correction protocol performs $N_c \gg 1$ syndrome measurement cycles and calls a decoder — a classical algorithm that takes as input the measured syndromes and outputs a guess of the final error on the data qubits. Error correction succeeds if the guessed and the actual error coincide modulo a product of check operators. In this case the two errors have the same action on any encoded (logical) state. Thus applying the inverse of the guessed error would return data qubits to the initial logical sate. Otherwise, if the guessed and the actual error differ by a non-trivial logical operator, error correction fails resulting in a logical error. Our numerical experiments are based on the Belief Propagation with an Ordered Statistics Decoder (BP-OSD) proposed by Panteleev and Kalachev [162]. The original work [162] described BP-OSD in the context of a toy noise model with memory errors only. Bravyi[3] et al. showed how to extend BP-OSD to the circuit-based noise model which closely followed Refs. [165, 166, 167, 168]. Bravyi[3] et al. also showed that BP-OSD can be applied to other problems in quantum fault-tolerance such as estimating the distance of a quantum LDPC code. These tasks can be accomplished with a relatively minor extension of the publicly available BP-OSD software developed by Roffe et

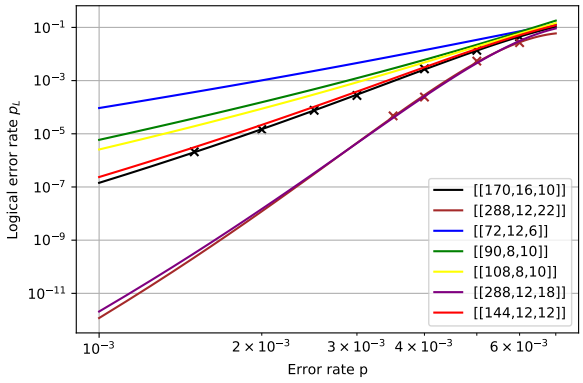


Figure 4.6: Logical vs physical error rate for Bivariate Bicycle LDPC codes. A numerical estimate of p_L (diamonds) was obtained by simulating d syndrome cycles for a distance-d code. Most of the data points have error bars $\approx p_L/10$ due to sampling errors.

$[[n,k,d]]$	c_0	c_1	c_2
$[[170,16,10]]$	1.738	1511	−11005
$[[288,12,22]]$	3.098	4085	−36207

Table 4.4: Parameters c_0, c_1, c_2 in the fitting formula $p_L(p) = p^{d_{circ}/2} e^{c_0 + c_1 p + c_2 p^2}$ for BB LDPC codes found by NuQES shown in Table 4.3.

al. [169]

Let $P_L(N_c)$ be the logical error probability after performing N_c syndrome cycles. Define the logical error rate as $p_L = 1 - (1 - P_L(N_c))^{1/N_c} \approx P_L(N_c)/N_c$. Informally, p_L can be viewed as the logical error probability per syndrome cycle. Following common practice, we choose $N_c = d$ for a distance-d code. Fig 4.6 shows the logical error rate achieved by codes from Table 4.3. The logical error rate was computed numerically for $p \geq 10^{-3}$ and extrapolated to lower error rates using an ansatz $p_L = p^{d'_{circ}/2} e^{c_0 + c_1 p + c_2 p^2}$, where c_0, c_1, c_2 are fitting parameters and shown in Table 4.4, and d'_{circ} is an upper bound on d_{circ} from Table 4.3.

4.4 Quantum Lego Code Synthesis

In this section, we first review more of the formalism behind Quantum Lego (QLego). For more details, please see [170]. Then we discuss searching for codes with low logical error rates with NuQES based on the Quantum Lego representation.

4.4.1 Channel-State Duality

The rules for combining these modular codes are inspired by tensor networks and will have a graphical construction. To make this connection, we make use of the *channel state duality*, wherein we can reinterpret the encoding map V, from the two logical qubits to the four physical qubits,

$$\mathcal{V} = \sum_{i_1,\ldots,i_6} V_{i_1 i_2 i_3 i_4 i_5 i_6} \left|i_3, i_4, i_5, i_6\right\rangle \left\langle i_1, i_2\right| \tag{4.5}$$

as a quantum state,

$$\left|\psi_V\right\rangle = \sum_{i_1,\ldots,i_6} V_{i_1 i_2 i_3 i_4 i_5 i_6} \left|i_1, i_2, i_3, i_4, i_5, i_6\right\rangle . \tag{4.6}$$

Matrix elements of V are simply the wave function coefficients for basis elements on both input and output legs, which are now on equal footing. See Fig. 4.7. Note that going from a state to a channel is not necessarily unique, as there is a freedom in choosing which (and how many) of the legs to choose as logical. For example, we could interpret the tensor arising from the $[[4,2,2]]$ code as a $[[5,1,2]]$ code,

$$\mathcal{V}' = \sum_{i_1,\ldots,i_6} V_{i_1 i_2 i_3 i_4 i_5 i_6} \left|i_2, i_3, i_4, i_5, i_6\right\rangle \left\langle i_1\right| . \tag{4.7}$$

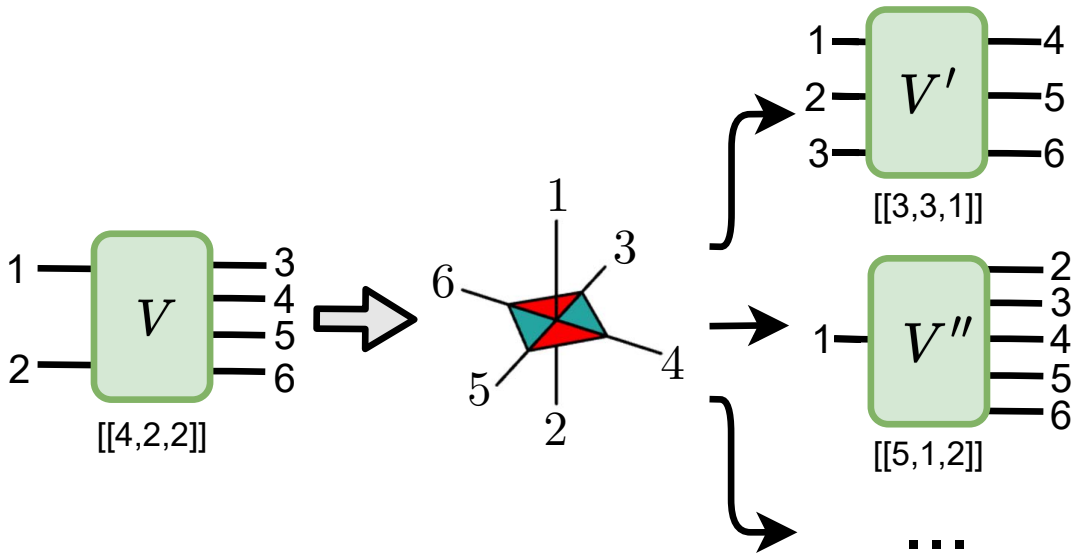


Figure 4.7: Channel state duality. The encoding map V of a QECC, taking logical qubits to their physical counterparts, can be interpreted as a state on all qubits. The tensor describing the state $|\psi_V\rangle$ is simply given by the matrix elements of V. When V is the encoding map for the $[[4,2,2]]$ code with stabilizer generators $\langle XXXX, ZZZZ\rangle$, we call this tensor the T6 lego. Note the freedom in going from a tensor to a QECC that comes from assigning each of the legs to be a logical or physical qubit. Because of this freedom, for example, we could interpret $|\psi_V\rangle$ as either a $[[4,2,2]]$ code or a $[[5,1,2]]$ code.

To standardize the notation, let us denote the input legs as 1 and 2, with the output legs corresponding to $3,4,5,6$.

For stabilizer codes, the resulting state $|\psi_V\rangle$ is a stabilizer state. The stabilizers of that state e.g. $S : S|\psi_V\rangle = +1|\psi_V\rangle$ are formed by taking $O^{(\mathrm{L})} \otimes O^{(\mathrm{P})}$ where $O^{(\mathrm{L})}$ denotes a logical Pauli operator and $O^{(\mathrm{P})}$ denotes the corresponding physical operator. To go from the state back to a code, one can isolate a set of physical legs P', and refactor $S = O^{(\mathrm{L})} \otimes O^{(\mathrm{P})} = O^{(\mathrm{L}')} \otimes O^{(\mathrm{P}')}$. If O is not a Pauli operator, one should take extra care in this conversion by taking the complex conjugate such that a stabilizer for the dual quantum state is given by $O^{(\mathrm{L})^*} \otimes O^{(\mathrm{P})}$, but for Pauli matrices $O = O^*$ up to global phase factors.

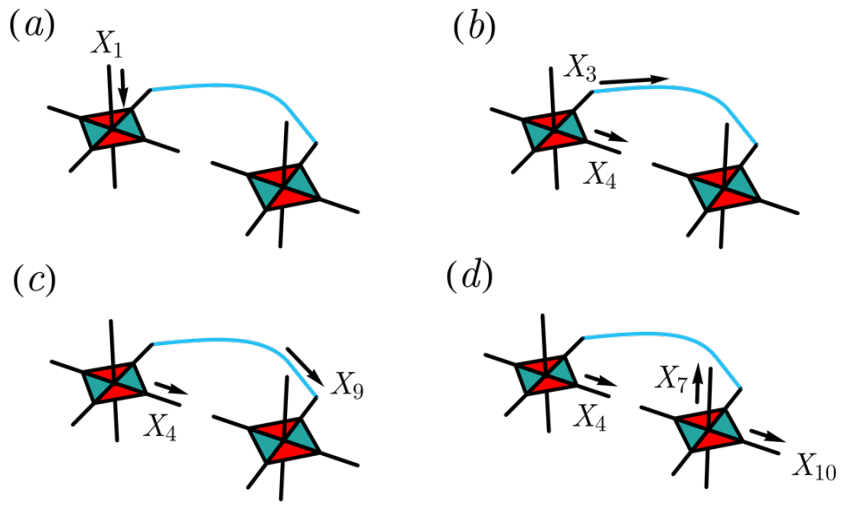


Figure 4.8: Single-trace operation via operator flow. Given two copies of T6, we glue the legs of two tensors by projecting onto a maximally entangled state (blue). The stabilizer of the resulting state can be found by performing operator pushing. a) We start with $X_1^{(\mathrm{L})} = X_1$ acting on the first qubit. b) Pushing through the first tensor yields $X_3 X_4$. c) Since qubits 3 and 9 are entangled, $X_3 \sim X_9$. d) Finally, pushing through the second tensor, we get $X_1^{(\mathrm{P})} = X_4 X_7 X_{10}$, so $X_1^{(\mathrm{L})} \otimes X_1^{(\mathrm{L})} = X_1 X_4 X_7 X_{10}$ is a stabilizer of the resulting tensor network after contraction.

