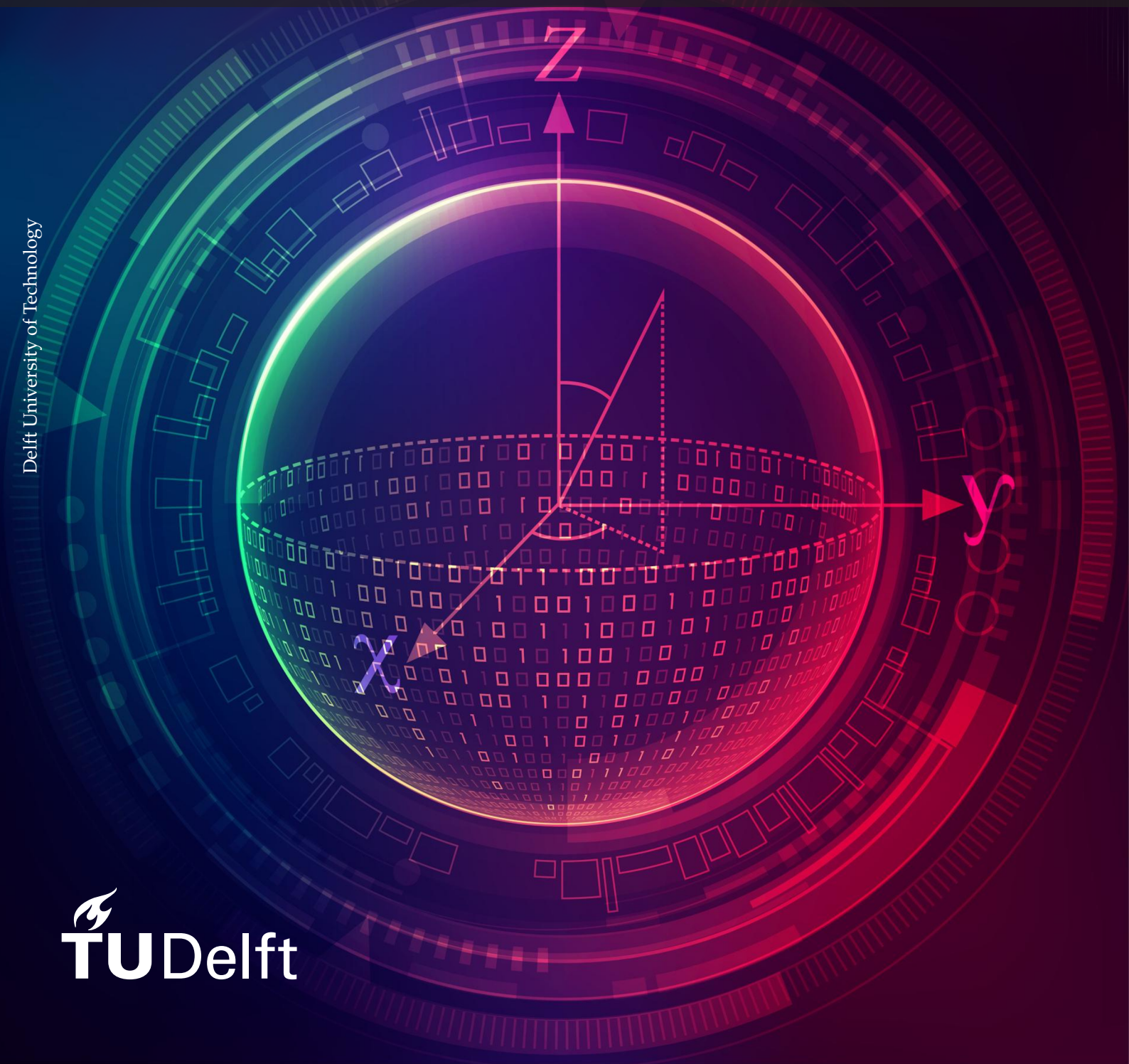
QPack: A cross-platform quantum benchmark-suite

Quantitative performance metrics for application-oriented quantum computer benchmarking

Master Thesis

Huub Donkers



Delft University of Technology

TU Delft

QPack: A cross-platform quantum benchmark-suite

Quantitative performance metrics for application-oriented quantum computer benchmarking

by

Huub Donkers

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on July 6 at 2 PM

Student number:	4475941
Project Duration:	November 8, 2021 - Friday 1st July, 2022
Institution:	Delft University of Technology
	Faculty Electrical Engineering, Mathematics and Computer Science
Thesis Committee:	Dr. ir. Zaid Al-Ars
	Dr. rer. nat. Matthias Möller
	Ir. Aritra Sarkar
	Ir. Koen Mesman

Cover Image: Visualization of the qubit concept - Jackie Niam – stock.adobe.com

TU Delft

Preface

This thesis concludes my journey at the Delft University of Technology, where I have spent seven amazing years of my life, discovering new challenges and meeting a lot of interesting people along the way. I enjoyed working on this thesis the past year, as it allowed me to work on state-of-the-art quantum technologies and to help develop the |Lib⟩ quantum library. Although quantum computing is in its infancy, I am very excited to see what the new way of computing will bring us in the future. The outlook for quantum computing is promising, but there is still a long way to go. Hopefully, this work can contribute to that journey and help in the development of quantum technology in the coming years.

I would like to thank Zaid Al-Ars and Matthias Möller for their guidance during this project. I especially liked the casual style of weekly meetings on Discord, which is a unicum among master students from what I understand. Also, many thanks to Koen Mesman for introducing me to this project, help along the way, and discussions about quantum computer benchmarking in general. I would also like to thank the rest of the quantum team of the research group for the enjoyable weekly meetings and sharing their interesting topics about quantum computing technology. Lastly, I would like to thank my friends and family who supported me in other ways, be it by accompanying me during late night study sessions at the library or allowing me to exploit their coffee machine.

Huub Donkers
Delft, July 2022

Abstract

As the technology of quantum computers improves, the need to evaluate their performance also becomes an important tool for indexing and comparing of quantum performance. Current benchmarking proposals either focus on gate-level evaluation, are centered around a single performance metric, or only evaluate in-house quantum computers. This gives rise to the need for a holistic, application-oriented, and hardware-agnostic benchmarking tool that can provide fair and varied insight into quantum computer performance. This thesis continues the development of the QPack benchmark, which collects quantum computer data by running noisy intermediate-scale quantum (NISQ)-era applications and transforms this data into an overall performance score, which is decomposed into four subscores. These scores are quantitative metrics of quantum performance that allow for easy and quick comparisons between different quantum computers.

The QPack benchmark is an application-oriented cross-platform benchmarking suite for quantum computers and simulators, which makes use of scalable Quantum Approximate Optimization Algorithm and Variational Quantum Eigensolver applications. Using a varied set of benchmark applications, an insight into how well a quantum computer or its simulator performs on a general NISQ-era application can be quantitatively made. QPack is built on top of the cross-platform library $|\text{Lib}\rangle$ (pronounced: libket), which allows for a single expression of a quantum circuit and execution on multiple quantum computers.

Using the QPack benchmarking scores, a comparison is made between various quantum computer simulators, running both locally and on vendors' remote cloud services. Tested local simulators include Qiskit Aer, Cirq, Rigetti QVM, and QuEST. For remote simulators, the IBMQ, IonQ, and Rigetti simulators have been benchmarked. The QPack benchmark is also executed on the Rigetti Aspen-M-1 and a selection of available quantum hardware from the IBMQ aviary, namely the Nairobi, Jakarta, Perth, Lagos, Quito, and Manila processors. For all quantum computers, an analysis is made of their individual performance in the QPack benchmark, as well as an evaluation of how these simulators or hardware implementations compare to each other. Based on the results of the QPack benchmark, the local QuEST simulator, the remote IBMQ QASM simulator and the IBMQ Nairobi and Quito quantum computers achieve best performance compared to the other tested backends.

This work shows that the QPack benchmark is capable of providing holistic quantum computer performance for quantum computers, be it physical implementation or their simulator counterparts. The latest version of the QPack benchmark and all the results collected can be found in the repository: `https://gitlab.com/libket/qpack/-/tree/stable`.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
#AQ	Number of Algorithmic Qubits
CLOPS	Circuit Layer Operations Per Second
CPI	Clock-cycles Per Instruction
CPU	Central Processing Unit
DSP	Dominating Set Problem
FLOPS	Floating-point Operations Per Second
GPU	Graphics Processing Unit
IC	Ising Chain
NISQ	Noisy Intermediate-Scale Quantum
MCP	MaxCut Problem
MIS	Maximal Independent Set
QAOA	Quantum Approximate Optimization Algorithm *or* Quantum Alternating Operator Ansatz
QASM	Quantum Assembly Language
QPE	Quantum Phase Estimation
QPU	Quantum Processing Unit
QV	Quantum Volume
RH	Random Hamiltonian
TSP	Traveling Salesperson Problem
VQA	Variational Quantum Algorithm
VQE	Variational Quantum Eigensolver

Symbols

Symbol	Definition	Unit
A	Adjacency matrix	-
B	QAOA mixer Hamiltonian	-
C	QAOA cost Hamiltonian	-
E	Edge	-
F_p	QAOA expectation value for p	-
G	Graph	-
M_p	QAOA maximum expectation value for p	-
N	QPack problem size	-
p	QAOA iterations	-
P	QPack problem	-
T^{QE}	Circuit execution time	[s]
T^{QJob}	Quantum job time	[s]
S	Subset	-
V	Vertex	-
W	Weights matrix	-
α	Pauli-base set, $\alpha \in \{x, y, z\}$	-
σ^{α}	Pauli-operator	-

1 Introduction

Over the past two decades, quantum computer technology has evolved rapidly. Since the realization of the first physical quantum gate by NIST in 1995 [1], DiVincenzo's famous list of criteria to realize a quantum computer proposed in 1996 [2] and the Oxford university's implementation of a two-qubit quantum computer that solves Deutsch's problem [3], quantum computers have achieved many milestones. One such milestone was achieved in 2014 by the Kavli Institute of Nanoscience at the TU Delft. They teleported information between two qubits separated by 3 meters. Although quantum teleportation has been observed before, the Kavli researchers were able to do so reliably using diamond spin qubits [4]. Another milestone occurred in 2019, when Google claimed that it had achieved quantum supremacy: the ability of a quantum computer to perform a given task faster than a classical computer [5]. Their 53-qubit system could perform a task in 200 seconds, where today's classical supercomputers would need about ten thousand years [6]. However, IBM criticized this claim to actually be 2.5 days [7]. Current state-of-the-art quantum computers push quantum technology even further, with systems such as Google's 72-qubit Bristlecone [8], Rigetti's 80-qubit Aspen-M-1 [9] and IBM's 127-qubit Eagle [10] quantum processors at the forefront of quantum technology.

Still, despite these advances, the current state of quantum computing is labeled the Noisy Intermediate-Scale Quantum (NISQ) era [5], emphasizing the fact that although quantum computing is becoming an important tool to push technological boundaries, it is still in its infancy and many challenges still need to be solved to scale up quantum computing performance.

It should be noted that the NISQ-era is a term used to describe the current state of universal quantum computers. Another quantum computing method called quantum annealing, is a technology that can already operate on many thousands of qubits. Leading this technology is the company D-Wave, with its 5000+ qubit Advantage system being the largest quantum computer currently in existence [11]. However, quantum annealers are mainly used to solve optimization problems. This makes them only useful for certain problems, whereas the universal quantum computer, as the name suggests, will be able to run any quantum algorithm. Scaling up these universal quantum computers is currently one of the most research topics in the field of quantum computing and will therefore also be the type of quantum computer that this thesis focuses on.

When scaling up quantum computers, increasing the qubit count is often pursued. However, the number of qubits is not the only important goal. Qubit relaxation and decoherence time, along with gate fidelity, play an important role in the ability to increase the number of qubits of a quantum processor. As the number of qubits increases, noise and crosstalk tend to rapidly increase the error rate of unwanted qubit rotations [12]. So, a quantum system with a large number of qubits may seem impressive at first sight, but these error rates deserve extra attention, especially in the NISQ-era. After all, what is the benefit of a 1000-qubit quantum computer that cannot produce a meaningful output? Thus, it is important to benchmark this low-level noisy behavior and see where performance gains are feasible. Much work has been put into Quantum Characterization, Verification, and Validation (QCVV), which measures the noise behavior of single or dual qubit gates [13]. Although important, these low-level performance characteristics do not capture the quantum computers' overall performance, giving rise to the problem that the system-level performance of a quantum computer cannot properly be understood by only evaluating its individual components. This illustrates the need for holistic quantum benchmarks

that allow the possibility of comparing the performance of different hardware platforms at the system level.

There have been multiple proposals for benchmarking applications at the system-level using NISQ-era quantum applications [14, 15, 16, 17, 18, 19], but most of them focus on a single performance metric (often evaluation of the quantum computer output), use only a single type of circuit to collect data, or have different scoring metrics per application circuit. QPack aims to overcome these issues by presenting a benchmark evaluation approach that is based on multiple aspects of quantum computer performance and a combination of measurements on a variety of quantum applications.

This thesis presents the QPack cross-platform quantum computer benchmarking suite, which is a continuation of previous work in which QAOA-based applications were used to benchmark IBMQ quantum computers [20, 21]. This work focuses on the further development of the benchmark, where the data obtained during the execution of quantum applications is analyzed and transformed into performance metrics. The goal of the QPack benchmark is the ability to make a quantitative comparison between quantum computers (both physical realizations and simulators) by scoring a quantum computer on multiple characteristics, based on quantum runtime, accuracy, scalability and capacity. This gives insight into the areas where a quantum computer excels and where it can improve performance. It can also provide a general idea of what type of hardware should be used for a desired application. For example, an application may require a more accurate result without runtime being a concern or vice versa. Alternatively, some applications might benefit from a denser connectivity of qubits rather than a large number of qubits. As such, a quantum computer can be selected on the basis of benchmark performance.

