

# DeQompile: quantum circuit decompilation using genetic programming for explainable quantum architecture search

Shubing Xie<sup>1,2</sup>, Aritra Sarkar<sup>2,3</sup>, and Sebastian Feld<sup>2,3</sup>

<sup>1</sup>*Instituut-Lorentz, Leiden University, The Netherlands*

<sup>2</sup>*Quantum Machine Learning research group, Quantum Computing division, QuTech, The Netherlands*

<sup>3</sup>*Department of Quantum & Computer Engineering, Delft University of Technology, The Netherlands*  
 [s.feld@tudelft.nl](mailto:s.feld@tudelft.nl)

## Abstract

Demonstrating quantum advantage using conventional quantum algorithms remains challenging on current noisy gate-based quantum computers. Automated quantum circuit synthesis via quantum machine learning has emerged as a promising solution, employing trainable parametric quantum circuits to alleviate this. The circuit ansatz in these solutions is often designed through reinforcement learning-based quantum architecture search when the domain knowledge of the problem and hardware are not effective. However, the interpretability of these synthesized circuits remains a significant bottleneck, limiting their scalability and applicability across diverse problem domains.

This work addresses the challenge of explainability in quantum architecture search (QAS) by introducing a novel genetic programming-based decompiler framework for reverse-engineering high-level quantum algorithms from low-level circuit representations. The proposed approach, implemented in the open-source tool DeQompile, employs program synthesis techniques, including symbolic regression and abstract syntax tree manipulation, to distill interpretable Qiskit algorithms from quantum assembly language. Validation of benchmark algorithms demonstrates the efficacy of our tool. By integrating the decompiler with online learning frameworks, this research potentiates explainable QAS by fostering the development of generalizable and provable quantum algorithms.

## 1 Introduction

Quantum computing (QC) represents a paradigm shift in information processing, utilizing the principles of quantum mechanics to solve problems that are intractable for classical computers. In recent years, significant advances have been made in engineering gate-based quantum computing, which relies on quantum bits (qubits) and quantum gates to manipulate quantum states and process data in a superposition of states [1]. The current state of the art in quantum computing is exemplified by breakthroughs from quantum processor manufacturers in qubits with better fidelity, scalability, real-time control, and error-correction [2, 3, 4]. Conventional quantum algorithms like integer factorization [5] and search [6] ensure provable resource complexity advantages over their classical counterparts. However, demonstrating the classical-to-quantum advantage crossover for applications via these algorithms requires considerable improvements in the fidelity and multiplicity of qubits. The current noisy intermediate-scale quantum computing (NISQ) [7, 8, 9, 10] devices have motivated the automated synthesis of quantum circuits while accounting for the device constraints [11, 12].

The quantum circuit synthesis methods are often referred to using the broader umbrella term of quantum machine learning (QML). Similar to classical neural networks, these methods typically consider a parametric quantum circuit (PQC) [13, 14]. The structure/ansatz of the circuit can either be based on the problem structure or the hardware constraints or automatically constructed via quantum architecture search (QAS) [15, 16, 17]. The parameters, in turn, are iteratively adjusted by the variational principle mediated by a classical optimizer in the loop. QAS is typically mediated by neural-network-based reinforcement learning (RL-QAS) [18, 19] or population-based evolutionary algorithms [20, 21].

QAS-based PQC shares both the strengths and limitations of classical machine learning (ML) based neural architecture search (NAS). A major challenge in deep neural networks has been the issue of explainability [22] of both the architecture and the learned model. Various model explainability techniques have been developed in ML and subsequently in QML [23, 24, 25, 26, 27]. Interpretable NAS techniques [28, 29] have also been employed for post-doc understanding of the architecture based on empirical performance. Similarly to NAS, while QAS-based solutions have been shown to perform against hardware constraints [30], the trained ansatz is not human-interpretable. This restricts the generalizability of these solutions to larger problem sizes and a deeper understanding of the underlying governing symmetries that led the QAS to converge on the solution.

Recently, explainable QAS has been attempted variously via quantum information theoretic approach [31], interpretable learning models [32], and online learning of a gadget library (GRL-QAS) [33]. In this work, we develop a complementary approach toward explaining a family of quantum circuits via a genetic programming-based decompiler. We demonstrate our framework using examples from conventional quantum algorithms and suggest their integration within a GRL-QAS framework to augment their capability beyond simple gadgets.

Program synthesis provides a promising avenue for addressing the explainability problem in quantum algorithm design automation [34]. It aims to distill a family of circuits into a high-level, interpretable algorithm, often by using methods like genetic programming or neural networks. Symbolic regression, for example, can be used to find mathematical expressions that describe the behavior of quantum circuits [35]. For more holistic and expressive descriptions, high-level program synthesis and concept learning [36, 37] can be employed. Decompilation closely resembles inductive inference in the human brain, such as when we deduce the next item in a sequence by identifying underlying patterns. In the context of quantum computing, such methods would facilitate the understanding and generalization of quantum algorithms across different quantum hardware and related problem instances. For quantum circuits, if the circuits are known at a limited scale and we wish to extend the corresponding design, we must identify the shared structures and recognize the patterns of these circuits. Inspired by quantum techniques for Solomonoff’s induction [38, 39, 40, 41], in this research, the rules or patterns of these circuits are represented by finding the underlying code that can generate them. With that background, in this article, we address this research question of reverse-engineering high-level quantum algorithms (in Qiskit) from their low-level representations (QASM) using genetic programming (GP) on the abstract syntax tree (AST) representation. We develop a framework to tackle this question, including a method to initialize syntactically valid Qiskit ASTs, a multi-objective fitness function, the genetic operators, and strategies to improve the convergence of the GP. We validate the decompiler’s performance on common variational ansatz and quantum algorithms like GHZ state preparation, quantum Fourier Transform, and quantum phase estimation. Further, we investigate the increased difficulty in decompiling circuits that have been transpiled to the constraints of underlying quantum hardware. The software implementation of our technique, called DeQompile, is available as open-source software for further research and development.

## 2 Background

This section introduces the necessary concepts of decompilation, genetic programming, and abstract syntax tree representation.

### 2.1 Decompilation

Decompilation is the process of translating compiled code (e.g., low-level assembly code) back into a more readable form of source code (e.g., a high-level language) or a close approximation of it without having access to the original source code. This process is crucial for understanding, analyzing, and optimizing software systems, especially when the original source code is obfuscated or unavailable. In classical computing, decompilation has numerous applications, including software reverse engineering, explainable AI (XAI), and programming by example. For instance, in software reverse engineering, decompiling a program’s binary allows for vulnerability assessment, malware analysis, and system optimization. Decompilation plays an important role in programming by example, such as Excel’s Flash Fill, where it is used to automate string processing tasks based on input-output examples, significantly reducing the need for manual coding [42, 43, 44]. In the realm of XAI, decompiling machine learning models [45, 46] help enhance interpretability and transparency, which is vital for ensuring the trustworthiness of AI systems.

Decompilation, however, is inherently a difficult task due to several challenges. Fundamentally, the undecidability of the halting problem implies that it is not always possible to determine whether a synthesized program will terminate with the intended effect or run indefinitely. Additionally, the program space is non-differentiable, which prevents the direct application of traditional gradient-based optimization methods. The semantic variable names are typically absent in low-level codes, making it difficult to understand the decompiled code. Despite these difficulties, various techniques have been developed to address the challenges of decompilation. For instance, genetic programming [47], a technique rooted in evolutionary algorithms, has been popularly used for program synthesis and software reverse engineering. Neural networks [48] have also been applied to decompilation tasks, where deep learning models can learn mappings between machine code and high-level languages, thus automating and enhancing the decompilation process.

Despite the similarities with classical decompilation, quantum circuit decompilation has not yet been explored for quantum algorithm reverse engineering from learned quantum circuits. Quantum decompilation faces unique challenges that differentiate it from classical decompilation. Quantum phenomena like superposition, entanglement, non-stabilizer magic, and interference do not have phenomenological analogs in human daily experience (or in propositional logic), thus requiring a certain degree of commitment to mathematical models

in their interpretability. It is also preferable to evaluate the decompiler purely syntactically instead of testing by executing the decompiler artifact to circumvent the exponential computational cost of classical simulation of quantum programs. Besides this, quantum transformations have an added continuous yet non-measurable degree of freedom in their global phase, which makes it difficult to generalize across circuits where the pattern gets obscured by this phase factor.

Decompiling quantum circuits could uncover the underlying principles of quantum architecture and algorithm design, ultimately enabling the discovery of more interpretable, scalable, and provable quantum algorithms. This would allow us to transition from QAS-based PQC solutions in the NISQ era to conventional quantum algorithms in the fault-tolerant quantum computing (FTQC) era.

## 2.2 Genetic programming

The basic idea behind evolutionary techniques such as genetic algorithms (GA), genetic programming (GP), and genetic expression programming (GEP) is to evolve solutions to problems by iteratively modifying a population of candidate solutions. These candidates are typically represented as tree structures, which can be subjected to operations like selection, crossover, and mutation to explore the solution space. GP [47] allows computer programs to evolve to solve a variety of complex problems, including symbolic regression, classification, and optimization. In GP, the nodes of the tree represent functions or operations, and the leaves represent variables or constants. The fitness function evaluates the performance of the individuals (programs) and guides the evolution process. Quantum genetic programming (QGP) [49, 50] extends the concept of classical GP into the realm of quantum program synthesis.

