Quantum Circuit Decompiler: Pattern Recognition of Quantum Circuits By Genetic Algorithm

Xie, Shubing

Quantum Circuit Decompiler – Pattern Recognition of Quantum Circuits By Genetic Algorithm

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
PHYSICS

Author :	Shubing Xie
Student ID :	3554309
Delft Supervisor :	Sebastian Feld
Daily Supervisor :	Aritra Sarkar
Leiden Corrector :	Vedran Dunjko

Leiden, The Netherlands, August 21, 2024

Quantum Circuit Decompiler – Pattern Recognition of Quantum Circuits By Genetic Algorithm

Shubing Xie

Institute-Lorentz, Leiden University
Qutech, TU Delft
P.O. Box 9500, 2300 RA Leiden, The Netherlands
Lorentzweg 1, 2628 CJ Delft, The Netherlands

August 21, 2024

Abstract

For gate-based quantum computing, the design of quantum circuits is a critical procedure for implementing specific quantum algorithms. Currently, many quantum circuits are designed using Quantum Architecture Search, where heuristic algorithms are often applied, resulting in circuits that are not human-interpretable. To understand the underlying logic and specific patterns of the quantum circuits, we have developed a QASM-to-Qiskit transformation called the quantum decompiler. This transformation acts as a form of reverse engineering, converting the relatively low-level representation of a quantum circuit – the QASM file into a more understandable, high-level representation, the Python Qiskit code. To implement this method, we combined Genetic Algorithms (GA) and Abstract Syntax Trees (AST). In our work, we primarily focus on developing the concept and testing this proof-of-concept on some simple, commonly used quantum circuits (GHZ, QFT, QPE) with a limited number of qubits. At the end of this research, the metrics used for the evaluation of the output of our decompiler is also discussed.

Contents

List of Acronyms

AST	Abstract Syntax Tree
CX	Controlled-NOT Gate
GA	Genetic Algorithm
GHZ	Greenberger-Horne-Zeilinger (state)
H	Hadamard Gate
HDL	Hardware Description Language
LCS	Longest Common Subsequence
LLM	Large Language Model
QAS	Quantum Architecture Search
QASM	Quantum Assembly Language
QCS	Quantum Circuit Synthesis
QD	Quantum Decompiler
QFT	Quantum Fourier Transform
QML	Quantum Machine Learning
QPE	Quantum Phase Estimation
QPU	Quantum Processing Unit
RL	Reinforcement Learning

Version of August 21, 2024– Created August 21, 2024 - 06:42

Acknowledgements

I would like to express my deepest gratitude to my daily supervisor, Aritra Sarkar, for his invaluable guidance, continuous support, and insightful feedback throughout the development of this thesis. His expertise and encouragement have been instrumental in shaping this work. I am also profoundly grateful to my Delft supervisor, Sebastian Feld, for his constructive suggestions and for providing a broader perspective on the research. His input has greatly enriched the quality of this thesis. I extend my sincere thanks to my Leiden supervisor, Vedran Dunjko, for his support and valuable insights. Additionally, I thank my parents, my friends and all the colleagues in my internship office who encouraged me and shared me with their ideas. Without the support from them, both physically and emotionally, I can't make it.

Special thanks to ChatGPT*, I use it to refine my words and polish my sentences a lot. This tool plays a significant role in my project. Also, I use it to debug my Python code (mainly related to plotting) and generate some trivial functions for the final script.

I've spent two years in the quaint town of Leiden, and I often recall the sea breeze by the coast and the dim streetlights by the canals in the evening. Watching the reflections on the water, it feels like time has stood still. My life has forever been connected to this ancient town, and I can truly sense that it is a fortress of freedom, where I have encountered many great souls of the past.

In the blink of an eye, I have been a student for 19 years. My two years in Leiden felt like a rare escape from my familiar surroundings, with each day bringing something new. Admittedly, this period was also filled with struggles, as I racked my brain over projects and endured countless sleepless nights, all of which remain vividly in my memory.

Now, I stand at another crossroads in my life. But I choose to believe that life is a journey. During my two years in the Netherlands, I have no regrets.

*https://openai.com/chatgpt/

Chapter 1

Introduction

> The measure of greatness in a scientific idea is the extent to which it stimulates thought and opens up new lines of research.
>
> — Paul Dirac

Circuit decompilation involves the reverse engineering of digital logic circuits to reconstruct a high-level representation that approximates the original design, typically expressed in a hardware description language. In quantum computing, defining decompilation is challenging. Unlike traditional computers, for gate-based quantum computing, quantum algorithms are usually implemented as quantum circuits on quantum computers and compiled quantum circuits can be represented in Quantum Assembly (QASM). Furthermore, the design of these quantum circuits can be in Qiskit code written in Python. Naturally, we can define reverse engineering from QASM to a higher-level Qiskit code as a process of quantum decompilation. This QASM-to-Qiskit process closely resembles inductive inference in the human brain, such as when we deduce the next item in a sequence by identifying underlying patterns. For quantum circuits, if we know the corresponding design on a limited scale and wish to extend it, we must identify the shared structures or recognize the patterns of these circuits. In this thesis, the method used to represent the rules or patterns of these circuits is by finding the underlying code that can generate them. Therefore, this research aims to discover a program synthesis method that generates a piece of Qiskit code as a high-level representation of the quantum circuits. We define this as a quantum decompiler, which combines the genetic algorithm (GA) and Abstract Syntax Tree (AST) to find patterns in quantum algorithms

within the circuit representations.

Motivation

Due to the inherent complexity in establishing the necessary mathematical frameworks and designing circuit-level logic, only a few human-designed quantum algorithms currently exist. Modern quantum applications often leverage various optimization techniques, including reinforcement learning (RL) [1–4] and genetic algorithms [5–7], for quantum architecture search. In these methods, an agent, whether an RL agent or a genetic programming approach, designs quantum circuits by evaluating the quality of solutions or by evolving solutions to fit a desired outcome. However, these automatically generated circuits frequently lack human interpretability, which obscures the logic driving their success and makes them difficult to scale or adapt for new tasks.

In response to these challenges, this work develops a Qasm-to-Qiskit quantum circuit decompiler. Utilizing genetic programming, we evolve the abstract syntax tree to create a Qiskit program that can generate a comparable set of Qasm circuits based on the size of the problem. This methodology is applied to simple variational ansatz patterns and established quantum algorithms, aiming to enhance the practical usability of quantum circuits while preserving or improving their computational efficiency.

Research Questions

The project will explore several key questions:

1. How can we reverse-engineer quantum circuits from QASM to Python Qiskit code? What methods would be effective?

2. How can we measure the correctness of our decompilation? How do we define the similarity between two groups of QASM files?

3. To what extent can the decompiler generalize across different types of quantum circuits, and what are the limitations of its applicability?

4. If the initial quantum circuits are decomposed into different forms, can our decompiler still reconstruct the design and generate the corresponding Python code?

Problem Defination

To be more clear with the goal we want to achieve and specifically illustrate what we are going to implement in our thesis, the problem is defined as follows:

Phase I: Practical Deliverable

Phase I involves the initial practical deliverables of the project. First, we will select a quantum algorithm A of choice, such as, quantum Fourier transform, quantum phase estimation. Quantum circuits for A will be generated for different problem sizes, ranging from 2 to 10 qubits, using a Python program $P_A(n)$ with the Qiskit package. The resulting circuits, $C_A^2, C_A^3, \ldots, C_A^{10}$, will be represented in either OpenQASM or unrolled Qiskit format, depending on the method chosen (compression/ML-based methods will use QASM, while software engineering methods will work at the Python level). This set of circuits C_A^n serves as the data set. Given this set as input, the designed decompiler should be able to build a 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. $P'_A(n)$ can then be used to generate $C'^{11}_A, C'^{12}_A, \ldots$, which will be compared to $C_A^{11}, C_A^{12}, \ldots$ to validate the generalization capability.

Phase II: Extended Goal

Phase II addresses the extended goals of the project by exploring the limits of the tool developed in Phase I through empirical analysis of decompiling a highly optimized code with a lesser algorithmic structure. We will take circuits C_A^n and translate them into the corresponding unitaries U_A^n. These unitaries will be decomposed using Qiskit into the native gate set of a quantum processing unit (QPU), such as IBM's, to obtain circuits C'^n_A. This optimization is the recompiler. Given this set of circuits C'^n_A, the decompiler will infer $P''_A(n)$. We expect $P''_A(n)$ to be less explainable and more abstract than $P'_A(n)$, thus making it harder to generalize from optimized circuits. This will be tested by checking if $C''^{11}_A, C''^{12}_A, \ldots$ have lower similarity scores with target circuits compared to $C'^{11}_A, C'^{12}_A, \ldots$, with respect to $C_A^{11}, C_A^{12}, \ldots$.

Version of August 21, 2024– Created August 21, 2024 - 06:42

Chapter 2

Preliminaries

> Quantum mechanics is the way nature works and it is fascinating.
>
> Richard Feynman

In this chapter, we introduce the fundamental concepts and tools relevant to our research, aiming to build a bridge from quantum mechanics to quantum computing, and further to the context of our quantum decompiler.

2.1 Qubits

Classical computers use binary digits, 0 and 1, to represent logical computation. However, as Richard Feynman famously stated, everything in the world obeys the rules of quantum mechanics. Naturally, this leads us to consider how to represent a number or a certain state in a "quantum computer." Thus, the concept of the qubit was born.

In the realm of quantum computation, the qubit is the fundamental unit of quantum information, analogous to the classical bit in classical computation. However, unlike a classical bit which can be either 0 or 1, a qubit can exist simultaneously in a superposition of both states $|0\rangle$ and $|1\rangle$. This property is a direct consequence of the principles of quantum mechanics.

A qubit is mathematically described as a quantum state in a two-dimensional Hilbert space, where the state of the qubit can be expressed as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{2.1}$$

where α and β are complex probability amplitudes. The coefficients α and

β determine the probabilities of measuring the qubit in the states $|0\rangle$ and $|1\rangle$, respectively. Specifically, the probability P of measuring the state $|0\rangle$ is $|\alpha|^2$ and the probability of measuring the state $|1\rangle$ is $|\beta|^2$, with the normalization condition ensuring that the total probability is one:

$$|\alpha|^2 + |\beta|^2 = 1. \tag{2.2}$$

The state of an n-qubit system is an arbitrary superposition over 2^n basis states with normalized complex amplitudes as coefficients, with an irrelevant global phase. Mathematically, it is a vector in a 2^n-dimensional Hilbert space (the complex generalization of Euclidean space).

2.2 Quantum Computation and its Principles

Building on the foundational concepts of quantum information science, we explore the various models of quantum computation and their principles.

2.2.1 Schema For different Quantum Computers Models

Quantum computation is a revolutionary paradigm that leverages the principles of quantum mechanics to process information in fundamentally new ways. Unlike classical computation, which relies on bits as the smallest units of information, quantum computation uses qubits, which can exist in superpositions of states and exhibit entanglement. These unique properties endow quantum computers with capabilities far beyond those of classical systems.

Several models of quantum computers have been proposed and developed, each with its own approach to harnessing the power of qubits. The primary models include:

- **Quantum Circuit Model**: This is the most widely studied model of quantum computation, often referred to as the gate model or gate-based quantum computing. In this approach, quantum computations are performed using a sequence of quantum gates, which are unitary operations that manipulate the states of qubits. Quantum algorithms, such as Shor's algorithm [8] for factoring and Grover's algorithm [9] for search, are typically expressed within this framework.

  In this thesis, we focus on the gate-based quantum computing model and specifically work on QASM representation of it. Some common quantum gates are shown in Table 2.1

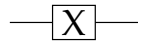
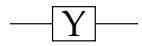
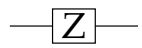
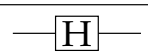
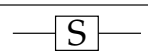
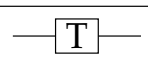
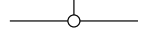
Operator	Gate(s)	Matrix
Pauli-X (X)		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)		$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$

Table 2.1: *Commonly used quantum gates*

Version of August 21, 2024– Created August 21, 2024 - 06:42

- **Adiabatic Quantum Computing (AQC)**: This model relies on the principle of adiabatic evolution [10], where the system remains in its ground state while the Hamiltonian of the system is slowly varied from an initial to a final form. Quantum annealers, such as those developed by D-Wave Systems [11], operate based on this principle and are particularly suited for solving optimization problems.

- **Topological Quantum Computing**: In this approach, qubits are encoded in the global properties of topological states of matter, which are inherently protected from local noise and decoherence. This model leverages anyons, particles that exist in two-dimensional spaces and follow non-Abelian statistics, to perform quantum computations [12]. Topological quantum computing promises greater fault tolerance compared to other models.

- **Measurement-Based Quantum Computing (MBQC)**: Also known as the one-way quantum computer, this model uses a highly entangled initial state, known as a cluster state, to perform computations. Computation is carried out by performing a sequence of measurements on individual qubits, with the outcomes of these measurements determining the subsequent operations.

2.2.2 Physical Implementation for Quantum Computation

As of now, the most prevalent and widely used computation model for quantum computers is gate-based quantum computation [13]. Specifically, in the quantum circuit model, David DiVincenzo, a leading researcher in the field, has outlined several critical requirements for the physical implementation of quantum computation [14]. These criteria provide a framework for understanding and developing quantum technologies. The key requirements are as follows:

- A scalable physical system with well-characterized qubits: For a quantum computer to function effectively, it must have a scalable system where qubits can be reliably created, manipulated, and measured. Qubits, which can be represented by various physical systems such as spin states of particles or energy levels of atoms must have well-defined and controllable properties.

- The ability to initialize the state of the qubits to a simple fiducial state, such as $|000...\rangle$: Before a quantum computation can begin, the qubits need to be initialized to a known state. This ensures that the computation starts

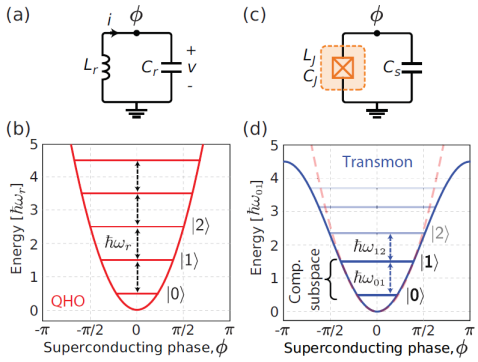
from a predictable configuration, which is essential for both practical operations and error correction protocols.

- Long relevant decoherence times, much longer than the gate operation time: Decoherence, the process by which a quantum system loses its quantum properties due to interactions with its environment, must be minimized. The coherence time of the qubits should be significantly longer than the time required to perform quantum gate operations, allowing quantum information to be preserved throughout the computation.

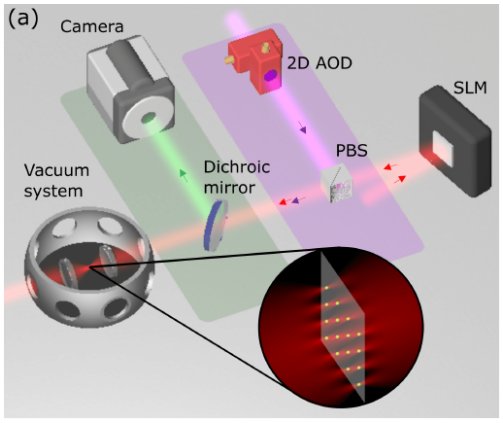
- A universal set of quantum gates: To perform arbitrary quantum computations, it is necessary to have a set of quantum gates that can be combined to implement any quantum algorithm. These gates must operate with high precision and reliability to ensure the accurate manipulation of quantum states.

- Efficient qubit-specific measurement: Finally, the ability to measure the state of individual qubits accurately and efficiently is crucial. Measurements allow the extraction of computational results and the implementation of error correction protocols, which are essential for maintaining the integrity of quantum information.

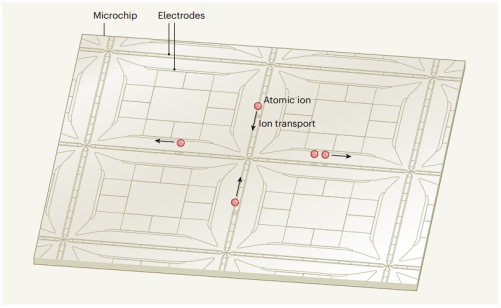
Guided by these foundational principles, quantum computing has ventured into varied experimental realms. Each method offers distinct benefits and challenges, and they adhere to DiVincenzo's criteria to different degrees. As the field has matured, researchers have explored several physical implementations for building prototype quantum computers. These include superconducting circuits [15], trapped ions [16], photonic systems [17], and, more recently, Rydberg atoms [18]. Such innovations have marked significant achievements, notably achieving quantum supremacy, where a quantum computer surpasses the best classical supercomputers at specific tasks.