4.4.2 Gluing Legos

Let us take two copies of this code, labeling the legs of A as 1-6 and B as 7-12 with the convention for numbering the same as before. Consider contracting legs 3 and 9, which amounts to a Bell fusion, or a projection on the maximally entangled state. We would like to understand the stabilizers for the remaining ten qubit state. Suppose we take stabilizers of the individual tensors $X_1^{(\mathrm{L})} \otimes X_1^{(\mathrm{P})} = X_1 I_2 X_3 X_4 I_5 I_6$ and $X_7^{(\mathrm{L})} \otimes X_7^{(\mathrm{P})} = X_7 I_8 X_9 X_{10} I_{11} I_{12}$, then $X_1 I_2 X_4 I_5 I_6 X_7 I_8 X_{10} I_{11} I_{12}$, would be a stabilizer of the remaining qubits. Here we make use of the fact that $X^2 = I$. An alternate way to understand this construction of the resulting stabilizer is to push X_3 "through" the other tensor. This is illustrated in Fig. 4.8.

In constructing the full set of stabilizers, we simply perform this operation for all operators acting on the glued legs, taking care to ensure operators acting on the traced legs are matched.

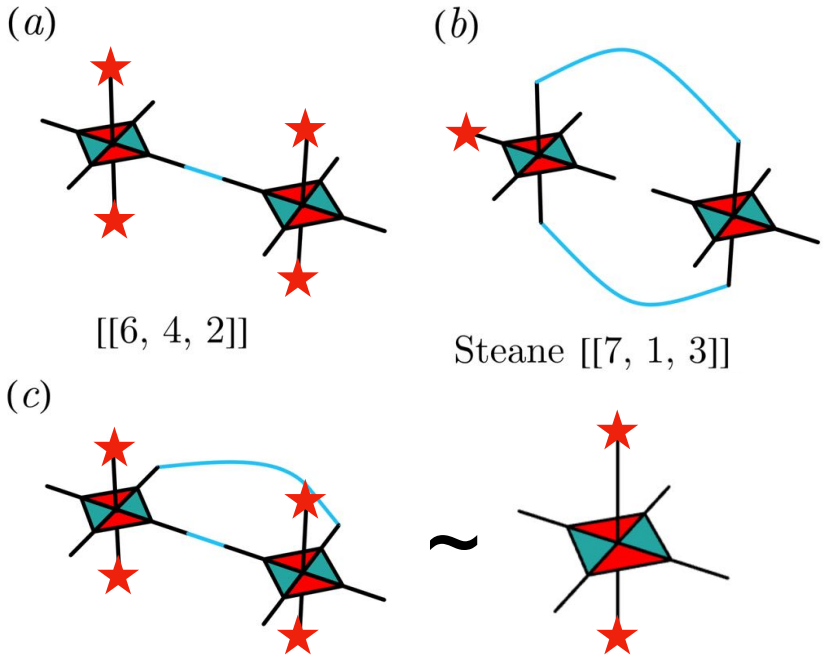


Figure 4.9: Examples of code construction using Quantum Lego. Given just two copies of the T6 lego, we already see a variety of behavior. (a) A single contraction yields a code that encodes more physical qubits with a better encoding rate, at the same distance as the original code. (b) Picking a single logical leg after tracing two legs reproduces the well-studied [[7, 1, 3]] Steane code. (c) Performing two traces on different choices of legs reduces to a single copy of the [[4,2,2]]. This demonstrates the rich and unexpected behavior that these simple contractions can produce.

Although naively the total number of stabilizer matchings is exponential in the system size, in the case of stabilizer codes, there is an efficient algorithm to keep track of the glued codes via a check matrix operation known as conjoining (Appendix D of [170]).

Note that any two legs may be glued together. When two legs of the same tensor (more generally, a connected component) are glued, we refer to it as a self-trace. Otherwise, we refer to the operation as a single-trace.

In general, [170] showed that tracing together stabilizer codes yield another stabilizer code. If the individual legos are (self-dual) CSS codes, then up to re-definitions via the Choi-Jamiolkowski isomorphism, so are the resulting codes obtained from gluing their physical legs. Note that although $[[4, 2, 2]]$ is self-dual, the $[[5, 1, 2]]$, $[[6, 0, 3]]$ codes from the same tensor are not, because of weight-3 stabilizers. Furthermore, tracing of two non-self-dual codes can be self-dual, e.g. the Steane code example in Fig. 4.9b. Therefore, the codes we obtain in this work are CSS codes, but the self-duality property need not be preserved during the game.

4.4.3 QLego Tensor Network Description

We describe the construction of a QLego tensor network T with two lists: (1) tnL indicating the list of tensors in the network and (2) opL indicating the list of operations to construct the network. T starts with the first tensor in tnL and we sequentially apply the operations in opL to T. There are three possible operations in opL.

- Trace operation in the format $(trace, t_1, l_1, t_2, l_2)$. This operation traces the leg indexed by l_1 of a tensor $tnL[t_1]$ in the network with the leg indexed by l_2 by a tensor $tnL[t_2]$'s that is not in the network. This operation will insert $tnL[t_2]$ into the network.

- Self-trace operation in the format $(self, t_1, l_1, t_2, l_2)$. It traces the tensor $tnl[t_1]$'s leg l_1 and the tensor $tnL[t_2]$'s leg l_2. Both tensors should be already in the network.

- Logical leg operation $(setLog, t, l)$. It sets the tensor $tnL[t]$'s leg l as the logical leg.

For example, the tensor network to construct the $[[7, 1, 3]]$ code in Fig 4.9(b) with two T6 tensors can be described by:

$$tnL = [T6, T6]$$

$$opL = [(trace, 0, 1, 1, 1), (self, 0, 4, 1, 6), (setLog, 0, 6)]$$

For efficient search, we restrict opL comes with some trace operations first, then some self-trace operations, and finally the logical leg operations. We use the following tensor set as the basic tensor in the synthesis.

- T6 $[[6, 0, 3]]$ tensor extended from the stabilizer generators of the $[[4, 2, 2]]$ codes which are also used by Su et al[145]. Legs indexed by 0 and 1 are the logical legs.

- Rank-6 perfect tensor derived from the perfect $[[5, 1, 3]]$ code.

- Rank-8 tensor derived from the $[[7, 1, 3]]$ Steane code and defined by the stabilizer generators.

- GHZ Tensor defined by thestabilizer generators $\langle IZZ, XXX, ZZI \rangle$.

- The H,S and $|0\rangle$ tensor.

4.4.4 Synthesize QLego Codes with NuQES

Code Criteria In practice, given that not all physical noise is created equally, designing codes that are tailored to particular shortcomings of a device may accelerate reaching fault tolerant computation. To highlight the flexibility of our methods for code design, we let NuQES minimize the probability of a logical error under a biased noise model. As a proof of principle, we consider biased noise models that each qubit goes through a noise channel with independent bit flip and phase error probabilities p_x and p_z respectively, similar to Su et al. [145].

We follow the same method used by the previous quantum code search framework[145] for estimating the logical error rate of the best-known codes and the codes NuQES found. Given a code, we estimate the probability of an undetectable logical error p_L, which is the probability that a non-trivial logical operation has been applied to the encoded state due to bit-flip and phase-flip errors on individual qubits. In other words, this is the lower bound for the logical error rate regardless of the choice of decoder. Because bit flip and phase errors occur independently, for each non-trivial logical Pauli operator $\bar{L}$ such that the X-weight $wt_X(\bar{L}) = w_x$ and Z-weight $wt_Z(\bar{L}) = w_z$, the total probability that it comes from a physical error is $p(w_x, w_z) = p_x^{w_x} p_z^{w_z} (1-p_x)^{n-w_x} (1-p_z)^{n-w_z}$. Therefore,

$$p_L = \sum_{w_x, w_z = 0}^{n} C_{w_x, w_z}\, p(w_x, w_z), \tag{4.8}$$

where C_{w_x,w_z} enumerates the number of non-identity logical operators with the same X and Z weights.

Then we consider a detection-based decoder which resets the system upon measuring a

non-trivial syndrome. Such a measure is highly relevant for fault-tolerant protocols like magic state distillation. This is the logical error probability *given* that one measures trivial error syndrome, meaning no decoding needs to be applied. The corresponding logical error probability is

$$p_L^{\text{norm}} = p_L / p_{s=0}, \tag{4.9}$$

where $p_{s=0}$ is the probability of measuring trivial syndromes (assuming no measurement errors). Both p_L and $p_{s=0}$ are computed exactly using the double quantum weight enumerator polynomials [171].

Based on this metric, we search for codes with small logical error rate p_L under three kinds of bit flip and phase error probabilities, $(p_X = 0.01, p_z = 0.05), (p_x = 0.05, p_z = 0.01)$ and $(p_x = 0.01, p_z = 0.01)$. Due to the large search space and computation cost of exact logical error rate calculation, we focus on codes with $n \leq 13$ and $k \leq 2$.