The steps of initialization, evaluation, selection, crossover, and mutation characterize the GP methodology. The following algorithm outlines the baseline genetic programming process.

---

**Algorithm 1** Baseline Genetic Programming Algorithm

---

- 1: Initialize population  $P$  with random individuals (programs in chosen representation)
  - 2: **for** each generation  $g$  **do**
  - 3:     Evaluate the fitness of each program in  $P$  based on a fitness function
  - 4:     Select the fittest individuals from  $P$  to form a mating pool
  - 5:     Apply crossover to pairs of individuals in the mating pool to create offspring
  - 6:     Apply mutation to the offspring with a certain probability
  - 7:     Replace the least fit individuals in  $P$  with the new offspring
  - 8: **end for**
  - 9: Return the best individual from the final population
- 

A critical challenge in both classical and quantum GP is the encoding of the candidate solutions to allow efficient manipulation and evolution during the search process. This is where abstract syntax trees (AST) come into play and will be discussed in more detail subsequently.

## 2.3 Abstract syntax tree

Abstract syntax trees are a hierarchical, tree-like representation of the syntactic structure of source code or expressions, abstracted from the details of the underlying syntax. Unlike the raw code or textual representation, the AST focuses on the logical structure and relationships within the code, omitting unnecessary syntactic details like punctuation and formatting. Each node in the tree corresponds to a construct or operation in the source code, such as a variable, function, operator, or control flow structure.

ASTs are widely used in compilers, interpreters, and program analysis tools for efficient parsing, optimization, and transformation of code. AST's hierarchical and structured representation of quantum circuits or programs makes them ideal for encoding in evolutionary algorithms. By using ASTs, we can effectively manipulate the structure of quantum programs and optimize them during the evolutionary process, facilitating the exploration of the quantum program space.

## 3 Quantum circuit decompilation

In this section, we explain the decompilation process for QASM circuits. A population of Qiskit programs is initialized in the AST representation. The fitness for each corresponding Qiskit code for each AST is evaluated by comparing the generated QASM from the code and the training corpus (i.e., the list of QASM to be decompiled). The GP guides the evolution towards fitter individuals that can decompile the corpus.

### 3.1 Problem formulation

We study two primary objectives. Firstly, we select a quantum algorithm  $A$  of choice, such as simple ansatzes, GHZ state preparation, quantum Fourier transform, quantum phase estimation, etc. Quantum circuits for  $A$  are generated for different problem sizes, ranging from 2 to 30 qubits, using a Python program  $P_A(n)$  using the Qiskit package [51]. The resulting circuits,  $C_A^2, C_A^3, \dots, C_A^{30}$ , is represented in OpenQASM [52]. This set of circuits  $C_A^n$  serves as the training data set. Given this set as input, the designed decompiler evolves an optimal program  $P'_A(n)$  that closely approximates  $P_A(n)$ . The closeness metrics can be either the process distance between the synthesized unitaries or the difference between the generated QASMs, as discussed later.  $P'_A(n)$  can then be used to generate  $C_A'^{31}, C_A'^{32}, \dots$ . The data set can also be divided into training and test sets to validate the generalization capability.

Secondly, we explore the limits of the tool through empirical analysis of decompiling a highly optimized code with a lesser algorithmic structure. We take circuits  $C_A^n$  and translate them into the corresponding unitaries  $U_A^n$ . These unitaries are decomposed using Qiskit's transpilation into the native gate set of a quantum processing unit (QPU), such as IBM's, to obtain circuits  $C'^n_A$ . Given this set of circuits  $C'^n_A$ , the decompiler will infer  $P''_A(n)$ . We show our experiments agree with the expectation that  $P''_A(n)$  is less explainable and more abstract than  $P'_A(n)$ , thus making it harder to generalize from optimized circuits.

The software architecture of DeQompile is shown in the Figure 1.

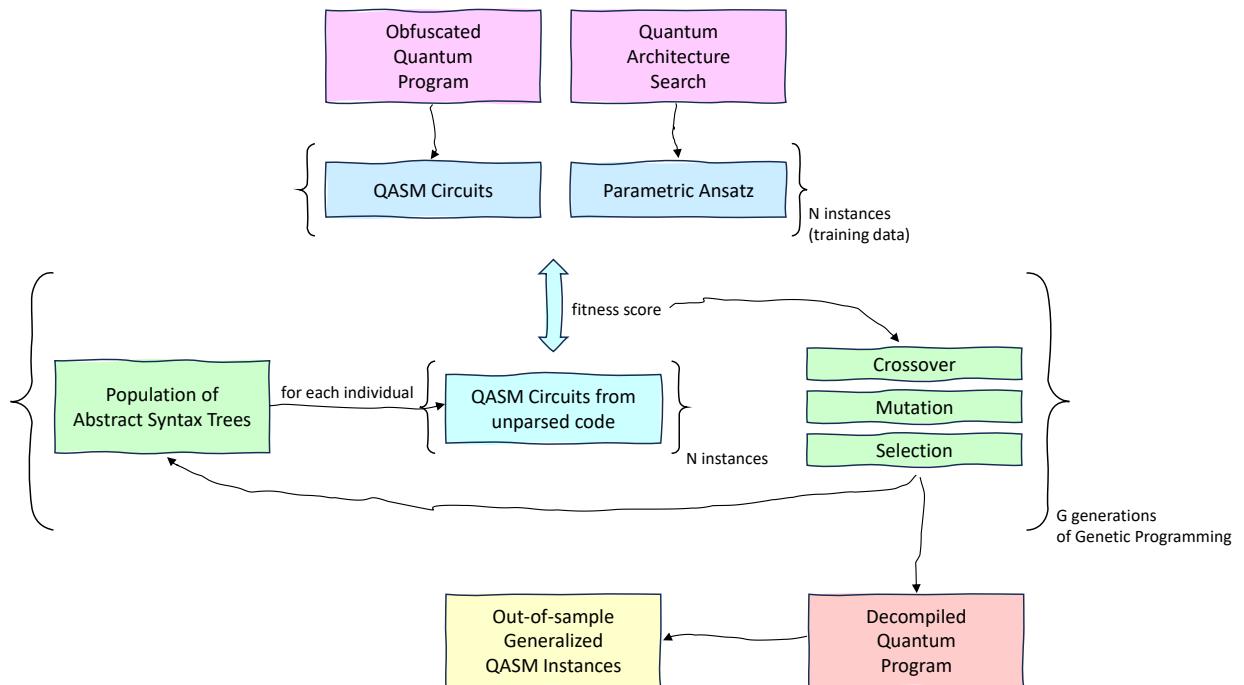


Figure 1: Software architecture of DeQompile

### 3.2 AST representation

In this project, we use the `ast` module of Python to generate a population of individuals for the GP. The `parse` and `unparse` functions in the `ast` package allow the conversion between the AST and code string representation. Each AST in the population represents a Python function that takes as an argument a circuit size  $n$  and returns a Qiskit quantum circuit object of  $n$  qubits. The operations in the quantum circuit are (potentially) conditioned on  $n$  as a proxy for the problem instance size. Each node in the AST represents a fundamental construct in the source code, such as quantum gate operations, assignments, and control flow.

The nodes of the AST include:

- Module: Represents the entire quantum program, containing all quantum operations and gates.
- FunctionDef: Represents a function definition, such as a quantum circuit initializer.
- Assign and Constant: Represent assignments, including qubit initializations and gate parameters.
- Expr and BinOp: Represent mathematical expressions and quantum gate operations.
- Control Flow: Represent loops (`for` loops), crucial for parameterized quantum operations.

An example of a simple AST in Qiskit is as follows. Consider the function `rx_c` that initializes a quantum circuit, and applies a series of `rx` rotations on each of the  $i \in [1, n]$  qubits with angle scaling by a factor of  $\pi/i$  for  $i \in [1, n]$ , and returns the constructed circuit.

---

```

1 def rx_c(n):
2     qc = QuantumCircuit(n)
3     angle = pi
4     for i in range(n):
5         qc.rx(angle, i)
6         angle /= 2
7     return qc

```

---

Listing 1: Qiskit example for a rx\_c module

The corresponding AST structure includes:

- Module: The root node representing the quantum program.
- FunctionDef: Defines the function `rx_c` with arguments.
  - Assign: Assignment and storing of the circuit object `qc = QuantumCircuit(n)`.
  - Assign: Assignment of the initial angle `angle = math.pi`.
  - For: A loop iterating over qubits.
    - AugAssign: Augmented assignment `angle /= 2`.
    - Call: Expression `qc.rx(angle, i)`.
  - Load: The return statement `return qc`.

Figure 2 shows the AST for the `rx_c` function, illustrating how the AST captures the quantum circuit structure and operations.