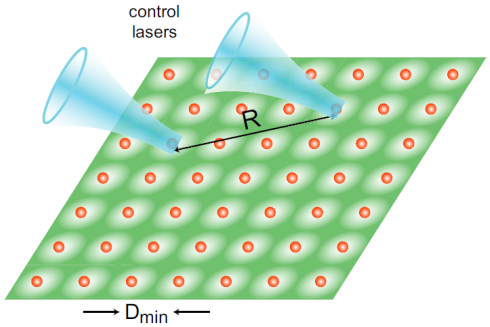
(a) Superconducting Circuits [15]



(b) Photonic Systems [17]



(c) Trapped Ions [17]



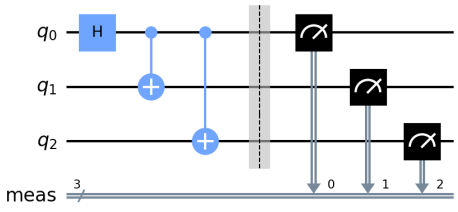
(d) Rydberg Atoms[19]

Figure 2.1: *Different physical implementations of quantum computers.*

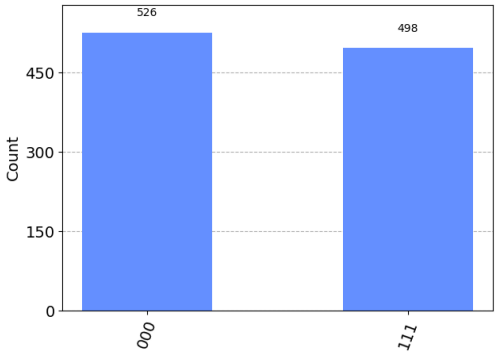
2.3 Qiskit Codes and QASM Files

Qiskit, developed by IBM, is an open-source framework specifically designed for interacting with quantum computers at the levels of pulses, circuits, and algorithms [20]. To illustrate Qiskit's workflow, we present an example of a quantum circuit designed to generate a GHZ state [21], accompanied by the simulation results of its measurement, as depicted in Figure 2.2. A GHZ state of N qubits is defined as:

$$|\text{GHZ}\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle^{\otimes N} + |1\rangle^{\otimes N}\right), \tag{2.3}$$



(a) GHZ Circuits with measurements



(b) Simulation Histogram

Figure 2.2: *Visualizations of the GHZ state quantum circuit and simulation results. They were here obtained by using the code provided in Appendix C,*

Alongside these tools, Quantum Assembly Language (QASM), particularly OpenQASM, serves as a hardware-agnostic platform for specifying quantum circuits [22]. It standardizes the notation for quantum operations, measurements, and control logic, enabling the sharing of circuit designs across various quantum platforms supported by Qiskit. Essentially, OpenQASM offers a textual representation of quantum circuits, such as the GHZ circuit described previously:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg meas[3];
h q[0];
cx q[0],q[1];
cx q[0],q[2];
barrier q[0],q[1],q[2];
measure q[0] -> meas[0];
measure q[1] -> meas[1];
measure q[2] -> meas[2];
```

Native Gates of IBM Quantum Hardware

Native gates are the fundamental operations that a quantum processor can perform directly without needing further decomposition. In IBM Quantum systems, the standard native gates generally include 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, as they directly influence the implementation and efficiency 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 within IBM Quantum systems, as detailed in the official IBM documentation [23]. To utilize R_y operations, they must be constructed through the synthesis of available native gates, predominantly R_z and X. This synthesis process is crucial because it significantly influences the complexity and fidelity of the operations executed on the quantum processor.

In the experimental section of this thesis, we focus on testing the reverse engineering process involving R_y gates, both pre and post-decomposition. Quantum circuits containing decomposed R_y gates may exhibit enhanced efficiency on specific quantum platforms. However, these decomposed forms often suffer from reduced readability compared to their original configurations. Despite performing identical functions as their undecomposed counterparts, these circuits are considered to have lower "readability," complicating the identification of the quantum algorithms they implement.

Version of August 21, 2024– Created August 21, 2024 - 06:42

Chapter 3

Background

> The computing scientist's main challenge is not to get confused by the complexities of his own making.
>
> Edsger Dijkstra

The quest to decipher and simplify the intrinsic complexities of computational systems extends from the classical to the quantum realm. While classical computing has developed sophisticated methods like decompilation to translate machine code back to higher-level programming languages, quantum computing faces similar challenges. The process of decompiling quantum circuits, particularly starting from their QASM representations to uncover underlying patterns and the original high-level Qiskit code, is crucial for advancing our understanding of quantum algorithms and improving quantum architecture designs. In this chapter, we will explore the concept of "decompilation." The first section delves into what decompilation entails in the realm of classical computing. Subsequently, the second section introduces quantum compilers and discusses their role in quantum computing. This is followed by an examination of various approaches to searching for optimal quantum circuit architectures. Finally, we conclude with a discussion on similar reverse engineering efforts applied to quantum circuits.

3.1 Classical Decompiler

Decompilation is a critical process in computer science and software engineering, essential for understanding and analyzing compiled code. At its core, decompilation involves translating machine code or low-level intermediate representations back into higher-level programming languages that are more readable and understandable by humans, just as Figure 3.1 shows. This reverse engineering process is invaluable for various applications, including software debugging, optimization, security analysis, and intellectual property protection. [24].

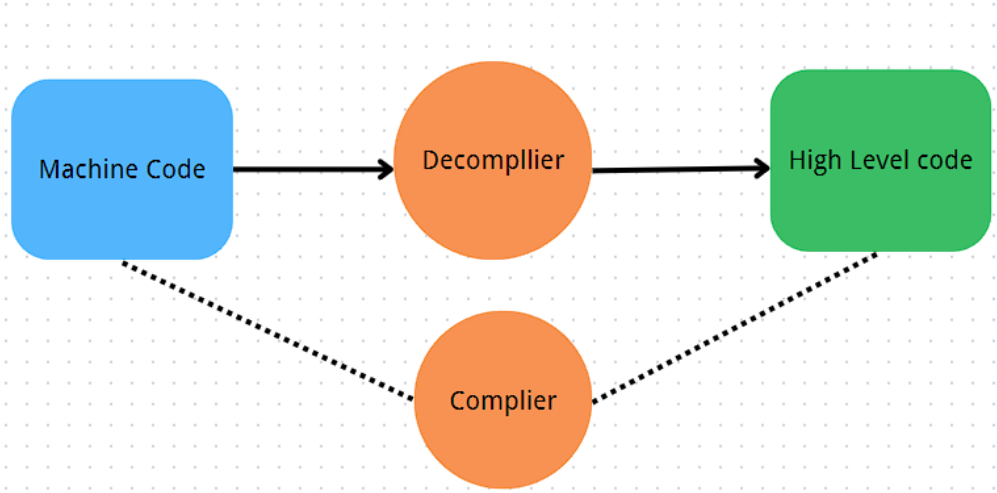


Figure 3.1: *A schema of how Decompilation works*

One of the primary motivations for decompilation is the need to understand the inner workings of software systems, especially when the source code is unavailable. This situation often arises in legacy systems where the original source code has been lost or when working with third-party software. Decompilation allows developers and engineers to recover the higher-level structure and logic of a program, facilitating maintenance, updates, and integration with new systems [25].

In the context of security, decompilation is a powerful tool for vulnerability assessment and malware analysis. By decompiling malicious code, security experts can uncover the strategies and mechanisms employed by attackers, enabling the development of effective countermeasures. This process also aids in the verification of software to ensure it adheres to security standards and does not contain hidden backdoors or unauthorized functionalities [25, 26].

The complexity of decompilation varies depending on the target architecture

and the optimization level of the compiled code. High-level languages with rich syntax and semantics pose significant challenges for decompilers, requiring sophisticated analysis techniques to reconstruct the original code accurately. Advances in artificial intelligence and machine learning have further enhanced decompilation capabilities, enabling more precise and automated recovery of high-level code structures [27].

Several methods are employed in decompilation to tackle these challenges. Pattern matching is a common technique where known code patterns are identified in the binary and replaced with their high-level equivalents. Data flow analysis helps in understanding how data moves through the program, which is essential for reconstructing the logic and control flow. Control flow analysis, on the other hand, focuses on the program's structure by identifying loops, conditionals, and other control structures. These methods are often combined to improve the accuracy and efficiency of the decompilation process [28].

There is a very similar work to quantum circuit decompilation, Hardware Decompilation, which also tries to find some underlying pattern from the hardware circuits, particularly through techniques like Hardware Loop Rerolling [29], plays a critical role. This method focuses on identifying repetitive logic within netlists and transforming these patterns back into loop structures in higher-level Hardware Description Language (HDL) code. Figures 3.2 and 3.3 illustrate these concepts. Figure 3.2 displays the detailed structure of a hardware module processed via HDL, while Figure 3.3 shows the transformation of netlist patterns into loops, exemplifying the loop rerolling process.

```
module ripple_carry_adder #(parameter N)
 (input [N-1:0] a, input [N-1:0] b, output logic
      [N-1:0] sum);
 logic cin, cout;
 always_comb begin
   cin = 1'b0;
   for (int i=0; i < N; i++) begin
     sum[i] = a[i] ^ b[i] ^ cin;
     cout = a[i] & b[i] | a[i] & cin | b[i] & cin;
     cin = cout;
   end
 end
endmodule
```

```
module accumulator (input [3:0] x, input clk, output
      reg [3:0] acc);
  logic [3:0] sum;
  ripple_carry_adder #(.N(4)) adder(acc, x, sum);
  always @(posedge clk) begin
    acc <= sum;
  end
endmodule
```

Figure 3.2: *A hardware module represented in HDL. [29]*

A practical example of decompilation is its use in recovering lost source code. Imagine a company that has a legacy software system critical to its operations, but the original source code has been lost due to poor archival practices. By decompiling the binary executables, the company can recover a high-level representation of the software, which can then be maintained and updated. This

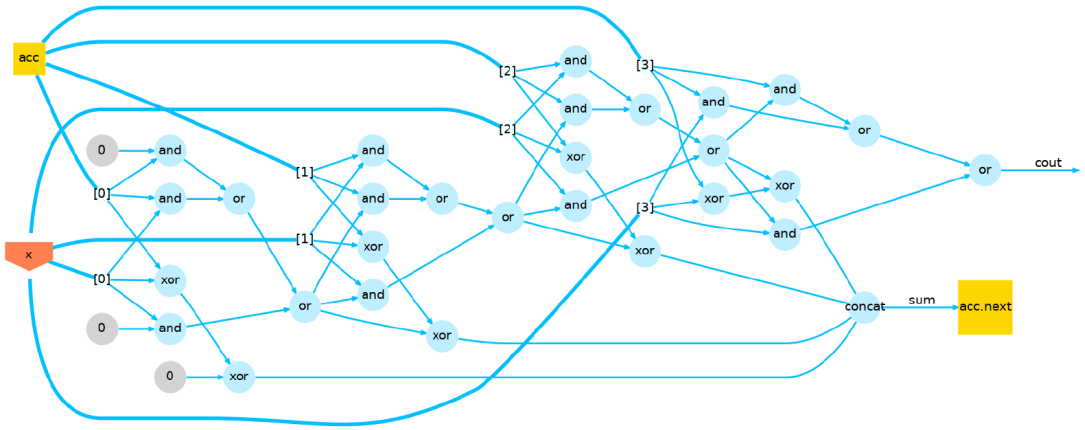


Figure 3.3: *Illustration of the loop rerolling process applied to netlists. [29]*

recovered code, while not identical to the original source code, provides a functional and understandable version that developers can work on [30].

Another use case is in malware analysis. Security researchers often encounter malware in binary form, with no access to the source code. Decompilation allows them to translate the binary back into a high-level language, revealing the malware's behaviour and functionalities. This process is crucial for developing anti-malware strategies and understanding how the malware interacts with systems to exploit vulnerabilities [31]. Recently, advancements in Large Language Models (LLM) have spurred research into their application for optimizing decompilers, such as the DeGPT model [32]. This innovative approach enhances the readability and utility of decompiled code through a structured workflow that maintains semantic integrity.

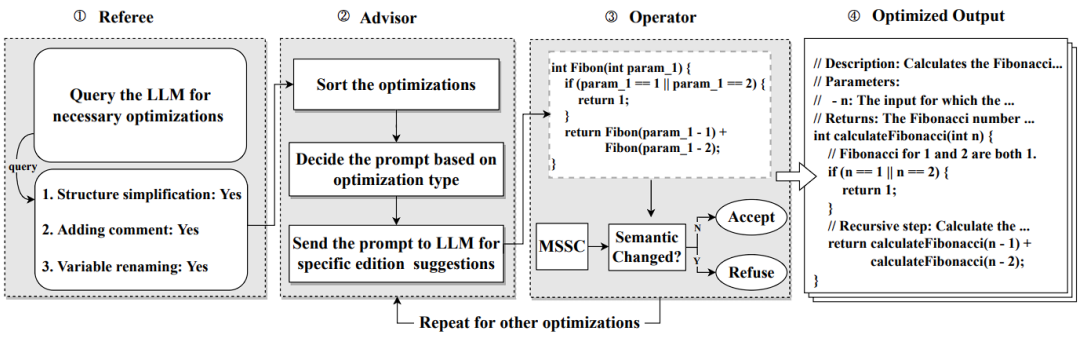


Figure 3.4: *Workflow of the DeGPT model showing the integration of LLMs to optimize decompiler outputs.[32]*

Figure 3.4 visually encapsulates the workflow of the DeGPT system, illustrating how each component of the framework interacts to refine the output of decompilers.

3.2 Quantum Compiler

Constructing a fully programmable quantum computer based on the circuit model, a system stack composed of several layers is required. [33] One of the structures propossed so far is shown in Figure 3.5

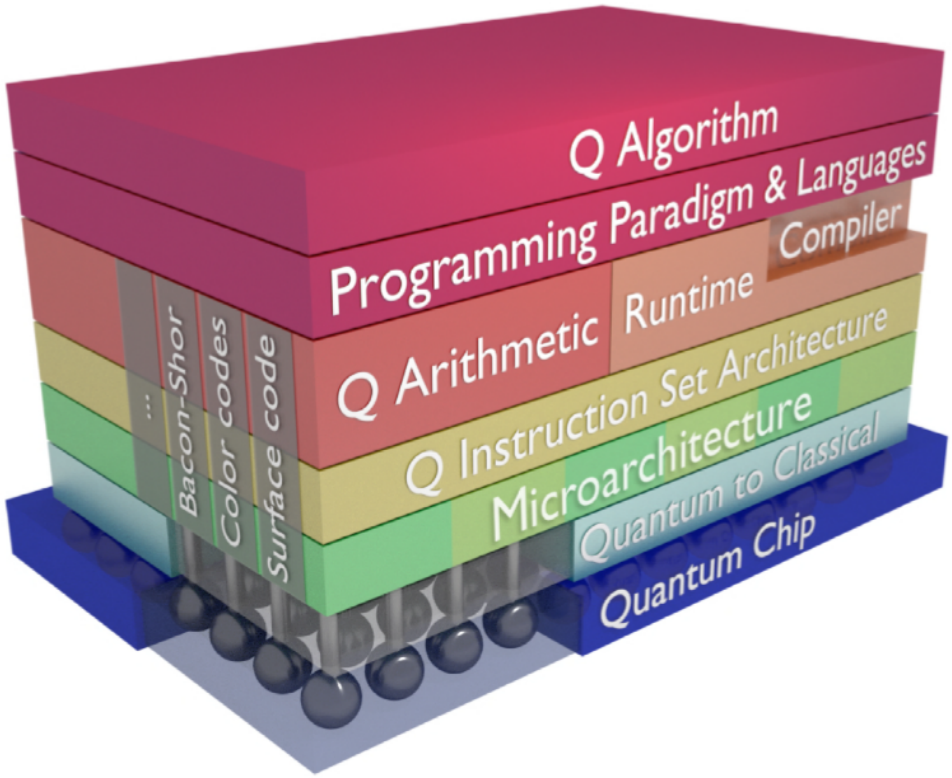


Figure 3.5: *One type of full stack Quantum Computer Architecture introduced by [34]. Starting from the bottom, it includes the quantum chip (physical qubits), the quantum-classical interface (ADCs, DACs, and controls), microarchitecture (timing controls and instruction pipelines), quantum instruction set architecture (runtime operations), quantum runtime unit (scheduling and error correction), compiler and programming language (high-level abstraction), and quantum algorithm descriptions (task-specific designs for the compiler). This comprehensive stack is crucial for interfacing quantum processors with algorithmic descriptions.*

Among these layers, quantum compilers play a crucial role. They translate high-level quantum algorithms into low-level instructions that can be executed on quantum hardware. These compilers optimize quantum circuits to reduce errors, improve fidelity, and ensure efficient use of quantum resources.

Quantum compilers are akin to classical compilers in their function. In classical computing, a compiler translates high-level programming languages (like C++ or Python) into machine code that a computer's processor can execute. Similarly, quantum compilers convert high-level quantum algorithms into quantum gate sequences that quantum processors can understand and execute. The key components of quantum compilers are as follows:

- **Decomposition**: This involves breaking down high-level quantum operations into a sequence of elementary gates that can be executed on a quantum computer. This step ensures that complex quantum gates are expressed in terms of a universal gate set (e.g., Clifford+T). For instance, a SWAP gate can be decomposed into three CNOT gates.

- **Optimization**: Optimization in quantum compilers aims to reduce the number of qubits (circuit width) and the number of gates (circuit depth) used in quantum circuits. High-level optimizations might involve simplifying sequences of gates, such as cancelling out consecutive inverse operations. Low-level optimizations could include minimizing the number of required quantum operations after decomposition.

- **Scheduling**: Scheduling ensures that quantum operations are executed in an order that respects the dependencies between them. This process generates a Quantum Instruction Dependency Graph (QIDG), which helps in determining the optimal sequence of operations to minimize the overall execution time while considering qubit connectivity and gate times.

- **Mapping**: Mapping involves assigning logical qubits in the quantum algorithm to physical qubits on the quantum hardware. This step must account for the physical layout and connectivity of qubits to ensure efficient execution. For example, a SWAP operation might be necessary if the physical qubits required by a CNOT gate are not directly connected.

- **Fault-Tolerant Synthesis**: This component focuses on creating fault-tolerant quantum circuits that can operate reliably despite the presence of errors.

- **Physical Realizations**: The compilation process needs to adapt to various physical implementations of quantum computers.

The challenges faced by quantum compilers are distinct and complex due to the fundamental principles of quantum mechanics. Quantum compilers must address quantum-specific issues such as gate fidelity, qubit connectivity, decoherence, and error correction. These aspects necessitate advanced techniques to ensure the effective execution of quantum programs on quantum hardware.

3.3 Quantum Architecture Search

Quantum Architecture search (QAS) is an active field where researchers utilize heuristic algorithms to optimize quantum structures, primarily focusing on quantum circuits and QASM representations. Techniques employed encompass Bayesian Optimization with Weisfeiler-Lehman Kernels [35], metric-based quantum circuit optimizations [36], differentiable architecture [37] methods that integrate quantum circuit parameters, along with Reinforcement Learning (RL) [1–4] and Evolutionary Algorithms [5–7]. These methods, while effective in certain contexts, often yield quantum architectures that lack interpretability, presenting substantial challenges in generalizing these structures or finding the high-level code that generates them. This lack of interpretability highlights the critical need for reverse engineering of quantum circuits. By reverse compiling these obscurely discovered quantum architectures, we aim to uncover the underlying principles that guide their design, thereby enabling us to effectively scale or adapt these complex structures.

3.4 Reverse Engineering on Quantum circuits

Decompiling logic circuits presents challenges due to the significant abstraction gap between their physical realizations and high-level descriptions. Furthermore, optimizations during the synthesis process may significantly modify the circuit's structure, thus obscuring its original design intentions. Similarly, these challenges are evident in the reverse engineering of quantum circuits. As discussed in the previous section, the compilation by a quantum compiler is inherently complex. Our research does not aim to reverse engineer all components but focuses specifically on using QASM files as a starting point. Our goal is to reconstruct the Qiskit Python code that originally generated these QASM files. This process is similar to pattern recognition within quantum circuits, aiming to simplify complex quantum instructions back into their intuitive, high-level forms. For this QASM-to-Python conversion, there is limited related research available. One notable study that aligns closely with this topic is "Reverse Engineering of Classical-Quantum Programs" by Luis Jimenez-Navajas et

al. [38]. This research introduces a reverse engineering method that examines both quantum (Qiskit) and classical (Python) code, creating a unified abstract model that integrates classical and quantum components.

In this project, we utilize Abstract Syntax Trees (AST) to represent python code, providing a structured and hierarchical representation of the code that facilitates easier manipulation and analysis. Additionally, Genetic Algorithms are employed to optimize the synthesis process, leveraging evolutionary techniques to explore a vast search space and identify near-optimal solutions. For the AST and GA, we will introduce them in detail in Chapter 4.

The combination of these methods allows us to effectively decompile intricate quantum circuits and synthesize high-level programs that are both efficient and explainable. This approach not only aids in the refinement and adaptation of quantum algorithms but also contributes significantly to the broader field of quantum software engineering by promoting a deeper understanding and more robust utilization of quantum computational resources. This process is essential for optimizing and adapting quantum circuits to different quantum hardware platforms, ensuring that the implementations are not only efficient but also maintainable and scalable [39].

There are various efforts in the realm of optimizing quantum circuits and exploring reverse engineering and pattern recognition within quantum systems. Techniques such as AlphaTensor have shown significant promise in enhancing circuit performance by identifying and implementing optimal transformations of quantum gates [40]. In a similar way, the application of genetic algorithms for quantum circuit optimization explores vast search spaces to identify nearly optimal designs [41].

Research into the automatic deobfuscation of executable code and program synthesis for identifying quantum circuit components offers valuable insights into simplifying complex code into more understandable elements [42, 43]. These strategies are adaptable for quantum decompilation, enhancing the comprehensibility and optimization of quantum circuits.

Moreover, quantum pattern recognition, employed in photonic circuits and real quantum processing units, showcases the practical utility of quantum algorithms in discerning intricate patterns and structures, underscoring the importance of advanced quantum program synthesis and decompilation techniques [43, 44].

Integrating these methodologies, our project seeks to push the boundaries of quantum decompiling and program synthesis, aiming to boost the efficiency, scalability, and clarity of quantum computing systems.

Chapter 4

Methods

> The essence of a breakthrough is not a matter of knowledge, but a matter of thinking differently.
>
> Albert Einstein

In this chapter, the methods related to decompiling quantum circuits are described, including using Abstract Syntax Trees (AST) to initialize qiskit code for quantum circuit generation and employing Genetic Algorithms to optimize the problem.

4.1 Abstract Syntax Tree

Abstract Syntax Trees (AST) play a pivotal role in the decompilation and synthesis of quantum circuits. An AST provides a tree-like representation of the abstract syntactic structure of source code written in a programming language. Each node in the tree denotes a construct occurring in the source code. The hierarchical nature of ASTs allows for efficient parsing, manipulation, and analysis of the code, making them an invaluable tool in the reverse engineering process.

4.1.1 Structure of AST

The structure of an AST consists of various types of nodes, each representing a different element or construct in the source code. Here, we introduce the basic concepts and methods used to create and manipulate AST nodes.

Nodes in AST:

AST nodes are categorized based on the type of construct they represent. Some of the common node types include:

- **Module Node**: The root of the AST, representing the entire source file. For example, it includes all functions, classes, and statements in a Python script.
- **FunctionDef Node**: Represents a function definition. For instance,