Fig 4.10 shows the overview of synthesizing QLego codes with NuQES. A QLego code is decided by the edges that glue the small codes together. Fig 4.10(a)(b) shows two different kinds of blue edges to glue two tensors of a $[[4, 2, 2]]$ code (See Figure 4.7 in Appendix 4.4 for the definition of this tensor) and generate a $[[6, 4, 2]]$ code or the famous $[[7, 1, 3]]$ Steane [30] code respectively. The heuristic function generated in the heuristic function synthesis step is an edge score function that giving a priority to each potential connecting edge between tensors.

At each iteration of the heuristic function synthesis step, NuQES builds a prompt by combining two programs sampled from the program database (favoring high-scoring ones) and adding a task description to generate an edge score function $\mathbf{F}$ in the correct format. The prompt is then fed to the pretrained LLM and new edge score functions are created and fed to the code

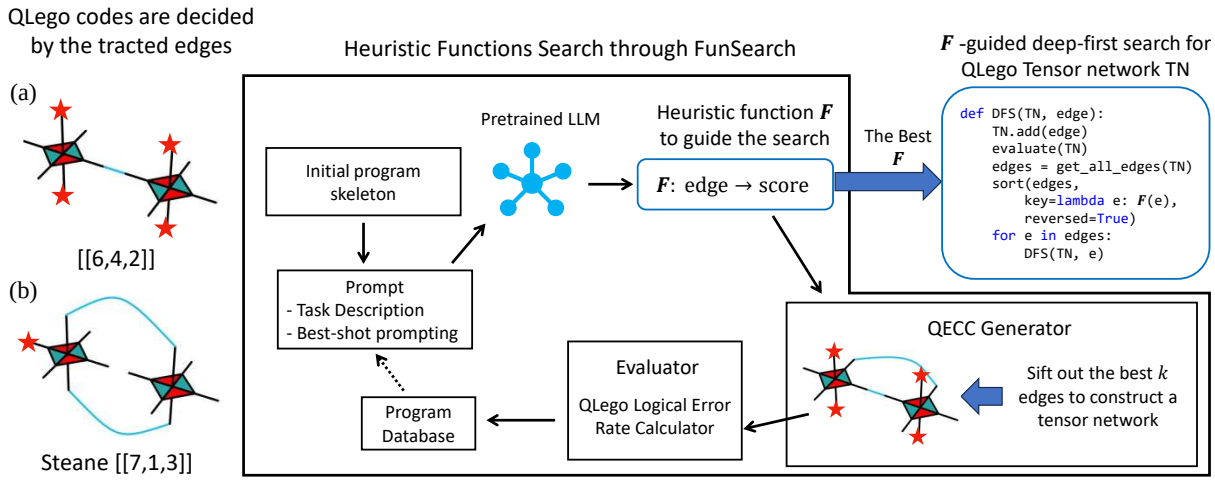


Figure 4.10: (a) $[[6, 4, 2]]$ codes constructed by two $[[4, 2, 2]]$ tensors. (b) Steane $[[7, 1, 3]]$ code constructed by two $[[4, 2, 2]]$ tensors in another way. (c) Quantum Lego codes synthesis through NuQES. NuQES utilize FunSearch to find a heuristic function $\mathbf{F}$ which gives a score to each possible edge when glueing the tensors. Each candidate function $\mathbf{F}$ is evaluated by the logical error rate of the codes constructed based on the edges chosen by $\mathbf{F}$. The best $\mathbf{F}$ is used to guide depth-first search (pseudo-code `DFS` on the right) in the large code space.

generator. Each edge score function $\mathbf{F}$ maps a valid edge that glues small code together to a float number as the score of this edge. Given an edge score function $\mathbf{F}$, the QECC generator picks M valid edges with the highest score according to $\mathbf{F}$ and used them to construct a QLego tensor network. We choose $M = 8$ in this case. Each edge score function is scored by the logical error rate of the code generated in this way. Then, the edge score function will be stored in the program database and may be used to construct the prompt for future iterations.

After the heuristic function synthesis step, the best edge score function will be used to guide the Depth-first search for larger QLego tensor network code. As shown in the pseudo-code at the right of Fig 4.10, the `DFS` function receives the current tensor network `TN` (which is set as empty in the initial step) and an edge to be added. `DFS` first adds the `edge` to `TN` and evaluates the logical error rate of the codes constructed by this new tensor network. Then `DFS` executes the depth-first recursive search by exploring the high score edge first.

Codes	$p_L \cdot 10^{-6}$				$p_L \cdot 10^{-8}$	
	$p_x = 0.01, p_z = 0.05$		$p_x = 0.05, p_z = 0.01$		$p_x = 0.01, p_z = 0.01$	
$[[n, k, d]]$	Best Known	NuQES	Best Known	NuQES	Best Known	NuQES
$[[5, 1, 3]]$	175[174]	**161**	175[174]	**161**	841[172]	841
$[[6, 1, 3]]$	50.9[174]	**25.9**	50.9[174]	**25.9**	433[174]	**334**
$[[7, 1, 3]]$	32.2[174]	**13.5**	25.8[174]	**13.5**	229[174]	**25.3**
$[[8, 1, 3]]$	53.6[174]	**6.85**	14.4[174]	**4.89**	229[174]	**15.0**
$[[9, 1, 3]]$	15.3[174]	**2.14**	8.62[174]	**3.10**	68.7[172]	**0.517**
$[[10, 1, 4]]$	1.82[174]	1.82	1.82[174]	1.82	4.54[174]	**0.496**
$[[11, 1, 5]]$	0.999[172]	**0.465**	0.999[172]	**0.385**	0.320[174]	**0.314**
$[[12, 1, 5]]$	0.999[172]	**0.376**	0.999[172]	**0.209**	0.341[172]	**0.331**
$[[13, 1, 5]]$	0.404[174]	**0.130**	0.219[174]	**0.147**	0.181[174]	**0.221**
$[[5, 2, 2]]$	359[172]	**166**	359[172]	**166**	841 [172]	841
$[[6, 2, 2]]$	359[172]	**83.6**	359[172]	**69.4**	841[172]	**334**
$[[7, 2, 2]]$	324[172]	**59.4**	324[172]	**34.2**	841[172]	**25.3**
$[[8, 2, 3]]$	119[174]	**13.4**	57.6[174]	**10.5**	68.7[172]	**20.3**
$[[9, 2, 3]]$	189[174]	**12.7**	52.0[174]	**10.8**	68.7[172]	**16.9**
$[[10, 2, 4]]$	6.36[174]	**6.14**	6.37[174]	**6.14**	15.3[174]	**4.93**

Table 4.5: Logical error rate p_L for best-known codes and codes found by NuQES. We consider biased noise models that each qubit goes through a noise channel with independent bit flip and phase error probabilities p_x and p_z respectively.

Synthesis Result We compare the codes found by NuQES with the existing codes we have collected from (1) the best code documented in the QECC zoo CodeTables and the Best Known Quantum Codes library of MAGMA[172]. (2) codes found by the recently proposed QECC searching frameworks [145, 173, 174]. (3) Small rotated surface codes on which many experiments focus[175]. Table 4.5 shows the comparison result. The best-known existing codes used as benchmarks are chosen based on its p_L^{norm}. NuQES successfully codes with smaller logical error rate compared with the best-known codes in all cases.

Chapter 5: Conclusion and Future Directions

In summary, this dissertation has laid the groundwork for building automated quantum frameworks to synthesize quantum programs at the high-level language, assembly language, and hardware control level language.

We first present QSynth, the first quantum program synthesis framework, including a new inductive quantum programming language, its specification, a sound logic for reasoning, and an encoding of the reasoning procedure into SMT instances. QSynth is equipped with a QSynth-spec language that enables input-output style specification for inductive quantum programs, a hypothesis-amplitude ($h - \alpha$) specification for scalable verification of quantum programs, a new programming language ISQIR which supports recursively defined families of quantum unitary circuits, and a Hoare-type logic for proving the correctness of ISQIR program. By leveraging existing SMT solvers, QSynth successfully automates the verification of inductive quantum programs and synthesizes ten quantum unitary programs. QSynth can generate programs better than the standard solutions in the quantum subtractor and quantum conditional adder cases. These programs can be readily transpiled to executable programs on major quantum platforms, e.g., Q#, IBM Qiskit.

Then We present MQCC, a meta-programming framework, to assist NISQ application designers to identify the best balance of trade-offs among heterogeneous factors specific to the

targeted application and quantum hardware in an automatic way. We demonstrate MQCC's expresiveness and generality by using it to implement several case studies. We employ MQCC to automate the selection of a syndrome extraction scheme of Chao and Reichardt [4], where we easily handle the heterogeneous qubit case in MQCC. We also implemented an automated procedure to trade accuracy for savings of circuit volume in implementations of Quantum Fourier Transformation (QFT) and Quantum Phase Estimation (QPE). Lastly, we implemented both the multi-programming and Crosstalk mitigation optimizations and also implemented a novel composition of these two optimizations which balances the tradeoff among the three attributes MQCC makes this composition simple to express. Ideally, it would be desirable to develop more automation by leveraging the MQCC framework from domain problems, as well as improving the scalability of MQCC with more efficient generation of cost expression.