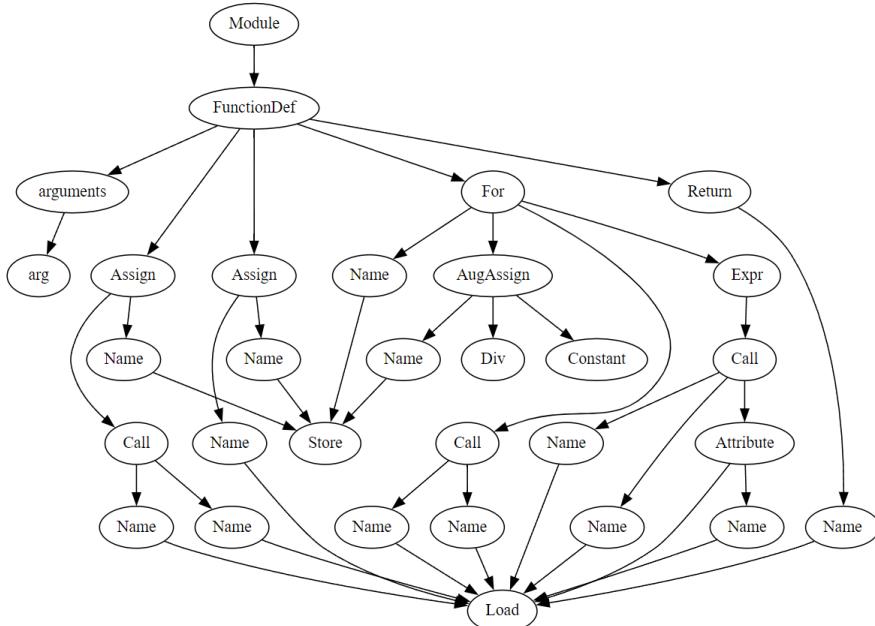


Figure 2: AST structure for a simple Qiskit function

### 3.3 Initialization of the population

By leveraging ASTs, we programmatically generate the initial population for evolution. In each generation, a preset percentage of the individuals are also culled and regenerated. Below are the key functions that contribute to this process.

#### 3.3.1 Qubit index expressions

The `random_expr` function generates random expressions for qubit indices using arithmetic operations, modulus, and simple variables. Parameters include:

- `depth`: Number of loop variables.
- `max_expr_operators`: Number of binary operations in expression.
- `var_depth`: Number of additional variables.

For example,

- `random_expr(0, 1, 2)` could be a simple operation like  $n - n - n$ , where  $n$  is a variable or index.

- `random_expr(1, 2, 3)` might generate a more complex expression such as  $i0 - n + 3 + 2$ , involving loop variables ( $i0$ ), constants (3, 2), and operations.

### 3.3.2 Loop structures

Loops are essential for repeated quantum gate operations. The `loop_index` function helps generate dynamic loop structures for complex quantum algorithms enabling the iterative application of quantum gates.

Below is an example illustrating nested loops in a quantum circuit:

---

```

1 for i0 in range(n):
2     for i1 in range(abs(i0 - n)):
3         for i2 in range(abs(i0 + i1 + i1 + 1)):
4             qc.crx(pi * (1 / (2 ** (i1 + n) + i1)), (n + 1) % n)

```

---

Listing 2: Nested loops in quantum circuit generation

- First loop: The index `i0` iterates from 0 to `n`, setting the basic framework for subsequent nested loops.
- Second loop: Dependence on `i0` and `i1` ranges dynamically, increasing the complexity of operations performed in this layer.
- Third loop: The deepest layer uses both `i0` and `i1` in determining its range, illustrating an advanced level of dependency and complexity in a loop structure.

### 3.3.3 Angle expressions for rotation gates

The `random_phase_expr` function generates phase expressions for gates such as `rx`, `ry`, and `rz`. This function creates an expression of the form:

$$expr_{phase} = (\pi \cdot \frac{1}{2^a + b + c}) \quad (1)$$

where  $a$  is the expression related to the number of qubits  $n$ ,  $b$  is the expression of the loop index  $i_j$ ,  $0 < j < d$ , and  $c$  are random numbers from a Gaussian distribution

$$X \sim \mathcal{N}(\mu, \sigma^2). \mu = 0, \sigma = 1$$

The function's only input is the depth of the current loop location, which is used to decide which symbols can be included in the expression. As an example, `random_phase_expr(2)` can generate the expression `pi * (1 / (2 ** (n + 0 + n - 0) + (i0 + 0 + 0 - n + 0)))`.

### 3.3.4 Quantum gate calls

The `generate_gate_call` function constructs calls to quantum gates, accommodating single, multi-qubit, and rotational gates. It uses randomly generated expressions for qubit indices and phases to define the gates' applications. Some examples of the function `generate_gate_call(depth: Any, gate: Any)` are given as Table 1. It takes the current loop depth and a specific gate type as inputs:

Function call	Generated quantum gate operation
<code>generate_gate_call(2, 'h')</code>	<code>qc.h((i1 - 0 - n) % n)</code>
<code>generate_gate_call(1, 'rx')</code>	<code>qc.rx(pi * (1 / (2 ** (i0 + 0 - n) + (i0 - n + n + 0))), (n - 0) % n)</code>
<code>generate_gate_call(1, 'cx')</code>	<code>qc.cx((n - 0 + n) % n, (i0 + n - 0) % n)</code>
<code>generate_gate_call(1, 'cp')</code>	<code>qc.cp(-(pi * (1 / (2 ** (n - 0 - n) + (n - n + n + 2)))), (i0 + 0) % n, (n - 0 - n) % n)</code>

Table 1: Examples of generating quantum gate calls using `generate_gate_call` function.

The `random_qiskit_ast_generator` function in DeQompile assembles all these components together, constructing a complete quantum circuit. It takes as arguments a list of allowed operations, the maximum number of nodes, and the maximum cyclometric complexity (i.e., the levels of loop nesting). The function iterates through AST nodes, adding gate operations and returning the final circuit. This approach allows for the generation of random quantum circuits with arbitrary complexity.

## 3.4 Evolution operations

The main goal is to evolve Qiskit programs (in AST representation) that can reproduce the set of QASM circuits based on the input while optimizing for resources like gate counts, node counts, etc. The GP workflow in each generation of DeQompile after the population initialization is described in this section.

### 3.4.1 Fitness evaluation

The fitness of each candidate quantum circuit is evaluated by comparing it to the target circuit using a set of custom evaluation metrics inspired by natural language processing (NLP). Similarity can broadly be classified as semantic or syntactic. Semantic similarity can be estimated by the process fidelity between the unitary of the circuit generated via the induced program and that of the QASM target. While semantic similarity allows generalization over syntactic differences (e.g., global phase), constructing the unitary scales exponentially with the problem size becoming restrictive beyond a few small samples. Instead, we introduce methods that compare the syntactic structure and operational similarities between circuits. We use a combination of three metrics to measure the fitness of each QASM pair.

- Gate sequence similarity: Compares the sequences of quantum gates in the candidate and target circuits using the Levenshtein distance [53]. This metric calculates the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one sequence to the other. The similarity score  $S_{\text{seq}}$  is defined as:

$$S_{\text{seq}} = 1 - \sqrt{\frac{D(A, B)}{\max(|A|, |B|)}}$$

where  $D(A, B)$  is the Levenshtein distance between the sequences  $A$  and  $B$ , and  $|A|$  and  $|B|$  are their lengths. The square root emphasizes differences in larger sequences.

- Gate frequency similarity: Assesses how similar the usage frequency of each gate type is between the candidate and target circuits. By creating frequency vectors for each circuit and calculating the cosine similarity between them, we obtain the score  $S_{\text{freq}}$ :

$$S_{\text{freq}} = \frac{\mathbf{f}_1 \cdot \mathbf{f}_2}{\|\mathbf{f}_1\| \|\mathbf{f}_2\|}$$

where  $\mathbf{f}_1$  and  $\mathbf{f}_2$  are the frequency vectors of the candidate and target circuits, respectively.

- Longest common subsequence: By finding the longest common subsequence (LCS) of lines in the QASM representations of the candidate and target circuits, we evaluate the structural similarity. The LCS score  $S_{\text{LCS}}$  is calculated as:

$$S_{\text{LCS}} = \frac{\text{Length of LCS}}{\text{Total lines in target QASM}}$$

A dynamic programming approach is used to efficiently compute the LCS.

The overall fitness score  $S_{\text{total}}$  for a candidate circuit is the geometric mean of the individual similarity scores. This combined score ensures that a low similarity in any one metric significantly affects overall fitness, promoting well-rounded candidate solutions. The total fitness of the individual includes the average  $S_{\text{total}}$  for all problem sizes, and an additional quantum description complexity [54] score  $S_{\text{KC}}$  as a parsimony pressure [55] estimated by counting the number of nodes in the AST representation of the individual:

$$S_{\text{total}} = S_{\text{KC}} + \sum_n (S_{\text{seq}} \times S_{\text{freq}} \times S_{\text{LCS}})^{1/3}$$

These evaluation metrics focus on the structural and operational similarities between circuits, guiding the evolutionary process toward candidates that closely resemble the target circuit regarding gate sequence, usage, and overall structure with an inductive bias toward succinct representation.

### 3.4.2 Genetic operations

Genetic programming employs genetic operations such as selection, crossover, and mutation to evolve candidate Qiskit codes over successive generations.

Selection involves choosing individuals with better fitness scores for reproduction, allowing the propagation of superior solutions. DeQompile implements popular selection algorithms like roulette wheel, tournament, rank-based, random, and weighted roulette wheel. We refer the readers to [56] for a pedantic coverage of these methods.

Crossover combines parts of two parents' codes to explore new potential solutions. An example is shown in Figure 3. Crossover is always done at the outmost indent level (i.e., not within a loop) to preserve the syntactic validity of the children.