```
def add(a, b):
    return a + b
```

- **Assign Node** and **Constant Node**: The **Assign Node** represents an assignment operation where a variable is assigned a value, while the **Constant Node** specifically denotes the immutable value in the assignment. For example:

```
x = 10
```

  In this case, 'x = 10' features an 'Assign Node' linking the variable 'x' to a 'Constant Node' representing the value '10'.

- **Expr Node**: Represents an expression. Example:

```
print("Hello World")
```

- **Call Node**: Represents a function call. Example:

```
sum([1, 2, 3])
```

- **BinOp Node**: Represents a binary operation (e.g., addition, subtraction). Example:

```
a + b
```

- **Name Node**: Represents a variable name. Example: In 'x = 5', 'x' is a *Name* node.

- **For Node**: Represents a for loop. Example:

```
for i in range(10):
    print(i)
```

Basic Methods in AST:

The following methods are commonly used to work with ASTs:

- **ast.parse(source)**: Parses the source code into an AST.
- **ast.NodeVisitor**: A base class that walks the abstract syntax tree.
- **ast.NodeTransformer**: A base class for modifying an abstract syntax tree.
- **ast.dump(node)**: Returns a formatted string of the AST node.

Example: Creating and Analyzing an AST The following Python code 4.1 demonstrates how to create an AST for a simple function and analyze its structure. The example plot of AST can be seen in Figure 4.1

```
source_code = """
def add(a, b):
    result = a + b
    return result
"""
# Parse the source code into an AST
parsed_ast = ast.parse(source_code)
show_ast(parsed_ast)
```

Listing 4.1: *Creating and Analyzing an AST*

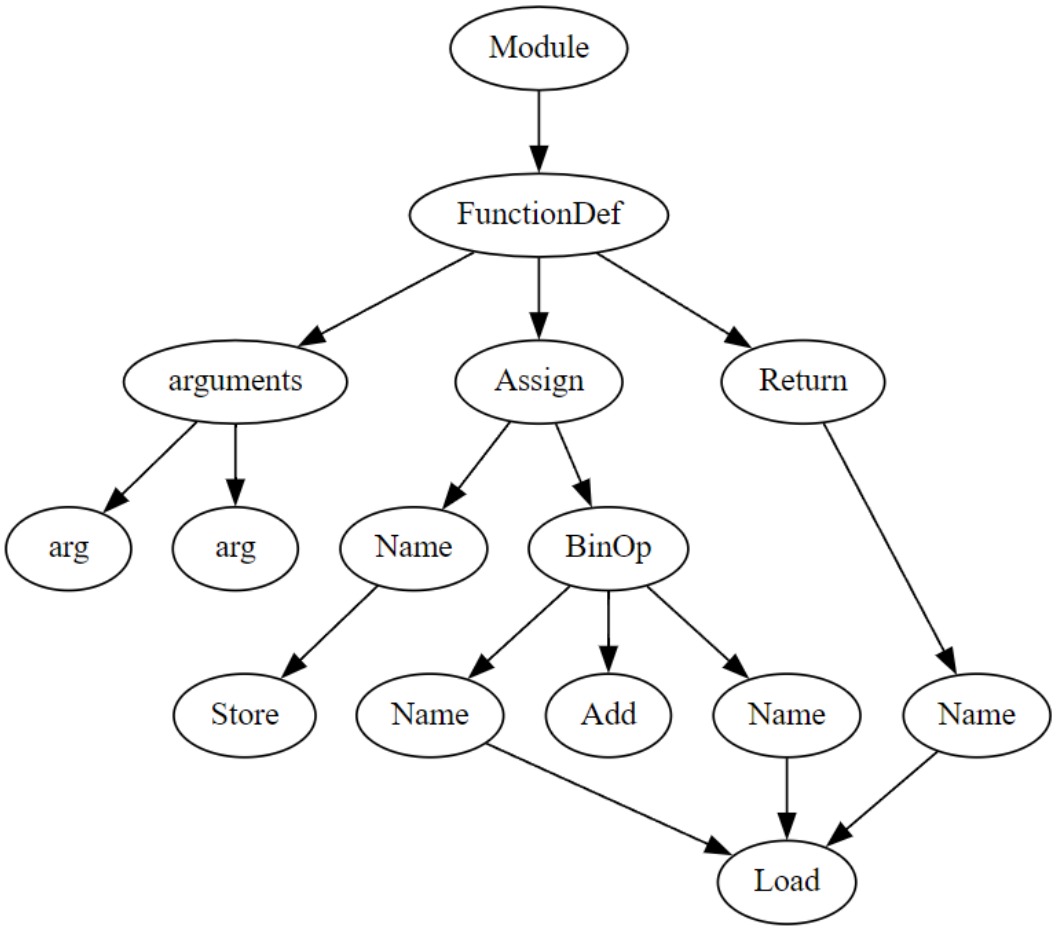


Figure 4.1: *An AST for a simple function* `add`*. The figure shows the hierarchy of nodes, with* `Module` *as the root representing the entire source code. The* `FunctionDef` *node represents the function definition for* `add`*. The* `arguments` *node represents the function parameters* `a` *and* `b`*. The* `Assign` *node represents the assignment statement* `result = a + b`*, where* `Name` *nodes represent the variables, and the* `BinOp` *node represents the binary operation* `+`*. The* `Return` *node (Load) represents the return statement* `return result`*, where the* `Name` *node under it represents the variable* `result`*.*

In this example, the AST is created by parsing a simple function using the `ast.parse` method. The structure of the AST is then printed using the `ast.dump` method. A function `showast` is also defined to traverse and print the types of nodes in the AST. The corresponding figure illustrates the hierarchy of nodes in the AST.

4.2 Applying AST to Qiskit code

By leveraging the hierarchical and modular nature of ASTs, we can efficiently represent, analyze, and manipulate qiskit code which is used to generate quantum circuits. The following sections will detail how ASTs can be utilized for various tasks such as generating random qubit index expressions, creating loop structures, and constructing quantum gate calls.

4.2.1 AST Representation for Qiskit

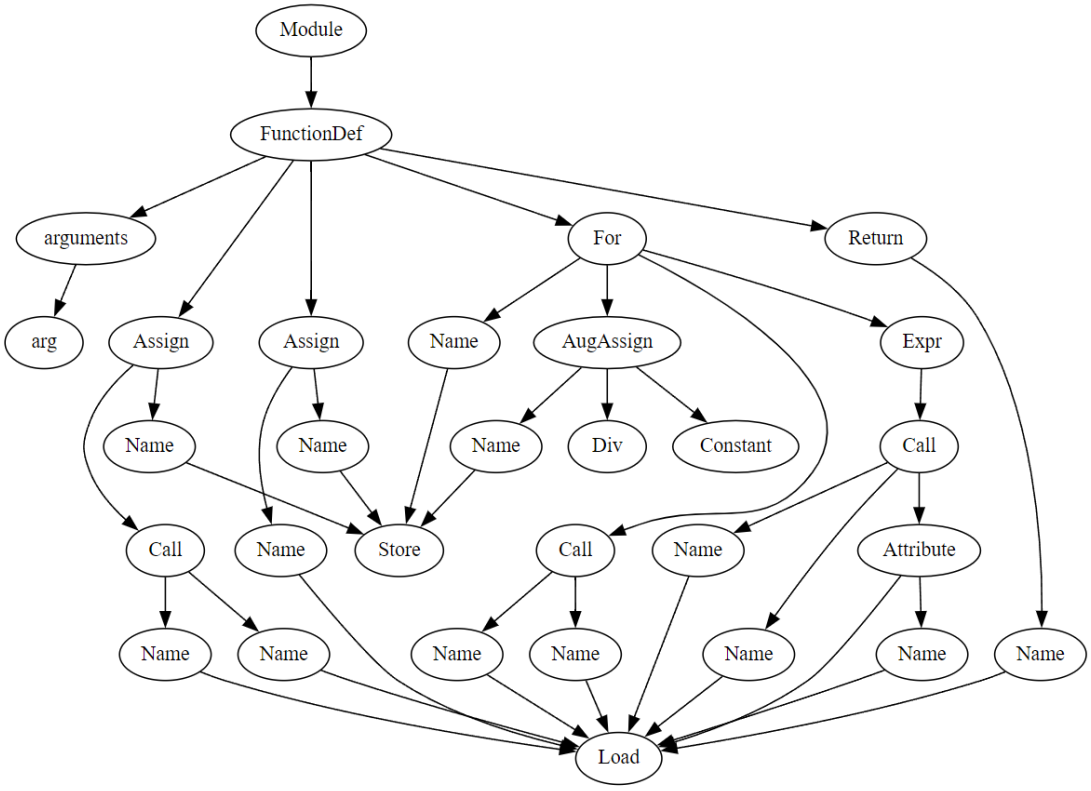


Figure 4.2: *AST structure for a simple qiskit function*

In the context of quantum circuit decompilation, ASTs are utilized to initialize the qiskit code for a certain series of quantum circuits. This structured representation captures the essential components and their relationships within the quantum algorithm. By converting the low-level quantum gate operations into

a high-level AST representation, we can more easily identify patterns, optimize the circuit, and translate it back into high-level code.

Figure 4.2 represents the Abstract Syntax Tree (AST) for the function `rx_c` 4.2. This function initializes a quantum circuit with a given number of qubits, applies a series of `rx` rotations with decreasing angles, and returns the constructed quantum 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 4.2: *qiskit Example for a rx_c circuit*

The AST for the `rx_c` function is structured as follows:

- **Module**: The root of the AST.
- **FunctionDef**: Represents the function definition `rx_c`.
  - **arguments**: Represents the function arguments, including `n`.
  - **Assign**: Represents assignment statements such as `qc = QuantumCircuit(n)` and `angle = pi`.
  - **For**: Represents the for loop `for i in range(n)`.
    - **Expr**: Represents the expression `qc.rx(angle, i)`.
    - **AugAssign**: Represents the augmented assignment `angle /= 2`. The AugAssign node in AST represents augmented assignment statements in programming. These statements combine a binary operation with an assignment operation. In simpler terms, augmented assignments are shorthand operations that update the value of a variable using operators like +=, -=, *=, /=, etc.
  - **Load**: Represents the return statement `return qc`.

4.2.2 Initialization of Qiskit Code using AST

The initialization of Qiskit code using ASTs allows us to generate and manipulate quantum circuits programmatically. This section details the process and the functions involved in creating Qiskit code for random quantum circuits using ASTs.

- **Generating arbitrary Qubit Index Expressions:** This involves creating random expressions for qubit indices using arithmetic operations, modulus, and simple variables. The function `random_expr(depth, max_expr_operators, var_depth)` generates an arbitrary expression with a specified depth, number of binary operations, and additional variables. The parameters are:
  - **depth** d: Number of loop variables.
  - **max_expr_operators**: Number of binary operations to perform.
  - **var_depth**: Number of additional variables.

  Example Function Call and Output:
  - **Input:** `random_expr(0, 1, 2)`
  - **Output:** The expression could be a simple operation like $n - n - n$, where n is a variable or index.
  - **Input:** `random_expr(1, 2, 3)`
  - **Output:** This might generate a more complex expression such as $i0 - n + 3 + 2$, involving loop variables ($i0$), constants (3, 2), and operations.
- **Creating Loop Structures:** Loops are crucial in quantum algorithms for performing repeated operations. The `loop_index` function is designed to generate loop indices that facilitate complex loop structures. These loops are integral in quantum circuit generation, 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 4.3: *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:** Dependent on `i0`, `i1` ranges dynamically, creating a dependency that increases 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.

This example demonstrates how nested loops are used to dynamically adjust quantum operations, crucial for complex quantum algorithms that require precise control over multiple qubits.

- **Creating Phase Expressions for Phase-Related Gates:** For quantum gates that involve phases (such as `rx`, `ry`, `rz`), the `random_phase_expr` function generates a arbitrary phase expression. This function creates an expression of the form:

$$expr_{phase} = (\pi \cdot \frac{1}{2^a + b + c}) \tag{4.1}$$

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 Gaussion Distribution

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

. The function `random_phase_expr(depth:)` is used to create Phase Expression, where the only input is the depth for the current loop location since it needs to decide which symbol is included for the expression. To have a better understanding of how it works, we can show an example in the following:
Example Function call and Output:

  - **Input:** `random_phase_expr(2)`
  - **Output:** `pi * (1 / (2 ** (n + 0 + n - 0) + (i0 + 0 + 0 - n + 0)))`

- **Constructing Quantum Gate Calls:** The `generate_gate_call` function constructs calls to quantum gates, accommodating single, multi-qubit, and rotational gates. It uses random expressions for qubit indices and phases, incorporating the generated loops and expressions to define where and how the gates are applied.
The example usage of the function `generate_gate_call(depth: Any, gate: Any)` is given as Table 4.1. It takes the current loop depth and a specific gate type as inputs:

Function Call	Quantum Gate Operation
`generate_gate_call(2, 'h')`	`qc.h((i1 - 0 - n) % n)`
`generate_gate_call(1, 'rx')`	`qc.rx(pi * (1 / (2 ** (i0 + 0 - n) + (i0 - n + n + 0))), (n - 0) % n)`
`generate_gate_call(1, 'cx')`	`qc.cx((n - 0 + n) % n, (i0 + n - 0) % n)`
`generate_gate_call(1, 'cp')`	`qc.cp(-(pi * (1 / (2 ** (n - 0 - n) + (n - n + n + 2)))), (i0 + 0) % n, (n - 0 - n) % n)`

Table 4.1: *Examples of generating quantum gate calls using `generate_gate_call` function.*

- **Assembling the Circuit:** The `generate_random_circuit_ast` function brings all these components together, generating a complete quantum circuit. It defines a function that initializes a quantum circuit, iterates through nodes to add gate operations, and returns the constructed circuit.*. The example usage of function `generate_random_circuit_ast(num_nodes, operations, max_loop_depth)` is listed in the following:
  - **Input:** `generate_random_circuit_ast(4, operations, 3)`,here `operations` is a group of pre-defined quantum operations
  - `Output:`

```
1  def generate_random_circuit_ast(n):
2      qc = QuantumCircuit(n)
3      for i0 in range(n):
4          for i1 in range(abs(n + n - 1)):
5              qc.u3(pi * (1 / (2 ** (n - 0 - 0 + n) + (i0 - 0
                   + 0 - 0 + 1))), (i1 - 0 - n) % n)
6              qc.u3(-(pi * (1 / (2 ** (n - n - 0 + 0) + (i1 -
                   n - 0 - n + 1)))), (i0 + 0 + n) % n)
7              qc.u3(-(pi * (1 / (2 ** (n + 0 - n - n) + (i1 +
                   0 + n - 0 + 0)))), (i1 - n) % n)
8      qc.cu1(pi * (1 / (2 ** (n - 0) + (n + 0 + 0))), (n - 0)
           % n)
9      qc.cu1(pi * (1 / (2 ** (n + n) + (n + n + 0))), (n - 0)
           % n)
10     return qc
```

Listing 4.4: *An example result for the function call of generate_random_circuit_ast*

*detail of this part can be seen in Appendix A

Version of August 21, 2024– Created August 21, 2024 - 06:42

4.3 Genetic Programming

Genetic Programming (GP) is an evolutionary algorithm-based methodology inspired by biological evolution to find approximate solutions to optimization and search problems. In the context of program synthesis, GP evolves computer programs to solve specific tasks, starting from an initial population of random programs and iteratively applying genetic operations such as selection, crossover, and mutation.

Recent research has demonstrated the efficacy of GP in various aspects of program synthesis. For instance, Banzhaf et al. (1998) [45] highlighted the application of GP in evolving algorithms that perform specific computational tasks. Similarly, Olsson (1995) [46] explored the use of GP for automatic program induction, showcasing its potential in generating and optimizing complex software systems. More recent advancements by Forstenlechner et al. (2017) [47] and Krawiec and O'Reilly (2014) [48] further refined these techniques, enabling more efficient and effective program synthesis across diverse domains.

In this thesis, we utilize GP to optimize the decompilation quantum circuits and synthesis of qiskit codes. The primary goal is to bridge the gap between high-level quantum algorithms and their low-level circuit implementations.

4.3.1 Baseline of Genetic Programming

The Genetic Programming (GP) methodology is characterized by the following steps:

- **Initialization**: Generating an initial population .
- **Selection**: Choosing the fittest individuals from the population based on their performance.
- **Crossover**: Combining parts of two or more parent programs to produce offspring.
- **Mutation**: Making random changes to a program to explore the search space.
- **Evaluation**: Assessing the fitness of each candidate program based on a predefined objective.

The following algorithm outlines the baseline genetic programming process and an example of how GP works is shown in Figure 4.3:

Algorithm 1 Baseline Genetic Programming Algorithm

1: Initialize population P with random individuals
2: **for** each generation g **do**
3: Evaluate the fitness of each program in P
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

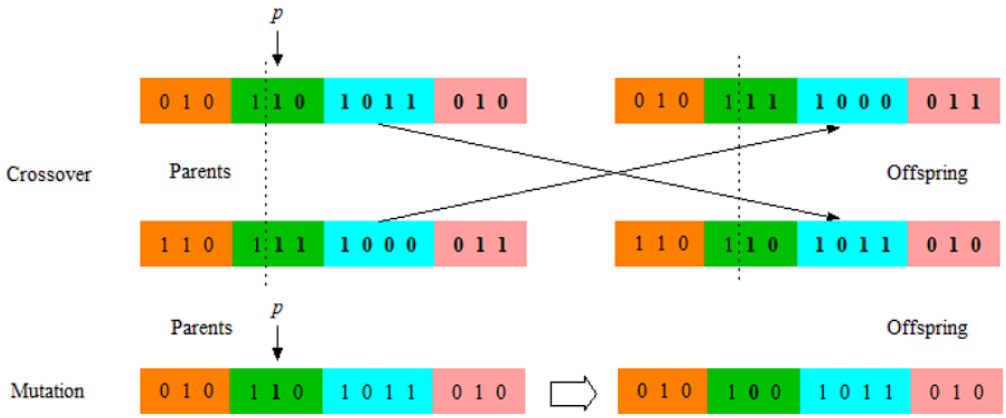


Figure 4.3: *An example of GP applying Crossover and Mutation on Binary digits [49]*

4.3.2 Implementation of Genetic Decompiler

In this project, we employ Genetic Programming (GP) to evolve the Abstract Syntax Tree (AST) model-generated quantum circuits. The fitness of each individual in the population is evaluated based on the similarity between the generated QASM files and the target QASM files. This similarity serves as the evaluation criterion for the population, guiding the evolutionary process toward optimal solutions.

The implementation of the genetic decompiler involves several key components, including the initialization of the population, the application of genetic operators (mutation and crossover), the selection of parents, and the evaluation of the fitness of each candidate solution. Below is a detailed description of each

step, followed by a pseudocode representation of the genetic decompiler algorithm. The hyperparameter of the overall genetic decompiler can be seen in Table 4.2

Table 4.2: *Hyperparameters for the genetic decompiler.*

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.

Initialization The population is initialized with random candidate solutions, each representing a qiskit code encoded as an Abstract Syntax Tree (AST). The function `generate_initial_population` creates an initial population of a specified size.

To calculate the parameter space size for generating a random qiskit code, we consider the parameters used in the `generate_random_circuit_ast` function and its called functions. The main parameters include:

- **num_nodes**: The number of nodes (or gates) in the qiskit code.
- **operations**: The list of possible quantum gates (e.g., `['h', 'x', 'cx']`).
- **max_loop_depth**: The maximum depth of nested loops in the circuit.

For the functions called within `generate_random_circuit_ast`, we consider:

- `random_expr`:
  - **depth**: Number of loop variables.
  - **max_expr_operators**: Number of binary operations to perform.
  - **var_depth**: Number of additional variables.
- `random_qubit_expr`:
  - **expr**: The expression for the qubit index.
- `random_phase_expr`:
  - **depth**: Number of loop variables.
- `loop_index`:
  - **depth**: Depth of the loop.

Assuming typical values for these parameters, such as **num_nodes** = 10, **operations** = ['h', 'x', 'cx'] (3 possible operations), **max_loop_depth** = 2, **depth** = 2, **max_expr_operators** = 3, and **var_depth** = 2, we can estimate the total parameter space size. The number of possible qiskit code is influenced by the combinations of these parameters:

$$\text{Total Parameter Space} \approx 3^{10} \times 2^2 \times 10^5 \approx 2.36 \times 10^9$$

This large space is what the genetic algorithm explores to evolve and optimize qiskit code. The vastness of the parameter space highlights the necessity of using genetic programming. This approach efficiently navigates the complex and large search space, enabling the decompiler to approximate the ground truth, which is the actual qiskit code used to create quantum circuits.

Mutation Mutation is a crucial genetic operator that introduces diversity into the population by making random changes to candidate solutions. In the genetic decompiler, we employ two types of mutation methods: insertion and modification.

Insertion:

In the insertion mutation, a piece of qiskit code creating a new quantum gate is added to the quantum circuit at a randomly selected position.

Modification:

In the modification mutation, an existing qiskit code for generating quantum gates in the circuit is replaced with a new one. These mutation methods ensure diversity within the population of quantum circuits, enabling the genetic algorithm to explore a broader search space and avoid local optima.

Crossover The crossover operator in genetic programming combines parts of two parents AST to produce offspring by exchanging genetic material for potentially better solutions. Specifically, it involves randomly selecting one node from each of two parent AST, splitting each AST at the selected node into two fragments, and then recombining these fragments in a new arrangement to create two new offspring ASTs. This process blends features from both parents, aiming to generate offspring with improved or desired characteristics by introducing novel structural combinations. A simple case can be seen in the following listing 4.4, where " − " lines represent splitting for parent codes, while" ∗ " lines represent the position children codes crossover.