To access a multitude of quantum computers in a hardware-agnostic approach, QPack is built on top of the cross-platform library $|\text{Lib}\rangle$ (pronounced: libket) [22, 23], which allows for a single implementation of a quantum circuit and its execution on a variety of quantum computers and simulators. In this thesis, we consider a quantum computer or backend as a system that takes a quantum instruction set as input and returns a state distribution histogram as a result. This covers the whole system of the actual quantum hardware, control system, qubit mapping, and gate scheduling, or simulation thereof. QPack runs NISQ-era quantum applications to collect execution data during runs of quantum computer applications. In this work, the quantum approximate optimization algorithm (QAOA) [24] and the variational quantum eigensolver (VQE) [25] are used as quantum applications, as these variational quantum circuits can already give meaningful results on small and noisy quantum computers. The measured data is then transformed into quantum performance subscores, and finally an overall score based on these subscores is computed.

This thesis starts by highlighting some general knowledge about benchmarking in Chapter 2. Here, the main concept of classical and quantum benchmarking is presented along with some recent proposals for quantum benchmarking. Chapter 3 follows up with some core concepts and motivation for building the QPack benchmark. In Chapter 4, an overview of how the QPack benchmark functions, what quantum execution data is obtained during benchmarking, and the criteria for benchmark scores are presented. The $|\text{Lib}\rangle$ implementations of the VQE and QAOA applications are presented in Chapters 5 and 6, respectively. These chapters now form the foundation for the QPack benchmark scores, which are defined in Chapter 7. Their application on a selection of seven local and remote quantum simulators is presented in Chapter 8, followed by the results obtained on seven actual quantum hardware implementations from Rigetti and the IBMQ aviary. The thesis wraps up with a discussion of the work in Chapter 9 and a conclusion along with some future recommendations in Chapter 10.

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. We also acknowledge the use of IBM Quantum services for this work and the advanced services provided by the IBM Quantum Researchers Program. The views expressed are those of the authors, and do not reflect the official policy or position of IBM or the IBM Quantum team.

2

Measuring computer performance

This chapter presents a theoretical basis for benchmarking for computing systems. An overview of classical computer benchmarking is given, after which some commonly used benchmarks for classical computing systems are highlighted. We then shift our focus to quantum computer benchmarking, where some key similarities but also differences are pointed out. This is then followed up by some recent quantum benchmark proposals, to get a feel of the current state of application-oriented quantum computer benchmarking.

2.1. Benchmarking for classical computers

In *Computer Architecture: A Quantitative Approach* [26], benchmarks are described as programs that are used to establish the relative performance between computers. To evaluate a new system, a user would simply run the same benchmark program on the new device and compare the results. This is supported by Kistowsk et al. [27] who defines a benchmark as "A standard tool for the competitive evaluation and comparison of competing systems or components according to specific characteristics, such as performance, dependability, and security". Typically, computer benchmarks fall into three categories: specification-based, kit-based, and a hybrid method between those two. Specification-based benchmarks focus on the output of a function for a specified input set without requiring a specific implementation, whereas kit-based benchmarks provide the implementation as a required part of benchmark execution. Usually, a benchmark takes characteristics from both methods, making the hybrid version a more common occurrence in benchmarking. One can also make a distinction in benchmarks between the levels at which performance is measured, either at the component- or system-level. Here, as the name suggests, the benchmark focuses on the performance of either a specific component of a computer (e.g., CPU, GPU, or memory) or the system as a whole. Another way of distinguishing benchmarks is their composition, referring to either a synthetic or application benchmark. Synthetic benchmarks are generated by combining basic computer functions that provide an indicative measure of the performance capacity of the tested machine. Application benchmarks, on the other hand, measure performance by evaluating real-life user applications, giving a more insightful measure to see how well a computer performs when handled by a user.

2.1.1. Design

Designing a workload for hardware can be difficult, as one often has to balance several (often conflicting) criteria for usable benchmarks [27, 28]. There are several key properties for benchmarking computer hardware:

1. Relevance: Benchmarks should measure important features.
2. Representativeness: Benchmark performance metrics should be broadly accepted by industry and academia.
3. Equity: All systems should be fairly compared.
4. Repeatability: The benchmark results should be verifiable.
5. Cost-effectiveness: Benchmark tests should be economical.
6. Scalability: Benchmark tests should measure from a single server to multiple servers.
7. Transparency: Benchmark metrics should be readily understandable.

Standardization of a benchmark must ensure these criteria. However, even if all those criteria can be balanced, there are some common pitfalls when designing a benchmark [26]:

- Benchmark only runs a small part of a real-life application, making computers appear faster than they would when running the whole program.
- Unclear benchmark conditions, e.g., improve benchmark performance by using benchmark-specific compiler flags.
- A benchmark program may not reflect the overall performance of a computer.

To overcome these issues, a collection of benchmark applications called *benchmark suites* are widely used to measure performance using a variety of benchmark applications. Designers must specify benchmarking conditions, and it should also be made clear which modifications to the source code are allowed.

A final fallacy in the design of benchmarks is that benchmarks will remain valid indefinitely. Performance metrics may change over time, and as a benchmark becomes more popular, hardware designers may want to modify their systems to perform well on a benchmark, making the benchmark itself a lesser indicator of overall system performance. Updates are thus an important requirement for the longevity of a benchmark.

2.1.2. Classical benchmarks

With the commercialization of computing systems, the need for performance measurements also increased due to the need of the customer. Typically, measuring the speed of a computer is a dominant metric to determine performance. In early computer days, typically the MIPS (Million Instructions Per Second) was a popular measurement to determine a computer's performance, but has lost its significance with the introduction of reduced instruction set computers (RISC), which could most often perform the same high-level instruction of a complex instruction set computer (CISC) in a similar time, but with more instructions. It is only still interesting if you combine this number with the clock-cycles per instruction (CPI) to determine the total runtime. With this in mind, benchmark applications were created and the runtime of program execution could be used to compare computing systems [29].

With some basics covered, the rest of this subsection lists some commonly used benchmarks in classical computing and shows what metrics are used to measure computer performance.

LINPACK

The LINPACK benchmark first appeared in 1979 as an application to measure the number of million floating-point operations per second (Mflops) [30]. The benchmark was originally used to solve a linear system $Ax = b$, for a 100x100 matrix A. With the improvement in computer performance over the years, larger sizes have been proposed as well [29, 30] (see an overview in Table 2.1) often using giga-flops (Gflops) as a more convenient unit of measurement.

Benchmark	Matrix dimensions	Optimization allowed	Parallel processing
LINPACK 100	100	Compiler	Compiler only
LINPACK 1000	1000	Manual	Multi-core only
LINPACK Parallel	1000	Manual	Yes
HPLinpack	arbitrary	Manual	Yes

Table 2.1: LINPACK benchmark overview [30]

Manufacturers often refer to the peak performance rate when indexing their systems

$$R_{\text{peak}} = N \cdot f \tag{2.1}$$

where N is the theoretical number of floating-point operations per clock cycle and f the clock rate of the processor. This is of course a bad indication of the performance of a processor, as the average Mflops usually do not come close to the peak rate. For the devices tested by Dongarra et al. [30], the average Mflops were approximately 25% of the peak rate [30]. A later report about the Sunway TaihuLight System [31] stated that it was able to achieve a 74.15% rate of it's 125.4 Pflops peak rate for the HPL benchmark, but only 1.2% for a real-life application (cloud-resolving atmospheric simulation in

this case). This indicates that the simplicity of the LINPACK benchmark may give a better performance indication than the theoretical rate, but can also be far from the performance of real-life applications.

LINPACK is currently still used to measure computer performance. Its success over the years can mainly be credited to its large historical database, its simplicity of operation, and its ability to capture the best and worst of programming in a single number [32].

SPEC
The Standard Performance Evaluation Corporation (SPEC) was founded because many experts felt that previously developed benchmarks were inadequate and could not be compared with real programs regarding, for example, testing system memory. The goal of SPEC is to distribute and standardize large programs that can be used as benchmarks [29]. SPEC covers a wide variety of benchmarks, targeting many computing areas such as CPU performance, graphics, HPC, cloud computing, machine learning, and more [33]. They have defined two main units of measurement:

- SPECspeed: The time to complete a workload.
- SPECrate: Work completed per time unit; in this case, jobs/hour.

Depending on the chosen benchmark and the targeted area of computing, these metrics may have more specific definitions to give a better understanding of computing performance within certain constraints, for example, SPECint or SPECfp.

BAPCo
BAPCo takes a more creative approach to determining a system's performance and is more geared toward user experience than raw computational metrics [34]. In their main benchmark SYSMark25, they define an overall benchmark score based on three scenarios, shown in Figure 2.1.

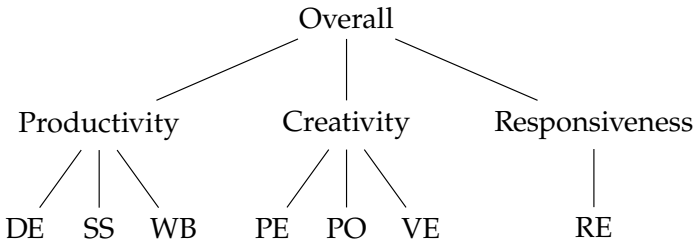


Figure 2.1: Sysmark25 model overview [35]

The productivity scenario contains user interaction with a system such as document editing (DE), spreadsheets (SS), and web browsing (WB). The creativity scenario focuses more on photo editing (PE), photo organization (PO), and video editing (VE). Responsiveness is a more general scenario that measures application launches and file openings, taken from the other scenarios. The ratings are based on the geometric mean of the response times of the subscenarios. Due to the benchmark using industry-standard programs such as Microsoft Office and Adobe, this benchmark is currently only used for Windows operating systems. BAPCo also provides other benchmarks, such as MobileMark25 and CrossMark, to benchmark Android and iOS devices.

EEMBC
The Embedded Microprocessor Benchmark Consortium (EEMBC) is a non-profit organization that has developed a variety of benchmark suites to measure the performance of all kinds of embedded processor implementations, with their ULPMark (ultra low power performance), CoreMark (processor core functionality), and SecureMark (efficiency of cryptographic solutions) being their most popular benchmarks [36].

CoreMark aims to provide a simple and standardized benchmark which can reflect meaningful information about the CPU, by using common big data structures and algorithms often found in practical applications: list processing, matrix manipulation, state machines, and cyclic redundancy checks. CoreMark not only measures performance during computations, but also verifies that they were completed

correctly. The CoreMark score is defined as benchmark iterations per second. For a more scalable metric, CoreMark/MHz is used as well [37].

2.2. Benchmarking for quantum computers

The current state of quantum computing is called the Noisy Intermediate-Scale Quantum (NISQ) era, where quantum computers have been realized, but can only produce meaningful results for a limited number of qubits and circuit depth. In order to build a quantum computer, Divicenzo [38] lists five (plus two) criteria for the physical implementation of quantum computers:

1. A scalable physical system with well characterized qubits.
2. The ability to initialize the state of the qubits to a simple fiducial state, such as $|000...\rangle$.
3. Long relevant decoherence times, much longer than the gate operation time.
4. A "universal" set of quantum gates.
5. A qubit-specific measurement capability.
6. The ability to interconvert stationary and flying qubits.[1]
7. The ability to faithfully transmit flying qubits between specified locations.[1]

These requirements will be considered fulfilled for the purpose of benchmarking, as benchmarks are meant to evaluate the performance of working machines, rather than to test if a system is working at all. Benchmarking, however, is a very useful tool for finding areas where a quantum computer can improve performance. For example, benchmarking can be used to see how well qubits are initialized or if a system is scalable.

2.2.1. Quantum benchmarking levels

Similar to classical benchmarking, quantum benchmarking can be divided into two benchmark levels, namely component- and system-level benchmarking. The prior focuses mainly on the behavior of individual components of the quantum computer, while the latter evaluates the performance of the system as a whole.

Component-level benchmarking

As the growth of quantum computing hardware accelerates, so does the need to evaluate its components. Especially in the NISQ-era of quantum computing, gate noise and qubit decoherence time have a large influence on the capability of quantum systems to handle increasing circuit sizes.

The analysis of these low-level components is performed using quantum characterization, verification, and validation (QCVV), which targets measurements of the noise behavior of single or dual qubit gates [13]. Characterization determines the effect of noise and control on a quantum system and is therefore the lowest-level information about a quantum computer on which other information is founded. Verification concerns itself with how well quantum operations are executed, e.g., how close an experimental operation is to an ideal operation. In practice, verification overlaps much with characterization, as error rates strongly relate to how well an operation can be executed. Validation methods are used to confirm that an implementation is executed as designed [39].

Examples of low-level benchmarking methods are state-preparation-and-measurement [40], randomized benchmarking [41] and gate set tomography [42].

System-level benchmarking

System-level benchmarking evaluates the performance of the complete quantum system. In the field of quantum computing, a well-known system-level performance metric is the Quantum Volume, a term coined by IBM [43] when they introduced it in 2019. Since then, a variety of other volumetric benchmarks [14, 15, 16] have been proposed. A volumetric benchmark has the following characteristics [13]:

1. It maps the width and depth of the circuit to ensembles of quantum circuits $C(w, d)$ (single circuits or a set of (randomly selected) circuits).
2. It rules detailing constraints on how the circuit may be compiled to native gates.