The mutation operator alters parts of a single code to introduce diversity and avoid converging to suboptimal solutions. DeQompile selects a node in the AST to mutate. It allows either inserting a new AST by extending the body of the node or modifying the body of the node to the new AST. Deletion is not allowed to maintain syntactic validity.

---

```

1 # Parent 1
2 qc.h((n - 1) % n)
3 ----- split
4 for i0 in range(n):
5     qc.h((n - 1 - i0) % n)
6     qc.h((i0 - 1) % n)

```

---

```

1 # Child 1
2 qc.h((n - 0) % n)
3 **** Crossover
4 for i0 in range(n):
5     qc.x((i0 + n + 1) % n)
6     qc.x((i0 - n) % n)

```

---

```

1 # Parent 2
2 qc.h((n - 0) % n)
3 ----- split
4 for i0 in range(n):
5     qc.x((i0 + n + 1) % n)
6     qc.x((i0 - n) % n)

```

---

```

1 # Child 2
2 qc.h((n - 1) % n)
3 **** Crossover
4 for i0 in range(n):
5     qc.h((n - 1 - i0) % n)
6     qc.h((i0 - 1) % n)

```

---

Figure 3: An example of the effect of AST crossover on Qiskit codes

### 3.5 Improvement strategies

To enhance the performance of the genetic decompiler, we introduced three key strategies:

#### 3.5.1 Random initialization of new individuals

At each generation, a portion of the population is replaced with randomly initialized individuals to increase exploration and avoid local optima. The updated population for generation  $g + 1$  is given by:

$$\mathcal{P}_{g+1} = \mathcal{E}_g \cup \mathcal{N}_g$$

where  $\mathcal{E}_g$  represents elite individuals and  $\mathcal{N}_g$  represents newly generated individuals, ensuring both diversity and retention of high-quality solutions.

#### 3.5.2 Annealed mutation rate

An annealing mechanism is applied to the mutation rate, which decreases over generations to balance exploration and exploitation:

$$m_r 2(g) = \max(m_r 2^0 \cdot d^g, m_r 2^{\min})$$

where  $m_r 2^0$  is the initial mutation rate,  $d$  is the decay factor ( $0 < d < 1$ ), and  $m_r 2^{\min}$  is the minimum allowable rate. This ensures broad exploration early on and refined optimization in later stages.

These strategies improve the decompiler's ability to explore complex search spaces and converge to high-quality solutions efficiently.

#### 3.5.3 Simplification of expressions

Code bloat is a known GP issue that affects the decompilation process. To tackle this, we introduce the parsimony pressure as described in the previous section. However, to effectively calculate this value, it is required to simplify the expressions. This is done by using the symbolic simplification method in the SymPy package [57] for every generated expression (for qubits, angles, and loop limits). Algebraic simplification [58] also maintains the intelligibility of the expressions in the decompiled code.

## 3.6 Workflow

The workflow of the DeQompile software is summarized in the pseudocode presented in Algorithm 2.

---

**Algorithm 2** Genetic Programming Algorithm for Quantum Circuit Optimization

---

```
1: Initialize: Population  $\mathcal{P} \leftarrow \text{generate\_initial\_population}(\text{pop\_size})$ 
2: for generation = 1 to generations do
3:   Evaluate fitness  $\mathcal{F}$  for each individual in  $\mathcal{P}$ 
4:   Sort  $\mathcal{P}$  by  $\mathcal{F}$  in descending order
5:   Select best individual  $\mathcal{I}_{\text{best}}$  with fitness  $f_{\text{best}}$ 
6:   Initialize new population  $\mathcal{P}_{\text{new}} \leftarrow \{\mathcal{I}_{\text{best}}\}$  ▷ Elitism
7:   for  $i = 1$  to crossover_count do
8:     Select parents  $(\mathcal{P}_1, \mathcal{P}_2)$  using tournament selection
9:     Generate offspring  $(\mathcal{C}_1, \mathcal{C}_2)$  via crossover
10:    Add  $\mathcal{C}_1, \mathcal{C}_2$  to  $\mathcal{P}_{\text{new}}$ 
11:   end for
12:   for  $i = 1$  to mutation_count do
13:     Mutate a random individual  $\mathcal{I} \in \mathcal{P}_{\text{new}}$ 
14:   end for
15:   Introduce new random individuals to maintain diversity
16:   Update  $\mathcal{P} \leftarrow \mathcal{P}_{\text{new}}$ 
17: end for
18: return best individual  $\mathcal{I}_{\text{best}}$ 
```

---

The hyperparameter of DeQompile is listed in Table 2

Table 2: Hyperparameters for DeQompile

Hyperparameter	Default value	Description
algorithm_name	N/A	The name of the quantum algorithm to be decompiled.
qubit_limit	20	The maximum number of qubits in the generated quantum circuits.
generations	100	The number of generations the genetic algorithm will run.
pop_size	50	The size of the population in each generation.
max_length	10	The maximum number of operations in the generated quantum circuits.
crossover_rate	0.3	The rate at which crossover operations occur.
new_gen_rate	0.2	The rate at which new random individuals are introduced to the population.
mutation_rate	0.1	The rate at which mutation operations occur.
compare_method	‘l_by_l’	The method used to compare the generated QASM files with the target QASM files.
max_loop_depth	2	The maximum depth of nested loops in the generated qiskit codes.
selection_method	‘tournament’	The method used to select parents for crossover.
operations	[‘h’, ‘x’, ‘cx’]	The list of quantum gate operations that can be used in the circuits.

## 4 Challenges

The following challenges were critical in the implementation of the DeQompile software.

### 4.1 Syntactic correctness

Syntactic correctness is a critical aspect of the quantum circuit generation process, particularly in the context of quantum genetic programming. Ensuring the correct syntax of quantum gates and operations is essential for both the execution of the generated circuits for fitness calculation and the final decompilation result.

#### 4.1.1 Decompile-time handling of runtime constraints

At decompile time, the focus is on generating quantum circuits that adhere to the syntax of quantum programming languages, such as QASM. This includes ensuring that the qubit indices, gate types, and angles are well-formed according to the quantum circuit model. However, runtime execution introduces additional constraints, including hardware-specific requirements such as qubit connectivity and gate fidelity. While syntactic correctness ensures that the generated quantum circuit can be parsed and compiled into executable code, runtime correctness requires that the quantum circuit also adheres to the physical limitations of the underlying quantum processor, which are not satisfied by the decompilation.

#### 4.1.2 Qubit argument for multi-qubit gates

A common issue encountered during the generation of quantum circuits is ensuring that the qubit indices fall within valid bounds of  $n$ . To avoid syntax errors, such as an index exceeding the number of qubits, we apply a modulo operation to the expression.

$$\text{expr}_{\text{qubit}} = \text{expr}_{\text{qubit}} \bmod n \quad (2)$$

This approach works well for local operations (single-qubit gates).

However, when dealing with non-local operators like the CNOT gate for two qubits or the Toffoli gate for three qubits, two/three arbitrary expressions need to be evaluated to distinct values. If the generated indices for both the control and target qubits are the same (i.e., CX(i, i)), the operation becomes invalid. This problem often arises when the randomly generated expressions for qubit indices do not properly account for the total number of qubits in the circuit. Since the evaluation of an expression is unknown at decompile time, such syntactic errors could not be prevented. During fitness calculation, if such errors are encountered, the fitness for the comparison is set to 0.

#### 4.1.3 Divide-by-zero error for rotation angle expressions

A common challenge in maintaining syntactic correctness during quantum gate parameter calculations is the risk of divide-by-zero errors, particularly in gates like  $R_y$  where parameters involve division. When the denominator approaches zero, this can lead to undefined behavior and runtime failures. To address this, one approach is to label such cases as invalid expressions and redo the sampling, ensuring that only valid parameter sets are used. This helps maintain the integrity of decompiled circuits and prevents execution issues.

### 4.2 Convergence

Convergence in evolutionary algorithms is a key factor, particularly when using genetic programming techniques to approximate a target quantum circuit. Achieving convergence in quantum circuit generation is challenging due to the high dimensionality of the parameter space and the complexity of quantum operations.

#### 4.2.1 Convergence in parameter space

The parameter space of quantum circuits is typically large, as each quantum gate may involve several parameters (e.g., rotation angles for single-qubit gates). The complexity of the parameter space increases with the number of qubits and gates, making it difficult for the genetic algorithm to converge on an optimal solution. Convergence depends on the ability of the algorithm to effectively explore this high-dimensional space, which often requires fine-tuning the crossover and mutation operations to direct the search toward more promising regions of the solution space. Achieving convergence in such a complex space is non-trivial and often requires careful balance between exploration (searching new areas of the space) and exploitation (refining known good solutions) via hyperparameter tuning.

#### 4.2.2 Distribution of gates and expression operators

Another challenge in achieving convergence lies in the distribution of gates and operators in the generated quantum circuits. The population of circuits evolved by the genetic algorithm may be biased toward certain types of gates, especially if those gates are overrepresented in the initial population or in the fitness evaluation function. For instance, an overabundance of single-qubit gates like the  $U3$  gate may lead to premature convergence, where the algorithm favors these gates without exploring more diverse or complex solutions. To mitigate this bias and encourage the exploration of a wider variety of circuits, the distribution of gates and operators must be managed to ensure a more balanced search across the solution space. This can be achieved by modifying the genetic algorithm to include mechanisms that prevent over-representation of specific gates or by introducing diversity-promoting techniques such as niching or speciation.