Finally, we propose NuQES for synthesizing QEC codes with a low logical error rate and high encoding rate. NuQES is equipped with a two-step synthesis approach to handle the challenge for synthesizing large-size QEC codes. This approach leverages both symbolic search strategies and heuristic-driven exploration, offering a scalable solution to the QEC code synthesis problem. We utilize NuQES to found (1) Bivariate Bicycle (BB) LDPC Code[3]; (2) QLego codes[36]. NuQES successfully found two $[[170, 16, 10]]$ and $[[288, 12, 22]]$ BB codes, which have higher encoding rates and error thresholds than the best codes found by Bravyi et al. [3]. In the QLego formalism, NuQES also produce stabilizer codes which outperform their counterparts of similar length, rate, and distance under biased noise. In particular, the QLego codes are no longer restricted to $k = 1$ CSS codes as in Su et al. [145].

These contributions above provide fundamental tools for advancing quantum software development, but they also present new challenges and limitations that must be addressed in future

research.

- QSynth currently only supports unitary quantum programs without oracles, whereas extending its capability to handle oracle-based quantum programs would be highly beneficial, given the popularity of oracle algorithms in quantum computing.

- NuQes can be further improved by synthesizing low-density weight-check quantum error correction codes while considering decoder and hardware constraints, which would enhance its practicality in real-world applications.

Beyond extending existing projects to overcome their limitations, quantum program synthesis offers several promising directions for future development.

- One such area is quantum analog machines, which serve as powerful platforms for solving Hamiltonian simulation problems. These machines can naturally emulate complex quantum systems by evolving their own quantum states according to a given Hamiltonian, making them highly effective for studying many-body physics, material science, and optimization problems. However, despite their advantages, quantum analog machines face significant challenges related to hardware constraints and error suppression. One major limitation is the difficulty of implementing conventional quantum error correction techniques on these machines. To address this, quantum program synthesis can be leveraged to generate weight-reducted penalty Hamiltonians from existing quantum error correction codes, enabling more effective error mitigation strategies.

- Quantum network protocols often involve complex asynchronous patterns, making it challenging to write efficient and error-free network control programs. A particularly quantum-

specific challenge is decoherence time—the limited duration over which quantum information remains stable before it is lost due to environmental noise. Unlike classical networks, where delays do not degrade stored information, quantum network protocols must optimize timing to ensure operations are complete before qubits decohere. Quantum program synthesis can assist by automatically generating network control programs that account for decoherence constraints, dynamically adapting execution schedules, and optimizing resource allocation to minimize errors.

- The diversity of quantum hardware platforms, such as superconducting qubits and ion-trap systems, has led to the fragmentation of quantum programming languages and instruction sets, making code migration a time-consuming task. Given the demonstrated success of large language models in classical code translation, integrating them into quantum program synthesis could significantly ease the burden of migrating quantum code across different hardware architectures, improving interoperability and accelerating quantum software development.

By addressing the existing limitations and expanding the capabilities of new automated synthesis frameworks, we aim to push the boundaries of quantum computing and bring us closer to achieving real-world quantum advantages. The tools developed in this dissertation represent important steps toward that goal, and we hope that their continued refinement will accelerate the development of practical, efficient, and scalable quantum computing solutions.

Appendix 6: Appendix

6.1 QSynth Appendix

6.1.1 Proofs

Proof of Theorem 2.4.5

Proof. We verify each rule's soundness by reasoning about the semantics of the program constructs.

WEAKEN For $(n, x, y) \in h' \subset h$, there is $\langle y| [\![\{\!\{S\}\!\}(n)]\!] |x\rangle = \alpha(n, x, y)$, hence $h \triangleright S \leftrightarrow \alpha$.

CONST For any $(n, x, y) \in \mathbb{N}^3$, note $\{\!\{\textbf{const P}\}\!\}(n) = P$ and $\alpha(n, x, y) = [\![P]\!](x, y)$. This makes $\langle y| [\![\{\!\{\textbf{const P}\}\!\}(n)]\!] |x\rangle = \alpha(n, x, y)$, and $\mathbb{N}^3 \triangleright \textbf{const P} \leftrightarrow \alpha$.

REPLACE For any $(n, x, y) \in h$, since $\langle y| [\![\{\!\{S\}\!\}(n)]\!] |x\rangle = \alpha(n, x, y) = \alpha'(n, x, y)$, we conclude $h \triangleright S \leftrightarrow \alpha'$.

RELABEL For any $(n, x, y) \in h$, there is $\langle y| [\![\{\!\{S\}\!\}(n)]\!] |x\rangle = \alpha(n, x, y)$. Because $\pi(n)$ is injective, for any $(n, u, v) \in \pi \circ h$, there exists x, y such that $\pi(n, x) = u$ and $\pi(n, y) = v$.

Then $\langle u|\,[\![\{\!\{\textbf{relabel}\ \pi\ S\}\!\}(n)]\!]\,|v\rangle = \langle \pi(n,y)|\,[\![\{\!\{\textbf{relabel}\ \pi\ S\}\!\}(n)]\!]\,|\pi(n,x)\rangle = \langle y|\,[\![\{\!\{S\}\!\}(n)]\!]\,|x\rangle = \alpha n, x, y = \alpha'(n,u,v)$.

SEQ For any $(n,x,y) \in h$, note $\langle y|\,[\![\{\!\{\textbf{seq}\ S_1\ S_2\}\!\}(n)]\!]\,|x\rangle = \langle y|\,[\![\{\!\{S_2\}\!\}(n)]\!][\![\{\!\{S_1\}\!\}(n)]\!]\,|x\rangle = \sum_z \langle y|\ [\![\{\!\{S_2\}\!\}(n)]\!]\,|z\rangle\,\langle z|\,[\![\{\!\{S_2\}\!\}(n)]\!]\,|x\rangle$. For any z, if $(n,x,z) \in h_1 \wedge \alpha_1(n,x,z) = 0$, then $\langle z|\,[\![\{\!\{S_2\}\!\}(n)]\!]\,|x\rangle = \alpha_1(n,x,z) = \alpha_1(n,x,z)\alpha_2(n,z,y)$. Similarly, if $(n,z,y) \in h_2 \wedge \alpha_2(n,z,y) = 0$, it equals to $\alpha_1(n,x,z)\alpha_2(n,z,y)$. If $(n,x,z) \in h_1 \wedge (n,z,y) \in h_2$, the term also becomes $\alpha_1(n,x,z)\alpha_2(n,z,y)$. Hence we have $h \triangleright \textbf{seq}\ S_1\ S_2 \leftrightarrow \alpha$.

FIX Similarly, we denote $\textbf{fix}_k\ \pi\ P_0\ \cdots\ P_{k-1}\ S_L\ S_R$ as $\textbf{fix}_k$. For any $(i,x,y) \in h$ where $i < k$, by $(h_i, \alpha_i) \equiv_i (h, \alpha)$, we know $(i,x,y) \in h_i$ and $\alpha(i,x,y) = \alpha_i(i,x,y)$. Since $h_i \triangleright \textbf{const}\ \mathrm{P}_i \leftrightarrow \alpha_i$ and $(h,\alpha) \equiv_i (h_i, \alpha_i)$, we have $\langle y|\,[\![\{\!\{\textbf{fix}_k\}\!\}(i)]\!]\,|x\rangle = \langle y|\,[\![P_i]\!]\,|x\rangle = \alpha_i(i,x,y) = \alpha(i,x,y)$. For any $(i,x,y) \in h$ such that $i \geq k$, note $\{\!\{\textbf{fix}_k\}\!\} = \{\!\{S_L\}\!\}(i); \texttt{map\_qb}(\pi(i), \{\!\{\textbf{fix}_k\}\!\}(i-1)); \{\!\{S_R\}\!\}(i)$. According to the proof for RELABEL and SEQ, with $(h,\alpha) \equiv_i (h_L, \alpha_L) \otimes (\pi \circ \text{pred}\ (h,\alpha)) \otimes (h_R, \alpha_R)$, we have $\langle y|\,[\![\{\!\{\textbf{fix}_k\}\!\}(i)]\!]\,|x\rangle = \alpha(i,x,y)$. □

Proof of Theorem 2.5.3

Proof. We first prove that given a SQIR program P and we have $\mathbb{N}^3 \triangleright \textbf{const}\ \mathrm{P} \leftrightarrow \alpha$, then α is sparse.

Since P is a SQIR program, P is a sequence of applications of gates to fixed number of

qubits and we have

$$\alpha(n,x,y) = \{\!\{P\}\!\}_{xy} \tag{6.1}$$

Without loss of generality, suppose $\{\!\{P\}\!\}$ is a unitary applied to qubits $q_0, q_1, ..., q_m, m \in \mathbb{N}$. Let function $\mathcal{X}, \mathcal{Y}$ be

$$\mathcal{X}(n,y) = \begin{cases} \{k \mid k < 2^{m+1}, k \in \mathbb{N}\},\ y < 2^{m+1} \\ \{\mathbf{mk}(y,0,m) + k \mid k < 2^{m+1}, k \in \mathbb{N}\},\ \text{otherwise} \end{cases} \tag{6.2}$$

$$\mathcal{Y}(n,x) = \begin{cases} \{k \mid k < 2^{m+1}, k \in \mathbb{N}\},\ x < 2^{m+1} \\ \{\mathbf{mk}(x,0,m) + k \mid k < 2^{m+1}, k \in \mathbb{N}\},\ \text{otherwise} \end{cases} \tag{6.3}$$

where $\mathbf{mk}(x,0,m)$ means to set from the 0th bit to the mth bits (in the order from low to high) in x to 0 (e.g. $\mathbf{mk}((1111)_2,0,1)) = (1100)_2 = (12)_{10}$). It is easy to see that for arbitrary inputs, the size of the sets returned by $\mathcal{Y}, \mathcal{X}$ is always 2^{m+1}. Now we prove that

$$\forall n,x,y \in \mathbb{N}, \alpha(n,x,y) \neq 0 \rightarrow y \in \mathcal{Y}(n,x) \tag{6.4}$$

Formula $\alpha(n,x,y) \neq 0 \rightarrow x \in \mathcal{X}(n,y)$ can be proved in the same way. When $\alpha(n,x,y) \neq 0$:

- if $x < 2^{m+1}$, since $\{\!\{P\}\!\}$ is a unitary applied to qubits $q_0, q_1, ..., q_m$, $\langle x| \{\!\{P\}\!\} |y\rangle = 0$ for every $y \geq 2^{m+1}$. So we know that $y < 2^{m+1}$ and $y \in \mathcal{Y}(n,x)$.
- Otherwise, since $\{\!\{P\}\!\}$ does not change the state of qubits other than qubits $q_0, q_1, q_2, ..., q_m$, we should have $\mathbf{mk}(x,0,m) = \mathbf{mk}(y,0,m)$ if $\alpha(n,x,y) \neq 0$. Then we have $y \in \mathcal{Y}(n,x)$

So we have $\alpha \trianglelefteq (\mathcal{X}, \mathcal{Y})$ and we prove that α is sparse.