[1] The last two of Divicenzo's requirements are not necessarily required for building a quantum computer, but rather for communication between quantum computers.

3. It defines a measure of "success", e.g., at what threshold does a circuit "pass" a test or how well does a circuit score.
4. It defines measure of "overall success" on an ensemble of circuits.
5. Optional: It provides an experimental design of how circuits should be run.

Non-volumetric performance metrics have also been proposed for system-level benchmarking and will be highlighted in Section 2.3.

2.2.2. Quantum benchmarking types

Blume-Kohout and Young [13], divide benchmark circuits into three classes for volumetric benchmarking: random circuits, periodic circuits, and application circuits.

A random circuit is defined by randomized subroutines that build up larger circuits. For example, Magesan et al. [44] proposed a sequence of $m+1$ quantum operations with the first m operations chosen uniformly at random from a group of unitaries, with the last operation chosen such that the complete circuit is the identity operation. Other examples of randomized circuits have been proposed by Google to demonstrate quantum supremacy [45] or by IBM to run their QV benchmark [43].

Periodic circuits lend their name to the fact that they have a periodic, rather than a random structure. Examples of these are Rabi sequences, gate set tomography [42], and even Grover's algorithm [46]. The benefit of periodic circuits is that they amplify coherent errors, which would not be as apparent when using randomized circuits.

Application circuits are not as clearly defined as random or periodic circuits, but cover the whole range of circuits that are expected to be used in real-world applications. These circuits are neither random nor periodic, but are rather formed to fit a required problem solving algorithm. Examples of such application-oriented circuits used in proposed benchmarks will be given in Section 2.3.

2.3. Proposed system-level benchmarks

This section will cover an overview of some recently proposed benchmarks, both volumetric and non-volumetric. All benchmarks use application circuits, except for the IBM benchmark which makes use of randomized circuits, but is included as their system-level benchmarks lay an important foundation for holistic quantum computer benchmarking. The benchmark proposals covered in this section have been published in the past three years and are listed in chronological order.

2.3.1. IBM: Quantum Volume

IBM was the first to propose a widely accepted benchmark in October 2019 [43]. They proposed the Quantum Volume as a single-number performance metric for near-term quantum computers of modest size (< 50 qubits). Quantum execution data is obtained by running randomized circuits (see Figure 2.2) with a width of m qubits and a depth of d.

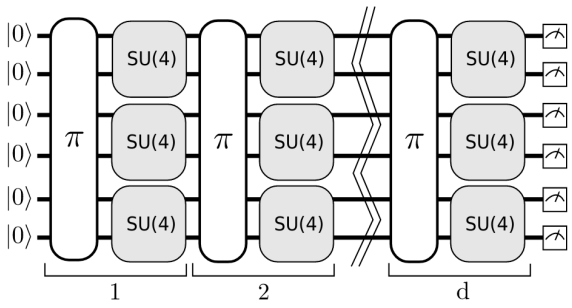


Figure 2.2: Model circuit with qubit width m and number of layers d

The QV is then defined as 2^m for the largest square circuit ($m = d$) that a quantum computer can successfully implement. In other words:

$$\log_2 V_Q = \underset{m}{\text{argmax}} \min(m, d(m)) \tag{2.2}$$

For the QV, success means a probability of sampling *heavy output* above $\frac{2}{3}$. *Heavy output* is defined as the sum of all the state probabilities that are above the median probability of the total state distribution. In principle, the higher a quantum computer's quantum volume, the more complex a circuit can be successfully executed.

2.3.2. Zapata: Fermionic benchmark

In March 2020, Zapata proposed the ground state problem of the Fermi-Hubbard model as a benchmark [17]. This model can be solved exactly using the Bethe ansatz [47] for finite and infinite chains. Using VQE, the ground state can be obtained with quantum computers. As the chain length L increases, the ground energy approaches the infinite chain limit E_0. For a NISQ-era quantum computer, an effective fermionic length L^* is a characterization of a quantum computer's performance, since at this length it will diverge from E_0. In other words, the benchmark scores a quantum computer on how well it can simulate one-dimensional fermionic behavior.

2.3.3. Atos: Q-score

In February 2021, Atos Quantum Laboratories proposed a new non-volumetric benchmark, dubbed the Atos Q-score [14]. The Q-score is defined as the maximum number of qubits that can be used effectively to solve the MaxCut combinatorial optimization problem using the QAOA. The score is computed by running 100 random Erdös-Renyi graphs [48] $G(n, p = 0.5)$ and seeing for which n the algorithm has an average energy above a certain threshold $\beta^\star$. Here, $\beta^\star$, has a value between 0 and 1, where 0 indicates a random result and 1 the result of an exact solver. The Q-score is then the maximum n qubits for which this threshold is reached:

$$n^\star \equiv \max\{n \in \mathbb{N}, \beta(n) > \beta^\star\} \tag{2.3}$$

Atos decided on $\beta^\star = 0.2$ for an arbitrary threshold value and the score $\beta(n)$ is:

$$\beta(n) = \frac{C(n) - \frac{n^2}{8}}{\lambda n^{3/2}} \tag{2.4}$$

This score depends on the average energy (accuracy) $C(n)$ produced by the QAOA circuit, fitted with parameter $\lambda = 0.178$ such that $\beta(n)$ is bounded by the average score of the MaxCut problem.

2.3.4. TU Delft: QPack

In a recent paper by Mesman et al. [21] published in 2021, an application-oriented benchmark was proposed using QAOA applications for the evaluation of quantum performance. A schematic overview of the benchmark is given in Figure 2.3.

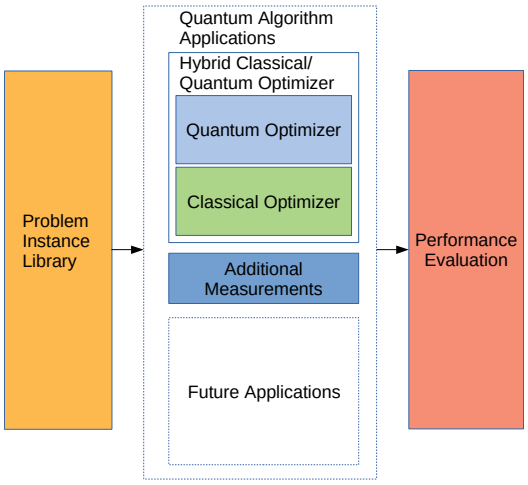


Figure 2.3: Overview of the QPack benchmark [21]

Other than volumetric benchmarks, this benchmark implementation considers metrics more related to the actual useful aspects of quantum computing hardware, namely runtime, accuracy, and scalability. Current implementations of the problem set library are the maximum cut, dominating set and traveling salesperson problem for QAOA. These problems are mapped to a 4-regular graph, which is increased in size (number of vertices) to scale the QAOA problems for more qubits. The benchmark has presented runtime and accuracy results for these problems for the IBMQ QASM simulator [49], as well as the IBMQ Montreal and Nairobi quantum processors [50]. The paper stresses the importance of runtime as a quantum performance metric, as it will likely become the dominant metric as quantum computing moves on from the NISQ-era.

2.3.5. QuSoft

An alternative approach to quantum benchmarking was proposed by QuSoft in 2021 [18]. They proposed a hardware-agnostic benchmark using real-world NISQ applications. QuSoft mainly emphasizes the interplay between various relevant characteristics of quantum computers, such as qubit count, connectivity, and gate and measurement fidelity. This benchmark defines a different score for each quantum application and takes a mean over normalized scores to present a single-number score. Moreover, instead of just a one-dimensional number to represent a quantum computer's performance, QuSoft also proposes a visual representation of the benchmark outcome, which allows for an unambiguous and single-glance comparison between quantum computers. These visual representations can also reveal interesting features of noise in quantum computers, such as systematic errors in deformed images.

2.3.6. QED-C: Volumetric performance benchmark

Another volumetric benchmark was proposed by the Quantum Economic Development Consortium (QED-C) in October 2021 [15]. This open-source benchmark probes a quantum backend with small applications for which the problem sizes are varied, mapping the fidelity of the results as a function of circuit width and depth, hence making it a volumetric benchmark. QED-C determines the success of a quantum application by comparing the normalized fidelity of the output distribution P_{output} of a quantum computer with the output distribution P_{ideal} of an ideal quantum computer, defined by:

$$F(P_{\text{ideal}}, P_{\text{output}}) = \frac{F_s(P_{\text{ideal}}, P_{\text{output}}) - F_s(P_{\text{ideal}}, P_{\text{uni}})}{1 - F_s(P_{\text{ideal}}, P_{\text{uni}})} \tag{2.5}$$

where P_{uni} is the uniform distribution and

$$F_s(P_{\text{ideal}}, P_{\text{output}}) = \left(\sum_x \sqrt{P_{\text{output}}(x) P_{\text{ideal}}(x)} \right)^2 \tag{2.6}$$

where x is the bitstring that encodes the state. An application is then considered successful if $F > \frac{1}{2}$. In addition to this primary metric, the benchmark also provides insight into runtime and the ratio between the programmed and transpiled circuit depth. QED-C makes the remark that runtime metrics are currently rudimentary, as quantum providers can have different definitions of quantum execution time.

2.3.7. IBM: CLOPS

Stepping away from measuring the performance of the output state of a quantum computer, IBM introduces yet another performance metric based on execution time in October 2021 [51]. The CLOPS metric (Circuit Layer Operations Per Second) measures how many layers of a QV circuit a quantum computer can execute per time unit, using Qiskit-runtime [52]. It is defined as the number of QV layers executed per second using a set of parameterized circuits with $\log_2(\text{QV})$ layers. The execution time that is measured includes updating the circuit parameters, submitting the job to the quantum processing unit (QPU), executing the circuit on the QPU, and sending back the results to be processed. Circuit compilation is not included in this time measurement, and all sets of parameterized circuits are already compiled to the target machine's native gate set before the execution timer is started. This benchmarking approach may give more insight into how well quantum computers perform in contrast to classical computers and may identify where performance improvements with respect to runtime might be gained.

2.3.8. Super.tech: SupermarQ

Another recent proposal for application-oriented quantum benchmarking was published by Super.tech in April 2022 [19]. Super.tech focuses on applications that are not only chosen based on their real-world purpose, but also their coverage of the application space, described using feature vectors *program communication*, *critical-depth*, *entanglement-ratio*, *parallelism*, *liveness* and *measurement*. A quantum computer is then evaluated for each benchmarking application with an application-specific benchmark score. The SupermarQ benchmark is built upon the cross-platform framework SuperstaQ [53], which allows an application written in OpenQASM to be executed on multiple backends that do not support OpenQASM natively.

2.3.9. IonQ: Algorithmic Qubits

IonQ also joined the pursuit of defining a single performance metric for quantum computing, proposing the Algorithmic Qubits (#AQ) in February 2022 [16]. Taking inspiration from the QED-C work (discussed in Subsection 2.3.6), it also derives its benchmark metrics from NISQ-executable quantum applications. IonQ defines $\#AQ = N$ for the largest box with N qubits and a circuit depth of N^2 in which all quantum circuits meets a success criterion. The set of algorithms that have been selected for this benchmark is the same as found in the QED-C repository [54]. The main differences between the QED-C and IonQ benchmarks are the definition of circuit depth (both single- and two-qubit gates vs. only two-qubit gates, respectively) and the success criteria of a quantum circuit. For IonQ, success is also determined using the fidelity as described in Equation 2.6 with

$$F_s(P_{\text{ideal}}, P_{\text{output}}) - \epsilon_s > t \tag{2.7}$$

where ϵ_s is the statistical error based on the number of shots ($\epsilon_s = \sqrt{\frac{F_s(1-F_s)}{\#\text{shots}}}$) and $t = \frac{1}{e} \approx 0.37$ is the threshold.

2.3.10. This work

This thesis covers the ongoing work on the QPack benchmark, by implementing the QPack benchmark on top of the cross-platform library |Lib⟩, expanding the problem library, and a quantitative approach to score the performance of quantum computers. As covered in this section, most benchmarking proposals focus on a single performance metric (often evaluation of the quantum computer output), use only a single type of circuit to collect data, or have different scoring metrics per application circuit. A comparison of the presented benchmark proposals other than QPack is shown in Table 2.2. Here, a benchmark is checked if it evaluates multiple performance metrics (runtime, output state, etc.), is application oriented, uses multiple quantum circuits to make measurements, and if the performance store is a single figure of merit. It can be observed that no current benchmark proposal ticks off all categories, which may give a limited view of quantum computer performance. QPack aims to overcome these limitations by presenting a benchmark evaluation approach that is based on multiple aspects of quantum computer performance and a combination of measurements on a variety of quantum applications.

Table 2.2: Characteristics of presented benchmark proposals

	IBM QV	Zapata L^*	Atos Q-score	QuSoft	QED-C	IBM CLOPS	SupermarQ	IonQ #AQ
Multiple performance metrics	✗	✗	✗	✗	✓	✗	✗	✗
Application-oriented	✗	✓	✓	✓	✓	✗	✓	✓
Multiple benchmark circuits	✗	✗	✗	✓	✓	✗	✓	✓
Single-number benchmark score	✓	✓	✓	✓	✗	✓	✗	✓

3

Benchmark building blocks

Now that some foundational knowledge about benchmarking and some recently proposed quantum benchmarks have been highlighted in Chapter 2, we can move on to constructing the QPack benchmark. Designing a quantum computer benchmark can be done in various ways. This chapter will focus on four main building blocks of the QPack benchmark. These are:

1. A cross-platform quantum-programming library to be used for the creation and execution of hardware-agnostic circuits on multiple quantum computers.
2. Quantum algorithms to be used as benchmarking applications.
3. A set of problems to be solved by these quantum algorithms.
4. A classical optimizer to find the solutions of the chosen problem sets.