## 5 Experiments with DeQompile

To evaluate the performance of the genetic decompiler implementation, DeQompile, we conducted a series of experiments on quantum circuit datasets, ranging from simple patterns to more complex quantum algorithms. This section presents the results for each dataset, along with fitness score trends and key observations.

### 5.1 1-qubit ansatz

The first set of experiments focuses on one-qubit patterns to validate the decompiler's ability to reconstruct and optimize simple circuits. These include single loops, multiple operations in loops, and nested loops. These examples are motivated by common ansatz in variational algorithms like quantum alternating operator ansatz (QAOA) [59], the generalization of the quantum approximate optimization algorithm (of the same acronym, QAOA). Many common algorithms like quantum (inverse) Fourier transformation also include similar patterns.

**Single loop:** We tested circuits applying a single gate iteratively to a qubit. For example:

- Applying multiple Hadamard gate ( $h_0(n)$ ) to first qubit.

$$qc = \prod_{i=0}^{n-1} H_0$$

- Applying the Hadamard gate ( $h_c(n)$ ) sequentially to all qubits.

$$qc = \prod_{i=0}^{n-1} H_i$$

- Rotational gate  $R_x$  applied to each qubit with exponentially decreasing angles ( $rx_c(n)$ ).

$$qc = \prod_{i=0}^{n-1} R_x \left( \frac{\pi}{2^i} \right)_i$$

The decompiler achieved a perfect fitness score of 1.0 within 30 generations, demonstrating its effectiveness in reconstructing these simple patterns. The results of combined score over 3 trials are shown in Figure 4.

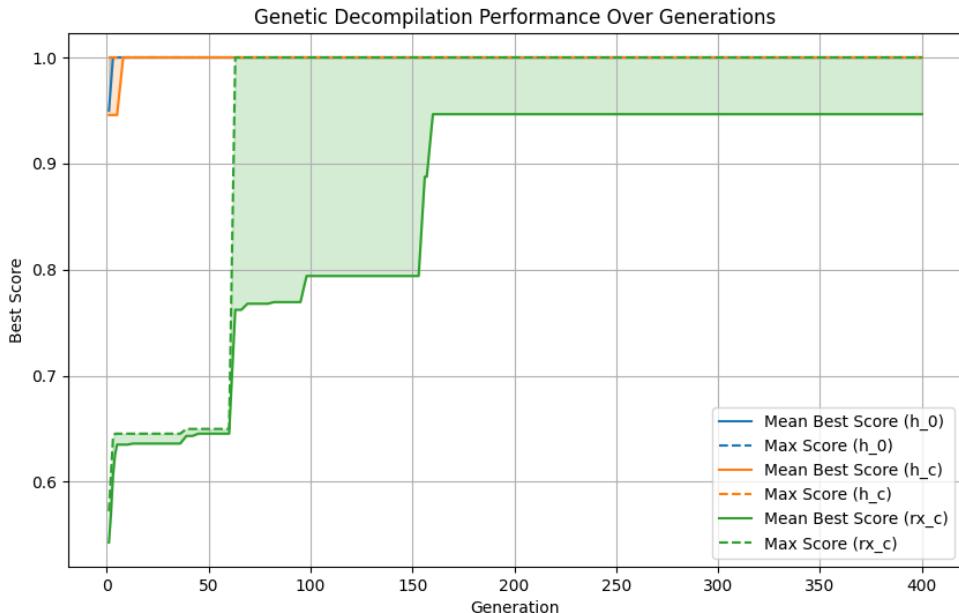


Figure 4: DeQompile performance for 1-qubit ansatz. The mean best score and maximum score across generations demonstrate steady improvement. The parameters used for generating the plot are: `mutation_rate=0.3`, `pop_size=40`, `generations=100`, `rep=3`, `total_qubit=20`, `max_length=10`, `perform_crossover=True`, `crossover_rate=0.3`, `new_gen_rate=0.2`, `max_loop_depth=2`, `mutation_rate_2=0.5`

As the figure shows, all three simple datasets get perfectly decompiled by our decompiler, the Highest scores of duplicate experiments for each quantum circuit all achieve a combined score 1, which means all individual metrics also reach score 1, we can also confirm it by checking the final qiskit code generated by our decompiler, which matches the certain pattern of these quantum circuits.

```

def generate_random_circuit_ast(n):
    qc = QuantumCircuit(n)
    for i0 in range(n):
        qc.h((i0 + 1 - 1) % n)
    return qc

```

(a) Best code for  $h_c$

```

def generate_random_circuit_ast(n):
    qc = QuantumCircuit(n)
    for i0 in range(n):
        qc.h((i0 + 0 + 0) % n)
    return qc

```

(b) Best code for  $h_0$

```

def generate_random_circuit_ast(n):
    qc = QuantumCircuit(n)
    for i0 in range(n):
        qc.rx(pi * (1 / (2 ** (i0 + 0 - 0) + (n - n - 0 + 0)))), 
              (i0 - n + 0 - n + 0) % n
    return qc

```

(c) Best code for  $rx_c$

Figure 5: Best decompiled codes for 1-qubit ansatz

**Multiple operations in a loop:** We extended the experiments to circuits with multiple operations within a loop, such as alternating  $H$  and  $X$  gates. The fitness scores improved steadily, with convergence requiring slightly more generations due to increased complexity.

**Nested loops:** Nested loops were also evaluated, involving interdependent operations such as combining  $R_x$  gates on one qubit and  $H$  gates on another. Fitness scores displayed evolutionary jumps around 50 generations, indicating the decompiler’s exploration and refinement capabilities.

## 5.2 Explainability-efficiency tradeoff in gate set transpilation

Native gates are the fundamental operations that a quantum processor can perform directly without needing further decomposition. Based on IBM’s quantum systems [60], we consider the native gates  $R_z$ ,  $X$ , and  $\sqrt{X}$  (the square root of  $X$ ), along with  $CNOT$  as a common two-qubit gate. These gates form the basis of quantum circuit design on these platforms and directly influence the implementation and efficiency [61, 62] of quantum algorithms.

In contrast to  $R_z$  and  $X$ , the  $R_y$  gate (rotation around the y-axis) is not commonly included as a native gate. To utilize  $R_y$  operations, they must be constructed through the synthesis of available native gates, predominantly  $R_z$  and  $X$ . We tested both undecomposed and decomposed versions of  $R_y$  circuits to evaluate the decompiler’s handling of circuit structure and decomposition. Specifically, we compared the following two scenarios as a proof of concept.

**Undecomposed  $R_y$ :**  $R_y(\theta)$  gates applied directly to all qubits.

**Decomposed  $R_y$ :** Circuits where  $R_y$  gates are expressed using  $R_x$ ,  $R_z$ , and  $H$  gates. The  $R_y$  gate can be decomposed using  $RX$  and  $RZ$  gates, which are also native gates for some quantum hardware.

$$R_y(\theta) = R_z\left(\frac{\pi}{2}\right) \cdot RX(\theta) \cdot R_z\left(-\frac{\pi}{2}\right)$$

As an alternate common decomposition, we consider using  $H$  and  $RX$  gates

$$R_y(\theta) = H \cdot RX(\theta) \cdot H$$

The results in Figure 6 show that:

- Undecomposed circuits achieved perfect fitness scores within 40 generations.
- Decomposed circuits exhibited lower scores, with  $H$ - $R_x$  decompositions achieving better results than  $R_x$ - $R_z$  decompositions, which requires multiple-parameter convergence.

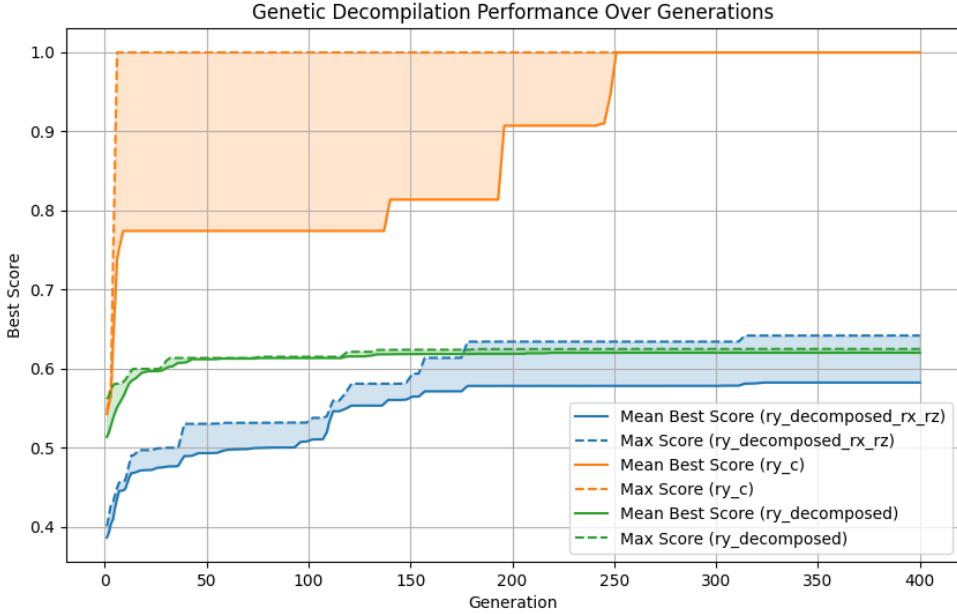


Figure 6: Fitness evolution for RY circuits for various native gate availability. Undecomposed circuits converge faster and achieve higher scores compared to decomposed versions. Here *ry\_decomposed* means the decomposition of *r\_x* and *h* for the *ry\_c* circuit, and *ry\_rx\_rz* means the decomposition of *r\_y* and *r\_z* for the *r\_c* circuit. The hyperparameter settings for the  $R_y$  gate experiments are as follows: `mutation_rate=0.3, new_gen_rate=0.3, crossover_rate=0.2, mutation_rate_2=0.99, max_length=4, max_loop_depth=3, qubit_limit=10, pop_size=50, generations=400, rep=3`.