Next we prove that given a sparse amplitude function α and an injective mapping π, function $\pi \circ \alpha$ is sparse.

Since α is sparse, there exists functions $\mathcal{X}, \mathcal{Y}$ and we have $\alpha \trianglelefteq (\mathcal{X}, \mathcal{Y})$. Let functions $\mathcal{X}^\pi, \mathcal{Y}^\pi$ be

$$\mathcal{X}^\pi(n, y) = \{\pi^{-1}(k) \mid k \in \mathcal{X}(n, \pi(y))\} \tag{6.5}$$

$$\mathcal{Y}^\pi(n, x) = \{\pi^{-1}(k) \mid k \in \mathcal{Y}(n, \pi(x))\} \tag{6.6}$$

Then we have

$$\begin{aligned}
(\pi \circ \alpha)(n, x, y) \neq 0 &\implies \alpha(n, \pi(x), \pi(y)) \neq 0 \\
&\implies \pi(x) \in \mathcal{X}(n, \pi(y)) \wedge \pi(y) \in \mathcal{Y}(n, \pi(x)) \\
&\implies x \in \{\pi^{-1}(k) \mid k \in \mathcal{X}(n, \pi(y))\} \wedge y \in \{\pi^{-1}(k) \mid k \in \mathcal{Y}(n, \pi(x)\} \\
&\implies x \in \mathcal{X}^\pi(n, y) \wedge y \in \mathcal{Y}^\pi(n, x)
\end{aligned}$$

So we have

$$\forall n\ x\ y, (\pi \circ \alpha)(n, x, y) \neq 0 \rightarrow y \in \mathcal{Y}^\pi(n, x) \wedge x \in \mathcal{X}^\pi(n, y)$$

So we have , $\pi \circ \alpha \trianglelefteq (\mathcal{X}^\pi, \mathcal{Y}^\pi)$ and we prove that $\pi \circ \alpha$ is sparse.

Finally we prove that given two sparse amplitude function α_1, α_2, function $\alpha_1 * \alpha_2$ is sparse.

Since α_1, α_2 are sparse, suppose we have $\alpha_1 \trianglelefteq (\mathcal{X}_1, \mathcal{Y}_1)$ and $\alpha_2 \trianglelefteq (\mathcal{X}_2, \mathcal{Y}_2)$. We also have

$$(\alpha_1 * \alpha_2)(n, x, y) = \sum_{z \in \mathcal{X}_1(n,y)} \alpha_1(n, x, z)\alpha_2(n, z, y)$$

Let functions $\mathcal{X}, \mathcal{Y}$ be

$$\mathcal{X}(n, y) := \{\mathcal{X}_1(n, z) | z \in \mathcal{X}_2(n, y)\} \tag{6.7}$$

$$\mathcal{Y}(n, x) := \{\mathcal{Y}_2(n, z) | z \in \mathcal{Y}_1(n, x)\} \tag{6.8}$$

Then we have

$$(\alpha_1 * \alpha_2)(n, x, y) \neq 0 \Longrightarrow \exists z, \alpha_1(n, x, z) \neq 0 \wedge \alpha_2(n, z, y) \neq 0 \tag{6.9}$$

$$\Longrightarrow \exists z, z \in \mathcal{Y}_1(n, x) \wedge z \in \mathcal{X}_2(n, y) \wedge x \in \mathcal{X}_1(n, z) \wedge y \in \mathcal{Y}_2(n, z) \tag{6.10}$$

$$\Longrightarrow x \in \mathcal{X}(n, y) \wedge y \in \mathcal{Y}(n, x) \tag{6.11}$$

So we have

$$\forall n\ x\ y, (\alpha_1 * \alpha_2)(n, x, y) \neq 0 \rightarrow x \in \mathcal{X}(n, y) \wedge y \in \mathcal{Y}(n, x) \tag{6.12}$$

So $\alpha_1 * \alpha_2 \trianglelefteq (\mathcal{X}, \mathcal{Y})$ and we prove that $\alpha_1 * \alpha_2$ is sparse. □

6.2 MQCC Appendix

6.2.1 Proof of Theorem 3.3.1

Proof. Before we start, we show that for any statement S, $\sigma \in \Sigma$ and state $s : T$, where no **choice** is nested inside **case**, there is

$$A.\texttt{value}([\![S]\!](\sigma, s)) = A.\texttt{value}(s) + A.\texttt{value}([\![S]\!](\sigma, A.\texttt{empty})).$$

Notice that by choosing $s = A.\texttt{empty}$ in above equation, we have $A.\texttt{value}(A.\texttt{empty}) = 0$.

This can be proved by induction on S. For the case that S is an operation or a **case** clause, it is by the additive properties of the attribute. For $S = S_1; S_2$, notice that

$$\begin{aligned}
& A.\texttt{value}([\![S_1; S_2]\!](\sigma, s)) \\
= {} & A.\texttt{value}([\![S_2]\!](\sigma, [\![S_1]\!](\sigma, s))) \\
= {} & A.\texttt{value}([\![S_1]\!](\sigma, s)) + A.\texttt{value}([\![S_2]\!](\sigma, A.\texttt{empty})) \\
= {} & A.\texttt{value}(s) + A.\texttt{value}([\![S_1]\!](\sigma, A.\texttt{empty})) \\
& \qquad + A.\texttt{value}([\![S_2]\!](\sigma, A.\texttt{empty})) \\
= {} & A.\texttt{value}(s) + A.\texttt{value}([\![S_2]\!](\sigma, [\![S_1]\!](\sigma, A.\texttt{empty}))) \\
= {} & A.\texttt{value}(s) + A.\texttt{value}([\![S_1; S_2]\!](\sigma, A.\texttt{empty})).
\end{aligned}$$

For $S = \textbf{choice}(\mathit{var})\{\overline{i \to S_i}\}$, let k be $\sigma[\mathit{var}]$. We equates:

$$
\begin{aligned}
& A.\texttt{value}([\![S]\!](\sigma, s)) \\
=\ & A.\texttt{value}([\![S_k]\!](\sigma, s)) \\
=\ & A.\texttt{value}(s) + A.\texttt{value}([\![S_k]\!](\sigma, A.\texttt{empty})) \\
=\ & A.\texttt{value}(s) + A.\texttt{value}([\![S]\!](\sigma, A.\texttt{empty})).
\end{aligned}
$$

We now prove the theorem by induction on S. Notice that $\sum_{i \in \Sigma_{\mathit{var}}} \delta^i_{\mathit{var}} = 1$, so we have $\sum_{\sigma \in \Sigma} \delta_{\mathit{Vars},\sigma} \cdot r = r$ for any constant $r \in \mathbb{R}$.

For the base case where $S =$ *opID(exps, regs)*, and $S = \textbf{case}(\mathit{creg})\{\overline{i \to S_i}\}$, it is true by the above equation.

To show the target for $S = S_1; S_2$, we have

$$
\begin{aligned}
& \mathrm{cost}^+_A(S_1; S_2) \\
=\ & \mathrm{cost}^+_A(S_1) + \mathrm{cost}^+_A(S_2) \\
=\ & \mathrm{cost}_A(S_1) + \mathrm{cost}_A(S_2) \\
=\ & \sum_{\sigma \in \Sigma} \delta_{\mathit{Vars},\sigma} \cdot (A.\texttt{value}(A.\texttt{empty}) + \texttt{value}([\![S_1]\!](\sigma, A.\texttt{empty})) \\
& \qquad\qquad + A/\texttt{value}([\![S_2]\!](\sigma, A.\texttt{empty}))) \\
=\ & \sum_{\sigma \in \Sigma} \delta_{\mathit{Vars},\sigma} \cdot \texttt{value}([\![S_1; S_2]\!](\sigma, \texttt{empty})).
\end{aligned}
$$

Notice that $\sum_{i \in \bar{i}} \delta^i_{var} \delta_{Var,\sigma} = \delta_{Vars,\sigma}$ since

$$\delta^i_{var} \delta_{Var,\sigma} = \delta^i_{var} \delta^{\sigma[var]}_{var} \prod_{u \in Vars \setminus \{var\}} \delta^{\sigma[u]}_u$$

is non-zero only when $i = \sigma[var]$. So for $S = \textbf{choice}(var)\{\overline{i \to S_i}\}$ we have

$$\begin{aligned}
& \mathrm{cost}^+_A(S) \\
= & \sum_{i \in \bar{i}} \delta^i_{var} \mathrm{cost}^+_A(S_k) \\
= & \sum_{i \in \bar{i}} \delta^i_{var} \sum_{\sigma \in \Sigma} \delta_{Var,\sigma} A.\mathtt{value}([\![S_{\sigma[var]}]\!](\sigma, A.\mathtt{empty})) \\
= & \sum_{\sigma \in \Sigma} \delta_{Var,\sigma} A.\mathtt{value}([\![S_{\sigma[var]}]\!](\sigma, A.\mathtt{empty})) \\
= & \, \mathrm{cost}_A(S).
\end{aligned}$$

□

Bibliography

[1] Poulami Das, Swamit S Tannu, Prashant J Nair, and Moinuddin Qureshi. A case for multi-programming quantum computers. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 291–303, 2019.