```
# Parent 1
qc.h((n - 1) % n)
---------------- split
for i0 in range(n):
    qc.h((n - 1 - i0) % n)
    qc.h((i0 - 1) % n)
```

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

```
# Child 1
qc.h((n - 0) % n)
**************** Crossover
for i0 in range(n):
    qc.x((i0 + n + 1) % n)
    qc.x((i0 - n) % n)
```

```
# Child 2
qc.h((n - 1) % n)
**************** Crossover
for i0 in range(n):
    qc.h((n - 1 - i0) % n)
    qc.h((i0 - 1) % n)
```

Figure 4.4: *An Example of Crossover on qiskit codes*

Selection The selection process involves choosing the fittest individuals from the population to serve as parents for the next generation. Several selection methods are implemented, including tournament selection, roulette wheel selection, and rank selection.

Roulette Wheel Selection:

- Calculates the total fitness of the population.
- Assigns a selection probability to each individual based on their fitness.
- Randomly selects parents according to these probabilities, favouring higher fitness individuals.

The probability P_i of selecting an individual i is given by:

$$P_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

where f_i is the fitness of individual i and N is the population size.

Tournament Selection:

- Randomly selects a subset of individuals from the population.
- Chooses the individual with the highest fitness from this subset as a parent.
- Repeats the process of selecting the second parent.

Rank Selection:

- Ranks the population based on fitness.
- Assigns selection probabilities based on ranks, with higher-ranked individuals having higher probabilities.
- Selects parents based on these probabilities.

If r_i is the rank of individual i (with 1 being the highest rank), the probability P_i of selecting individual i is:

$$P_i = \frac{2(r_i - 1)}{N(N-1)}$$

where N is the population size.

Weighted Roulette Wheel Selection:

- Similar to roulette wheel selection but applies a weighting factor to fitness scores.

- Increases the likelihood of selecting higher fitness individuals more aggressively.

The weighted probability P_i of selecting an individual i is given by:

$$P_i = \frac{f_i^w}{\sum_{j=1}^{N} f_j^w}$$

where f_i is the fitness of individual i, N is the population size, and w is the weighting factor.

These parent selection methods ensure diversity and maintain the evolutionary pressure towards better solutions. By combining the strengths of different individuals, crossover operators play a crucial role in evolving high-quality quantum circuits.

Evaluation Ton evaluate the fitness of each single quantum circuit compared to the target circuit, other than the typical way to compare two quantum circuits(Process Fidelity), we invent a set of evaluation metrics which are inspired by comparing two text documents in NLP(Natural Language Processing).

Process Fidelity:

- Measures how closely two unitary matrices (representing quantum circuits) align.
- Given two unitary matrices U and V, the process fidelity F is defined as:

$$F(U, V) = \left| \frac{\mathrm{Tr}(U^\dagger V)}{N} \right|^2$$

  where Tr denotes the trace, $U^\dagger$ is the conjugate transpose of U, and N is the dimension of the unitary matrices.

Gate Sequence Similarity:

- Compares the sequences of quantum gates applied in two circuits.
- Uses Levenshtein distance [50] to measure the similarity between two gate sequences. The Levenshtein distance is a string metric for measuring the difference between two sequences by counting the minimum number of operations required to transform one sequence into the other. These operations include insertions, deletions, or substitutions of single characters.

Version of August 21, 2024– Created August 21, 2024 - 06:42

- The similarity score S between two sequences A and B is:

$$S(A,B) = 1 - \sqrt{\frac{D(A,B)}{\max(|A|,|B|)}}) \tag{4.2}$$

  where $D(A,B)$ is the Levenshtein distance between sequences A and B, and $|A|$ and $|B|$ are the lengths of the sequences. For more details on the Levenshtein distance [51], refer to [50]. We apply the square root of the normalized Levenshtein distance since we want to scale up the importance of the sequence similarity term. Otherwise, when the quantum circuit sequence gets larger, the normalized distance will always be close to 0.

The general form of Levenshtein Distance is as follows:

$$\text{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \text{lev}_{a,b}(i-1,j) + 1, \\ \text{lev}_{a,b}(i,j-1) + 1, \\ \text{lev}_{a,b}(i-1,j-1) + \mathbf{1}(a_i \neq b_j) \end{cases} & \text{otherwise} \end{cases} \tag{4.3}$$

Where:

- $lev_{(i,j)}$ is the Levenshtein distance between the character i from string a and the character j from string b

In the following table 4.3, we show a simple example to calculate Levenshtein distance between two sequences occuring in QASM file as "rx(pi/4) q[0]" and "ry(pi/8) q[0]". The Levenshtein distance matrix is as follows:

To populate the matrix, we use the following rules:

- If the characters are the same, the cost is the same as the top-left diagonal cell.
- Otherwise, the cost is 1 plus the minimum of the left cell, top cell, or top-left diagonal cell.

The Levenshtein distance is the value in the bottom-right cell of the matrix, which in this case is 2. This indicates that 2 operations are needed to transform "rx(pi/4) q[0]" into "ry(pi/8) q[0]".

Gate Frequency Similarity:

- Compares the frequency of each type of gate in two circuits.

Table 4.3: *Levenshtein distance matrix for "rx(pi/4) q[0]" and "ry(pi/8) q[0]"*

		r	y	(	p	i	/	8	)		q	[	0	]
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
r	1	0	1	2	3	4	5	6	7	8	9	10	11	12
x	2	1	1	2	3	4	5	6	7	8	9	10	11	12
(	3	2	2	1	2	3	4	5	6	7	8	9	10	11
p	4	3	3	2	1	2	3	4	5	6	7	8	9	10
i	5	4	4	3	2	1	2	3	4	5	6	7	8	9
/	6	5	5	4	3	2	1	2	3	4	5	6	7	8
8	7	6	6	5	4	3	2	2	3	4	5	6	7	8
)	8	7	7	6	5	4	3	3	2	3	4	5	6	7
	9	8	8	7	6	5	4	4	3	2	3	4	5	6
q	10	9	9	8	7	6	5	5	4	3	2	3	4	5
[	11	10	10	9	8	7	6	6	5	4	3	2	3	4
0	12	11	11	10	9	8	7	7	6	5	4	3	2	3
]	13	12	12	11	10	9	8	8	7	6	5	4	3	2

- For each circuit, count the occurrences of each gate type.
- Calculates the cosine similarity between two frequency vectors.
- The similarity score S between two frequency vectors $\mathbf{f}_1$ and $\mathbf{f}_2$ is:

$$S(\mathbf{f}_1, \mathbf{f}_2) = \frac{\mathbf{f}_1 \cdot \mathbf{f}_2}{\|\mathbf{f}_1\| \|\mathbf{f}_2\|}$$

  where $\cdot$ denotes the dot product, and $\|\mathbf{f}_i\|$ is the Euclidean norm of $\mathbf{f}_i$.

Longest Common Subsequence Comparison:

- Compares the QASM (Quantum Assembly Language) code of two circuits by finding the longest common subsequence (LCS) of lines.
- Uses a dynamic programming approach to efficiently compute the LCS.
- Evaluates the similarity based on the proportion of the LCS to the total lines in the target QASM.
- The similarity score S is:

$$S = \frac{\text{length of the longest common subsequence}}{\text{total lines in target QASM}}$$

Version of August 21, 2024– Created August 21, 2024 - 06:42

The dynamic programming method for calculating the LCS is as follows:

1. Let $X = [x_1, x_2, \ldots, x_m]$ be the lines of the first QASM file and $Y = [y_1, y_2, \ldots, y_n]$ be the lines of the target QASM file.

2. Define a 2D table L where $L[i][j]$ represents the length of the LCS of $X[1 \ldots i]$ and $Y[1 \ldots j]$.

3. Initialize $L[0][j] = 0$ for all j and $L[i][0] = 0$ for all i.

4. Fill the table L using the following recurrence relation:

$$L[i][j] = \begin{cases} L[i-1][j-1] + 1 & \text{if } x_i = y_j \\ \max(L[i-1][j], L[i][j-1]) & \text{if } x_i \neq y_j \end{cases}$$

5. The length of the LCS is given by $L[m][n]$.

Combined Score:

- Integrates multiple evaluation metrics to provide a comprehensive fitness score.

- The combined score S is the average of individual scores:

$$S = \left(S_{\text{seq}}(A, B) * S_{\text{freq}}(\mathbf{f}_1, \mathbf{f}_2) * S_{\text{Lcs}}\right)^{1/3}$$

  where $S_{\text{seq}}(A, B)$ is the gate sequence similarity, $S_{\text{freq}}(\mathbf{f}_1, \mathbf{f}_2)$ is the gate frequency similarity, and S_{line} is the line-by-line comparison score. Due to the exponential growth of the unitary's fidelity with the number of qubits, in the actual implementation code, we only used gate sequence similarity, gate frequency similarity, and line-by-line comparison.

These specific metrics were chosen because they capture different aspects of the circuit's structure and behaviour. By taking the geometric mean of these scores, we ensure that the combined score reflects a balanced assessment, where a low score in one metric significantly impacts the overall fitness. This holistic approach drives the evolutionary process towards solutions that are well-rounded in multiple aspects of circuit similarity.

Version of August 21, 2024– Created August 21, 2024 - 06:42

Algorithm 2 Genetic Decompiler Run Method

Initialize population $\mathcal{P} \leftarrow$ generate_initial_population(pop_size)
for generation $= 1$ to `generations` **do**
  Evaluate fitness $\mathcal{F}$ for each individual in $\mathcal{P}$
  Sort $\mathcal{P}$ by $\mathcal{F}$ (descending)
  Select best individual $\mathcal{I}_{\text{best}}$ and score f_{best}
  Initialize new population $\mathcal{P}_{\text{new}}$
  Preserve elite individuals:
  for $i = 1$ to `elite_count` **do**
    $\mathcal{P}_{\text{new}} \leftarrow \mathcal{P}_{\text{new}} \cup \{\mathcal{P}[i]\}$
  end for
  Apply crossover:
  while $|\mathcal{P}_{\text{new}}| <$ `elite_count` $+$ `crossover_count` **do**
    $(\mathcal{P}_1, \mathcal{P}_2) \leftarrow$ select_parents$(\mathcal{P}, \mathcal{F}, \text{method})$
    $(\mathcal{C}_1, \mathcal{C}_2) \leftarrow$ crossover$(\mathcal{P}_1, \mathcal{P}_2)$
    $\mathcal{P}_{\text{new}} \leftarrow \mathcal{P}_{\text{new}} \cup \{\mathcal{C}_1, \mathcal{C}_2\}$
  end while
  Apply mutation:
  for $i = 1$ to `mutation_count` **do**
    if $|\mathcal{P}_{\text{new}}| > 0$ **then**
      $\mathcal{I}_{\text{mutate}} \leftarrow$ random.choice$(\mathcal{P}_{\text{new}})$
      $\mathcal{P}_{\text{new}} \leftarrow \mathcal{P}_{\text{new}} \cup \{\text{mutate}(\mathcal{I}_{\text{mutate}})\}$
    end if
  end for
  Generate new individuals:
  $\mathcal{P}_{\text{new}} \leftarrow \mathcal{P}_{\text{new}} \cup$ generate_initial_population(`new_gen_count`)
  Update $\mathcal{P} \leftarrow \mathcal{P}_{\text{new}}$
end for
Return best individual $\mathcal{I}_{\text{best}}$ and scores

Evolution of the Generation After introducing all the key components, the main Loop of the genetic decompiler to get the best code for the target quantum circuit is shown as 2, and the whole details for the genetic algorithm are listed in appendix B

4.3.3 Improvement Strategies

In order to enhance the performance of the genetic decompiler, we have introduced two key improvement strategies:

1. Random Initialization of New Individuals At the beginning of each generation, a portion of the new population is randomly initialized with new individuals. This strategy aims to increase the exploration capability of the decompiler and prevents it from getting trapped in local optima. Mathematically, the population update process can be described as follows:

Let $\mathcal{P}_g$ be the population at generation g, and $\mathcal{E}_g$ be the set of elite individuals selected from $\mathcal{P}_g$. The new population $\mathcal{P}_{g+1}$ is formed by combining the elite individuals $\mathcal{E}_g$ with a set of newly initialized individuals $\mathcal{N}_g$:

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

where $|\mathcal{E}_g| = e_c$ (elite count) and $|\mathcal{N}_g| = n_c$ (new generation count). This ensures that the new population contains both well-performing individuals from the previous generation and fresh, unexplored solutions, thereby enhancing the diversity of the population.

2. Annealed Mutation Rate To balance exploration and exploitation, we have introduced an annealing mechanism for the mutation rate. The mutation process is controlled by two parameters: m_r (mutation rate) and m_r2 (feature mutation rate). The former defines the proportion of the population that undergoes mutation, while the latter determines the probability of mutating each feature within an individual.

The mutation probability m_r2 is subject to exponential decay, modelled as follows:

$$m_r2(g) = \max(m_r2^0 \times d^g, m_r2^{\min})$$

where:

- m_r2^0 is the initial mutation probability.
- d is the decay factor, typically $0 < d < 1$.
- g is the current generation number.
- $m_r2^{\min}$ is the minimum allowable mutation rate, set to 0.2.

By applying this annealing process, the mutation rate m_r2 decreases over generations, allowing for extensive exploration in the early stages when the genetic pool is less adapted, and more refined, effective mutations in later stages when the population has converged towards an optimal solution. This approach helps maintain a balance between exploration and exploitation, promoting efficient convergence to high-quality solutions.

Version of August 21, 2024– Created August 21, 2024 - 06:42

These improvement strategies collectively enhance the genetic decompiler's ability to navigate the complex search space, improving the likelihood of finding the optimal or near-optimal quantum circuit representations.

4.4 Challenges

In the previous part of this section, we introduced how to initialize Qiskit code in the AST (Abstract Syntax Tree) form as the initial generation for the Genetic Algorithm. This process involves generating quantum circuits and converting them into a series of QASM files based on the number of qubits. These files are then evaluated for their fitness scores by comparing them with target QASM files using specific evaluation methods. By running the Genetic Algorithm, our decompiler evolves the circuit code generation to approximate the ground truth, which is the actual Qiskit code used to create quantum circuits. For this method, there are still some detailed procedures that present challenges to be tackled.

4.4.1 Syntax problem for the qubit index

To accurately reflect the logic of quantum circuits, we need to incorporate 'n' as the total number of qubits or 'i' as the loop index into the expression for the qubit index, indicating the position of each quantum operator. To avoid syntax errors, such as an index exceeding the number of qubits 'n', we will apply a modulo operation to the expression by 'n' to ensure it stays within valid bounds.

$$Expr_{qubit} = Expr_{qubit} \bmod n \tag{4.4}$$

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, we generate two random expressions or add certain expressions to the first qubit expression. To ensure all these expressions remain within the valid range for the number of qubits 'n', they are all subjected to the modulo operation by 'n'. This creates a challenge. It is difficult to generate two random expressions involving linear combinations of 'n' and 'i' that ensure different values modulo 'n' for a given range of 'n' (e.g., 2-20 qubits). It is nonsensical to apply a multi-qubit gate to the same qubit.

To address this issue, the decompiler currently assigns a fitness score of 0 to any syntax error, resulting in many invalid data points during the genetic decompiler run. Finding a method to generate two or more random expressions that remain distinct under modulo 'n' while keeping the form simple would significantly enhance the efficiency of our decompiler.

Example: Consider generating a CNOT gate within a circuit of 'n' qubits.

1. **Generate Random Expressions:**

$$expr_1 = i + n$$
$$expr_2 = 2i + 3$$

2. **Apply Modulo Operation:**

$$\text{qubit}_1 = (i + n) \bmod n$$
$$\text{qubit}_2 = (2i + 3) \bmod n$$

$$\text{If } i = 3, \text{ and } n = 2, \text{ then:}$$
$$\text{qubit}_1 = (3 + 3) \bmod 3 = 0$$
$$\text{qubit}_2 = (2 \cdot 3 + 3) \bmod 3 = 0$$

In this example, both two expressions return 0 when certain i and n are given, which will lead to syntax error when running the code containing a CNOT gate on two same qubit

4.4.2 Fitness Evaluation

Accurately and efficiently evaluating the fitness of generated circuits is complex and resource-intensive. It evaluates the similarity between generated circuits and target circuits, which can be computationally expensive. The unitary matrix of a quantum circuit provides the most comprehensive representation of the circuit's behaviour, capturing all its quantum properties. However, the size of the unitary matrix grows exponentially with the number of qubits, leading to significant computational challenges.

Alternatively, other fitness evaluation methods treat the quantum circuit as pure text information, such as using gate sequence similarity or gate frequency similarity. These methods involve comparing the generated quantum circuits to the target circuits by examining their text representations. While these approaches are computationally less demanding, they might lose some quantum-specific characteristics. By focusing solely on the textual representation, subtle but important quantum properties may be overlooked, potentially leading to suboptimal solutions.

4.4.3 Balancing Exploration and Exploitation

Genetic algorithms need to balance the exploration of new solutions and the exploitation of known good solutions. This balance is crucial for the algorithm's efficiency and effectiveness but is challenging to tune and often requires empirical adjustment.

Fine-grained crossover and mutation, and local search definition are methods that can aid in achieving this balance. However, the process of selecting which specific traits to mutate or crossover is entirely random. Unlike gradient descent methods, where the direction of parameter adjustments is guided by gradients, genetic algorithms lack such directionality. We do not have a clear understanding of the distribution of parameters in the parameter space or how to define their neighbourhoods.

This lack of directionality makes it difficult to guide mutations in a beneficial direction, increasing the difficulty of achieving convergence. The randomness in choosing specific traits for mutation and crossover means that we may need to rely on a larger number of generations and a more extensive search space to find optimal solutions, thus making the algorithm less efficient and more computationally expensive.

Chapter 5

Results and Discussion

> An experiment is a question
> which science poses to Nature,
> and a measurement is the
> recording of Nature's answer.
>
> Max Planck

In this chapter, we delve into the practical implementation and performance evaluation of the genetic decompiler for quantum circuits. The experiments are designed to validate the efficacy and robustness of our approach in decompiling quantum circuits and synthesizing high-level quantum algorithms. First, we had our decompiler test some very simple quantum line patterns, which we call the Vanilla dataset, this part is mainly used as a proof of concept, to see whether our decompiler works or not. Then, we tested the performance of our decompiler before and after the Then, we test the performance of our decompilers quantum lines before and after decomposition, the purpose of this test is to detect how the effect of those decomposed quantum lines, which are more efficient in some hardware platforms, being decompiled compares with the effect of those before they are not decomposed, and here we interpret the result of the reverse compilation as the readability of this part of the quantum lines. Finally, we tested our decompiler on quantum lines represented by some classical algorithms to check the ability of our decompiler to be decompiled. Finally, we discuss the results of the three sets of experiments and, in order to verify the generalization ability of the decompiler and to check whether he accurately recognizes the pattern features of the quantum lines, we tested it on lines with more quantum bits than the test dataset and discuss the different evaluation metrics separately.

Version of August 21, 2024– Created August 21, 2024 - 06:42

5.1 Vanilla data-set

In this section, we introduce and test some simple quantum circuits. These circuits are designed to demonstrate basic quantum gate operations and evaluate the performance of our genetic decompiler on these fundamental data sets.

- **Hadamard Gate Circuit on All Qubits Function:** `h_c(n)`

  This circuit applies a Hadamard gate on each qubit. Mathematically, this circuit can be represented as:

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

  where H_i denotes the Hadamard gate applied to the i-th qubit.

- **Hadamard Gate on the First Qubit**

  Function: `h_0(n)`

  This circuit applies a Hadamard gate on the first qubit for all iterations. Mathematically, this circuit can be represented as:

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

  where H_0 denotes the Hadamard gate applied to the first qubit.

- **Rotational Gate Circuit**

  Function: `rx_c(n)`

  This circuit applies a rotational gate $R_x(\theta)$ to each qubit, with each qubit's rotation angle being half of the previous qubit's angle. Mathematically, this circuit can be represented as:

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

  where $R_x(\theta)$ denotes the rotational gate with angle θ, and the angle θ decreases exponentially with the qubit index i.

The decompilation results of these simple datasets are shown in Figure 5.1. We run the decompiler 3 times and plot both the average and highest scores of each algorithm. We perform all the tricks we used before including crossover, mutation and annealing mutation to balance the exploration and exploitation.

Version of August 21, 2024– Created August 21, 2024 - 06:42

The evaluation method used to get the fitness scores is a combined method of sequence similarity, sequence frequency and line-by-line correctness.We plot the combined scores to evaluate performance over generations. 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 final scores are calculated based on our defined 'combined' scores metric. The evaluation is performed on the best individual from each generation (a segment of Python code) produced by the genetic algorithm. These individuals are executed to generate quantum circuits within a limited qubit range (2-10 qubits). The generated QASM files are then compared to a known target QASM file. The final score is the average of the comparison results for each QASM file within the 2-10 qubit range. To test it, we show the best code corresponding to the last generation of three algorithms as Figure 5.2

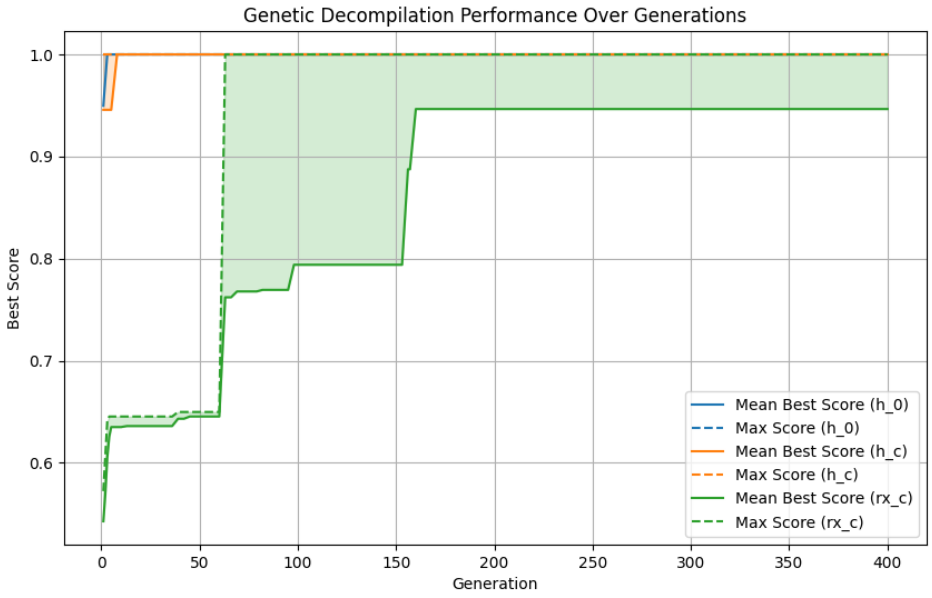


Figure 5.1: *Genetic Decompilation Performance Over Generations.*
The parameters used for generating the plot are: `mutation_rate=0.3`, `pop_size=40`, `generations=100`, `rep=3`. `total_qubit=20`, `max_length=10`, `perform_crossover=True`, `crossover_rate=0.3`, `new_gen_rate=0.2`, `max_loop_depth=2`, `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

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

(a) Best code for h_c