This experiment indicates that while decomposition improves hardware efficiency, it reduces readability, making the circuits harder to decompile. Further aspects of explainability-efficiency tradeoff can be explored via DeQompile, especially in the context of variational algorithms [63].

### 5.3 Conventional quantum algorithms

After conducting our decompiling experiments on simple quantum circuits, we extend the approach to more complex and commonly used quantum algorithms. Specifically, we selected GHZ state preparation circuits, quantum Fourier transform (QFT), and quantum phase estimation (QPE) as these exhibit a gradual increase in hierarchical and compositional complexity. In this section, we provide a brief review of the circuit structure before presenting the decompilation results.

**GHZ state preparation:** Greenberger-Horne-Zeilinger (GHZ) [64] state is an entangled quantum state involving multiple qubits. It is used in quantum communication, quantum error correction, and tests of quantum mechanics. The circuit is shown in Figure 7.

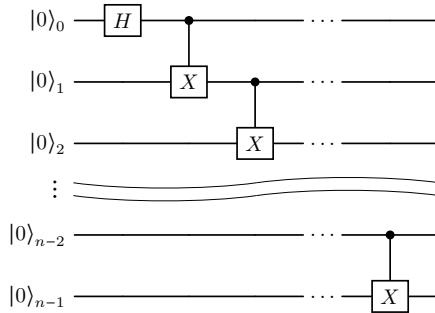


Figure 7: Quantum circuit for GHZ state preparation

The key steps for constructing a GHZ state preparation circuit are:

1. Hadamard gate: Apply a Hadamard gate to the first qubit to create a superposition state.

- CNOT gates: Apply CNOT gates between consecutive qubits (or, alternatively, from the first qubit to other qubits) to entangle them, creating the GHZ state.

**Quantum Fourier transform:** The quantum Fourier transform is a linear transformation on quantum bits, analogous to the discrete Fourier transform in classical computation [65]. It is a crucial component in many quantum algorithms, including Shor's algorithm for factoring. The key steps for QFT are:

- Superposition: Apply Hadamard gates to put the qubits into a state of superposition, encoding the input in quantum parallelism.
- Phase rotation: Apply a series of controlled phase rotation gates to entangle the qubits and encode the Fourier transform coefficients. Each subsequent Z-rotation involves a smaller angle by a factor of 2, i.e.,  $\frac{\pi}{2}, \frac{\pi}{4}$ , etc.

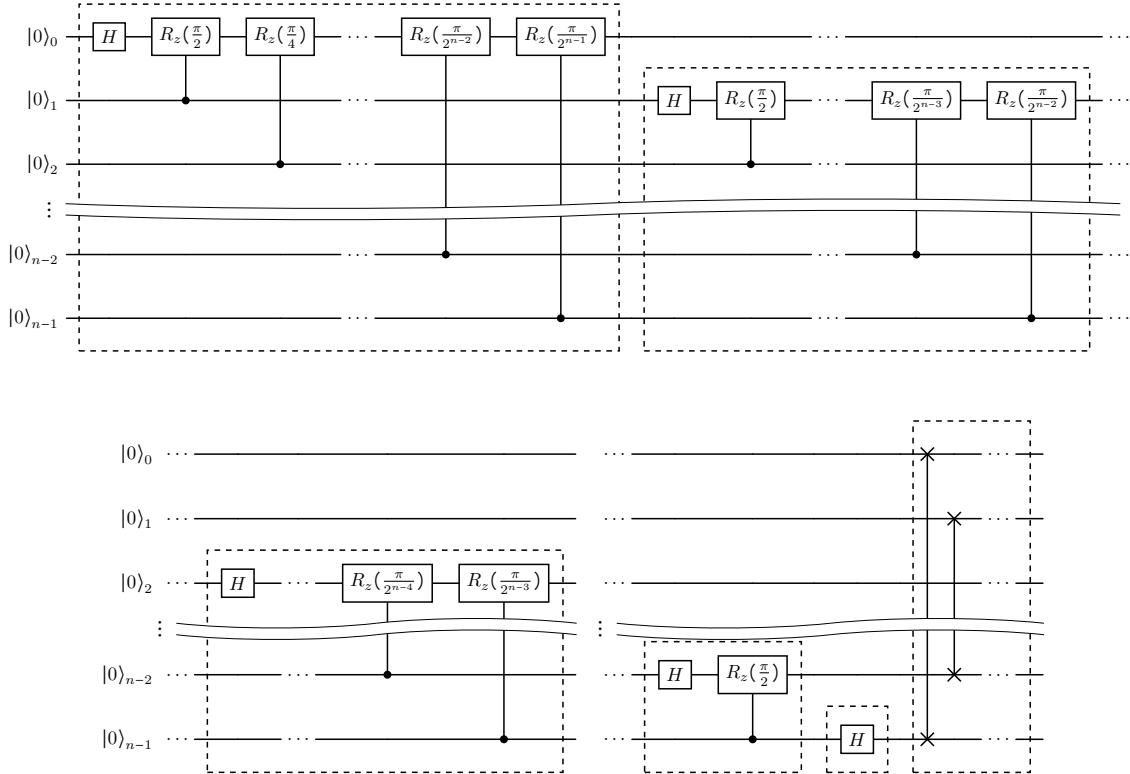


Figure 8: Quantum circuit for quantum Fourier transform (little endian)

**Quantum phase estimation:** Quantum phase estimation [66] is a fundamental algorithm used to estimate the phase (eigenvalue) introduced by a unitary operator. It has applications in various fields, including factoring, cryptography, and quantum chemistry. The key steps for QPE are:

- State preparation: Initialize two registers: the first with qubits in superposition to act as controls, and the second with an eigenstate of the unitary operator.
- Controlled unitary operations: Apply controlled unitary operations that evolve the second register based on the state of the first register.
- Inverse quantum Fourier transform ( $\text{QFT}^\dagger$ ): Apply the inverse QFT on the first register to convert the quantum phase information into a readable binary format.

The results of the decompiling on these algorithms is shown in Figure 10. The Qiskit code generated by the decompiler for the best individual from last generation is available in [56].

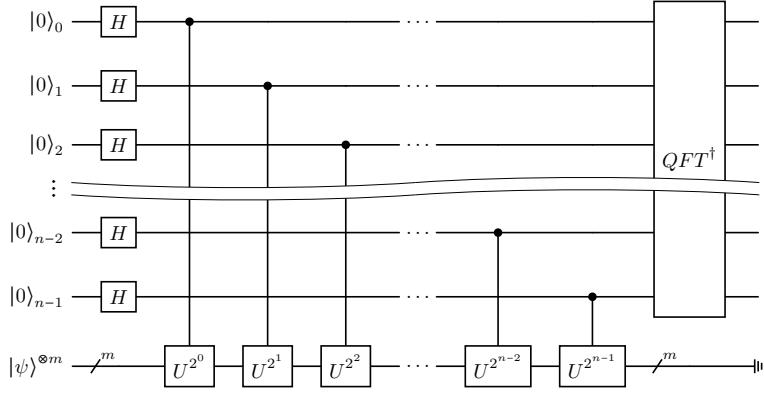


Figure 9: Quantum circuit for quantum phase estimation.  $QFT^\dagger$  refers to the inverted circuit of Figure 8.

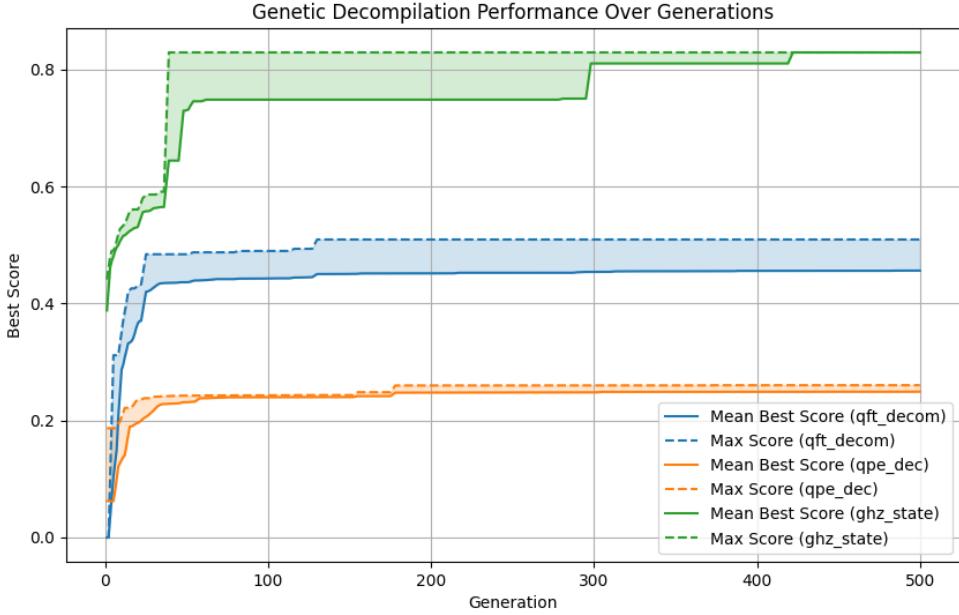


Figure 10: Genetic decompilation performance over generations. The plot shows the mean best score and the maximum score across generations for different algorithms: QFT decomposition (`qft_decom`), QPE decomposition (`qpe_dec`), and GHZ state preparation (`ghz_state`). The mean best score represents the average score of the best individual across all experiments for each generation, while the max score indicates the highest score achieved among all repetitions for each generation. The hyperparameter settings are as follows: `mutation_rate=0.3`, `new_gen_rate=0.3`, `crossover_rate=0.2`, `mutation_rate_2=0.99`, `max_length=4`, `max_loop_depth=3`, `qubit_limit=10`, `pop_size=40`, `generations=500`, `rep=3`.