[2] Ang Li, Samuel Stein, Sriram Krishnamoorthy, and James Ang. Qasmbench: A low-level qasm benchmark suite for nisq evaluation and simulation. *arXiv preprint arXiv:2005.13018*, 2021.

[3] Sergey Bravyi, Andrew W Cross, Jay M Gambetta, Dmitri Maslov, Patrick Rall, and Theodore J Yoder. High-threshold and low-overhead fault-tolerant quantum memory. *Nature*, 627(8005):778–782, 2024.

[4] Rui Chao and Ben W Reichardt. Quantum error correction with only two extra qubits. *Physical review letters*, 121(5):050502, 2018.

[5] Ben W Reichardt. Fault-tolerant quantum error correction for steane's seven-qubit color code with few or no extra qubits. *Quantum Science and Technology*, 6(1):015007, 2020.

[6] Emanuel Knill. Quantum computing with realistically noisy devices. *Nature*, 434(7029): 39–44, 2005.

[7] Prakash Murali, Jonathan Baker, Ali Javadi-Abhari, Frederic Chong, and Margaret Martonosi. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. pages 1015–1029, 04 2019. doi: 10.1145/3297858.3304075.

[8] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*, volume 2. Cambridge university press Cambridge, 2001.

[9] Seth Lloyd. Universal quantum simulators. *Science*, 273(5278):1073–1078, 1996.

[10] Hefeng Wang, Sabre Kais, Alán Aspuru-Guzik, and Mark R Hoffmann. Quantum algorithm for obtaining the energy spectrum of molecular systems. *Physical Chemistry Chemical Physics*, 10(35):5388–5393, 2008.

[11] Markus Reiher, Nathan Wiebe, Krysta M Svore, Dave Wecker, and Matthias Troyer. Elucidating reaction mechanisms on quantum computers. *Proceedings of the national academy of sciences*, 114(29):7555–7560, 2017.

[12] Yuri Alexeev, Dave Bacon, Kenneth R Brown, Robert Calderbank, Lincoln D Carr, Frederic T Chong, Brian DeMarco, Dirk Englund, Edward Farhi, Bill Fefferman, et al. Quantum computer systems for scientific discovery. *PRX quantum*, 2(1):017001, 2021.

[13] Robert Wille, Rod Van Meter, and Yehuda Naveh. Ibm's qiskit tool chain: Working with and developing for real quantum computers. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1234–1240. IEEE, 2019.

[14] Ali J Abhari, Arvin Faruque, Mohammad J Dousti, Lukas Svec, Oana Catu, Amlan Chakrabati, Chen-Fu Chiang, Seth Vanderwilt, John Black, and Fred Chong. Scaffold: Quantum programming language. Technical report, Princeton Univ NJ Dept of Computer Science, 2012.

[15] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S Bishop, Steven Heidel, Colm A Ryan, Prasahnt Sivarajah, John Smolin, Jay M Gambetta, et al. Openqasm 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, 2022.

[16] Daniel Sierra-Sosa, Michael Telahun, and Adel Elmaghraby. Tensorflow quantum: Impacts of quantum state preparation on quantum machine learning performance. *IEEE Access*, 8: 215246–215255, 2020.

[17] Vivek V Shende, Stephen S Bullock, and Igor L Markov. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1000–1010, 2006.

[18] Timothee Goubault de Brugiere. *Methods for optimizing the synthesis of quantum circuits*. PhD thesis, Universite Paris-Saclay, 2020.

[19] Matthew Amy, Dmitri Maslov, Michele Mosca, and Martin Roetteler. 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):818–830, 2013.

[20] Mehdi Saeedi, Robert Wille, and Rolf Drechsler. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing*, 10(3):355–377, 2011.

[21] A Yu Kitaev. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, 52(6):1191, 1997.

[22] Ed Younis, Koushik Sen, Katherine Yelick, and Costin Iancu. Qfast: Conflating search and numerical optimization for scalable quantum circuit synthesis. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 232–243. IEEE, 2021.

[23] Chan Gu Kang and Hakjoo Oh. Modular component-based quantum circuit synthesis. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):348–375, 2023. doi: https://doi.org/10.1145/3586039.

[24] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):1–49, 2012.

[25] Matthew Amy. Towards large-scale functional verification of universal quantum circuits. *arXiv preprint arXiv:1805.06908*, 2018.

[26] Yu-Fang Chen, Kai-Min Chung, Ondrej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. An automata-based framework for verification and bug hunting in quantum circuits. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1218–1243, 2023. doi: https://doi.org/10.1145/3591270.

[27] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi: 10.1145/3434318. URL https://doi.org/10.1145/3434318.

[28] Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:R2493–R2496, Oct 1995. doi: 10.1103/PhysRevA.52.R2493. URL https://link.aps.org/doi/10.1103/PhysRevA.52.R2493.

[29] A. R. Calderbank and Peter W. Shor. Good quantum error-correcting codes exist. *Phys. Rev. A*, 54:1098–1105, Aug 1996. doi: 10.1103/PhysRevA.54.1098. URL https://link.aps.org/doi/10.1103/PhysRevA.54.1098.

[30] Andrew M Steane. Error correcting codes in quantum theory. *Physical Review Letters*, 77 (5):793, 1996.

[31] Daniel Gottesman. *Stabilizer codes and quantum error correction*. California Institute of Technology, 1997.

[32] Andrew M Steane. A tutorial on quantum error correction. *Quantum Computers, Algorithms and Chaos*, pages 1–32, 2006.

[33] Daniel Gottesman. An introduction to quantum error correction and fault-tolerant quantum computation. In *Quantum information science and its contributions to mathematics, Proceedings of Symposia in Applied Mathematics*, volume 68, pages 13–58, 2010.

[34] Adriano Barenco, Artur Ekert, Kalle-Antti Suominen, and Päivi Törmä. Approximate quantum fourier transform and decoherence. *Physical Review A*, 54(1):139, 1996.

[35] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[36] ChunJun Cao, Michael J Gullans, Brad Lackey, and Zitao Wang. Quantum lego expansion pack: Enumerators from tensor networks. *arXiv preprint arXiv:2308.05152*, 2023.

[37] Krysta M Svore, Alfred V Aho, Andrew W Cross, Isaac Chuang, and Igor L Markov. A layered software architecture for quantum computing design tools. *Computer*, 39(1):74–83, 2006.

[38] Mohammad Javad Dousti, Alireza Shafaei, and Massoud Pedram. Squash 2: a hierarchical scalable quantum mapper considering ancilla sharing. *arXiv preprint arXiv:1512.07402*, 2015.

[39] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[40] IBM. Ibm q device information. https://quantum-computing.ibm.com/docs/manage/backends/, 2021.

[41] Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoit Valiron. A deductive verification framework for circuit-building quantum programs. *arXiv preprint arXiv:2003.05841*, 2020.

[42] Dax Enshan Koh, Mark D Penney, and Robert W Spekkens. Computing quopit clifford circuit amplitudes by the sum-over-paths technique. *Quantum Information & Computation*, 2017. doi: https://doi.org/10.26421/QIC17.13-14-1.

[43] Sergey Bravyi and David Gosset. Improved classical simulation of quantum circuits dominated by clifford gates. *Physical review letters*, 116(25):250501, 2016.

[44] Matthew Amy, Parsiad Azimzadeh, and Michele Mosca. On the controlled-not complexity of controlled-not–phase circuits. *Quantum Science and Technology*, 4(1):015002, 2018.

[45] Matthew Amy, Dmitri Maslov, and Michele Mosca. Polynomial-time t-depth optimization of clifford+ t circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2014. doi: 10.1109/TCAD.2014.2341953.

[46] Matthew Amy and Michele Mosca. T-count optimization and reed–muller codes. *IEEE Transactions on Information Theory*, 65(8):4771–4784, 2019. doi: 10.1109/TIT.2019.2906374.

[47] Dave Bacon, Wim van Dam, and Alexander Russell. Analyzing algebraic quantum circuits using exponential sums. *unpublished: see tinyurl. com/qpo7s2*, 2008.

[48] Christopher M Dawson and Michael A Nielsen. The solovay-kitaev algorithm. *arXiv preprint quant-ph/0505030*, 2005.

[49] Ashley Montanaro. Quantum circuits and low-degree polynomials over. *Journal of Physics A: Mathematical and Theoretical*, 50(8):084002, 2017. doi: 10.1088/1751-8121/aa565f.

[50] Jay Gambetta. Ibm quantum roadmap to build quantum-centric supercomputers, Aug 2022. URL https://research.ibm.com/blog/ibm-quantum-roadmap-2025.

[51] Richard P Feynman. Quantum mechanical computers. *Optics news*, 11(2):11–20, 1985.

[52] Steven A Cuccaro, Thomas G Draper, Samuel A Kutin, and David Petrie Moulton. A new quantum ripple-carry addition circuit. *arXiv preprint quant-ph/0410184*, 2004.

[53] Seth Lloyd. Quantum algorithm for solving linear systems of equations. In *APS March Meeting Abstracts*, volume 2010, pages D4–002, 2010. doi: https://doi.org/10.1103/PhysRevLett.103.150502.

[54] Charles H Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K Wootters. Teleporting an unknown quantum state via dual classical and einstein-podolsky-rosen channels. *Physical review letters*, 70(13):1895, 1993. doi: https://doi.org/10.1103/PhysRevLett.70.1895.

[55] Don Coppersmith. An approximate fourier transform useful in quantum factoring. *arXiv preprint quant-ph/0201067*, 2002.