For each building block, an overview of its purpose is given, potential implementations are reviewed, and finally a choice for the selected implementation is motivated.

3.1. Quantum cross-platform libraries

In order for the benchmark to be applicable to a variety of quantum computers, a cross-platform library for quantum computers is desirable, as a quantum circuit can be built once in the library environment and executed on multiple quantum computers. This will not only allow for efficient access to current quantum computers, but for easy integration of future quantum computers as well.

3.1.1. XACC

The Oak Ridge National Laboratory presented XACC in November 2019 [55]. XACC is a framework for hybrid quantum-classical computing. It allows the user to create quantum expressions or circuits in their own *xasm* or Rigetti's *quilc* language [56]. The framework features a quantum coprocessor programming model, low-level system-software interfaces, cross-platform operation, and supports modular and extensible functionality. Programs can be written natively in C++ or Python using their API. XACC currently supports the following quantum backends, listed in Table 3.1.

Table 3.1: XACC supported backends [56]

Provider	IBMQ	Qiskit	Rigetti	IonQ	Qsim	Qpp	DWave	TNQVM	Atos QLM	QuaC	Qrack
Hardware	Yes	No	Yes	Yes	No	No	Yes	No	No	No	No
Simulator	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Access	Remote	Local	Local/remote	Remote	Local	Local	Remote	Remote	Remote	Local	Local

3.1.2. |Lib⟩

A year later in 2020, TU Delft presented a new cross-platform programming framework called |Lib⟩ (pronounced: libket) [22]. This framework uses a CUDA-inspired [57] streaming concept to use quantum hardware as an accelerator, like one would with a GPU. |Lib⟩ is designed based on the principles of QPU-accelerated computing, concurrent task offloading, single-source quantum-classical programming, using generic expressions, development on top of existing tools, and seamless integration into the status quo. Programs are written natively in C++, but some early stage Python and C APIs are

developed as well. According to the online documentation [23], |Lib⟩ supports both remote quantum backends as well as local simulators, listed in Table 3.2.

Table 3.2: |Lib⟩ supported backends [23]

Provider	IBMQ	Qiskit	Rigetti	IonQ	Atos QLM	Cirq	QuEST	QX
Hardware	Yes	No	Yes	Yes	No	No	No	No
Simulator	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Access	Remote	Local	Local/remote	Remote	Remote	Local	Local	Local

3.1.3. SuperstaQ

In August 2021, Super.tech announced a new cross-platform library: SuperstaQ [53, 58]. SuperstaQ takes input from Qiskit, Cirq, or OpenAPI environments and can execute data on various quantum systems, listed in Table 3.3. This platform aims to improve quantum computing performance via optimizations in the entire system stack, down to control pulses that perform quantum operations.

Table 3.3: SuperstaQ supported backends [53]

Provider	IBMQ	Qiskit	Rigetti	IonQ	Berkeley-AQT
Hardware	Yes	No	Yes	Yes	Yes
Simulator	Yes	Yes	Yes	Yes	Yes
Access	Remote	Local	Both	Remote	Remote

3.1.4. QPack platform of choice

Using a cross-platform library is clearly beneficial when building a universal quantum benchmark. All highlighted platforms have their own benefits and downsides. For QPack, |Lib⟩ is the platform of choice. SuperstaQ is currently limited in the number of quantum computers it can run circuits on and is not yet available for public use. Although XACC has a larger range of providers than |Lib⟩, the benefit of using |Lib⟩ is that it is also developed by TU Delft, allowing for customization of the source code which is necessary to suit the QPack needs for data acquisition.

3.2. Quantum algorithms

In this section, an overview of applications that are suitable for the QPack benchmark is presented. Specifically, different variations of hybrid classical-quantum algorithms, called variational quantum algorithms (VQAs), will be highlighted. The reason that we consider VQAs as benchmarking applications is that they have been proposed as a leading strategy to work within the constraints of the NISQ-era quantum computers. This makes VQAs an appropriate choice to use as benchmarking applications in the current state of quantum technology. However, some pure quantum algorithms exist that could be used for benchmarking, such as quantum phase estimation [59], Shor's factorization algorithm [60] or Grover's search algorithm [46] (see Appendix B). However, these were ultimately not chosen, as they are not suited so produce meaningful results in the NISQ era and can only reach their full potential when quantum computers become more fault-tolerant [61]. In the following subsections, a review and main purpose of each VQA are briefly stated.

3.2.1. VQA in general

VQAs use a classical optimizer to train a parameterized quantum circuit (see Figure 3.1). According to Cerezo et al. [61], these algorithms will pave the way for all applications for which quantum computers were envisioned. The main idea is that these parameterized circuits are small and thus already give meaningful results on NISQ-era computers, which can be processed further by a classic computer. The VQA consists of a classical and a quantum subroutine. The quantum circuit (or ansatz) contains a set of parameterized gates. A classical optimizer can change the parameters of these gates and evaluate the change in output. Using a cost function, the cost of the output state of the quantum circuit is determined. The goal of the optimizer is then to minimize this cost by changing the parameters of the

quantum circuit. When the optimal parameters are found, the optimizer outputs these parameters that encode the solution to a given problem.

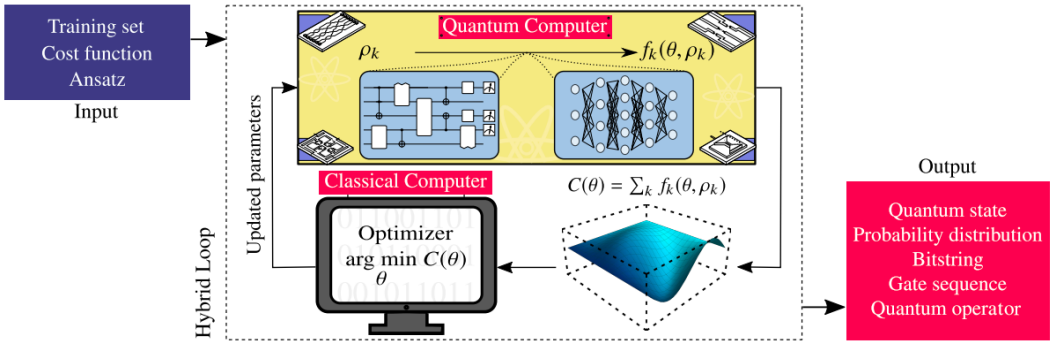


Figure 3.1: Schematic representation of a VQA [61]

VQAs have the advantage that their framework can be used for a variety of tasks. Essentially, all applications for quantum computing can be performed with VQAs and allow for universal quantum computing [61]. These other applications can make VQA very useful for many quantum related fields. VQAs, for example, can be used to perform simulations of a dynamical quantum system (e.g., with Trotterization [62]). Mathematical applications are also possible. For instance, solving systems of linear equations or integer factorization. VQAs could also be used for quantum error correction and compilation, decreasing circuit depth and allowing for a more robust quantum circuit in the NISQ-era.

3.2.2. The variational quantum eigensolver

In April 2013, the first VQA was proposed by Peruzzo et al.; the variational quantum eigensolver (VQE) [25]. This algorithm can be used to find the ground state of a given Hamiltonian and is mainly used in quantum chemistry.

The algorithm consists of two main parts. The first part is the quantum subroutine that computes the (partial) expectation values of the Hamiltonian H for a parameterized input state $|\psi(\boldsymbol{\theta})\rangle$. Any Hamiltonian may be written as

$$H = \sum_{i,\alpha} h_i^\alpha \sigma_i^\alpha + \sum_{i,j,\alpha,\beta} h_{i,j}^{\alpha,\beta} \sigma_i^\alpha \sigma_j^\beta + \ldots \tag{3.1}$$

for real h where i, j denotes the subspace on which the operators act and $\alpha, \beta \in \{x, y, z\}$ identifies the Pauli operator (only the first and second order terms are shown in Equation 3.1, but higher order terms are implied). The expectation value for an arbitrary state $|\psi\rangle$ of this Hamiltonian is then just the sum of the partial Hamiltonian operators, as

$$\begin{aligned} \langle H \rangle &= \langle \psi | H | \psi \rangle \\ &= \sum_{i,\alpha} h_i^\alpha \langle \psi | \sigma_i^\alpha | \psi \rangle + \sum_{i,j,\alpha,\beta} h_{i,j}^{\alpha,\beta} \langle \psi | \sigma_i^\alpha \sigma_j^\beta | \psi \rangle + \ldots \\ &= \sum_{i,\alpha} h_i^\alpha \langle \sigma_i^\alpha \rangle + \sum_{i,j,\alpha,\beta} h_{i,j}^{\alpha,\beta} \langle \sigma_i^\alpha \sigma_j^\beta \rangle + \ldots \end{aligned} \tag{3.2}$$

which means that the number of circuits to evaluate scales with K, where K is the number of terms in the decomposition of the Hamiltonian H. The total expectation value of H can be found by computing each partial expectation value and summing them up on a classical computer. In the special case that $|\psi\rangle = |\psi_{\text{min}}\rangle$ (ground state of H), the expectation value $\langle H \rangle$ is the ground state energy of the Hamiltonian H. The main idea behind VQE is to find this ground state energy by tuning a parameterized unitary to approximate the ground state.

The second part of the VQE consists of a classical optimizer which evaluates the computed expectation value. Based on this input, the optimizer then alters the input state $|\psi(\boldsymbol{\theta})\rangle$ by varying the vector

$\boldsymbol{\theta} = (\theta_0, \theta_1, ..., \theta_{n-1})$ with n optimization parameters, such that it minimizes the expectation value

$$\langle H \rangle = \langle \psi(\boldsymbol{\theta}) | H | \psi(\boldsymbol{\theta}) \rangle \tag{3.3}$$

where the state $|\psi(\boldsymbol{\theta})\rangle$ that gives the lowest expectation value is considered the ground state of the Hamiltonian H. An important aspect of VQE is choosing an appropriate ansatz unitary $U(\boldsymbol{\theta})$ that creates the parameterized input state

$$\psi(\boldsymbol{\theta}) = U(\boldsymbol{\theta}) |\psi_{\text{init}}\rangle \tag{3.4}$$

where ψ_{init} is the initial state of the system (often the zero state $|00...0\rangle$ or superposition $|+ + ...+\rangle$). The choice of ansatz is mainly dependent on the type of Hamiltonian that needs to be evaluated, as it should be able to cover all eigenstates (or at least the ground state) of the Hamiltonian. Such an ansatz for the single-qubit gate can be achieved with the $U_3(\lambda, \phi, \theta)$ unitary [63], where

$$U_3(\lambda, \phi, \theta) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\lambda+\phi)}\cos(\theta/2) \end{pmatrix} \tag{3.5}$$

which rotates a qubit over each axis of the Bloch sphere, covering all possible states. For the two-qubit case, an entangled circuit was proposed by Shende et al. [63], which uses 8 of these U_3 unitaries and 3 CNOT gates to create the ansatz. Other commonly used ansatzes in quantum chemistry are the unitary coupled cluster (UCC) [64] and hardware-efficient SU2 ansatzes [65].

3.2.3. Quantum approximate optimization algorithm

The quantum approximate optimization algorithm (QAOA) is a method used to find approximate solutions to combinatorial optimization problems, often expressed as a graph-based problems. It is a very common occurrence in VQAs, which has led to its use by Atos [14] and QPack [21] as a benchmarking application. QAOA was introduced by Fahri et al. in 2014 [24] and consists of two main parts: a cost unitary and a mixer unitary, based on cost and mixer Hamiltonians, respectively. The cost Hamiltonian C is based on the cost function f of the combinatorial optimization problem, usually as a sum of m clauses:

$$f(z) = \sum_{\alpha=1}^{m} f_\alpha(z) \tag{3.6}$$

where $z = z_1 z_2 ... z_n$ is the bit string to evaluate, m the number of clauses and $f_\alpha(z) = 1$ if z satisfies clause α and 0 otherwise. Because a quantum computer works in a 2^n-dimensional Hilbert space with computational basis vectors $|z\rangle$ and the cost function in Equation 3.6 is viewed as an operator that is diagonal in the computational basis, the cost Hamiltonian C encodes this cost function as:

$$C |z\rangle = f(z) |z\rangle \tag{3.7}$$

This cost Hamiltonian is then used as an operator in a unitary operator:

$$U_C(\gamma) = e^{-i\gamma C} = \prod_{\alpha=1}^{m} e^{-i\gamma C_\alpha} \tag{3.8}$$

which depends on angle $\gamma \in [0, 2\pi]$. Next, we define the mixer Hamiltonian B. In the original QAOA algorithm, this a sum of single-qubit σ^x operators.

$$B = \sum_{j=1}^{n} \sigma_j^x \tag{3.9}$$

where n is the number of qubits and σ_j^x the Pauli-X operation on qubit j. The unitary can then be defined as:

$$U_B(\beta) = e^{-i\beta B} = \prod_{j=1}^{n} e^{-i\beta\sigma_j^x} \tag{3.10}$$

with $\beta \in [0, \pi]$. These unitaries are then applied to an initial state $|s\rangle$ in superposition, i.e.,

$$|s\rangle = \frac{1}{\sqrt{2^n}} \sum_z |z\rangle = |+\rangle_1 |+\rangle_2 ... |+\rangle_n \tag{3.11}$$

The cost and mixer unitary can then be applied p times in alternating order to the initial state. We then arrive at the characteristic quantum state for QAOA, that is,

$$|\boldsymbol{\gamma}, \boldsymbol{\beta}\rangle = U_B(\beta_p)U_C(\gamma_p)...U_B(\beta_2)U_C(\gamma_2)U_B(\beta_1)U_C(\gamma_1)\,|s\rangle \tag{3.12}$$

where $\boldsymbol{\gamma} = (\gamma_1, \gamma_2, ..., \gamma_p)$ and $\boldsymbol{\beta} = (\beta_1, \beta_2, ..., \beta_p)$. This means that the algorithm will always have $2p$ parameters, where $p \in \mathbb{Z}^+$. We can now use this parameterized quantum state with a classical optimizer, which attempts to maximize (or minimize) the expectation value of C by finding the optimal parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$. The expectation value of C is then simply:

$$F_p = (\boldsymbol{\gamma}, \boldsymbol{\beta}) = \langle \boldsymbol{\gamma}, \boldsymbol{\beta}|\, C\, |\boldsymbol{\gamma}, \boldsymbol{\beta}\rangle \tag{3.13}$$

and the maximum value:

$$M_p = \max_{\boldsymbol{\gamma},\boldsymbol{\beta}} F_p(\boldsymbol{\gamma}, \boldsymbol{\beta}) \tag{3.14}$$

Note that

$$M_p \geq M_{p-1} \tag{3.15}$$

and thus increasing p will generally result in a higher maximum result. However, for NISQ-era quantum hardware, only small values for p are possible to use. Generally, only a few p iterations are used, because circuits need to be kept small. This results in a rough approximation of the Hamiltonian.