```python
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

```python
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 5.2: *Best code Decomplied by genetic algorithm for some simple quantum circuits*

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.

5.2 Rotation Ry Circuits with Decomposition

The R_y gate is not a native gate for some of the quantum hardware platforms supported by Qiskit, particularly IBM Quantum hardware. Instead, R_y gates are typically decomposed into a series of native gates that the hardware can execute directly. Below are two types of decompositions of the R_y gate for different hardware platforms using Qiskit:

- **Decomposition using RX and RZ Gates**

  The R_y gate can be decomposed using RX and RZ gates, which are also native gates for some quantum hardware.

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

- **Decomposition using H and RX Gates**

  Another common decomposition is to use H (Hadamard) gates along with RX gates.

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

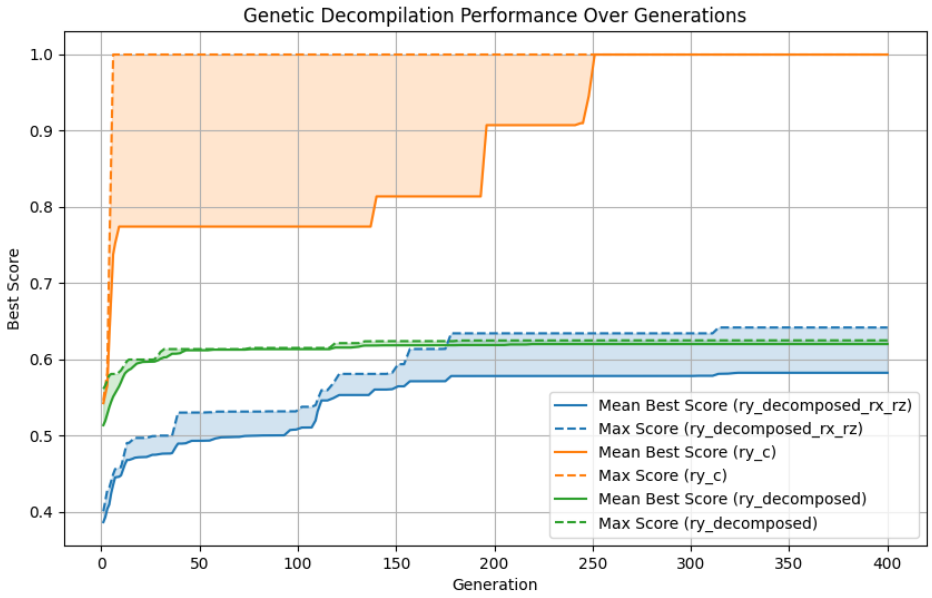


Figure 5.3: *Reverse compilation results for decomposed R_y gate circuits across different hardware platforms. Here ry_decomposed means the decomposition of r_x and h for the ry_c circuit and ry_rx_rz means the decomposition of r_y and r_z for the r_c circuit.The hyperparameter settings for the R_y gate 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`.

We will explore the reverse compilation results based on these decompositions and regard the hardness of the decompilation as a kind of readability for quantum circuits. The results are shown in Figure 5.3. Similar to the previous plot. 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.

We can see that the r_y circuit, without decomposition, achieves a perfect score of 1.0, indicating it exactly replicates the pattern of the quantum circuit as generated by the real code. For the circuits decomposed with a Hadamard gate and Rotation Rx (*ry_decomposed* in the figure) and those decomposed with Rotation Rx and Rotation Rz (*ry_decomposed_rx_rz* in the figure), our decompiler

Version of August 21, 2024– Created August 21, 2024 - 06:42

yields lower evaluation scores. To provide a clearer comparison, we juxtaposed the original code used to generate the dataset, referred to as "real code," with the final qiskit code produced by the decompiler from the best individual of the last generation, as shown in Figure D.1. The results clearly demonstrate that our decompiler is more adept at mimicking and learning from the original r_y circuits than from the decomposed r_y circuits. Moreover, the learning ability for decompositions involving r_x and h gates is slightly superior to that involving r_x and r_z gates. This might be due to the additional phase coefficient required for implementing a rotation gate in the qiskit code compared to the Hadamard Gate. After decomposition, these circuits might be more efficient on certain platforms, as the Ry gate is not inherently decomposed; however, generally, their patterns are more challenging for our decompiler to capture.

5.3 GHZ, QFT and QPE

After conducting our decompiling experiments on simple quantum circuits, we extend our approach to more complex and commonly used quantum algorithms. Specifically, we select Quantum Phase Estimation (QPE), Quantum Fourier Transform (QFT), and GHZ state preparation circuits

Quantum Fourier Transform (QFT)

The Quantum Fourier Transform is a linear transformation on quantum bits, analogous to the discrete Fourier transform in classical computation [52]. It is a crucial component in many quantum algorithms, including Shor's algorithm for factoring. The ket steps for QFT are:

1. **Superposition:** Apply Hadamard gates to put the qubits into a state of superposition, encoding the input in quantum parallelism.

2. **Phase Rotation:** Apply a series of controlled phase rotation gates to entangle the qubits and encode the Fourier transform coefficients.

Quantum Phase Estimation (QPE)

Quantum Phase Estimation is a fundamental algorithm used to estimate the phase (eigenvalue) introduced by a unitary operator. It has applications in various fields including factoring [53], cryptography [54], and quantum chemistry [55]. The key steps for QPE are:

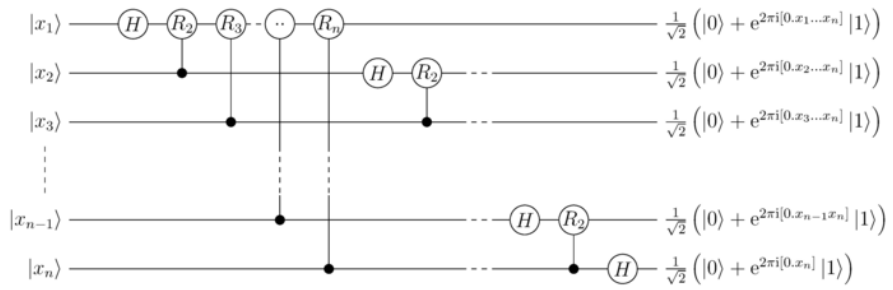


Figure 5.4: *Quantum Fourier Transform (QFT) Circuit*

1. **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.

2. **Controlled Unitary Operations:** Apply controlled unitary operations that evolve the second register based on the state of the first register.

3. **Inverse Quantum Fourier Transform (QFT):** Apply the inverse QFT on the first register to convert the quantum phase information into a readable binary format.

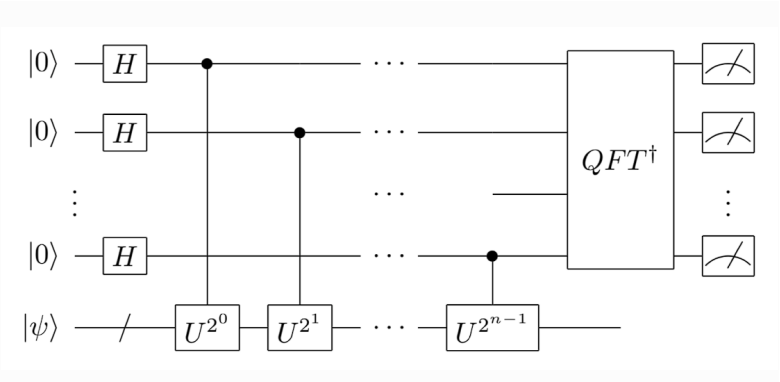


Figure 5.5: *Quantum Phase Estimation (QPE) Circuit*

GHZ State Preparation

The GHZ (Greenberger-Horne-Zeilinger)[21] state is an entangled quantum state involving multiple qubits. It is used in quantum communication, quantum error correction, and tests of quantum mechanics. we have already introduced

this circuit structure in Chapter 3 as Figure2.2. 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.
2. **CNOT Gates:** Apply CNOT gates between the first qubit and the rest to entangle them, creating the GHZ state.

The results of the decompiling on these algorithms can be seen in Figure 5.6. We test the similarity of the qasm files (2-10 qubits) generated by each best individual of generations with the target qasm files(2-10 qubit). Theqiskit code generated by decompiler for the best individual from last generation is listed in Figure D.2

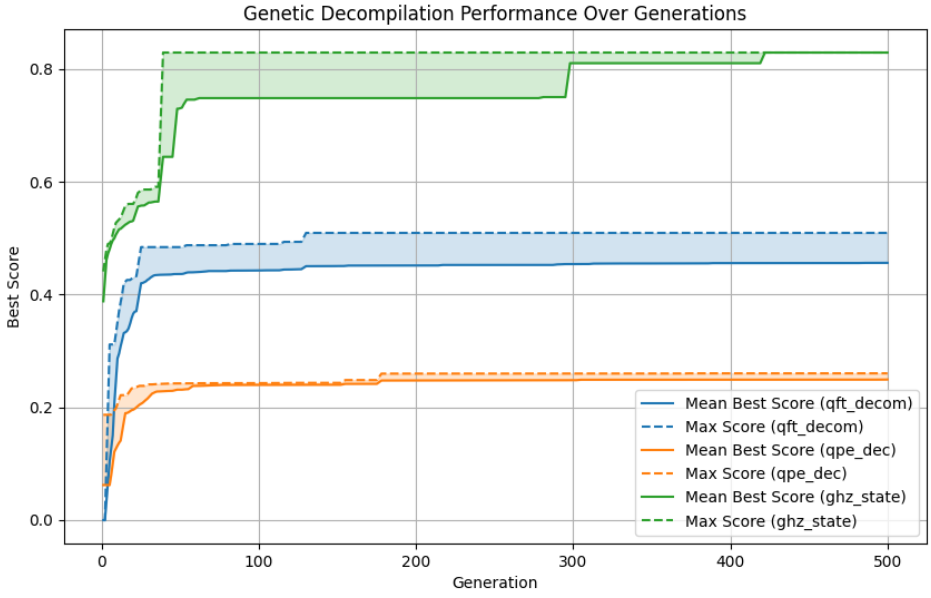
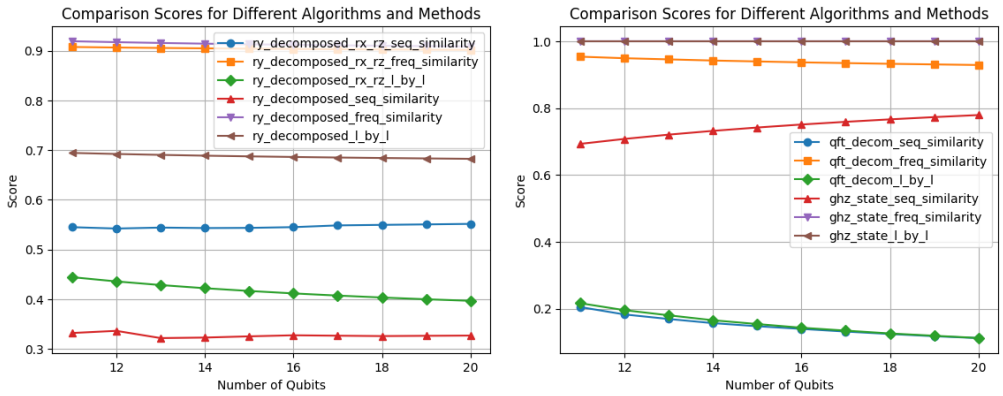


Figure 5.6: *Genetic decompilation performance over generations. The plot shows the mean best score and the maximum score across generations for different algorithms: QFT decomposition (`qft_decom`), QPE decomposition (`qpe_dec`), and GHZ state preparation (`ghz_state`). The mean best score represents 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`.

As the plot illustrates, the GHZ circuit achieves the best decompilation result, while the QFT is easier to decompile than the QPE. This may be because the GHZ circuit contains the fewest quantum gates and follows the simplest pattern compared to the other two. Meanwhile, since QFT is a component of QPE, the quantum circuits for QPE naturally exhibit a more complex pattern, making them harder to decompile.

5.4 Testing on more Qubits

To evaluate the performance of our genetic decompiler, we selected the best code after the evolution of the genetic algorithm and generated quantum circuits larger than the original size (2-10 qubits) used for initial evaluation. We then plotted the fitness scores for various comparison metrics at qubit sizes ranging from 11 to 20 to assess the generalization ability of the decompiler.



(a) different evaluation scores on ry experiments

(b) different evaluation scores on qft and GHZ-generation circuit

Figure 5.7: *Comparison of different evaluation scores on the qubit size from 11 to 20.*

From these two sets of experiments,5.7 we can see that sequence frequency similarity is the easiest feature to capture. In contrast, features of quantum circuits as sequences, such as sequence similarity and largest common sequence, are challenging to capture for both sets of experiments. This is evident since the frequency of quantum gates appearing in quantum circuits is an easily learned feature, whereas their order and logical relationships are relatively difficult to capture under the framework of using a genetic algorithm to manipulate the Abstract Syntax Tree.

This difficulty arises because the order and logical relationships of quantum gates in quantum circuits exhibit more complex structures and interdependen-

cies. While sequence frequency similarity can be easily obtained by counting occurrences, capturing the specific sequence and interactions of quantum gates within quantum circuits requires more advanced analytical methods and algorithms. Therefore, our research findings indicate that genetic algorithms is insufficient to comprehensively understand the characteristics of quantum circuits. This further underscores that in the field of quantum computing, understanding and optimizing the complexity of quantum circuits requires more research and innovative approaches.

5.5 Discussion

In this section, we discuss the outcomes of our experiments with the genetic decompiler.

5.5.1 Proof of Concept

We have demonstrated that using a genetic algorithm to manipulate the abstract syntax tree (AST) of Qiskit code can identify pattern features in quantum circuits. For simple datasets, our genetic decompiler achieved near-perfect accuracy, effectively reconstructing the original quantum circuits. For more complex and common quantum circuits, the decompiler partially reconstructed their features. Unfortunately, our results indicate that within limited computational time, using this framework to reverse-engineer quantum circuits may not be the optimal solution.

5.5.2 Explainability and Efficiency Trade-Off

Through our experiments, we observed a significant trade-off between the explainability and the efficiency of quantum circuits. For instance, the R_y gate, when decomposed into native gates such as H, CX, RX, and RZ, becomes more efficient for execution on hardware but less readable. This trade-off was evident in the adaptive scores obtained during reverse compilation. Decomposed circuits, while more efficient in execution, scored lower in terms of readability.

5.5.3 Selection of Evaluation Methods

Choosing the right evaluation method for fitness scoring is essential. Our current approach combines sequence similarity, gate frequency similarity, and line-by-line correctness. However, each method has its strengths and weaknesses:

- **Gate Frequency Similarity:** This method often yields high scores because it projects gate frequencies into vectors, allowing dominant gates to influence the global score. However, it does not account for the sequence of gates.
- **Gate Sequence Similarity and LCS:** These methods consider the order of gates, often resulting in lower scores but providing a more accurate reflection of the circuit's structure.

We currently use the cubic root of these three scores with equal weighting. Adjusting these weights may better guide the decompiler to capture the correct quantum circuit patterns more effectively.

5.5.4 Limitation For Genetic Algorithm

In our research, we utilize genetic algorithms (GA) to search the abstract syntax tree (AST) structure in the parameter space and evaluate the quantum circuits generated by qiskit code for the corresponding AST structure against the target circuits (the real circuits we want to decompile, our ground truth). We aim to identify patterns in these quantum circuits, but this process lacks some prior knowledge or empirical understanding of the quantum circuits. Each search is completely random and equally distributed among all possible parameters, such as the index for the qubit, the expression for the phase, or the selection for loop expression. However, there are some common rules when designing quantum circuits. For instance, the Hadamard gate is used to prepare entanglement, and there is a greater probability of applying a two-qubit gate in cases involving adjacent qubits or multiple qubits near the first or last qubit. But this approach is not generative enough and is difficult to define strictly mathematically. Overall, enhancing the search process with more experience-based knowledge could be a potential way to uncover the underlying patterns in quantum circuits

Chapter 6

Conclusion and Future Direction

> We choose to go to the Moon in this decade and do the other things, not because they are easy, but because they are hard.
>
> John F. Kennedy

6.1 Conclusion

There are many quantum circuits designed by heuristic algorithms that our human brains cannot easily recognize. Therefore, understanding the underlying logic behind these quantum circuits motivates us to develop a QASM to Python Qiskit decompiler. To review, the research questions for this thesis include whether such QASM-to-Qiskit reverse engineering can be accomplished, what methods can be employed, how we can evaluate the performance of our decompiler, and how our decompiler performs on quantum circuits with varying readability (before and after decomposition). The main contribution of this thesis is the so-called Genetic Quantum Decompiler, which uses a Genetic Algorithm to manipulate the Qiskit code in the format of an Abstract Syntax Tree, evolving the Qiskit code to approximate the true pattern of certain quantum circuits. To assess the correctness of this reverse engineering, we have defined a series of metrics inspired by text similarity measures used in Natural Language Processing (NLP), which include gate sequence frequency, gate sequence similarity, and line-by-line comparison. For the quantum circuits after decomposition, it is indeed more challenging for our decompiler to decompile, which reveals a lower readability for these quantum circuits.