[56] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017. ISSN 2325-1107. doi: 10.1561/2500000010. URL http://dx.doi.org/10.1561/2500000010.

[57] Daniel M Greenberger, Michael A Horne, and Anton Zeilinger. Going beyond bell's theorem. pages 69–72, 1989.

[58] Yan Xia, Chang-Bao Fu, Shou Zhang, Suc-Kyoung Hong, Kyu-Hwang Yeon, and Chung-In Um. Quantum dialogue by using the ghz state. *arXiv preprint quant-ph/0601127*, 2006.

[59] Man Zhong-Xiao and Xia Yun-Jie. Controlled bidirectional quantum direct communication by using a ghz state. *Chinese Physics Letters*, 23(7):1680, 2006.

[60] Mark Hillery, Vladimir Buzek, and André Berthiaume. Quantum secret sharing. *Physical Review A*, 59(3):1829, 1999.

[61] Ci-Hong Liao, Chun-Wei Yang, and Tzonelish Hwang. Dynamic quantum secret sharing protocol based on ghz state. *Quantum information processing*, 13(8):1907–1916, 2014.

[62] Chia-Chun Lin, Amlan Chakrabarti, and Niraj K Jha. Qlib: Quantum module library. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 11(1):1–20, 2014.

[63] Menghan Dou, Tianrui Zou, Yuan Fang, Jing Wang, Dongyi Zhao, Lei Yu, Boying Chen, Wenbo Guo, Ye Li, Zhaoyun Chen, et al. Qpanda: high-performance quantum computing framework for multiple application scenarios. *arXiv preprint arXiv:2212.14201*, 2022. doi: 10.48550/arXiv.2212.14201.

[64] Himanshu Thapliyal, Edgard Munoz-Coreas, TSS Varun, and Travis S Humble. Quantum circuit designs of integer division optimizing t-count and t-depth. *IEEE transactions on emerging topics in computing*, 9(2):1045–1056, 2019.

[65] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[66] A Yu Kitaev. Quantum measurements and the abelian stabilizer problem. *arXiv preprint quant-ph/9511026*, 1995. doi: https://doi.org/10.48550/arXiv.quant-ph/9511026.

[67] Mark Ettinger and Peter Hoyer. A quantum observable for the graph isomorphism problem. *arXiv preprint quant-ph/9901029*, 1999.

[68] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. doi: 10.1017/CBO9780511976667.

[69] Haowei Deng, Yuxiang Peng, Michael Hicks, and Xiaodi Wu. Automating nisq application design with meta quantum circuits with constraints (mqcc). *ACM Transactions on Quantum Computing*, 4(3):1–29, 2023. doi: https://doi.org/10.1145/3579369.

[70] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Synthesizing quantum-circuit optimizers. *Proceedings of the ACM on Programming Languages*, 7 (PLDI):835–859, 2023. doi: https://doi.org/10.1145/3591254.

[71] Emanuel Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *International workshop on approaches and applications of inductive programming*, pages 50–73. Springer, 2009.

[72] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 215–224. IEEE, 2010.

[73] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[74] Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. Search-based program synthesis. *Communications of the ACM*, 61(12):84–93, 2018.

[75] Qinheping Hu and Loris D'Antoni. Syntax-guided synthesis with quantitative syntactic objectives. In *International Conference on Computer Aided Verification*, pages 386–403. Springer, 2018.

[76] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[77] Sumit Gulwani, William R. Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, aug 2012. ISSN 0001-0782. doi: 10.1145/2240236.2240260. URL https://doi.org/10.1145/2240236.2240260.

[78] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

[79] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–32, 2021.

[80] Loris D'Antoni, Qinheping Hu, Jinwoo Kim, and Thomas Reps. Programmable program synthesis. In *International Conference on Computer Aided Verification*, pages 84–109. Springer, 2021.

[81] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–268, 2019.

[82] Qinheping Hu, John Cyphert, Loris D'Antoni, and Thomas Reps. Synthesis with asymptotic resource bounds. In *International Conference on Computer Aided Verification*, pages 783–807. Springer, 2021.

[83] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. *ACM SIGPLAN Notices*, 46(6):50–61, 2011.

[84] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310, 2016.

[85] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. Component-based synthesis for complex apis. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 599–612, 2017.

[86] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[87] Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

[88] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195. PMLR, 2013.

[89] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[90] Jennifer Paykin, Robert Rand, and Steve Zdancewic. Qwire: a core language for quantum circuits. *ACM SIGPLAN Notices*, 52(1):846–858, 2017. doi: https://doi.org/10.1145/3009837.3009894.

[91] Nengkun Yu and Jens Palsberg. Quantum abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 542–558, 2021.

[92] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W Cross, Frederic T Chong, and Ronghui Gu. Giallar: push-button verification for the qiskit quantum compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 641–656, 2022. doi: https://doi.org/10.1145/3519939.3523431.

[93] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A Acar, et al. Quartz: superoptimization of quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 625–640, 2022. doi: https://doi.org/10.1145/3519939.3523433.

[94] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. symqv: Automated symbolic verification of quantum programs. In Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker, editors, *Formal Methods*, pages 181–198, Cham, 2023. Springer International Publishing. doi: https://doi.org/10.1007/978-3-031-27481-7_12.

[95] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997. doi: 10.1137/S0097539796300921. URL https://doi.org/10.1137/S0097539796300921.

[96] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992. doi: 10.1098/rspa.1992.0167.

[97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997. ISSN 0097-5397. doi: 10.1137/S0097539795293172. URL https://doi.org/10.1137/S0097539795293172.

[98] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917855. doi: 10.1145/237814.237866. URL https://doi.org/10.1145/237814.237866.

[99] Prakash Murali, David C McKay, Margaret Martonosi, and Ali Javadi-Abhari. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1016, 2020.

[100] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, page 309–328, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660231. URL https://doi.org/10.1145/2660193.2660231.

[101] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. *SIGPLAN Not.*, 48(10):33–52, October 2013. ISSN 0362-1340. doi: 10.1145/2544173.2509546. URL https://doi.org/10.1145/2544173.2509546.

[102] Shih-Han Hung, Kesha Hietala, Shaopeng Zhu, Mingsheng Ying, Michael Hicks, and Xiaodi Wu. Quantitative robustness analysis of quantum programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[103] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3): 032324, 2012.

[104] Adam Holmes, Yongshan Ding, Ali Javadi-Abhari, Diana Franklin, Margaret Martonosi, and Frederic T Chong. Resource optimized quantum architectures for surface code implementations of magic-state distillation. *Microprocessors and Microsystems*, 67:56–70, 2019.

[105] Runzhou Tao, Yunong Shi, Jianan Yao, John Hui, Frederic T. Chong, and Ronghui Gu. Gleipnir: Toward practical error analysis for quantum programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 48–64, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454029. URL https://doi.org/10.1145/3453483.3454029.

[106] Jean-Christophe Filliâtre and Andrei Paskevich. Why3: Where programs meet provers. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, page 125–128, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 9783642370359. doi: 10.1007/978-3-642-37036-6_8. URL https://doi.org/10.1007/978-3-642-37036-6_8.

[107] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Vs3: Smt solvers for program verification. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 702–708, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-02658-4.

[108] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *POPL'10, January 17-23, 2010, Madrid, Spain*, January 2010. URL https://www.microsoft.com/en-us/research/publication/program-verification-program-synthesis/.

[109] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *PLDI'11, June 4-8, 2011, San Jose, California, USA*, June 2011. URL https://www.microsoft.com/en-us/research/publication/path-based-inductive-synthesis-program-inversion/.

[110] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI'11, June 4-8, 2011, San Jose, California, USA*, June 2011. URL https://www.microsoft.com/en-us/research/publication/synthesis-loop-free-programs/.

[111] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices*, 49(6):530–541, 2014.

[112] Prakash Murali, Ali Javadi-Abhari, Frederic Chong, and Margaret Martonosi. Formal constraint-based compilation for noisy intermediate-scale quantum systems. *Microprocessors and Microsystems*, 66, 02 2019. doi: 10.1016/j.micpro.2019.02.005.

[113] Andrew Cross. The ibm q experience and qiskit open-source quantum computing software. In *APS March Meeting Abstracts*, volume 2018, pages L58–003, 2018.

[114] Dave Bacon. Operator quantum error-correcting subsystems for self-correcting quantum memories. *Physical Review A*, 73(1):012340, 2006.

[115] Raymond Laflamme, Cesar Miquel, Juan Pablo Paz, and Wojciech Hubert Zurek. Perfect quantum error correcting code. *Physical Review Letters*, 77(1):198, 1996.

[116] David P DiVincenzo and Panos Aliferis. Effective fault-tolerant quantum computation with slow measurements. *Physical review letters*, 98(2):020501, 2007.

[117] Andrew M Steane. Active stabilization, quantum computation, and quantum state synthesis. *Physical Review Letters*, 78(11):2252, 1997.

[118] Andrew M Steane. Fast fault-tolerant filtering of quantum codewords. *arXiv preprint quant-ph/0202036*, 2002.

[119] Emanuel Knill. Scalable quantum computing in the presence of large detected-error rates. *Physical Review A*, 71(4):042322, 2005.

[120] Theodore J Yoder and Isaac H Kim. The surface code with a twist. *Quantum*, 1:2, 2017.

[121] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[122] David Beazley. Ply (python lex-yacc). https://www.dabeaz.com/ply/, 2018.

[123] Leonardo De Moura and Nikolaj Bjorner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[124] Giulia Meuli, Mathias Soeken, Martin Roetteler, and Thomas Häner. Automatic accuracy management of quantum programs via (near-) symbolic resource estimation. *arXiv preprint arXiv:2003.08408*, 2020.

[125] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on computing*, 26(5):1411–1473, 1997.

[126] Robin Harper, Steven T Flammia, and Joel J Wallman. Efficient learning of quantum noise. *Nature Physics*, 16(12):1184–1188, 2020.