Extended algorithm
Five years after the introduction of QAOA, an extension of the algorithm was proposed, named the *Quantum Alternating Operator Ansatz* to keep the abbreviation the same [66]. This extension allows the algorithm to have a more varied set of states than the original version. The paper takes a more general approach to optimization. An *optimization problem* is a pair (F, f), where F is the domain and $f : F \to \mathbb{R}$ is the objective function. In quantum terms, this corresponds to $\mathcal{F}$ being the Hilbert space of dimension $|F|$, with standard basis $\{|x\rangle : x \in F\}$. The QAOA circuit is then constructed using repeating unitaries:

- Phase separation operators $U_P(\gamma)$, depending on the objective function f
- Mixing operators $U_M(\beta)$, depending on the domain and its structure

The QAOA algorithm then comes down to three parts: The phase separation operator, mixing operator, and a starting state. This extension allows for more than just Hamiltonian-based cost functions and mixers and can be useful to map more complex problems to the QAOA algorithm.

3.2.4. Variational quantum linear solver

The variational quantum linear solver (VQLS) was introduced in 2019 by two teams of authors separately, Bravo-Prieto et al. [67] and Chen et al. [68]. Both algorithms are used to solve linear systems of the form $A\boldsymbol{x} = \boldsymbol{b}$. This is a more NISQ-era implementation to solve such systems, whereas other approaches such as the Harrow-Hassidim-Lloyd (HHL) quantum algorithm [69] will need larger and low-noise quantum computers to realize their full potential.

The VQLS algorithm works as follows. Consider a system of the form $A\,|x\rangle = |b\rangle$, with known matrix A and known vector $|b\rangle$, where the latter is the normalized version of $\boldsymbol{b}$. The VQLS algorithm attempts to find normalized $|x\rangle$, proportional to $\boldsymbol{x}$. A given square matrix A can be expressed as a linear combination of unitary matrices

$$A = \sum_{l=1}^{L} c_l A_l \tag{3.16}$$

where A_l are the unitaries to be implemented with a quantum circuit and c_l are complex numbers. It is assumed that $|A| \leq 1$, and that the A_l unitaries can be efficiently implemented as quantum circuits. The VQLS then aims to prepare a quantum state $|x\rangle$, such that $A\,|x\rangle$ is proportional to $|b\rangle$, i.e.,

$$|\Psi\rangle \equiv \frac{|\psi\rangle}{\langle\psi|\psi\rangle} = \frac{A\,|x\rangle}{\sqrt{\langle x|\, A^\dagger A\,|x\rangle}} \approx |b\rangle \tag{3.17}$$

where $|b\rangle = U_b |0\rangle$, for some unitary U_b that prepares the state $|b\rangle$. The solution $|x\rangle$ of the problem can be approximated with VQLS with a unitary circuit V, dependent on classical parameters $\boldsymbol{\alpha}$

$$|x\rangle = V(\boldsymbol{\alpha}) |0\rangle \tag{3.18}$$

The resulting state $|\Psi\rangle$ (Equation 3.17) can now be varied to find the minimal value. For this, two approaches are considered, using a global cost function C_G or a local cost function C_L. The global cost function approach considers the maximal overlap of the $|\Psi\rangle$ state with the $|b\rangle$ state, which can be expressed as

$$C_G = 1 - |\langle b|\Psi\rangle|^2 \tag{3.19}$$

Because we want to minimize the cost, C_G approaches zero as the overlap between the $|\Psi\rangle$ and $|b\rangle$ states increases. Using the global cost function can be difficult for optimization if the norm of $|\psi\rangle$ is small and if a large number of qubits are used, which results in many barren plateaus appearing in the cost function. To negate this, a local cost function is introduced, as

$$C_L = \frac{\langle x| H_L |x\rangle}{\langle\psi|\psi\rangle} \tag{3.20}$$

where the local Hamiltonian is

$$H_L = A^\dagger U_b \left(\mathbb{1} - \frac{1}{n} \sum_{j=1}^{n} |0_j\rangle \langle 0_j| \otimes \mathbb{1}_{\bar{j}} \right) U_b^\dagger A \tag{3.21}$$

Here, $|0_j\rangle$ is the zero state on qubit j and $\mathbb{1}_{\bar{j}}$ the identity operator on all qubits except qubit j. Finding the expectation value of the cost function C_G or C_L can be achieved using the Hadamard test [70], as depicted in Figure 3.2. The Hadamard test finds the expectation value of a unitary U for a state $|\phi\rangle$.

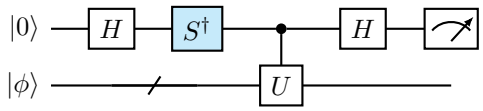


Figure 3.2: Hadamard test circuit to find the expectation value of U. The colored phase gate is not included when calculating the real part of the expectation value and is included when calculating the imaginary part of the expectation value.

For repeated measurements of the first qubit, the probability of measuring 0 and 1 can be estimated. The probability of these measurements is related to the real and imaginary components of the expectation value $\langle U\rangle$:

$$\mathrm{Re}\langle U\rangle = P(0) - P(1) = \frac{1}{2}(1 + \mathrm{Re}\langle U\rangle) - \frac{1}{2}(1 - \mathrm{Re}\langle U\rangle) \tag{3.22}$$

The imaginary part of the expectation value can be found by adding a $S^\dagger$ gate to the Hadamard test circuit (blue gate in Figure 3.2) and applying Equation 3.22 again for the measured probabilities. Using the Hadamard test is a simple way to compute the expectation value, but needs to have control over all unitaries V, A_l, and U_b. Another method that circumvents this is the Hadamard-Overlap test [67], which negates the need for control of the V and U_b unitaries at the cost of doubling the number of qubits.

3.2.5. Variational quantum classifier

A VQA approach is also applicable to perform tasks that are more commonly solved using machine-learning, such as classification of data points into two distinct sets based on feature vectors [71, 72, 73]. These types of algorithms are known as variational quantum classifiers (VQCs). For this review, we will follow the implementation by Havileck et al. [73], but others have a similar approach. The VQC follows the approach of supervised learning. Consider a training data set T and a test data set S. Both sets contain a set of feature vectors $\boldsymbol{x}$ that are labeled by a map $m : T \cup S \to \{+1, -1\}$, unknown to the algorithm. When training the algorithm, it only receives labels from T. The goal of the algorithms is then to find the approximate map $\tilde{m} : S \to \{+1, -1\}$, such that is agrees with a high probability with

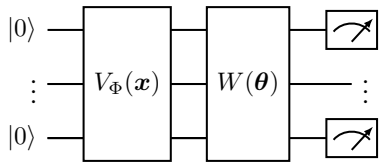


Figure 3.3: General circuit approach of a VQC

the true map m. Figure 3.3 visualizes the general approach of the VQC, where it can be observed that the VQC circuit consists of two unitary operators. The first unitary operator V_Φ 'loads' the data set onto the quantum computer using a quantum feature map:

$$\Phi : \boldsymbol{x} \in \Omega \rightarrow |\Phi(\boldsymbol{x})\rangle \langle\Phi(\boldsymbol{x})| \tag{3.23}$$

where Ω is the complete data set. For NISQ-era quantum computers, commonly used datasets for machine-learning (such as MNIST [74], Iris flowers [75], Palmer penguins [76], etc.) contain a vast set of feature vectors. Since a feature dimension is mapped to a qubit, these datasets need to be reduced and normalized in order to function within the constraints of NISQ-era quantum computers. Havileck et al. propose a mapping circuit that is not too deep and works well in their experiments:

$$V_\Phi(\boldsymbol{x}) = U_{\Phi(\boldsymbol{x})} H^{\otimes n} U_{\Phi(\boldsymbol{x})} H^{\otimes n} \tag{3.24}$$

for a circuit of n qubits and

$$U_{\Phi(\boldsymbol{x})} = \exp\left(i \sum_{S \subseteq [n]} \phi_S(\boldsymbol{x}) \prod_{i \in S} Z_i \right) \tag{3.25}$$

where coefficients $\phi_S(\boldsymbol{x}) \in \mathbb{R}$ encode the data $\boldsymbol{x}$. The feature state is then obtained as $|\Phi(\boldsymbol{x})\rangle = V_\Phi(\boldsymbol{x}) |0\rangle^n$. Now, the second unitary $W(\boldsymbol{\theta})$ is applied to the feature state, which is optimized to fit the training set T. Havileck et al. use the linear efficient SU2 ansatz [65], but others can be implemented as well. After applying $W(\boldsymbol{\theta})$, the state can be measured to determine the classification label $y \in \{+1, -1\}$. The probability of outcome y is

$$p_y(\boldsymbol{x}) = \langle\Phi(\boldsymbol{x})| W^\dagger(\boldsymbol{\theta}) M_y W(\boldsymbol{\theta}) |\Phi(\boldsymbol{x})\rangle \tag{3.26}$$

for the measurement operator $M_y = \frac{1}{2}(\mathbb{1} + y\boldsymbol{f})$ with the quantum parity function $\boldsymbol{f} = \sum_{z \in \{0,1\}^n} f(x) |z\rangle \langle z|$ using $f : \{0,1\}^n \rightarrow \{+1, -1\}$. The label $\tilde{m}(\boldsymbol{x}) = y$ is assigned when the empirical distribution $\hat{p}_y(\boldsymbol{x})$, obtained after repeated measurement, adheres to

$$\hat{p}_y(\boldsymbol{x}) > \hat{p}_{-y}(\boldsymbol{x}) - yb \tag{3.27}$$

where $b \in [-1, 1]$ is a bias operator that is optimized by the classical optimizer along with the optimization parameters $\boldsymbol{\theta}$. The cost function that the optimizer minimizes is defined by the empirical risk $R_{\text{emp}}(\boldsymbol{\theta})$, given by the probability of assigning an incorrect label to a sample of the training set T, defined as

$$R_{\text{emp}}(\boldsymbol{\theta}) = \frac{1}{|T|} \sum_{\boldsymbol{x} \in T} Pr(\tilde{m}(\boldsymbol{x}) \neq m(\boldsymbol{x})) \tag{3.28}$$

where the probability of assigning the wrong label is given by the binomial cumulative density function of the empirical distribution $\hat{p}_y(\boldsymbol{x})$ for a large number of shots ($\gg 1$).

$$Pr(\tilde{m}(\boldsymbol{x}) \neq m(\boldsymbol{x})) \approx \text{sig}\left(\frac{\sqrt{\#\text{ shots}}(\frac{1}{2} - (\hat{p}_y(\boldsymbol{x}) - \frac{yb}{2}))}{\sqrt{2(1 - \hat{p}_y(\boldsymbol{x}))\hat{p}_y(\boldsymbol{x})}} \right) \tag{3.29}$$

When the VQC has been trained on data set T by finding the optimal parameters $\boldsymbol{\theta}$ that minimize the cost function, the VQC can be used to label the unknown data set S. The achieved cost on the test set will then indicate how well the algorithm has been trained.

3.2.6. Selected algorithms for QPack

In this section, only a few of the many VQAs that exist were highlighted. It should be noted that there is a wide range of VQAs that exist or are still being developed, as they can be verified on NISQ-era quantum computers. Examples are quantum neural networks [77], the variational quantum thermalizer [78] or variational quantum factoring [79], to name a few. This promises a wide variety of NISQ-era applicable quantum circuits that can potentially be selected for benchmarking purposes.

The QAOA was already part of the original QPack benchmark [21, 20] and, as such, it is also selected to be implemented in this $|\text{Lib}\rangle$ version. In addition to QAOA, the VQE algorithm is chosen to be used for benchmarking, as it can be implemented with small circuits of a few qubits. This fills the gap in the evaluation of functionality of a quantum computer with very small qubit numbers (1, 2, or 3 qubits), where the current QAOA implementations usually start at 4 or 5 qubits. Other VQA-based applications have the potential to be included in QPack as well, but generally need a larger number of qubits to operate. As quantum computers improve and scale up, it could be interesting to implement algorithms such as VQLS or VQC.