However, during our discussions, it was evident that genetic programming is not yet sufficient to accurately determine the structure of the Qiskit code when problems become complex. On the other hand, since we treat the quantum circuits purely as text documents, this approach may lose some key information about the quantum algorithms they represent. Therefore, although our evaluating metrics ensure that the quantum circuits generated by the decompiled Qiskit code look similar and follow the same patterns as the target quantum circuits, they do not guarantee functional equivalence.

In conclusion, we have demonstrated the potential for reverse engineering quantum circuits with a QASM-to-Qiskit decompilation and have made a significant initial attempt to combine AST and GA as methods. However, to develop a better decompiler with more appropriate metrics to evaluate the closeness between quantum circuits, much more research and novel ideas are still required

6.2 Future Direction

As we look ahead, there are several exciting directions and improvements to consider for advancing our work on quantum circuit decompilation.

6.2.1 Hyper-Parameter Tuning

Detailed and meticulous hyper-parameter tuning is crucial for optimizing the performance of our genetic algorithm (GA). By systematically exploring various combinations of parameters, we can better balance the exploration and exploitation phases of the algorithm, thereby improving the overall efficiency and effectiveness of the decompiler.

6.2.2 Comprehensive Quantum Circuit Decomposition

Expanding the range of quantum circuit structures that our decompiler can handle is essential. This includes incorporating conditional operations such as ‘if‘ statements within the code and clearly specifying which qubits to measure in randomly generated circuits. These enhancements will allow for more complex and realistic quantum circuits to be effectively decompiled.

6.2.3 Exploring New Optimization Methods

Given the inherent randomness and inefficiency of GAs, exploring alternative optimization techniques such as reinforcement learning (RL) is a promising di-

rection. RL, with its potential for making discrete decisions for each feature, could offer more precise and efficient optimization.

6.2.4 Expanding GA-Manipulated AST Framework for Other Tasks

Our framework for manipulating the Abstract Syntax Tree (AST) using GA has broader applications beyond decompilation. For example, it could be used to create general quantum circuits aimed at achieving highly entangled quantum states. By changing the fitness score from the similarity between the generated QASM and target QASM to the degree of entanglement of the generated quantum state, we can tackle new challenges and expand the capabilities of our framework.

6.2.5 New Features for Describing Quantum Circuits

Currently, our approach to recognizing quantum circuit patterns involves treating the circuit as a text file and analyzing its features at the QASM level. However, exploring other methods to characterize these features could enhance our understanding and efficiency. While fidelity is a measure we've considered, it scales exponentially with the size of the system. Machine learning models, such as neural networks, could be trained to encode the features of quantum circuits into a latent space, providing a more nuanced and scalable approach to pattern recognition and feature extraction.

6.2.6 Decompilation on circuits by Quantum Architecture search

In our research, we want to start with a proof of concept to see whether our methodâcombining AST and GAâmight work. Initially, we only test our decompiler on some well-known quantum circuits that are designed by humans, and for which we already know the code to generate them. However, true to our initial motivation, we aim to generalize this QASM-to-qiskit process to a broader range of quantum circuits, including some circuit designs discovered by quantum Architecture search. These circuits lack human interpretability, and we do not know the underlying logic behind them. If we could decompile these quantum circuits on a limited qubit scale and then test the resulting qiskit code to generate larger quantum circuits, we could check if they still perform similarly to the smaller-sized quantum circuits. This approach could potentially improve the efficiency of human-designed quantum circuits and quantum Architecture Search.

Bibliography

[1] S. Y.-C. Chen, *Quantum reinforcement learning for quantum architecture search*, in *Proceedings of the 2023 International Workshop on Quantum Classical Cooperative*, pages 17–20, 2023.

[2] E.-J. Kuo, Y.-L. L. Fang, and S. Y.-C. Chen, *Quantum architecture search via deep reinforcement learning*, arXiv preprint arXiv:2104.07715 (2021).

[3] E. Ye and S. Y.-C. Chen, *Quantum architecture search via continual reinforcement learning*, arXiv preprint arXiv:2112.05779 (2021).

[4] Y. J. Patel, A. Kundu, M. Ostaszewski, X. Bonet-Monroig, V. Dunjko, and O. Danaci, *Curriculum reinforcement learning for quantum architecture search under hardware errors*, arXiv preprint arXiv:2402.03500 (2024).

[5] G. T. Pereira, I. B. Santos, L. P. Garcia, T. Urruty, M. Visani, and A. C. de Carvalho, *Neural architecture search with interpretable meta-features and fast predictors*, Information Sciences **649**, 119642 (2023).

[6] L. Ding and L. Spector, *Evolutionary quantum architecture search for parametrized quantum circuits*, in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 2190–2195, 2022.

[7] S.-X. Zhang, C.-Y. Hsieh, S. Zhang, and H. Yao, *Neural predictor based quantum architecture search*, Machine Learning: Science and Technology **2**, 045027 (2021).

[8] P. 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.

[9] R. Jozsa, *Searching in Grover's algorithm*, arXiv preprint quant-ph/9901021 (1999).

[10] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser, *Quantum computation by adiabatic evolution*, arXiv preprint quant-ph/0001106 (2000).

[11] D-Wave Systems, *D-Wave Quantum Computing*, 2024, Accessed: 2024-07-28.

[12] A. Stern and N. H. Lindner, *Topological quantum computationâfrom basic concepts to first experiments*, Science **339**, 1179 (2013).

[13] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, Cambridge university press, 2010.

[14] D. P. DiVincenzo, *The physical implementation of quantum computation*, Fortschritte der Physik: Progress of Physics **48**, 771 (2000).

[15] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver, *A quantum engineer's guide to superconducting qubits*, Applied physics reviews **6** (2019).

[16] W. K. Hensinger, *Quantum computer based on shuttling trapped ions*, 2021.

[17] L. Henriet, L. Beguin, A. Signoles, T. Lahaye, A. Browaeys, G.-O. Reymond, and C. Jurczak, *Quantum computing with neutral atoms*, Quantum **4**, 327 (2020).

[18] A. Browaeys and T. Lahaye, *Many-body physics with individually controlled Rydberg atoms*, Nature Physics **16**, 132 (2020).

[19] M. Saffman, T. G. Walker, and K. MÃžlmer, *Quantum information with Rydberg atoms*, Reviews of Modern Physics **82**, 2313 (2010).

[20] IBM, *Qiskit: An Open-source Framework for Quantum Computing*, 2024, Accessed: 2024-07-28.

[21] D. M. Greenberger, M. A. Horne, and A. Zeilinger, *Going beyond Bellâs theorem*, in *Bellâs theorem, quantum theory and conceptions of the universe*, pages 69–72, Springer, 1989.

[22] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, *OpenQASM 3: A Broader and Deeper Quantum Assembly Language*, arXiv (2021).

[23] IBM, *IBM Quantum Guides*, 2024, Accessed: 2024-07-28.

[24] C. Collberg, C. Thomborson, and D. Low, *A taxonomy of obfuscating transformations*, Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[25] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, *A survey on automated dynamic malware-analysis techniques and tools*, ACM computing surveys (CSUR) **44**, 1 (2008).

[26] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, *Polyunpack: Automating the hidden-code extraction of unpack-executing malware*, in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 289–300, IEEE, 2006.

[27] R. Baldoni, E. Coppa, D. C. Dâelia, C. Demetrescu, and I. Finocchi, *A survey of symbolic execution techniques*, ACM Computing Surveys (CSUR) **51**, 1 (2018).

[28] B. Schwarz, S. Debray, and G. Andrews, *Disassembly of executable code revisited*, in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 45–54, IEEE, 2002.

[29] Z. D. Sisco, J. Balkind, T. Sherwood, and B. Hardekopf, *Loop Rerolling for Hardware Decompilation*, Proc. ACM Program. Lang. (2023).

[30] M. Emmerik and T. Waddington, *Using a decompiler for real-world source recovery*, in *11th Working Conference on Reverse Engineering*, pages 27–36, IEEE, 2004.

[31] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, *A generic approach to automatic deobfuscation of executable code*, in *2015 IEEE Symposium on Security and Privacy*, pages 674–691, IEEE, 2015.

[32] P. Hu, R. Liang, and K. 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.

[33] X. Fu et al., *An experimental microarchitecture for a superconducting quantum processor*, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 813–825, 2017.

[34] X. Fu, L. Riesebos, L. Lao, C. G. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels, *A heterogeneous quantum computer architecture*, in *Proceedings of the ACM International Conference on Computing Frontiers*, pages 323–330, 2016.

[35] B. Ru, X. Wan, X. Dong, and M. Osborne, *Interpretable neural architecture search via bayesian optimisation with weisfeiler-lehman kernels*, arXiv preprint arXiv:2006.07556 (2020).

[36] T. Duong, S. T. Truong, M. Pham, B. Bach, and J.-K. Rhee, *Quantum neural architecture search with quantum circuits metric and bayesian optimization*, in *ICML 2022 2nd AI for Science Workshop*, 2022.

[37] S.-X. Zhang, C.-Y. Hsieh, S. Zhang, and H. Yao, *Differentiable quantum architecture search*, Quantum Science and Technology **7**, 045023 (2022).

[38] L. Jiménez-Navajas, R. Pérez-Castillo, and M. Piattini, *Reverse Engineering of Classical-Quantum Programs.*, in *ENASE*, pages 275–282, 2024.

[39] N. Nurgalieva, S. Mathis, L. del Rio, and R. Renner, *Thought experiments in a quantum computer*, 2022, arXiv:2209.06236 [quant-ph].

[40] F. J. Ruiz et al., *Quantum circuit optimization with alphatensor*, arXiv preprint arXiv:2402.14396 (2024).

[41] L. Sünkel, D. Martyniuk, D. Mattern, J. Jung, and A. Paschke, *GA4QCO: genetic algorithm for quantum circuit optimization*, arXiv preprint arXiv:2302.01303 (2023).

[42] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, *A generic approach to automatic deobfuscation of executable code*, in *2015 IEEE Symposium on Security and Privacy*, pages 674–691, IEEE, 2015.

[43] R. Wang, C. Hernani-Morales, J. D. Martín-Guerrero, E. Solano, and F. Albarrán-Arriagada, *Quantum pattern recognition in photonic circuits*, Quantum Science and Technology **7**, 015010 (2021).

[44] S. Das, J. Zhang, S. Martina, D. Suter, and F. Caruso, *Quantum pattern recognition on real quantum processing units*, Quantum Machine Intelligence **5**, 16 (2023).

[45] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*, Morgan Kaufmann Publishers Inc., 1998.

[46] R. Olsson, *Inductive functional programming using incremental program transformation*, Artificial intelligence **74**, 55 (1995).

[47] S. Forstenlechner, D. Fagan, M. Nicolau, and M. O'Neill, *Towards effective semantic operators for program synthesis in genetic programming*, in *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1119–1126, 2018.

[48] K. Krawiec, *Behavioral program synthesis with genetic programming*, volume 618, Springer, 2016.

[49] Y. Yuan, W. Wang, and W. Pang, *A Genetic Algorithm with Tree-structured Mutation for Hyperparameter Optimisation of Graph Neural Networks*, 2021.

[50] L. Yujian and L. Bo, *A normalized Levenshtein distance metric*, IEEE transactions on pattern analysis and machine intelligence **29**, 1091 (2007).

[51] V. I. Levenshtein et al., *Binary codes capable of correcting deletions, insertions, and reversals*, in *Soviet physics doklady*, volume 10, pages 707–710, Soviet Union, 1966.

[52] F. X. Lin, *Shorâs algorithm and the quantum Fourier transform*, McGill University (2014).

[53] V. Parasa and M. Perkowski, *Quantum phase estimation using multivalued logic*, in *2011 41st IEEE International Symposium on Multiple-Valued Logic*, pages 224–229, IEEE, 2011.

[54] H. Zbinden, H. Bechmann-Pasquinucci, N. Gisin, and G. Ribordy, *Quantum cryptography.*, Applied Physics B: Lasers & Optics **67** (1998).

[55] K. Sugisaki, S. Nakazawa, K. Toyota, K. Sato, D. Shiomi, and T. Takui, *Quantum chemistry on quantum computers: quantum simulations of the time evolution of wave functions under the S 2 operator and determination of the spin quantum number S*, Physical Chemistry Chemical Physics **21**, 15356 (2019).

Appendix A

Python Code for Circuit Intialization