[127] Mohan Sarovar, Timothy Proctor, Kenneth Rudinger, Kevin Young, Erik Nielsen, and Robin Blume-Kohout. Detecting crosstalk errors in quantum information processors. *Quantum*, 4: 321, 2020.

[128] Siyuan Niu and Aida Todri-Sanial. Analyzing crosstalk error in the nisq era. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 428–430. IEEE, 2021.

[129] Yongshan Ding, Pranav Gokhale, Sophia Fuhui Lin, Richard Rines, Thomas Propson, and Frederic T Chong. Systematic crosstalk mitigation for superconducting qubits via frequency-aware compilation. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 201–214. IEEE, 2020.

[130] Jay M Gambetta, Jerry M Chow, and Matthias Steffen. Building logical qubits in a superconducting quantum computing system. *npj quantum information*, 3(1):2, 2017.

[131] Pranav S Mundada, Aaron Barbosa, Smarak Maity, Yulun Wang, Thomas Merkh, TM Stace, Felicity Nielson, Andre RR Carvalho, Michael Hush, Michael J Biercuk, et al. Experimental benchmarking of an automated deterministic error-suppression workflow for quantum algorithms. *Physical Review Applied*, 20(2):024034, 2023.

[132] Nathalie P De Leon, Kohei M Itoh, Dohun Kim, Karan K Mehta, Tracy E Northup, Hanhee Paik, BS Palmer, Nitin Samarth, Sorawis Sangtawesin, and David W Steuerman. Materials challenges and opportunities for quantum computing hardware. *Science*, 372 (6539):eabb2823, 2021.

[133] Antti P Vepsäläinen, Amir H Karamlou, John L Orrell, Akshunna S Dogra, Ben Loer, Francisca Vasconcelos, David K Kim, Alexander J Melville, Bethany M Niedzielski, Jonilyn L Yoder, et al. Impact of ionizing radiation on superconducting qubit coherence. *Nature*, 584(7822):551–556, 2020.

[134] Ted Thorbeck, Andrew Eddins, Isaac Lauer, Douglas T McClure, and Malcolm Carroll. Two-level-system dynamics in a superconducting qubit due to background ionizing radiation. *PRX Quantum*, 4(2):020356, 2023.

[135] Yukai Wu, Sheng-Tao Wang, and L-M Duan. Noise analysis for high-fidelity quantum entangling gates in an anharmonic linear paul trap. *Physical Review A*, 97(6):062325, 2018.

[136] Matthew J Boguslawski, Zachary J Wall, Samuel R Vizvary, Isam Daniel Moore, Michael Bareian, David TC Allcock, David J Wineland, Eric R Hudson, and Wesley C Campbell. Raman scattering errors in stimulated-raman-induced logic gates in ba+ 133. *Physical Review Letters*, 131(6):063001, 2023.

[137] Andrew Addison Houck, JA Schreier, BR Johnson, JM Chow, Jens Koch, JM Gambetta, DI Schuster, L Frunzio, MH Devoret, SM Girvin, et al. Controlling the spontaneous emission of a superconducting transmon qubit. *Physical review letters*, 101(8):080502, 2008.

[138] Dorit Aharonov and Michael Ben-Or. Fault-tolerant quantum computation with constant error. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 176–188, 1997.

[139] Panos Aliferis, Daniel Gottesman, and John Preskill. Quantum accuracy threshold for concatenated distance-3 codes. *arXiv preprint quant-ph/0504218*, 2005.

[140] A Yu Kitaev. Fault-tolerant quantum computation by anyons. *Annals of physics*, 303(1): 2–30, 2003.

[141] Sergey B Bravyi and A Yu Kitaev. Quantum codes on a lattice with boundary. *arXiv preprint quant-ph/9811052*, 1998.

[142] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. Topological quantum memory. *Journal of Mathematical Physics*, 43(9):4452–4505, 2002.

[143] Austin G Fowler, Ashley M Stephens, and Peter Groszkowski. High-threshold universal quantum computation on the surface code. *Physical Review A—Atomic, Molecular, and Optical Physics*, 80(5):052312, 2009.

[144] Hsiang-Ku Lin and Leonid P Pryadko. Quantum two-block group algebra codes. *Physical Review A*, 109(2):022407, 2024.

[145] Vincent Paul Su, ChunJun Cao, Hong-Ye Hu, Yariv Yanay, Charles Tahan, and Brian Swingle. Discovery of optimal quantum error correcting codes via reinforcement learning. *arXiv preprint arXiv:2305.06378*, 2023.

[146] Artur d'Avila Garcez and Luis C Lamb. Neurosymbolic ai: The 3rd wave. *arXiv e-prints*, pages arXiv–2012, 2020.

[147] Shushan Arakelyan, Anna Hakhverdyan, Miltiadis Allamanis, Luis Garcia, Christophe Hauser, and Xiang Ren. Ns3: Neuro-symbolic semantic code search. *Advances in Neural Information Processing Systems*, 35:10476–10491, 2022.

[148] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[149] Jiankai Sun, Hao Sun, Tian Han, and Bolei Zhou. Neuro-symbolic program search for autonomous driving decision module design. In *Conference on Robot Learning*, pages 21–30. PMLR, 2021.

[150] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th international conference on software engineering*, pages 60–70, 2018.

[151] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. *arXiv preprint arXiv:1904.12584*, 2019.

[152] Leonardo Amado, Ramon Fraga Pereira, and Felipe Meneguzzi. Robust neuro-symbolic goal and plan recognition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 11937–11944, 2023.

[153] Reuben Feinman and Brenden M Lake. Learning task-general representations with generative neuro-symbolic modeling. *arXiv preprint arXiv:2006.14448*, 2020.

[154] Minkyu Choi, Harsh Goel, Mohammad Omama, Yunhao Yang, Sahil Shah, and Sandeep Chinchali. Neuro-symbolic video search. *arXiv preprint arXiv:2403.11021*, 2024.

[155] Thomas Eiter, Nelson Higuera, Johannes Oetsch, and Michael Pritz. A neuro-symbolic asp pipeline for visual question answering. *Theory and Practice of Logic Programming*, 22(5): 739–754, 2022.

[156] Claire Glanois, Zhaohui Jiang, Xuening Feng, Paul Weng, Matthieu Zimmer, Dong Li, Wulong Liu, and Jianye Hao. Neuro-symbolic hierarchical rule induction. In *International Conference on Machine Learning*, pages 7583–7615. PMLR, 2022.

[157] Joshua Grochow. New applications of the polynomial method: the cap set conjecture and beyond. *Bulletin of the American Mathematical Society*, 56(1):29–64, 2019.

[158] Steven S Skiena. *The algorithm design manual*, volume 2. Springer, 1998.

[159] J Alison Noble. Finding corners. *Image and vision computing*, 6(2):121–128, 1988.

[160] Noga Alon and Eyal Lubetzky. The shannon capacity of a graph and the independence numbers of its powers. *IEEE Transactions on Information Theory*, 52(5):2172–2176, 2006.

[161] Pavel Panteleev and Gleb Kalachev. Quantum ldpc codes with almost linear minimum distance. *IEEE Transactions on Information Theory*, 68(1):213–229, 2021.

[162] Pavel Panteleev and Gleb Kalachev. Degenerate quantum ldpc codes with good finite length performance. *Quantum*, 5:585, 2021.

[163] GAP. GAP – Groups, Algorithms, and Programming, Version 4.14dev. https://www.gap-system.org, this year.

[164] Leonid P Pryadko, Vadim A Shabashov, and Valerii K Kozin. Qdistrnd: A gap package for computing the distance of quantum error-correcting codes. *arXiv preprint arXiv:2308.15140*, 2023.

[165] Nicolas Delfosse and Adam Paetznick. Spacetime codes of clifford circuits. *arXiv preprint arXiv:2304.05943*, 2023.

[166] Matt McEwen, Dave Bacon, and Craig Gidney. Relaxing hardware requirements for surface code circuits using time-dynamics. *Quantum*, 7:1172, 2023.

[167] Oscar Higgott, Thomas C Bohdanowicz, Aleksander Kubica, Steven T Flammia, and Earl T Campbell. Improved decoding of circuit noise and fragile boundaries of tailored surface codes. *Physical Review X*, 13(3):031007, 2023.

[168] György P Gehér, Ophelia Crawford, and Earl T Campbell. Tangling schedules eases hardware connectivity requirements for quantum error correction. *PRX Quantum*, 5(1): 010348, 2024.

[169] Joschka Roffe, David R White, Simon Burton, and Earl Campbell. Decoding across the quantum low-density parity-check code landscape. *Physical Review Research*, 2(4):043423, 2020.

[170] ChunJun Cao and Brad Lackey. Quantum lego: Building quantum error correction codes from tensor networks. *PRX Quantum*, 3(2):020332, 2022.

[171] Chuangqiang Hu, Shudi Yang, and Stephen S-T Yau. Weight enumerators for nonbinary asymmetric quantum codes and their applications. *Advances in Applied Mathematics*, 121: 102085, 2020.

[172] Magma computational algebra system. https://magma.maths.usyd.edu.au/magma/handbo

[173] Caroline Mauron, Terry Farrelly, and Thomas M Stace. Optimization of tensor network codes with reinforcement learning. *arXiv preprint arXiv:2305.11470*, 2023.

[174] Jan Olle, Remmy Zen, Matteo Puviani, and Florian Marquardt. Simultaneous discovery of quantum error correction codes and encoders with a noise-aware reinforcement learning agent. *arXiv preprint arXiv:2311.04750*, 2023.

[175] Sergey Bravyi, Martin Suchara, and Alexander Vargo. Efficient algorithms for maximum likelihood decoding in the surface code. *Physical Review A*, 90(3):032326, 2014.