3.3. Problem sets

This section briefly shows what problems are possible to implement as a VQE or QAOA application. Selecting the right problems to solve is important, as together with the quantum algorithm, it forms a specific quantum circuit used as the quantum subroutine in the VQA.

3.3.1. VQE problem sets

The VQE can in essence be used to solve any problem, as long as it can be encoded into a Hamiltonian H and the solution is the ground state of H. Some problem-specific parameterized quantum circuits are proposed in [80], covering the *traveling salesperson* [81] and the *vertex cover* [82] problems.

The main challenge in VQE is to find an appropriate ansatz to approximate the ground state. VQE ansatzes that approximate a wave function very closely often use many optimization parameters or gate operations. As the number of optimization parameters increases, a classical optimizer can have trouble finding the global optimum, needing many optimization iterations and risking getting stuck at local optima. A large number of gate operations can be undesirable in the NISQ-era, as gate-induced noise can render the ansatz useless. Finding a good balance between the state approximation and ansatz resources is thus important to run VQE efficiently.

3.3.2. QAOA problem sets

In the original QAOA paper [24], solving the maximum cut (MaxCut) problem [83] was introduced as an application for QAOA. In the expansion paper of this algorithm [66], more families of problems were introduced. They make a distinction of five different types of QAOA problems, based on their mixer families. These are *X mixers*, *controlled-X mixers*, *XY mixers*, *controlled-XY mixers*, and *permutation mixers*.

X mixers

This mixer family originates from the QAOA proposal by Fahri et al. [24] and is simply an X-rotation on each qubit in the QAOA circuit, given by mixer Hamiltonian

$$B_X = \sum_j \sigma_j^x \tag{3.30}$$

Examples of QAOA problems that use this mixer are the *maximum cut* [83], *max-SAT* [84], *set splitting* [85] and *E3Lin2* [86] problems.

Controlled-X mixers

Similar to the X mixer family, this mixer is also implemented with a single X-rotation on each qubit, but is now dependent on the state of other qubits. The mixing rule here is that an element i can be swapped in or out some set V if some predicate $\chi(x_{\text{nbhd(i)}})$ is satisfied by the partial state of neighboring qubits. Here, χ is problem dependent. For each qubit, a partial mixer Hamiltonian is defined as:

$$B_{\text{CX},i} = H_{\chi(x_{\text{nbhd(i)}})} \otimes X_i \tag{3.31}$$

with the complete controlled-X mixer $B_{\text{CX}} = \sum_i B_{\text{CX},i}$. The complete mixer can be applied simultaneously or in a partitioned manner. Simultaneous mixers are often hard to compile, so using partitioned mixers can be a simpler option to use. However, partitioned mixers do have the downside that partial mixers do not always commute. Problems in this mixer family are the *maximal independent set* [87], *maximum clique* [88], *minimum vertex cover* [82], *maximum set packing* [89] and *minimum set cover* [90] problems.

XY mixers
These types of mixer can be used for problems that have been encoded onto a bitstring using the one-hot encoding scheme, i.e., a bitstring where a single bit is 1 and the rest is 0. The XY mixer that acts on the qubit state preserves the Hamming weight of the computational basis states. In other words, it shifts the 1-bit to another place in the bitstring. The mixer Hamiltonian for this operation is

$$B_{\text{XY}} = \sum_{a=0}^{d-1} (\sigma_a^x \sigma_{a+1}^x + \sigma_a^y \sigma_{a+1}^y) \tag{3.32}$$

where d is the number of qubits needed for the one-hot encoding scheme. Problems from this mixer family are *max-κ-colorable subgraph* [91], *maximum/minimum bisection* [92], and *maximum vertex κ-cover* [93].

Controlled-XY mixers
This mixer family is very similar to the controlled-X mixer family, but using the aforementioned XY mixers instead of X mixers. Using this knowledge, we can see that the partial controlled-XY mixer Hamiltonian can be described as

$$B_{\text{CXY},i} = H_{\chi(x_{\text{nbhd(i)}})} \otimes B_{\text{XY},i} \tag{3.33}$$

with the complete controlled-XY mixer $B_{\text{CXY}} = \sum_i B_{\text{CXY},i}$. Problems include the *maximum κ-colorable induced subgraph* [94], *minimum graph coloring* [95], and *minimum clique cover* [89] problems.

Permutation mixers
This family of mixers does not have a predefined gate operation to describe the mixer operation, unlike the other mixer families. Rather, it entails all types of mixer that deal with a configuration space that is a set of orderings, permutations, or schedule of a particular set of items. Problems that are present in this family are the *traveling salesperson problem* [81] and *single machine scheduling* [96] problems.

3.3.3. Selected problems for QPack

Unlike the QAOA problem mappings, selecting some problems for VQE is arbitrary. As any ground state of a Hamiltonian can be found by selecting the proper ansatz. As mentioned in Subsection 3.2.6, the VQE problem will be used to create very small circuit sizes. A very shallow and simple ansatz is the variable X-rotation on each qubit, such that it can explore the complete computational basis. A useful Hamiltonian for this is a diagonal matrix containing random variables. This random diagonal Hamiltonian (RH) can then be evaluated using this shallow ansatz to find the lowest eigenvalue. Checking the solution is then simple, as we can easily spot the lowest diagonal value, which corresponds to the lowest eigenvalue since all matrix columns are orthogonal. Another ansatz for VQE is the hardware-efficient SU2 ansatz, as it does not use too much resources compared to other commonly used ansatzes. This ansatz can be used for more complex Hamiltonians. In this case, we will model an Ising chain (IC) [97], but other Hamiltonians are also viable.

As also mentioned in Subsection 3.2.6, previous work on QPack already included three QAOA problems: The maximum cut problem (MCP), the dominating set problem (DSP), and the traveling salesperson problem (TSP). As such, they are also included in the $|\text{Lib}\rangle$ version of QPack. These three problems are part of the X mixer family (MCP and DSP) and the permutation mixer family using XY mixers (TSP). In order to have a varied set of QAOA applications, it would be beneficial to choose an additional problem from either the controlled-X or controlled-XY mixer family. Here, the controlled-X family is preferred, as it will use a less-complicated gate structure than the controlled-XY family, resulting in shallower circuits better suited for NISQ-era quantum computers. From this controlled-X family, the maximal independent set (MIS) problem was chosen to be implemented because it was better documented than the other problems of the mixer family.

3.4. Classical optimizers

The success of a VQA is highly dependent on the classical optimization method used [61]. It is important that the quantum circuit encodes a good approximation to the solution and that the optimizer can find the optimal parameters within an acceptable time frame and with sufficient accuracy [98]. Optimization is not a quantum-specific technique and is widely used in the classical computation field. Optimization methods can be divided into two categories, gradient descent and gradient-free methods. The gradient descent method is generally not used for VQAs, but a brief overview will be given to motivate the reasons for this. Some gradient-free methods will then be discussed from a comparison presented by Mesman et al. [21], which is used to choose an optimizer for the QPack benchmark.

3.4.1. Gradient descent method

Optimization using gradient descent uses, the gradient of the cost function to find the minimum value. In a nutshell, the gradient descent method works by calculating the cost for a set of input optimization parameters $\boldsymbol{\theta}$, calculating the gradient, and taking a step in the direction of the gradient. Repeat this process until the minimum cost value has been found, i.e., the point at which the gradient is approximately zero. Here, selecting an appropriate step value (learning rate) is a trade-off between the speed at which the optimization algorithm converges and whether the algorithm is able to converge at all. With this in mind, we can see that gradient descent methods can only work well for smooth cost functions, as noisy or discontinuous functions could mess with the gradient estimation. Gradient descent methods can also get stuck in local minima, as the gradient at these points is, of course, also zero. Cost functions encoded in quantum circuits of VQAs often exhibit this non-smooth behavior or have many local minima, making gradient descent methods not very suited to optimize quantum subroutines [61].

3.4.2. Gradient-free methods

Optimization methods that are better suited to work with the difficulties VQA optimization poses do not rely on the gradient of the cost function. In earlier work of QPack [21], a comparison of such classical optimization algorithms was presented, which will be used as a guideline for this work. Each optimization algorithm will be briefly discussed to give an idea of its approach to finding minimum values. Compared optimizers fall in either a local or global optimization category. Local optimizers are used to find the local minimum of a small region close to the evaluated point, but may not be able to find the global optimum. Global optimizers, as the name implies, look for the global optimum of the cost function, but can be coarse in the estimation of the optimum [99].

Dual Annealing (DA) [100]

Dual Annealing is a stochastic global optimization algorithm designed for cost functions that have a nonlinear response surface. It is based on the Simulated Annealing (SA) algorithm that uses a type of stochastic hill climbing, i.e., candidate solutions are modified in a random manner and replace the current candidate solutions probabilistically. The probability of replacement is high at the beginning of the search and decreases with each iteration, controlled by the so-called 'temperature' parameter. DA is then a hybrid method between classic simulated annealing (CSA) [101] and fast simulated annealing (FSA) [102], combined with a strategy to apply a local search for potential solution locations. This last part is especially desirable, as global optimizers are often good at finding the area for the optimal solution, and the local optimizer is then able to find the exact location of the optimum [103, 104].

Nelder-Mead simplex (NM) [105]

This method uses the simplex shape in n-dimensional space. The simplex is the shape that consists of $n+1$ points. For example, in a 2D space, the simplex is a triangle, and in a 3D space, it is a tetrahedron. All points $x_1, x_2, ..., x_{n+1}$ are evaluated by the cost function $f(x)$. The Nelder-Mead method (NM) then orders these points from highest (worst) to lowest (best) cost. Then, the centroid of all but the worst points is computed over which the simplex can be transformed, using reflection, expansion, contraction, or redefinition of the simplex shape. This process is repeated until some convergence or iteration limit is reached. NM is a very fast and simple method, but tends to get stuck on local minima [106].

Broyden-Fletcher-Goldfarb-Shanno (BFGS) [107]

This optimization method is not gradient-free, but it was included in the optimizer comparison as it can be useful for certain quantum optimization cases. BFGS is a second-order optimization algorithm that

uses the quasi-inverse of the Hessian matrix to determine in which way to move to find local minima. Without going into too much detail, the BFGS algorithm distinguishes itself by updating the inverse Hessian in a unique way, rather than every optimization iteration in normal second-order Newton algorithms [108].

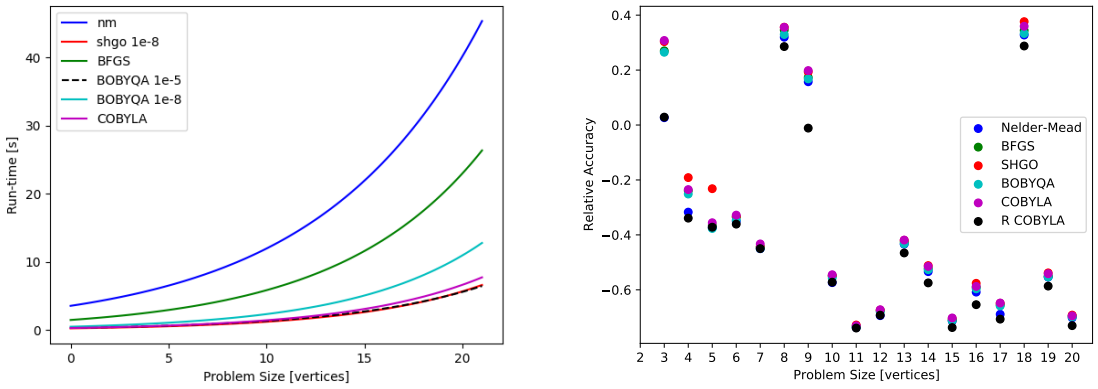
Constraint optimization by linear approximation (COBYLA) [109]
As the name implies, the COBYLA optimization method relies on linear approximation of the objective function using a simplex of $n + 1$ points for n dimensions as a constraint [110]. The value of the cost function is computed for each vertex of the simplex. With this information, an approximate linear representation of the cost function is generated. Using this approximation, the optimization problem is solved over a so-called 'trust'-region. The linear approximation and trust region are updated every optimization iteration. The trust region decreases as the algorithm converges to a solution. The COBYLA method terminates when some tolerance threshold is reached.

Bound optimization by quadratic approximation (BOBYQA) [111]
This iterative algorithm is used to find the minimum value of a function $f(\boldsymbol{x})$, subject to bounds $\boldsymbol{a} \leq \boldsymbol{x} \leq \boldsymbol{b}$. Similarly to COBYLA, BOBYQA tries to approximate the cost function, but this time by quadratic approximation. The algorithm uses m interpolation points to make this approximation, which are adjusted per optimization iteration. There is no predefined value for m, but $m = 2n + 1$ is commonly used for n dimensions [111]. Since BOBYQA constructs a quadratic approximation of the cost function, it may not be suitable for cost functions that are not twice differentiable [110].

Simplicial homology global optimization (SHGO) [112]
Unlike the name may suggest, the SHGO is a complex global optimization algorithm based on the application of simplicial integral homology and combinatorial topology. Going further into detail of how this algorithm functions is beyond the scope of this thesis. SHGO is able to find all local minima of a cost function and works especially well for applications such as energy landscape exploration.

3.4.3. QPack choice of optimizer

As shown in this section, there are many optimizers to choose from. In previous work on the QPack benchmark [20], a comparison between the previously mentioned optimizers has been made for their runtime and accuracy, as depicted in Figure 3.4.