The plot illustrates that the GHZ circuit achieves the best decompilation result, while the QFT is easier to decompile than the QPE. This matches the intuition of the complexity ordering among these examples.

QFT convergence was further inspected in the range of 2 – 10 qubits to assess scalability. Smaller QFT circuits (2 – 5 qubits) achieved fitness scores above 0.9 within 100 generations, while larger QFT circuits (6 – 10 qubits) converged more slowly, plateauing at scores around 0.85 after 150 generations. This indicates the decompiler’s effectiveness for smaller patterns, while larger circuits present additional challenges due to the exponential growth in gate sequences.

## 6 Conclusion

In the current era of ubiquitous software automation, this research champions the need to distill higher-level abstractions that are both understandable to human experts and amenable to formal analysis. Understanding the underlying algorithm behind a family of quantum circuits motivates us to develop DeQompile, a QASM

to Qiskit decompiler. Genetic programming is employed to evolve a candidate solution over generations, using the abstract syntax tree representation of Qiskit’s Python code. We designed a software framework consisting of strategies to generate syntactically valid and expressive individuals, evolution operations, and hyperparameter tuning. We defined a set of metrics like gate sequence frequency, gate sequence similarity, and line-by-line comparison to evaluate the fitness of the decompilation. The open-sourced DeQompile tool was demonstrated on a series of examples with increasing complexity, from common ansatz patterns to popular quantum algorithms. We also demonstrate the explainability-efficiency tradeoff in quantum algorithms during native gate transpilation. This research provides a novel explainability framework in quantum information processing.

We infer some promising future directions based on the experiments conducted. The DeQompile tool can be extended to include measurements and control flow structures that are essential for protocols like error correction. Our current approach to fitness calculation is based on comparing the textual QASM syntax, which fails to capture semantic similarities (e.g., commuting gates). However, assessing semantic similarity while avoiding the manipulation of exponential-sized unitary matrices remains an open question. It was evident that genetic programming, while being successful for symbolic regression fails to decompile or synthesize complex structures. While further investigation into hyper-parameter tuning is imperative, it is also worthwhile to explore alternate program synthesis methods based on neural networks. It is important to realize that while the neural network itself is not necessarily explainable (equivalent to the random mutation in our case), the overall architecture adheres to the neuro-symbolic paradigm [67] with its distinct opportunities. Advancements in foundational models [45, 46, 68] can augment quantum decompilation capabilities in the future. Additionally, prior knowledge can be used to guide the convergence of the decompilation. This can be incorporated via hierarchical reinforcement learning of a concept library [37, 69, 33]. Such an augmented framework would be equipped to decompile a suite of conventional or generated quantum circuits [70, 71, 72, 73]. Finally, as an alternate use case, the DeQompile can also be used to compress quantum circuits [74, 39, 38] and estimate the algorithmic information content [75, 39, 38] with multiple theoretical and practical implications.

## Software availability

The open-sourced code for the project, configuration files, output data, and plotting codes for the experiments presented in this article are available at: <https://github.com/Advanced-Research-Centre/DeQompile/>.

## Acknowledgements

A didactic introduction and additional details of the implementation presented in this article can be found in the corresponding master thesis [56]. We thank Vedran Dunjko for insightful discussions during the project planning and evaluation. AS acknowledges funding from the Dutch Research Council (NWO) through the project “QuTech Part III Application-based research” (project no. 601.QT.001 Part III-C—NISQ).

## Author contributions

Conceptualization, A.S.; Methodology, A.S., S.F. and S.X.; Software, S.X. and A.S.; Validation, S.X. and A.S.; Writing – Original Draft Preparation, S.X. and A.S.; Writing – Review & Editing, A.S. and S.F.; Visualization, S.X., S.F. and A.S.; Supervision, A.S. and S.F.; Project Administration, S.F. and A.S.;

## References

- [1] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.
- [2] Rajeev Acharya, Laleh Aghababaie-Beni, Igor Aleiner, Trond I Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Nikita Astrakhantsev, Juan Atalaya, et al. Quantum error correction below the surface code threshold. *arXiv preprint arXiv:2408.13687*, 2024.
- [3] Youngseok Kim, Andrew Eddins, Sajant Anand, Ken Xuan Wei, Ewout Van Den Berg, Sami Rosenblatt, Hasan Nayfeh, Yantao Wu, Michael Zaletel, Kristan Temme, et al. Evidence for the utility of quantum computing before fault tolerance. *Nature*, 618(7965):500–505, 2023.
- [4] Yutaro Akahoshi, Kazunori Maruyama, Hirotaka Oshima, Shintaro Sato, and Keisuke Fujii. Partially fault-tolerant quantum computing architecture with error-corrected clifford gates and space-time efficient analog rotations. *PRX Quantum*, 5(1):010337, 2024.
- [5] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM journal on computing*, 26(5):1484–1509, 1999.

- [6] Lov K Grover. A fast quantum mechanical algorithm for database search. *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.
- [7] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.
- [8] Frank Leymann and Johanna Barzen. The bitter truth about gate-based quantum algorithms in the nisq era. *Quantum Science and Technology*, 5(4):044007, 2020.
- [9] Xavier Waintal. The quantum house of cards. *Proceedings of the National Academy of Sciences*, 121(1):e2313269120, 2024.
- [10] Olivier Ezratty. Where are we heading with nisq? *arXiv preprint arXiv:2305.09518*, 2023.
- [11] Fred Chong. Closing the gap between quantum algorithms and machines with hardware-software co-design. In *Proceedings of the Great Lakes Symposium on VLSI 2023*, pages 83–84, 2023.
- [12] Zoltán Zimborás, Bálint Koczor, Zoë Holmes, Elsi-Mari Borrelli, András Gilyén, Hsin-Yuan Huang, Zhenyu Cai, Antonio Acín, Leandro Aolita, Leonardo Banchi, et al. Myths around quantum computation before full fault tolerance: What no-go theorems rule out and what they don’t. *arXiv preprint arXiv:2501.05694*, 2025.
- [13] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, 2016.
- [14] Marco Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, et al. Variational quantum algorithms. *Nature Reviews Physics*, 3(9):625–644, 2021.
- [15] Yuxuan Du, Tao Huang, Shan You, Min-Hsiu Hsieh, and Dacheng Tao. Quantum circuit architecture search for variational quantum algorithms. *npj Quantum Information*, 8(1):62, 2022.
- [16] Weiwei Zhu, Jiangtao Pi, and Qiuyuan Peng. A brief survey of quantum architecture search. In *Proceedings of the 6th International Conference on Algorithms, Computing and Systems*, pages 1–5, 2022.
- [17] Darya Martyniuk, Johannes Jung, and Adrian Paschke. Quantum architecture search: A survey. *arXiv preprint arXiv:2406.06210*, 2024.
- [18] Mateusz Ostaszewski, Lea M Trenkwalder, Wojciech Masarczyk, Eleanor Scerri, and Vedran Dunjko. Reinforcement learning for optimization of variational quantum circuit architectures. *Advances in Neural Information Processing Systems*, 34:18182–18194, 2021.
- [19] Akash Kundu. Reinforcement learning-assisted quantum architecture search for variational quantum algorithms. *arXiv preprint arXiv:2402.13754*, 2024.
- [20] Li Ding and Lee Spector. Evolutionary quantum architecture search for parametrized quantum circuits. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 2190–2195, 2022.
- [21] Anqi Zhang and Shengmei Zhao. Evolutionary-based searching method for quantum circuit architecture. *Quantum Information Processing*, 22(7):283, 2023.
- [22] L. H. Gilpin, R. Caruana, J. Gehrke, P. Koch, and D. H. Chau. Explaining explanations: An overview of interpretability of machine learning. In *Proceedings of the 2018 ICML Workshop on Human Interpretability in Machine Learning*, 2018.
- [23] Raoul Heese, Thore Gerlach, Sascha Mücke, Sabine Müller, Matthias Jakobs, and Nico Piatkowski. Explaining quantum circuits with shapley values: Towards explainable quantum machine learning. *arXiv preprint arXiv:2301.09138*, 2023.
- [24] Elies Gil-Fuster, Jonas R Naujoks, Grégoire Montavon, Thomas Wiegand, Wojciech Samek, and Jens Eisert. Opportunities and limitations of explaining quantum machine learning. *arXiv preprint arXiv:2412.14753*, 2024.
- [25] Lirandë Pira and Chris Ferrie. On the interpretability of quantum neural networks. *Quantum Machine Intelligence*, 6(2):52, 2024.
- [26] Luke Power and Krishnendu Guha. Feature importance and explainability in quantum machine learning. *arXiv preprint arXiv:2405.08917*, 2024.
- [27] Hsin-Yi Lin, Huan-Hsin Tseng, Samuel Yen-Chi Chen, and Shinjae Yoo. Quantum gradient class activation map for model interpretability. In *2024 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 165–170. IEEE, 2024.
- [28] Shuxiang Ru, Xin Tan, and Yifan Zhang. Interpretable quantum architecture search using weisfeiler-lehman kernels. *npj Quantum Information*, 6(1):1–9, 2020.
- [29] Gean T Pereira, Iury BA Santos, Luís PF Garcia, Thierry Urruty, Muriel Visani, and André CPLF de Carvalho. Neural architecture search with interpretable meta-features and fast predictors. *Information Sciences*, 649:119642, 2023.
- [30] Yash J Patel, Akash Kundu, Mateusz Ostaszewski, Xavier Bonet-Monroig, Vedran Dunjko, and Onur Danaci. Curriculum reinforcement learning for quantum architecture search under hardware errors. In *The Twelfth International Conference on Learning Representations*, 2024.
- [31] Abhishek Sadhu, Aritra Sarkar, and Akash Kundu. A quantum information theoretic analysis of reinforcement learning-assisted quantum architecture search. *Quantum Machine Intelligence*, 6(2):49, 2024.
- [32] Akash Kundu, Aritra Sarkar, and Abhishek Sadhu. Kanqas: Kolmogorov-arnold network for quantum architecture search. *EPJ Quantum Technology*, 11(1):76, 2024.
- [33] Akash Kundu and Leopoldo Sarra. From easy to hard: Tackling quantum problems with learned gadgets for real hardware. *arXiv preprint arXiv:2411.00230*, 2024.