```python
import ast
import random
import time
import matplotlib.pyplot as plt
from tqdm import tqdm
from graphviz import Digraph

def random_positive_gaussian_integers(mu=0, sigma=1):
    """Generate positive random integers from a Gaussian
       distribution, ensuring all numbers are within a given range
       [1, upper_bound].

    Args:
        mu (float): Mean of the Gaussian distribution.
        sigma (float): Standard deviation of the Gaussian
           distribution.
        num_samples (int): Number of samples to generate.
        upper_bound (int): Maximum value of the random integer (
           inclusive).
    """
    # Generate number, take absolute value, round, and apply upper
       bound
    number = random.gauss(mu, sigma)
    positive_integer = int(abs(number))
    return positive_integer

def random_expr(depth, max_expr_operators, var_depth):
```

```
23      """Generate a random qubit index expression using arithmetic,
            modulus, or simple variables.
24
25      Args:
26      depth (int): Number of loop variables.
27      max_expr_operators (int): Number of binary operations to perform
            .
28      var_depth (int): Number of additional variables.
29      """
30      # Generate variable names for loop indices and variables
31      loop_vars = [f"i{ind}" for ind in range(depth)]
32      vars = ['n']+[f"{ind}" for ind in range(var_depth+1)]
33
34      # All possible variables include 'n' and loop indices
35      choices = [ast.Name(id='n', ctx=ast.Load())] + \
36                [ast.Name(id=var, ctx=ast.Load()) for var in loop_vars
                      ]
37
38      # Start with a random variable
39      # expr = ast.Name(id=random.choice(vars),ctx=ast.Load())
40      expr=random.choice(choices)
41      # Add binary operations
42      for _ in range(max_expr_operators+1):
43          left = expr
44          right = ast.Name(id=random.choice(vars),ctx=ast.Load())
45          op = random.choice([ast.Add(), ast.Sub()])
46          expr = ast.BinOp(left=left, op=op, right=right)
47      return expr
48      # Apply modulo operation to ensure the result is within valid
            index range
49
50  def random_qubit_expr(expr):
51      mod_expr = ast.BinOp(left=expr, op=ast.Mod(), right=ast.Name(id=
            'n', ctx=ast.Load()))
52
53      return mod_expr
54
55  def print_qubit_indices(node, n_value):
56      """
```

```
    Traverse the AST and print each qubit index expression, evaluating them if possible,
    or replacing 'n' with a specific value.
    """
    class IndexPrinter(ast.NodeVisitor):
        def visit_Call(self, node):
            if isinstance(node.func, ast.Attribute) and node.func.attr in ['x', 'h', 'cx', 'rx', 'ry', 'rz']:
                for arg in node.args:
                    print(self.evaluate_expr(arg, n_value))
            self.generic_visit(node)

        def evaluate_expr(self, expr, n_value):
            if isinstance(expr, ast.Name) and expr.id == 'i':
                return 'i' # Return 'i' directly
            try:
                compiled_expr = compile(ast.Expression(expr), filename="<ast>", mode="eval")
                return str(eval(compiled_expr, {"n": n_value, "i": "i", "pi": 3.141592653589793}))
            except Exception as e:
                return f"Error evaluating expression: {e}"

    visitor = IndexPrinter()
    visitor.visit(node)

def random_phase_expr(depth):
    """Generate a random phase expression of the form pi * 1 / (2^a + b + c)."""
    loop_vars = [f"i{ind}" for ind in range(depth)]

    # Define a
    a = random_expr(depth,depth,0)

    # Define b
    b = random_expr(depth,depth,0)
    # Define c
    c = ast.Constant(value=random_positive_gaussian_integers())

    # Create the expression 2^a + b + c
```


Version of August 21, 2024– Created August 21, 2024 - 06:42

```
    expr_inner = ast.BinOp(
        left=ast.BinOp(
            left=ast.Constant(value=2),
            op=ast.Pow(),
            right=a
        ),
        op=ast.Add(),
        right=ast.BinOp(
            left=b,
            op=ast.Add(),
            right=c
        )
    )
    # Create the expression pi * 1 / (2^a + b + c)
    phase_expr = ast.BinOp(
        left=ast.Name(id='pi', ctx=ast.Load()),
        op=ast.Mult(),
        right=ast.BinOp(
            left=ast.Constant(value=1),
            op=ast.Div(),
            right=expr_inner
        )
    )
    return phase_expr

def loop_index(depth):
    if depth == 1:
        return ast.Name(id='n', ctx=ast.Load())
    else:
        # Generate variable names for loop indices
        vars = [f"i{ind}" for ind in range(depth-1)]
        choices = [ast.Name(id='n', ctx=ast.Load())]+\
        [ast.Name(id=var, ctx=ast.Load()) for var in vars]

        # Start with a random variable
        expr = random.choice(choices)

        # Add binary operations
        for _ in range(depth - 1):
            left = expr
```


Version of August 21, 2024– Created August 21, 2024 - 06:42

```
132             right = random.choice(choices)
133             op = random.choice([ast.Add(), ast.Sub()])
134             expr = ast.BinOp(left=left, op=op, right=right)
135 
136         # Random value to add/subtract
137         value = random.randint(0, depth-1) # Using a random integer
                instead of string
138         index = ast.BinOp(left=expr, op=random.choice([ast.Add(),
                ast.Sub()]),
139                           right=ast.Constant(value=value))
140 
141         return ast.Call(
142             func=ast.Name(id='abs', ctx=ast.Load()),
143             args=[index],
144             keywords=[])
145 
146 def set_specific_n(module, n_value):
147     print_qubit_indices(module, n_value)
148 
149 # Define example operations and number of nodes
150 rotation_gates = ['rx', 'ry', 'rz', 'u1', 'u2', 'u3', 'crx', 'cry',
        'crz', 'cp', 'cu1', 'cu3']
151 multi_qubit_gates = ['cx', 'cz', 'swap', 'ch', 'csx', 'cy', 'ccx',
        'cswap', 'cu', 'cp']
152 three_qubit_gates = ['ccx', 'cswap'] # Toffoli (CCX) gate and
        Fredkin (CSWAP) gate
153 
154 def generate_gate_call(depth, gate):
155     """Generate a gate call expression based on the gate type and
            depth."""
156     expr = random_expr(depth, 1, 2)
157     index = random_qubit_expr(expr)
158     if gate in multi_qubit_gates:
159         target_expr = random_expr(depth, 1, 2)
160         target_qubit_index = random_qubit_expr(target_expr)
161         if gate in rotation_gates:
162             phase = random_phase_expr(depth)
163             gate_call = ast.Expr(value=ast.Call(
164                 func=ast.Attribute(value=ast.Name(id="qc", ctx=ast.
                        Load()), attr=gate, ctx=ast.Load()),
```


Version of August 21, 2024– Created August 21, 2024 - 06:42

```
                args=[phase, index, target_qubit_index],
                keywords=[]
            ))
        else:
            gate_call = ast.Expr(value=ast.Call(
                func=ast.Attribute(value=ast.Name(id="qc", ctx=ast.
                    Load()), attr=gate, ctx=ast.Load()),
                args=[index, target_qubit_index],
                keywords=[]
            ))
    else:
        if gate in rotation_gates:
            phase = random_phase_expr(depth)
            gate_call = ast.Expr(value=ast.Call(
                func=ast.Attribute(value=ast.Name(id="qc", ctx=ast.
                    Load()), attr=gate, ctx=ast.Load()),
                args=[phase, index],
                keywords=[]
            ))
        else:
            gate_call = ast.Expr(value=ast.Call(
                func=ast.Attribute(value=ast.Name(id="qc", ctx=ast.
                    Load()), attr=gate, ctx=ast.Load()),
                args=[index],
                keywords=[]
            ))
    return gate_call

def generate_random_circuit_ast(num_nodes, operations,
    max_loop_depth):
    args = ast.arguments(
        posonlyargs=[],
        args=[ast.arg(arg='n', annotation=None)],
        vararg=None,
        kwonlyargs=[],
        kw_defaults=[],
        kwarg=None,
        defaults=[]
    )

```

```
    body = [
        ast.Assign(
            targets=[ast.Name(id="qc", ctx=ast.Store())],
            value=ast.Call(
                func=ast.Name(id='QuantumCircuit', ctx=ast.Load()),
                args=[ast.Name(id='n', ctx=ast.Load())],
                keywords=[]
            )
        )
    ]

    for i in range(num_nodes):
        depth = 0
        gate = random.choice(operations)
        if random.choice([True, False]): # Randomly decide to use a
            loop or a single operation
            loop_body = []
            loop_depth = random.randint(1, max_loop_depth)
            loop_vars = [f"i{ind}" for ind in range(loop_depth)]
            current_body = loop_body
            for j in range(loop_depth):
                depth += 1
                loop = ast.For(
                    target=ast.Name(id=loop_vars[depth-1], ctx=ast.
                        Store()),
                    iter=ast.Call(func=ast.Name(id='range', ctx=ast.
                        Load()), args=[loop_index(depth)], keywords
                        =[]),
                    body=[],
                    orelse=[]
                )
                current_body.append(loop)
                current_body = loop.body

                # Decide to add a gate call in the loop body
                add_gate=random.randint(0,2)
                for i in range(add_gate):
                    gate_call = generate_gate_call(depth, gate)
                    current_body.append(gate_call)

```


Version of August 21, 2024– Created August 21, 2024 - 06:42

```
237             choices = [ast.Name(id='n', ctx=ast.Load())] + [ast.Name
                    (id=var, ctx=ast.Load()) for var in loop_vars]
238             qubit_index = random.choice(choices)
239             gate_call = generate_gate_call(depth, gate)
240             current_body.append(gate_call)
241             body.extend(loop_body)
242         else:
243             gate_call = generate_gate_call(depth, gate)
244             body.append(gate_call)
245
246     body.append(ast.Return(value=ast.Name(id="qc", ctx=ast.Load())))
247
248     function_def = ast.FunctionDef(
249         name="generate_random_circuit_ast",
250         args=args,
251         body=body,
252         decorator_list=[],
253         returns=None,
254         type_comment=None
255     )
256
257     module = ast.Module(body=[function_def], type_ignores=[])
258     ast.fix_missing_locations(module)
259     return module
260 if __name__ == "__main__":
261
262
263     # data = []
264     # for i in range (1000):\
265     # data.append(random_positive_gaussian_integers(mu=0, sigma=2))
266
267     # plt.figure(figsize=(10, 6))
268     # plt.hist(data, bins=range(0, max(data) + 2), edgecolor='black
            ', alpha=0.75)
269     # plt.title('Histogram of 1000 Positive Gaussian Integers')
270     # plt.xlabel('Value')
271     # plt.ylabel('Frequency')
272     # plt.xticks(range(0, 11))
273     # plt.grid(True)
274     # plt.show()
```

```
    # print(random_positive_gaussian_integers())

    # Example usage
    operations = rotation_gates + multi_qubit_gates +
        three_qubit_gates # List of gates to use
    random_circuit = generate_random_circuit_ast(1, operations, 1)
    print(ast.unparse(random_circuit))
```

Appendix B

python code for Genetic Decomplier

```
import ast
from tqdm import tqdm
import subprocess
from circuit_generation import *


from copy import deepcopy
import os
import shutil
import importlib.util
from qiskit.circuit.exceptions import CircuitError
from qiskit import QuantumCircuit, Aer, transpile
from qiskit.quantum_info import Operator, state_fidelity,
    process_fidelity
import Levenshtein
import time
from tqdm import tqdm
rotation_gates = ['rx', 'ry', 'rz', 'u1', 'u2', 'u3', 'crx', 'cry',
     'crz', 'cp', 'cu1', 'cu3']
multi_qubit_gates = ['cx', 'cz', 'swap', 'ch', 'csx', 'cy', 'ccx',
    'cswap', 'cu', 'cp']
three_qubit_gates = ['ccx', 'cswap'] # Toffoli (CCX) gate and
    Fredkin (CSWAP) gate


def analyze_ast(node, output=False):
    gate_calls = []
```

```
    qc_calls = []
    parent_info = []
    index_depths = []

    def visit_node(node, depth=0):
        if isinstance(node, ast.Call) and isinstance(node.func, ast.
            Attribute) and isinstance(node.func.value, ast.Name) and
            node.func.value.id == 'qc':
            qc_calls.append(node)
            index_depth = 0
            args = [ast.unparse(arg) for arg in node.args]
            if output:
                print(f"{' ' * depth}Found qc call: {ast.dump(node)}
                    at depth {depth}")
                print(f"{' ' * depth}Arguments: {args}")
            for arg in node.args:
                arg_str = ast.unparse(arg)
                if 'pi' in arg_str:
                    continue
                # Check for 'i' and find the maximum number following
                    'i'
                for part in arg_str.split():
                    if 'i' in part:
                        i_pos = part.find('i')
                        if i_pos != -1 and i_pos < len(part) - 1:
                            num_str = ''.join(filter(str.isdigit, part
                                [i_pos+1:]))
                            if num_str:
                                index_depth = max(index_depth, int(
                                    num_str) + 1)
            index_depths.append(index_depth)
            if output:
                print(f"{' ' * depth}Index Depth: {index_depth}\n")
        for child in ast.iter_child_nodes(node):
            visit_node(child, depth + 1)

    visit_node(node)
    return gate_calls, qc_calls, parent_info, index_depths


```

```
58
59
60
61
62  class genetic_Decompiler:
63      def __init__(self, algorithm_name, qubit_limit=20, generations
            =100, pop_size=50, max_length=10,
64                  perform_crossover=True,crossover_rate=0.3,
                        new_gen_rate=0.2,mutation_rate=0.1,
65                  compare_method='l_by_l',max_loop_depth=2,
                        mutation_rate_2=0.5, perform_annealing=False,
66                 perform_mutation=True, selection_method='tournament',
                       operations = ['h', 'x', 'cx']):
67          self.algorithm_name = algorithm_name
68          self.qubit_limit = qubit_limit
69          self.generations = generations
70          self.pop_size = pop_size
71          self.max_length = max_length
72          self.crossover_rate=crossover_rate
73          self.mutation_rate=mutation_rate
74          self.mutation_rate_2 = mutation_rate_2
75          self.new_gen_rate=new_gen_rate
76          self.max_loop_depth=max_loop_depth
77          self.perform_crossover = perform_crossover
78          self.compare_method=compare_method
79          self.perform_annealing = perform_annealing
80          self.perform_mutation = perform_mutation
81          self.selection_method = selection_method
82          self.operations=operations
83          # Initialize the path for saving files related to the
                algorithm
84          self.path = os.path.join('genetic_deQ', self.algorithm_name)
85          self.qasm_path=os.path.join('genetic_deQ_qasm', self.
                algorithm_name)
86          os.makedirs(self.path, exist_ok=True) # Create the directory
                 if it does not exist
87          os.makedirs(self.qasm_path, exist_ok=True)
88
89
90      def generate_initial_population(self,size):
```


Version of August 21, 2024– Created August 21, 2024 - 06:42

```
91          population = []
92          for _ in range(size):
93              # num_qubits = random.randint(2, self.qubit_limit)
94              num_nodes = random.randint(1, self.max_length)
95              ast_circuit = generate_random_circuit_ast( num_nodes,
                    self.operations,max_loop_depth=self.max_loop_depth)
96              population.append(ast_circuit)
97          return population
98
99
100     def mutate(self, ast_circuit, mutation_rate_2=0.5, output=False)
            :
101         mutation_rate_2 = self.mutation_rate_2
102         # Create a deep copy of the AST to avoid modifying the
                original AST
103         ast_circuit_copy = deepcopy(ast_circuit)
104
105         # Analyze the AST
106         gate_calls, qc_calls, parent_info, index_depths =
                analyze_ast(ast_circuit_copy, output=False)
107
108         if not qc_calls:
109             return ast_circuit_copy # No gate calls to mutate
110
111         # Randomly choose mutation type
112         mutation_type = random.choices(['insert', 'modify'], weights
                =[0.2, 0.8])[0]
113
114         if mutation_type == 'insert':
115             # Randomly select a parent node to insert into
116             if not parent_info:
117                 return ast_circuit_copy # No parent nodes to insert
                        into
118
119             parent_node, parent_index = random.choice(parent_info)
120             new_gate = generate_gate_call(random.choice(self.
                    operations))
121             parent_node.body.insert(parent_index, new_gate)
122             if output:
```

```
123                 print(f"Inserted new gate: {ast.unparse(new_gate)} at
                        index {parent_index}")
124
125         elif mutation_type == 'modify':
126             # Randomly select a qc call to mutate with a probability
127             for qc_call, index_depth in zip(qc_calls, index_depths):
128                 if random.random() < mutation_rate_2:
129                     if output:
130                         original_code = ast.unparse(qc_call)
131
132                     # Extract arguments and classify them
133                     for i, arg in enumerate(qc_call.args):
134                         arg_str = ast.unparse(arg)
135                         if 'pi' in arg_str:
136                             # Mutate phase argument
137                             qc_call.args[i] = random_phase_expr(
                                    index_depth)
138                         else:
139                             # Mutate index argument
140                             new_expr = random_expr(index_depth, 3, 1)
141                             qc_call.args[i] = random_qubit_expr(
                                    new_expr)
142
143                     if output:
144                         new_code = ast.unparse(qc_call)
145                         print(f"Modified code from: {original_code} to
                                : {new_code}")
146
147         ast.fix_missing_locations(ast_circuit_copy)
148         return ast_circuit_copy
149
150     def crossover(self, parent1, parent2):
151         # Select crossover points
152         index1 = random.randint(1, len(parent1.body[0].body) - 2)
153         index2 = random.randint(1, len(parent2.body[0].body) - 2)
154
155         # Swap subcircuits
156         new_body1 = parent1.body[0].body[:index1] + parent2.body[0].
            body[index2:]
```


Version of August 21, 2024– Created August 21, 2024 - 06:42

```
157         new_body2 = parent2.body[0].body[:index2] + parent1.body[0].
                body[index1:]
158 
159         # Construct new ASTs
160         child1 = ast.Module(body=[ast.FunctionDef(
161             name=parent1.body[0].name,
162             args=parent1.body[0].args,
163             body=new_body1,
164             decorator_list=[]
165         )], type_ignores=[])
166 
167         child2 = ast.Module(body=[ast.FunctionDef(
168             name=parent2.body[0].name,
169             args=parent2.body[0].args,
170             body=new_body2,
171             decorator_list=[]
172         )], type_ignores=[])
173 
174         ast.fix_missing_locations(child1)
175         ast.fix_missing_locations(child2)
176 
177         return child1, child2
178 
179     def select_parents(self, population, fitness_scores,
            selection_method='tournament', k=3):
180         if selection_method == 'roulette':
181             return self.roulette_wheel_selection(population,
                    fitness_scores)
182         elif selection_method == 'tournament':
183             return self.tournament_selection(population,
                    fitness_scores, k)
184         elif selection_method == 'rank':
185             return self.rank_selection(population, fitness_scores)
186         elif selection_method == 'random':
187             return self.random_selection(population)
188         elif selection_method == 'weighted_roulette':
189             return self.weighted_roulette_wheel_selection(population
                    , fitness_scores)
190         else:
```

```
191             raise ValueError(f"Unknown selection method: {
                    selection_method}")
192
193     def roulette_wheel_selection(self, population, fitness_scores):
194         total_fitness = sum(fitness_scores)
195         probabilities = [score / total_fitness for score in
                fitness_scores]
196         selected_indices = random.choices(range(len(population)),
                weights=probabilities, k=2)
197         return population[selected_indices[0]], population[
                selected_indices[1]]
198
199     def tournament_selection(self, population, fitness_scores, k=3):
200         selected_indices = random.sample(range(len(population)), k)
201         selected_individuals = [(fitness_scores[i], population[i])
                for i in selected_indices]
202         parent1 = max(selected_individuals, key=lambda x: x[0])[1]
203         parent2 = max(selected_individuals, key=lambda x: x[0])[1]
204         return parent1, parent2
205
206     def rank_selection(self, population, fitness_scores):
207         sorted_population = sorted(zip(fitness_scores, population),
                key=lambda x: x[0])
208         rank_probabilities = [(i + 1) / len(sorted_population) for i
                 in range(len(sorted_population))]
209         selected_indices = random.choices(range(len(population)),
                weights=rank_probabilities, k=2)
210         return sorted_population[selected_indices[0]][1],
                sorted_population[selected_indices[1]][1]
211
212     def random_selection(self, population):
213         parent1, parent2 = random.sample(population, 2)
214         return parent1, parent2
215
216     def weighted_roulette_wheel_selection(self, population,
            fitness_scores, weight=2.0):
217         total_fitness = sum(fitness_scores)
218         weighted_fitness = [score ** weight for score in
                fitness_scores]
219         total_weighted_fitness = sum(weighted_fitness)
```

```
220         probabilities = [wf / total_weighted_fitness for wf in
                weighted_fitness]
221         selected_indices = random.choices(range(len(population)),
                weights=probabilities, k=2)
222         return population[selected_indices[0]], population[
                selected_indices[1]]
223
224     def save(self, population):
225     # Clear all files in the target directory before saving new
            files
226         for filename in os.listdir(self.path):
227             file_path = os.path.join(self.path, filename)
228             try:
229                 if os.path.isfile(file_path) or os.path.islink(
                        file_path):
230                     os.unlink(file_path) # Remove file
231                 elif os.path.isdir(file_path):
232                     shutil.rmtree(file_path) # Remove directory
233             except Exception as e:
234                 print(f'Failed to delete {file_path}. Reason: {e}')
235
236         # Iterate over the population and save each individual's
                Python code to a file
237         for index, individual in enumerate(population):
238             # Convert AST to Python code
239             python_code = ast.unparse(individual)
240
241             # Create the filename, including the algorithm name and
                    index
242             filename = os.path.join(self.path, f"{self.
                    algorithm_name}_{index}.py")
243
244             # Write Python code to the file
245             with open(filename, 'w') as file:
246                 file.write(python_code)
247
248     def get_quantum_gates_from_qasm(self):
249         target_qasm_dir = "Circuits"
250         all_gates = set()
251
```