(a) Runtime measurements fitted to exponential functions. Numbers in legend indicate convergence tolerance. If not stated, default values from their implementations in the Python libraries SciPy [113] or NLOpt [110] is used.

(b) Relative accuracy of optimizers compared to the Dual-Annealing (DA) optimization method [100]. The smaller the value, the closer a optimizer performs to the DA method. Negative values indicate that an optimization method outperforms DA.

Figure 3.4: Comparison of different gradient-free optimization methods presented by Mesman et al. [21].

Measurements were obtained by running the QAOA MaxCut problem for 3 to 25 nodes. Here, it was found that gradient-free optimizers such as COBYLA, BOBYQA, or SHGO have good runtime performance as well as accuracy. Especially COBYLA with randomized initialization parameters allows for a fast and accurate optimization routine. Therefore, in this work, the COBYLA optimization algorithm will be used, implemented using the open-source library for nonlinear optimization NLopt [110].

4

Benchmark design & criteria

This section covers the structure of the QPack benchmark. It contains an elaboration on how the benchmark functions, what data is collected, and the design criteria for a quantum score. The code that implements the VQE functionality in $|\text{Lib}\rangle$ can be found in the GitLab repository `https://gitlab.com/libket/qpack/-/tree/stable/Qpack`, specifically the files `main.cpp`, `BenchmarkLoops.hpp`, and `Optimizers.hpp`.

4.1. Benchmark outline

The QPack benchmark determines the performance of quantum backends by collecting quantum execution data when executing various VQAs. These VQAs are one of the first viable real-life applications in the NISQ-era of quantum computing [61] and have therefore been chosen as benchmark applications for QPack. Figure 4.1 shows an overview of the QPack benchmarking process. From here on, the term quantum computer or quantum backend will be used interchangeably. A quantum computer or backend covers the whole system of the actual quantum hardware, control system, qubit mapping, and gate scheduling, or the simulation thereof.

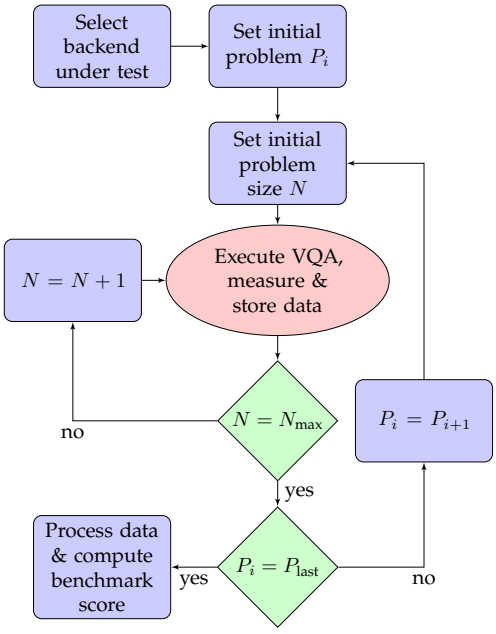


Figure 4.1: Benchmark process overview

The benchmark starts by selecting a quantum computer to be evaluated. Multiple quantum computers can be selected for sequential performance evaluation, but this overview will focus only on the evaluation of a single quantum backend. After selecting a backend under test, a problem P from the problem set can be selected. In the current implementation of QPack, $P \in \{\text{MCP, DSP, MIS, TSP, RH, IC}\}$ as

will be described in Chapters 5 and 6. This set contains four QAOA problems (MaxCut, dominating set, maximal independent set, and traveling salesperson problems) and two VQE problems (random diagonal Hamiltonian, and Ising chain model), respectively. There is no preferred order, so problems can be freely selected. However, to obtain the best and most varied comparison, QPack attempts to complete all problems for as large a problem size as possible.

For the selected problem, the initial problem size N is set. It should be noted that N is a symbolic value and does not represent the number of qubits. The number of qubits used for a given problem size can differ per benchmark application, hence a selection of problem sizes should be carefully considered. For example, the MaxCut problem scales qubits linearly with the problem size (i.e., problem size 5 needs 5 qubits), while the traveling salesperson problem scales quadratic (i.e., problem size 4 requires 16 qubits). For this reason, each problem has its own range of problem sizes.

When the problem P and problem size N are set, the VQA can be run on the backend under test. This entails the optimization of the parameterized quantum circuit by a classical optimizer (COBYLA [109] is used in this version of QPack). Once the optimizer has found the optimal minimal value, the VQA execution is finished and the measurement results are saved in JSON format for later use. This step is repeated multiple times, such that measurements are taken in multitude and outliers can be filtered. Specific details on what measurement data is collected is elaborated in Section 4.2.

After VQA execution is complete, QPack checks if the maximum problem size has been evaluated. If not, the problem size is incremented, and the VQA is executed for this new problem size. When all the problem sizes for a selected problem have finished, QPack checks if all problems have been evaluated. If not, the next problem is set up, and the VQA evaluation is repeated for this new problem. When all problems have been solved, QPack evaluates all measured data and calculates the quantum computer's benchmark score.

Almost all the steps as depicted in Figure 4.1 are implemented in C++ using the |Lib⟩ library. The last step, however, processing the benchmark data and computing the benchmark scores, is implemented using Python, as it is more convenient to handle and visualize data sets. The complete C++ code can be found in Appendix E and the Python code in Appendix F.

4.2. Quantum execution data

During the execution of a VQA, QPack takes multiple measurements per execution cycle, resulting in a set of quantum execution data. An example of such data can be found in the JSON snippet below, which shows the data for the first execution cycle of for P = MCP and $N = 5$ on the IBMQ QASM simulator. Descriptions of all quantum execution data can be found in Table 4.1. This raw execution data forms the basis of the QPack scores to be defined in Chapter 7.


```
{
  "Circuit execution durations [ms]": [10.979981, 10.870313, 10.715547, ...],
  "Depth": 95.0,
  "Expectation Value": -5.9697265625,
  "Optimal Expectation Value": -6.0,
  "Optimizer durations [ms]": [6757.02, 8492.6, 8000.625, ...],
  "Optimizer iterations": 100,
  "Optimizer params": [0.61316016681300, 0.88731605172780, 0.070652242307702, ...],
  "P": 3,
  "Problem": "MCP",
  "QJob durations [ms]": [6590.003728866577, 8407.450675964355, 7919.116973876953, ...],
  "QPU": "4010001",
  "Qubits": 5,
  "Queue durations [ms]": [677.0, 1656.0, 1107.0, ...],
  "Shots": 4096,
  "Size": 5,
  "Total Algorithm duration [s]": 1294.856
}
```

Table 4.1: Datasets collected by during execution of a VQA

Dataset	Description
Circuit execution durations	Duration of the circuit execution time T^{QE} of each quantum job of an optimizer iteration. This is the time that the quantum computer executes the quantum circuit for a given amount of shots and does not include circuit transpiling, optimization or other latencies.
Depth	The depth of the untranspiled quantum circuit as implemented in QPack. Although the circuit needs to be transpiled to fit a certain qubit layout, QPack does not take this transpiled depth into consideration when evaluating performance, as this is a job for the quantum computer under test.
Expectation value	Expectation value of the VQA problem with the optimized parameters.
Optimal expectation value	Theoretical best expectation value that can be achieved for a given problem and problem size.
Expectation value baseline[1]	Expectation value of the QuEST [114] noiseless simulator. This data serves as a baseline for how well the classical optimizer works on a given problem set.
Optimizer durations [ms]	Time for the optimizer to finish one optimization iteration.
Optimizer iterations	Number of iterations the optimizer needed to find the minimal value.
Optimizer params	Optimized parameters of the VQA by the classical optimizer.
P	QAOA p iterations. Default: 3 for QAOA, 0 for VQE (not applicable).
Problem	Tag of the problem P for current benchmark run.
QJob durations [ms]	Durations of all quantum jobs (QJobs) T^{QJob}. A QJob is defined as the point where the untranspiled circuit is send to to the quantum computer to the point where the results of the quantum job are returned. This includes communication time, compilation, validation and circuit execution.
QPU	Unique tag for the quantum computer under test.
Qubits	Number of qubits used for this problem size (circuit width).
Queue durations [ms]	Queue times for remote quantum providers if applicable.
Shots	Number of shots for current benchmark run (default: 4096).
Size	Problem size N of for the current benchmark run.
Total algorithm duration [s]	Total duration to execute the current VQA algorithm. This includes both classical and quantum durations.

[1] Not in JSON snippet. This value is not collected during VQA execution, but is added to the table for completeness

When plotted, the raw quantum execution can already reveal differences in execution characteristics between quantum backends. For example, consider the plots in Figure 4.2, which depict the execution data for the MIS problem on three different local quantum simulators. A clear distinction can be made between different quantum computer qualities. For example, with a growing problem size, it can be seen that the QuEST simulator has the lowest average QJob and optimizer duration and has the lowest average error amongst the tested quantum computer simulators. For QPack, the error is defined as the relative error between the output of a quantum backend and the baseline value of an ideal simulator. In contrast to the QuEST simulator, the Qiskit Aer simulator has the highest average QJob runtime and the highest error peaks of all three simulators. The Cirq simulator has performed somewhere in between. These observations already give a feeling about which backend outperforms the other, but we cannot yet have a solid foundation to verify this. These observations now need to be processed, such that a quantitative comparison between quantum computers is possible.

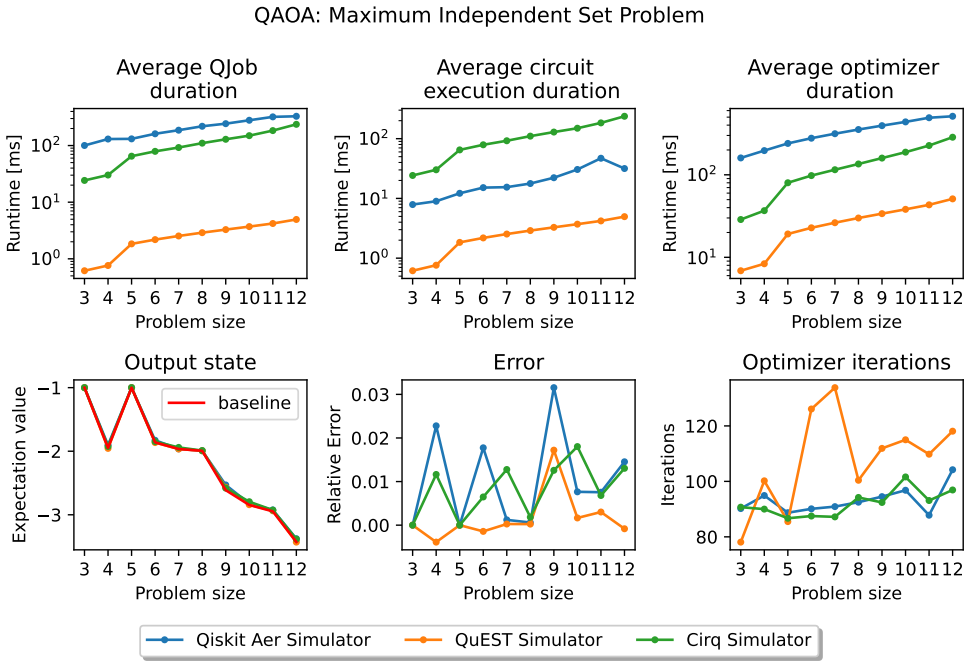


Figure 4.2: Quantum execution data from the simulation of the maximal independent set benchmark. (Top left): Average QJob duration per problem size. (Top center): Average circuit execution duration per problem size. (Top right): Average optimizer duration per problem size. (Bottom left): Expectation values of the output state per problem size, including baseline values. (Bottom center): Relative error between the quantum computer's expectation value and the baseline expectation value. (Bottom right): Average number of iterations that the classical optimizer needed to find the minimum value.

Notice that in the analysis of the example in Figure 4.2, the focus was mainly on the average duration of the QJob and the relative error of the simulators. This is because these are the main data points that are used to compute the QPack scores. However, it should be noted that other characteristics such as the average circuit execution duration and the number of optimizer iterations can also be relevant depending on which properties of a quantum computer are of interest.

4.3. Score criteria

Using the measurement data discussed in the previous section, performance scores can be distilled. For ease of comparison, a performance score should be a single "figure of merit", which allows for a clear distinction between different quantum computers, i.e., a quantum computer with a higher score is better than a quantum computer with a lower score. Nevertheless, the scores should not become too abstract as to not properly reflect the characteristics of a quantum computer. For instance, a quantum computer may be very fast but inaccurate, while another quantum computer may have high accuracy but takes a long time to execute. Which one is then considered better? Other factors such as scalability, maximum number of qubits, or serviceability also play a role in defining quantum computer performance.

With this in mind, a number of criteria for benchmark scores are defined in the design of QPack:

1. Benchmark score reflects application-level performance of quantum computers (simulators and hardware implementations).
2. Benchmark score is a composite of measurement data of multiple quantum applications.
3. Benchmark score is a single number (but may be split up into subscores).
4. Benchmark score is proportional to performance, i.e., a higher score means higher performance.
5. Benchmark score are scalable, i.e., score has no upper limit.
6. Benchmark score does not become too abstract from the data it is based on.
7. Subscores should be balanced, such that one sub-score does not become dominant in the overall score.

5

|Lib⟩ VQE implementation

This chapter describes the general implementation of the VQE algorithm, followed by two applications of the VQE algorithm in |Lib⟩. Finally, a state-readout approach is presented on how, for these two specific applications, a benchmarking runtime optimization can be realized. The code that implements the VQE functionality in |Lib⟩ can be found in the GitLab repository `https://gitlab.com/libket/qpack/-/tree/stable/Qpack/include`, specifically the files `VQA.hpp`, `VQE.hpp`, `VQE_Ansatzes.hpp`, `VQE_Hamiltonians.hpp`, and `VQE_loops.hpp`. All quantum gates that are used are defined in the |Lib⟩ documentation [23].