- [34] Aritra Sarkar. Automated quantum software engineering. *Automated Software Engineering*, 31(1):1–17, 2024.
- [35] Michael Schmidt and Hod Lipson. Distilling free-form mathematical expressions from experimental data. *Science*, 324(5923):81–85, 2009.
- [36] Matthew Bowers, Theo X Olausson, Lionel Wong, Gabriel Grand, Joshua B Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL):1182–1213, 2023.
- [37] Lea M Trenkwalder, Andrea López Incera, Hendrik Poulsen Nautrup, Fulvio Flamini, and Hans J Briegel. Automated gadget discovery in science. *arXiv preprint arXiv:2212.12743*, 2022.
- [38] Aritra Sarkar. *Applications of Quantum Computation and Algorithmic Information: for Causal Modeling in Genomics and Reinforcement Learning*. PhD thesis, Delft University of Technology, 2022.
- [39] Aritra Sarkar, Zaid Al-Ars, and Koen Bertels. Qksa: Quantum knowledge seeking agent. In *International Conference on Artificial General Intelligence*, pages 384–393. Springer, 2022.
- [40] Aritra Sarkar, Zaid Al-Ars, and Koen Bertels. Quantum circuit design for universal distribution using a superposition of classical automata. *arXiv preprint arXiv:2006.00987*, 2020.
- [41] Aritra Sarkar, Zaid Al-Ars, and Koen Bertels. Estimating algorithmic information using quantum computing for genomics applications. *Applied Sciences*, 11(6):2696, 2021.
- [42] Andrew Gauthier, Nathan Cho, Vidhya Narayanan, and Joseph Smith. Flash fill: A programming by example tool for spreadsheets. *ACM Conference on Human Factors in Computing Systems*, 2015.
- [43] Manuel Egele, Thorsten Holz, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. A survey on automated dynamic malware analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):1–42, 2008.
- [44] Pat Royal, Michael Hovis, Jared Reaves, Tim Goodwin, Timothy Hines, and Shane Packard. Polyunpack: Automated packing analysis for malicious code. *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.
- [45] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium* (2024). <https://api.semanticscholar.org/CorpusID>, volume 267622140, 2024.
- [46] Paul J Blazek, Kesavan Venkatesh, and Milo M Lin. Automated discovery of algorithms from data. *Nature Computational Science*, 4(2):110–118, 2024.
- [47] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.
- [48] Riccardo Baldoni, Andrea De Benedictis, and Emanuele Marchiori. A survey on neural decompilation: From machine code to high-level programs. *ACM Computing Surveys (CSUR)*, 51(4):1–36, 2018.
- [49] Lee Spector. *Automatic Quantum Computer Programming: a genetic programming approach*, volume 7. Springer Science & Business Media, 2004.
- [50] Lee Spector. A brief introduction to quantum genetic programming. *Genetic Programming and Evolvable Machines*, 4(1):1–19, 2003.
- [51] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D Nation, Lev S Bishop, Andrew W Cross, et al. Quantum computing with qiskit. *arXiv preprint arXiv:2405.08810*, 2024.
- [52] 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.
- [53] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.
- [54] Paul Vitanyi. Three approaches to the quantitative definition of information in an individual pure quantum state. *arXiv preprint quant-ph/9907035*, 2000.
- [55] Byoung-Tak Zhang and Heinz Mühlenbein. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38, 1995.
- [56] Shubing Xie. Quantum circuit decompiler: Pattern recognition of quantum circuits by genetic algorithm. Master’s thesis, Leiden University, 8 2024.
- [57] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.
- [58] Joel Moses. Algebraic simplification a guide for the perplexed. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pages 282–304, 1971.
- [59] Stuart Hadfield, Zhihui Wang, Bryan O’gorman, Eleanor G Rieffel, Davide Venturelli, and Rupak Biswas. From the quantum approximate optimization algorithm to a quantum alternating operator ansatz. *Algorithms*, 12(2):34, 2019.
- [60] IBM. Ibm quantum guides, 2024. Accessed: 2025-01-21.
- [61] Aritra Sarkar, Akash Kundu, Matthew Steinberg, Sibasish Mishra, Sebastiaan Fauquenot, Tamal Acharya, Jarosław A Miszczał, and Sebastian Feld. Yaqq: Yet another quantum quantizer–design space exploration of quantum gate sets using novelty search. *arXiv preprint arXiv:2406.17610*, 2024.

- [62] Bao Gia Bach, Akash Kundu, Tamal Acharya, and Aritra Sarkar. Visualizing quantum circuit probability: estimating quantum state complexity for quantum program synthesis. *Entropy*, 25(5):763, 2023.
- [63] Mauro ES Morales, Timur Tlyachev, and Jacob Biamonte. Variational learning of grover’s quantum search algorithm. *Physical Review A*, 98(6):062333, 2018.
- [64] Daniel M Greenberger, Michael A Horne, and Anton Zeilinger. Going beyond bell’s theorem. In *Bell’s theorem, quantum theory and conceptions of the universe*, pages 69–72. Springer, 1989.
- [65] Fang Xi Lin. Shor’s algorithm and the quantum fourier transform. *McGill University*, 2014.
- [66] A Yu Kitaev. Quantum measurements and the abelian stabilizer problem. *arXiv preprint quant-ph/9511026*, 1995.
- [67] Jennifer J Sun, Megan Tjandrasuwita, Atharva Sehgal, Armando Solar-Lezama, Swarat Chaudhuri, Yisong Yue, and Omar Costilla-Reyes. Neurosymbolic programming for science. *arXiv preprint arXiv:2210.05050*, 2022.
- [68] Paul J Blazek and Milo M Lin. Explainable neural networks that simulate reasoning. *Nature Computational Science*, 1(9):607–618, 2021.
- [69] Leopoldo Sarra, Kevin Ellis, and Florian Marquardt. Discovering quantum circuit components with program synthesis. *arXiv preprint arXiv:2305.01707*, 2023.
- [70] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. MQT Bench: Benchmarking software and design automation tools for quantum computing. *Quantum*, 2023. MQT Bench is available at <https://www.cda.cit.tum.de/mqtbench/>.
- [71] Boran Apak, Medina Bandic, Aritra Sarkar, and Sebastian Feld. Ketgpt–dataset augmentation of quantum circuits using transformers. In *International Conference on Computational Science*, pages 235–251. Springer, 2024.
- [72] Shunsuke Daimon, Kakeru Tsunekawa, Ryoto Takeuchi, Takahiro Sagawa, Naoki Yamamoto, and Eiji Saitoh. Quantum circuit distillation and compression. *Japanese Journal of Applied Physics*, 63(3):032003, 2024.
- [73] Natividad Ruiz, Kevin Ochoa, David Iglesias, Daniel Martinez, and Masashi Eguchi. Quantum circuit optimization using machine learning algorithms. *Quantum Engineering*, 2024.
- [74] Sibasish Mishra and Aritra Sarkar. Concept discovery of energy-optimized quantum instruction sets based on algorithmic complexity. [<https://github.com/Advanced-Research-Centre/QART>] (<https://github.com/Advanced-Research-Centre/QART>), 2024.
- [75] Matthew Steinberg, Medina Bandić, Sacha Szkudlarek, Carmen G Almudever, Aritra Sarkar, and Sebastian Feld. Lightcone bounds for quantum circuit mapping via uncomplexity. *npj Quantum Information*, 10(1):113, 2024.