```
252         for i in range(2, self.qubit_limit + 1):
253             target_qasm_file = os.path.join(target_qasm_dir, f"{self
                .algorithm_name}_{i}.qasm")
254 
255             if os.path.exists(target_qasm_file):
256                 with open(target_qasm_file, 'r') as file:
257                     qasm_str = file.read()
258 
259                 quantum_circuit = QuantumCircuit.from_qasm_str(
                    qasm_str)
260 
261                 for instruction in quantum_circuit.data:
262                     gate_name = instruction[0].name
263                     all_gates.add(gate_name)
264 
265         return list(all_gates)
266 
267     def save_qasm(self):
268         for filename in os.listdir(self.path):
269             if filename.endswith('.py'):
270                 full_py_path = os.path.join(self.path, filename)
271 
272                 # Read and modify the script as discussed above
273                 with open(full_py_path, 'r') as file:
274                     module_code = "from qiskit import QuantumCircuit\
                        nimport numpy as np\nimport random\nfrom math
                        import pi\n" + file.read()
275 
276                 local_namespace = {}
277                 exec(module_code, local_namespace)
278 
279                 # Set up the directory for QASM files
280                 file_base_name = filename[:-3] # Remove '.py'
                    extension
281                 qasm_dir_path = os.path.join(self.qasm_path,
                    file_base_name)
282                 os.makedirs(qasm_dir_path, exist_ok=True)
283 
284                 # Generate QASM files for each qubit count
285                 for i in range(2, self.qubit_limit + 1):
```

```
286                 try:
287                         # Print the generated Python code for
                               debugging
288                     # generated_code = module_code + f"\n\
                           ngenerate_random_circuit_ast({i})"
289                     # print(f"Running generated code for {
                           file_base_name} with {i} qubits:\n{
                           generated_code}")
290
291
292                     qc = local_namespace['
                           generate_random_circuit_ast'](i)
293
294                     modified_circuit = QuantumCircuit(qc.
                           num_qubits)
295                     for gate, qargs, cargs in qc.data:
296                         if gate.name == 'cx':
297                             control_qubit, target_qubit = qargs
298                             if control_qubit.index == target_qubit
                                   .index:
299                                 # Adjust target qubit index to be
                                       different from control qubit
                                       index
300                                 target_qubit = qc.qubits[(
                                       target_qubit.index + 1) % qc.
                                       num_qubits]
301                                 modified_circuit.cx(control_qubit,
                                       target_qubit)
302                             else:
303                                 modified_circuit.cx(control_qubit,
                                       target_qubit)
304                         else:
305                             modified_circuit.append(gate, qargs,
                                   cargs)
306
307                     qasm_output = modified_circuit.qasm()
308                 except (CircuitError, ZeroDivisionError) as e:
309                 # Handle both CircuitError and ZeroDivisionError
310                 # print(f"Error generating QASM for {filename}
                       with {i} qubits: {e}")
```

```
311                     qasm_output = "" # Save an empty QASM file if
                            there's an error
312
313                 qasm_filename = os.path.join(qasm_dir_path, f"{
                        file_base_name}_{i}.qasm")
314                 with open(qasm_filename, 'w') as f:
315                     f.write(qasm_output)
316
317
318     def qasm_to_unitary(self, qasm_file_path):
319         # Read QASM file and create a quantum circuit
320         with open(qasm_file_path, 'r') as file:
321             qasm_str = file.read()
322
323         quantum_circuit = QuantumCircuit.from_qasm_str(qasm_str)
324
325         # Use Aer simulator to get the unitary matrix
326         backend = Aer.get_backend('unitary_simulator')
327         transpiled_circuit = transpile(quantum_circuit, backend)
328
329         # Get the unitary matrix
330         job = backend.run(transpiled_circuit)
331         unitary_matrix = job.result().get_unitary(transpiled_circuit
                )
332
333         return unitary_matrix
334
335     def qasm_to_gate_sequence(self, qasm_file_path):
336         # Read QASM file and create a quantum circuit
337         with open(qasm_file_path, 'r') as file:
338             qasm_str = file.read()
339
340         quantum_circuit = QuantumCircuit.from_qasm_str(qasm_str)
341
342         # Extract gate sequence
343         gate_sequence = []
344         for instruction in quantum_circuit.data:
345             gate_name = instruction[0].name
346             qubits = [qubit.index for qubit in instruction[1]]
347             if gate_name in rotation_gates:
```

```
348                 params = [param for param in instruction[0].params]
349                 gate_sequence.append((gate_name, tuple(qubits),
                        params))
350             else:
351                 gate_sequence.append((gate_name, tuple(qubits)))
352 
353         return gate_sequence
354 
355     def gate_sequence_similarity(self, seq1, seq2):
356         seq1_str = ' '.join([f"{gate[0]}{gate[1]}{[f'{param:.6f}'
                for param in gate[2]]}" if len(gate) == 3 else f"{gate
                [0]}{gate[1]}" for gate in seq1])
357         seq2_str = ' '.join([f"{gate[0]}{gate[1]}{[f'{param:.6f}'
                for param in gate[2]]}" if len(gate) == 3 else f"{gate
                [0]}{gate[1]}" for gate in seq2])
358 
359         ### Debugging line
360         # print(f"Sequence 1: {seq1_str}")
361         # print(f"Sequence 2: {seq2_str}")
362 
363         max_len = max(len(seq1_str), len(seq2_str))
364         if max_len == 0:
365             return 1.0
366 
367         return 1 - (Levenshtein.distance(seq1_str, seq2_str) /
                max_len)**(1/2)
368 
369     def gate_frequency_similarity(self, qasm_file_path1,
            qasm_file_path2):
370         def get_gate_frequencies(qasm_file_path):
371             with open(qasm_file_path, 'r') as file:
372                 qasm_str = file.read()
373 
374             quantum_circuit = QuantumCircuit.from_qasm_str(qasm_str)
375             gate_count = {}
376             for instruction in quantum_circuit.data:
377                 gate_name = instruction[0].name
378                 if gate_name in gate_count:
379                     gate_count[gate_name] += 1
380                 else:
```

```
                gate_count[gate_name] = 1
        return gate_count

    freq1 = get_gate_frequencies(qasm_file_path1)
    freq2 = get_gate_frequencies(qasm_file_path2)

    all_gates = set(freq1.keys()).union(set(freq2.keys()))
    # Check if the gate types are the same
    if set(freq1.keys()) != set(freq2.keys()):
        return 0.0 # Directly return 0 if the gate types are
            different
    vec1 = [freq1.get(gate, 0) for gate in all_gates]
    vec2 = [freq2.get(gate, 0) for gate in all_gates]

    dot_product = sum([vec1[i] * vec2[i] for i in range(len(
        all_gates))])
    norm1 = sum([x ** 2 for x in vec1]) ** 0.5
    norm2 = sum([x ** 2 for x in vec2]) ** 0.5

    return dot_product / (norm1 * norm2)

def compare_qasm_lcs(self,qasm_lines, target_qasm_lines):
    def lcs_length(X, Y):
        m = len(X)
        n = len(Y)
        L = [[0] * (n + 1) for i in range(m + 1)]

        for i in range(m + 1):
            for j in range(n + 1):
                if i == 0 or j == 0:
                    L[i][j] = 0
                elif X[i - 1].strip() == Y[j - 1].strip():
                    L[i][j] = L[i - 1][j - 1] + 1
                else:
                    L[i][j] = max(L[i - 1][j], L[i][j - 1])

        return L[m][n]
    # Calculate the length of the longest common subsequence
    lcs_len = lcs_length(qasm_lines, target_qasm_lines)

```


Version of August 21, 2024– Created August 21, 2024 - 06:42

```
419         # Calculate similarity score based on the length of LCS over
                the total lines in target_qasm
420         score = lcs_len / len(target_qasm_lines) if
               target_qasm_lines else 0
421         return score
422
423     def compare_qasm(self, qasm, target_qasm):
424         def is_file_empty(file_path):
425             return os.path.getsize(file_path) == 0
426
427         if is_file_empty(qasm) or is_file_empty(target_qasm):
428             return 0
429         try:
430             if self.compare_method == 'fidelity':
431                 # Calculate unitary matrices for both QASM files
432                 unitary1 = self.qasm_to_unitary(qasm)
433                 unitary2 = self.qasm_to_unitary(target_qasm)
434
435                 # Calculate fidelity
436                 fidelity_score = process_fidelity(unitary1,unitary2)
437                 return fidelity_score
438
439             elif self.compare_method == 'seq_similarity':
440                 # Gate sequence similarity
441                 seq1 = self.qasm_to_gate_sequence(qasm)
442                 seq2 = self.qasm_to_gate_sequence(target_qasm)
443                 seq_similarity = self.gate_sequence_similarity(seq1,
                       seq2)
444                 return seq_similarity
445
446             elif self.compare_method == 'freq_similarity':
447                 # Gate frequency similarity
448                 freq_similarity = self.gate_frequency_similarity(qasm
                       , target_qasm)
449                 return freq_similarity
450
451
452             elif self.compare_method == 'l_by_l':
453                 with open(qasm, 'r') as file1, open(target_qasm, 'r')
                        as file2:
```

```
454                     qasm_lines = file1.readlines()
455                     target_qasm_lines = file2.readlines()
456 
457                     score = self.compare_qasm_lcs(qasm_lines,
                            target_qasm_lines)
458 
459                 return score
460 
461             elif self.compare_method == 'combined':
462                 # Gate sequence similarity
463                 seq1 = self.qasm_to_gate_sequence(qasm)
464                 seq2 = self.qasm_to_gate_sequence(target_qasm)
465                 seq_similarity = self.gate_sequence_similarity(seq1,
                        seq2)
466 
467                 # Gate frequency similarity
468                 freq_similarity = self.gate_frequency_similarity(qasm
                        , target_qasm)
469 
470                 with open(qasm, 'r') as file1, open(target_qasm, 'r')
                         as file2:
471                     qasm_lines = file1.readlines()
472                     target_qasm_lines = file2.readlines()
473 
474                     lcs_similarity = self.compare_qasm_lcs(qasm_lines
                            ,target_qasm_lines)
475 
476                 # combined_score = (seq_similarity + freq_similarity
                        + inter_section_score) / 3
477                 combined_score = (seq_similarity * freq_similarity *
                        lcs_similarity)**(1/3)
478                 return combined_score
479 
480 
481         except FileNotFoundError:
482             print(f"Error: One of the files not found ({qasm} or {
                    target_qasm}).")
483             return 0
484         except Exception as e:
485             print(f"Error comparing QASM files: {str(e)}")
```

```
            return 0

    def evaluate(self, individual, individual_index):
        qasm_dir = os.path.join(self.qasm_path, f"{self.
            algorithm_name}_{individual_index}")
        target_qasm_dir = "Circuits"

        # Calculate score for each QASM file
        scores = []
        for i in range(2, self.qubit_limit+1):
            qasm_file = os.path.join(qasm_dir, f"{self.
                algorithm_name}_{individual_index}_{i}.qasm")
            target_qasm_file = os.path.join(target_qasm_dir, f"{self
                .algorithm_name}_{i}.qasm")
             ## debug
            # print(qasm_file,target_qasm_file)
            score = self.compare_qasm(qasm_file, target_qasm_file)
            scores.append(score)

        # Return the average score

        average_score = sum(scores) / len(scores) if scores else 0
        return average_score


    def run(self):
        if not self.perform_crossover and not self.perform_mutation:
            print("Warning: Both crossover and mutation are disabled
                ; the population will not evolve.")

        best_score = float('-inf')
        best_individual = None
        best_generation_index = -1

        # Initialize lists to store scores
        best_scores = []
        all_scores = []
        best_code=[]

        # Generate initial population once at the beginning
```

```
522         population = self.generate_initial_population(self.pop_size)
523 
524         for generation in range(self.generations):
525             start_time = time.time()
526 
527             # Save the current population state and QASM data
528             self.save(population)
529             self.save_qasm()
530 
531             # Evaluate fitness for each individual
532             fitness_scores = [self.evaluate(individual, index) for
                    index, individual in enumerate(population)]
533 
534             # Save all fitness scores for this generation
535             all_scores.append(fitness_scores)
536 
537             # Sort population by fitness (descending order)
538             sorted_population = sorted(zip(fitness_scores,
                    population), key=lambda pair: pair[0], reverse=True)
539             sorted_scores, next_generation = zip(*sorted_population)
540             next_generation = list(next_generation)
541             sorted_scores = list(sorted_scores)
542 
543             # Select the best individual and corresponding score
544             best_individual = next_generation[0]
545             best_score = sorted_scores[0]
546 
547             # Save the best score for this generation
548             best_scores.append(best_score)
549 
550             # Find the index of the best individual in the original
                    population
551             best_individual_index = fitness_scores.index(best_score)
552 
553             # Print debugging information
554             # print(f"Algorithm : {self.algorithm_name} Generation {
                    generation + 1}: Best score = {best_score}")
555 
556             # # If the best score is 1, stop the iteration
557             # if best_score == 1:
```

```
558             # break
559             new_population = []
560 
561             # Number of individuals to be generated by each method
562             crossover_count = int(self.pop_size * self.
                    crossover_rate) if self.perform_crossover == True
                    else 0
563             mutation_count = int(self.pop_size * self.mutation_rate)
                     if self.perform_mutation == True else 0
564             new_gen_count = int(self.pop_size * self.new_gen_rate)
565             elite_count = self.pop_size - crossover_count -
                    mutation_count - new_gen_count
566 
567             # Preserve elite individuals
568             new_population.extend(next_generation[:elite_count])
569 
570             # Apply crossover to generate new individuals
571             while len(new_population) < elite_count +
                    crossover_count:
572                 parent1, parent2 = self.select_parents(
                        next_generation, sorted_scores, self.
                        selection_method)
573 
574                 child1, child2 = self.crossover(parent1, parent2)
575 
576                 new_population.extend([child1, child2])
577 
578             # Apply mutation to new individuals
579             for _ in range(mutation_count):
580                 if new_population:
581                     individual_to_mutate = random.choice(
                            new_population)
582                     new_population.append(self.mutate(
                            individual_to_mutate))
583 
584             # Generate new individuals
585             new_population.extend(self.generate_initial_population(
                    new_gen_count))
586             individual_code = ast.unparse(best_individual) if
                    best_individual else "No best individual found"
```

```
587
588             best_code.append(individual_code)
589
590             # Ensure the population size is correct after all
                    operations
591             # new_population = new_population[:self.pop_size]
592
593             population = new_population
594
595             # Apply annealing to the mutation_rate_2
596             if self.perform_annealing:
597                 self.mutation_rate_2 = max(self.mutation_rate_2 *
                        0.99, 0.1)
598
599             end_time = time.time()
600             time_taken = end_time - start_time
601             tqdm.write(f"Generation {generation + 1}/{self.
                    generations} completed in {time_taken:.2f} seconds")
602
603         # Unparse the AST of the best individual if found
604
605         return best_code, best_scores, all_scores
```


Version of August 21, 2024– Created August 21, 2024 - 06:42

Appendix C

Qiskit Code for Circuit Simulation

```
from qiskit import QuantumCircuit, transpile, Aer, assemble
from qiskit.visualization import plot_histogram, circuit_drawer
import matplotlib.pyplot as plt

# Create a quantum circuit with 3 qubits
qc = QuantumCircuit(3)

# Create a GHZ state
qc.h(0) # Apply Hadamard gate on qubit 0
qc.cx(0, 1) # Apply CNOT (Controlled-NOT) gate between qubit 0 and 1
qc.cx(0, 2) # Apply CNOT gate between qubit 0 and 2

# Simulate the circuit
simulator = Aer.get_backend('statevector_simulator')
result = simulator.run(qc).result()
statevector = result.get_statevector()

# Print the final statevector
print("Final statevector:")
print(statevector)

# Measure qubits and display histogram
qc.measure_all()
qc.draw(output='mpl', filename='ghz_state_measurement.png') # Save measurement circuit plot as image
print(qc)
```

```
26
27 # Simulate the measurement circuit
28 simulator = Aer.get_backend('qasm_simulator')
29 job = assemble(qc)
30 result = simulator.run(job).result()
31 counts = result.get_counts()
```

Appendix D

Qiskit code Generated by decompiler

```python
# Real code for ry_decomposed
angle = pi
for i in range(n):
    qc.h(i)
    qc.rx(angle, i)
    qc.h(i)
    angle /= 2
return qc
```

```python
# Real code for
   ry_decomposed_rx_rz
angle = pi
for i in range(n):
    qc.rz(-pi/2, i)
    qc.rx(angle, i)
    qc.rz(pi/2, i)
    angle /= 2
return qc
```

```python
# Decompiled code for ry_decomposed
qc.rx(pi * (1 / (2 ** (n - n) + (n - n + 0))), (n + n) % n)
for i0 in range(n):
    qc.h((i0 - n + n) % n)
    qc.h((i0 - 1 + 1 - n - 0) % n)
    for i1 in range(abs(n - n - 1)):
        qc.h((n - 1 - 1 - 0 - 1) % n)
qc.rx(pi * (1 / (2 ** (n - n) + (n - n + 0))), (n + n) % n)
qc.h((n - n - 0 - n - 1) % n)
return qc
```

```python
# Decompiled code for ry_decomposed_rx_rz
for i0 in range(n):
    qc.rz(pi * (1 / (2 ** (n - n) + (n - n + 1))), (n - 1 + n
        + 1 + 0) % n)
    qc.rz(pi * (1 / (2 ** (n - n) + (n - n + 1))), (n - 0 + 1
        - 0 + 1) % n)
    qc.rz(-(pi * (1 / (2 ** (n + 0 - n) + (n + 0 - n + 1)))),
        (i0 + n) % n)
qc.rx(pi * (1 / (2 ** (n - n) + (n - n + 1))), (n - 0 + 0 -
   n + n) % n)
return qc
```

Figure D.1: *Decomposed qiskit code for ry circuits in different decomposition*

```python
# Decompiled code code for ghz_state
    qc.h((n - n) % n)
    for i0 in range(n):
        qc.cx((i0 - 1 + 0) % n, (i0 + n) % n)
    return qc
```

```python
# Decompiled code for qpe_dec
qc.h((n - 1 + n - 1 - n) % n)
qc.h((n - 1 + n - 1 - n) % n)
qc.swap((n + 0) % n, (n - 1) % n)
qc.h((n - 1 + n - 1 - n) % n)
qc.h((n - 1 + n - 1 - n) % n)
qc.h((n - 1 + n - 1 - n) % n)
qc.h((n - 1 + n - 1 - n) % n)
for i0 in range(n):
    for i1 in range(abs(n + n - 1)):
        qc.cp(pi * (1 / (2 ** (i0 - 0 - 0) + (n - n - 0
            + 0))), (n - 1 + 0 + 1 - n) % n, (n + 0 +
           1 - 1 - 1) % n)
        qc.cp(pi * (1 / (2 ** (n - n) + (n - 0 + 0))),
           (n + 1 - 0 - 0 - 1) % n, (n + 1 + 1 - 1 + n
           ) % n)
return qc
```

```python
# Decompiled code for qft_decom
qc.h((n - n - 0 - 0 - 0) % n)
for i0 in range(n):
    qc.cp(-(pi * (1 / (2 ** (n - n) + (n - n + 1)))),
       (n + n + 0 + 0 - 1) % n, (n + 1 - 1 + 0 + 0) %
       n)
    qc.cp(pi * (1 / (2 ** (n - n + 0) + (i0 + 0 + 0 +
       1))), (i0 + 1) % n, (i0 - n + n) % n)
for i0 in range(n):
    qc.h((i0 + n - 0 + 1 + n) % n)
qc.swap((n + n - 1 + 0 + 1) % n, (n - 0 - 0 - 1 - n) %
    n)
return qc
```

Figure D.2: *Decomplied qiskit code for GHZ, qft_decom, and qpe_dec circuits*

Version of August 21, 2024– Created August 21, 2024 - 06:42