5.1. General VQE implementation

Like any VQA, the variational quantum eigensolver consists of a classical and quantum subroutine. An overview of the VQE architecture [25] is presented in Figure 5.1. The first algorithm consists of quantum circuits that compute the expectation value $\langle H \rangle$ of Hamiltonian $H = \sum_{i=0}^{K-1} H_i$. This means that K quantum circuits need to be evaluated to determine the total expectation value $\langle H \rangle$, which is simply the addition of the partial expectation values by a classical computer. The second algorithm attempts to minimize this expectation value by varying the quantum state $|\psi(\boldsymbol{\theta})\rangle$, also called the ansatz.

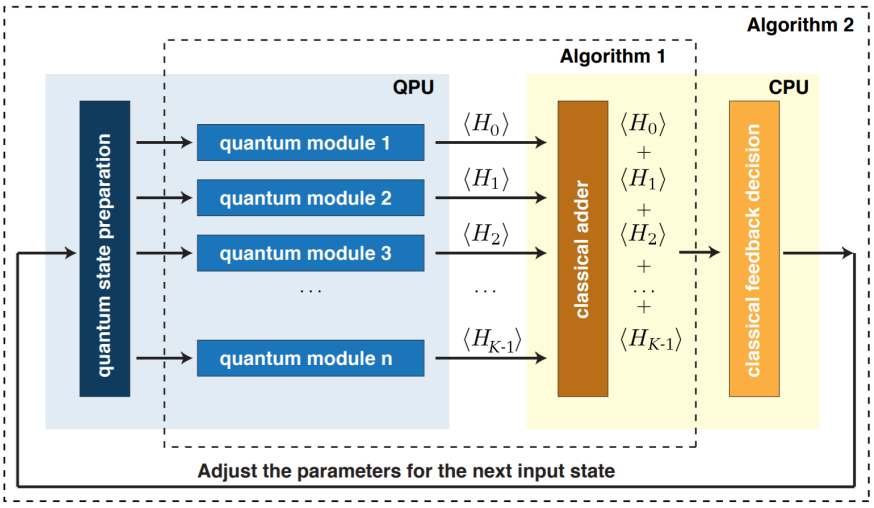


Figure 5.1: Architecture of the variational quantum eigensolver, modified from Peruzzo et al. [25]

A circuit of L qubits of a quantum module typically consists of two main parts. The ansatz unitary and a state basis change before measurement, as shown in Figure 5.2. An optional third step can be applied as well, which initializes the qubit state before applying the ansatz unitary. This typically consists of single-qubit Pauli rotations. For example, X-gates are applied to create a Hartree-Fock state [115] when VQE is used in quantum chemistry.

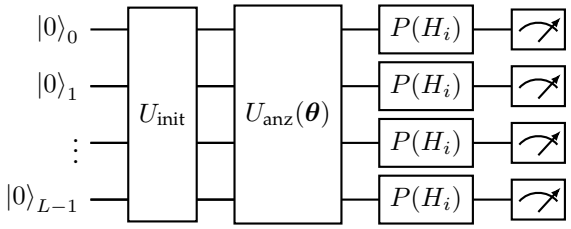


Figure 5.2: Typical VQE circuit. $P(H_i)$ denotes a Pauli operation $P \in \{I, X, Y, Z\}$ based on the i-th partial Hamiltonian basis set

5.1.1. Ansatz

The $U_{\text{anz}}(\boldsymbol{\theta})$ parameterized unitary consists of single- and multi-qubit gates to alter the wave function dependent on parameters $\boldsymbol{\theta}$. This unitary can vary from application to application. As such, variation circuits with a multitude of parameters can be implemented here to change the quantum wave function ansatz. This problem-specific unitary is defined later in this chapter. Before using the parameterized $U_{\text{anz}}(\boldsymbol{\theta})$ unitary, an initialization unitary U_{init} may be applied to set the inital state of the qubits.

5.1.2. Energy measurements

The measurement step consists of single-qubit rotations to measure the qubit state in the desired Pauli base before measurement, using a unitary operation $U_{\text{basis}}(H_i)$. This change in measurement basis depends on the partial Hamiltonian H_i that needs to be measured.

$$U_{\text{basis}}(H_i) = \bigotimes_{j=0}^{L-1} P_j(H_i) \tag{5.1}$$

where $P_j(H_i)$ denotes a Pauli operation $P \in \{I, X, Y, Z\}$ based on the i-th partial Hamiltonian basis set on qubit j. To measure qubits in bases other than the Pauli-Z basis, some single-qubit rotations [116] are needed to get the right measurements, if a quantum computer is not able to measure other bases natively. The change of basis required for each measurement can be found in Figure 5.3. A measurement in the I-basis will always yield 1, hence no measurement needs to be taken.

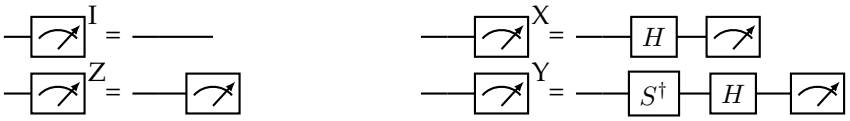


Figure 5.3: Pauli measurement basis circuits

5.1.3. Expectation value from state distribution

Since the measurement result is returned as a set of 2^L states, with count C_l for state l. This needs to be converted to get the expectation value of the state for every partial Hamiltonian H_i. This is simply the sum of the probability of each state multiplied by its eigenvalue λ_l. The probability p_l is simply the counts of that state divided by the number of shots. So for every partial Hamiltonian, we find the expectation value:

$$E_i = \sum_{l=0}^{2^L-1} p_l \lambda_l = \frac{1}{\#\text{shots}} \sum_{l=0}^{2^L-1} C_l \lambda_l \tag{5.2}$$

where the eigenvalue of the state is an eigenvalue of the $Z^{\otimes L}$ space. This coincidentally corresponds to the parity of the bitstring of the state and can thus be defined as:

$$\lambda_l = \begin{cases} 1 & \text{if state parity is even} \\ -1 & \text{if state parity is odd} \end{cases} \tag{5.3}$$

5.1.4. Hamiltonian energy from expectation value

Now that the expectation value can be extracted from the measured state distribution, the energy of the Hamiltonian can be estimated. Recall that

$$H = \sum_{i=0}^{K-1} H_i = \sum_{i=0}^{K-1} c_i \prod_{j=0}^{L-1} \sigma_{i,j}^{\alpha} \tag{5.4}$$

where the partial Hamiltonian H_i consists of a coefficient c_i and a set of Pauli operators $\sigma_{i,j}^{\alpha}$ for each particle j of the i-th term described by the Hamiltonian, with $\alpha \in \{x, y, z\}$. The Pauli-I operator is left out for readability. The complete energy of the Hamiltonian is thus simply the sum of all the expectation values of each partial Hamiltonian multiplied by its coefficients. The energy of the complete Hamiltonian thus becomes

$$\langle H \rangle = \sum_{i=0}^{K-1} \langle H_i \rangle = \sum_{i=0}^{K-1} c_i \langle \psi(\boldsymbol{\theta}) | \prod_{j=0}^{L-1} \sigma_{i,j}^{\alpha} | \psi(\boldsymbol{\theta}) \rangle = \sum_{i=0}^{K-1} c_i E_i \tag{5.5}$$

Since the ansatz is variational, a classical optimizer can now use this total energy measurement and use it to find the minimal eigenvalue of the Hamiltonian.

5.2. Random diagonal Hamiltonian (RH)

The first VQE problem added to QPack is a simple demonstration of the basic functionality of the VQE. The problem description, Hamiltonian, and ansatz will be highlighted, after which some results on the Qiskit Aer Simulator will be presented as a proof of implementation.

5.2.1. Problem description

This simple problem consists of a diagonal matrix with random values on the diagonal. The VQE aims to find the lowest eigenvalue of this matrix, which is trivial to find, since it is simply the lowest diagonal value, since all eigenvectors are orthogonal.

5.2.2. Hamiltonian

The Hamiltonian for this problem is a set of Pauli Z-rotations, where the number of partial Hamiltonians K is equal to the problem size N:

$$H_{\text{random}} = \sum_{i=0}^{N-1} r_i \sigma_i^z = \begin{pmatrix} R_0 & 0 & 0 & \dots & 0 \\ 0 & R_1 & 0 & \dots & 0 \\ 0 & 0 & R_2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & R_{2^N-1} \end{pmatrix} \tag{5.6}$$

where $r_i \in [-1, 1]$ is a random value, Z_i is the Pauli Z operator on qubit i and R_i is a random value based on a sum of r_i.

5.2.3. Ansatz

The ansatz used to find the lowest eigenvalue is simply an X-rotation on each qubit for different optimization parameters, shown in Figure 5.4. Since only values on the diagonal of the Hamiltonian are nonzero, this ansatz covers the complete solution space.

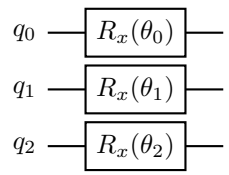


Figure 5.4: Random diagonal Hamiltonian ansatz for 3 qubits

As the problem size increases, the number of qubits and optimization parameters for this ansatz grow proportionally. This means that for a problem size N, we need N qubits for the ansatz and there are N optimization parameters.

5.2.4. Simulation results

The VQE algorithm was run on the random Hamiltonian problem, and the results are shown in Figure 5.5. A comparison is made between the exact solution of the RH problem and the approximation of the solution using the VQE algorithm on the Qiskit Aer simulator.

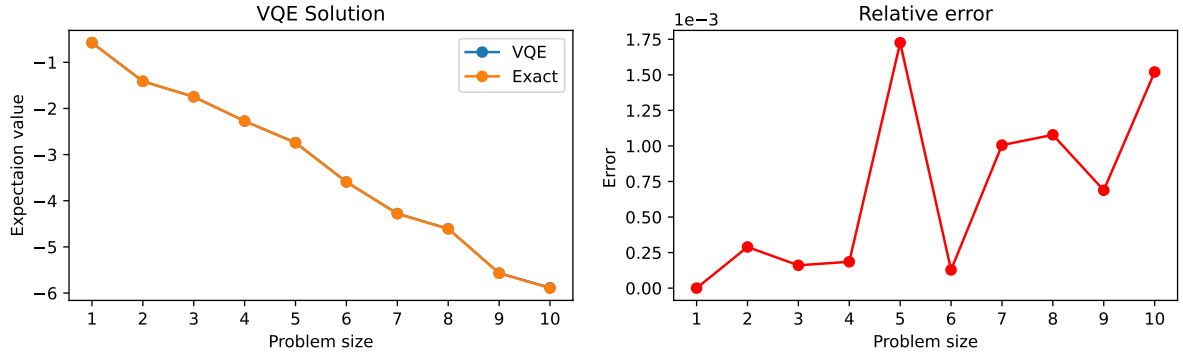


Figure 5.5: Results for the random diagonal Hamiltonian. Quantum circuits were run on the Qiskit Aer simulator with 4096 shots

It can be seen that the ansatz performs very well, since the line for the VQE estimation is hidden by the line of the exact solution. A more insightful graph on the difference between the exact solution and the VQE estimation is shown in the right figure, where the relative error is plotted, defined as $RE = \frac{E^{exact} - E^{VQE}}{E^{exact}}$. This also shows that the VQE estimation is very close to the exact solution.

5.3. Ising chain ground state (IC)

A VQE problem with a more interesting Hamiltonian to evaluate is the Transverse Field Ising Model (TFIM), which also includes XX-interaction along with a Z-component like in the random Hamiltonian VQE problem.

5.3.1. Problem description

This problem focuses on finding the ground state of a one-dimensional spin system. The model used is the 1D TFIM. It consists of spin-1/2 particles on a chain, where neighbors interact with each other via XX-couplings (Figure 5.6). A transverse magnetic field of strength h is applied in the Z-direction, which alters the spin of the particles in the chain. The goal of the VQE algorithm is to find the lowest energy state of this system [97]. For this problem, the problem size N is equal to the number of particles in the Ising chain.

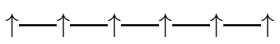


Figure 5.6: 1D Ising chain model

5.3.2. Hamiltonian

The Hamiltonian of this system is described by the XX-interaction between neighbors with coupling strength J and the transverse magnetic field of strength h. Since this problem is a chain, the first and last particle only interact with one neighbor. For this problem, the coupling strength is uniform. The Hamiltonian can thus be described as presented in Equation 5.7.

$$H_{\text{Ising}} = -J \sum_{i=0}^{N-2} \sigma_i^x \sigma_{i+1}^x - h \sum_{j=0}^{N-1} \sigma_j^z \tag{5.7}$$

An interesting phenomenon known as a quantum phase transition occurs when $J = h$ [117], which is the point where the transverse field strength is equal to the coupling strength of the spin particles. For this problem, the Hamiltonian will be constructed with this feature. The total number of partial Hamiltonians for a problem size N is $K = 2N - 1$.

5.3.3. Ansatz

Ansatzes for this problem can become very hardware demanding as the problem size grows. To negate this, the hardware efficient SU2 [65] will be used with full entanglement, see Figure 5.7. This circuit can be used to model an arbitrary wave function, but also needs significantly more optimization parameters, 4 for every qubit. This will put more load on the classical optimizer than the other VQE problems.

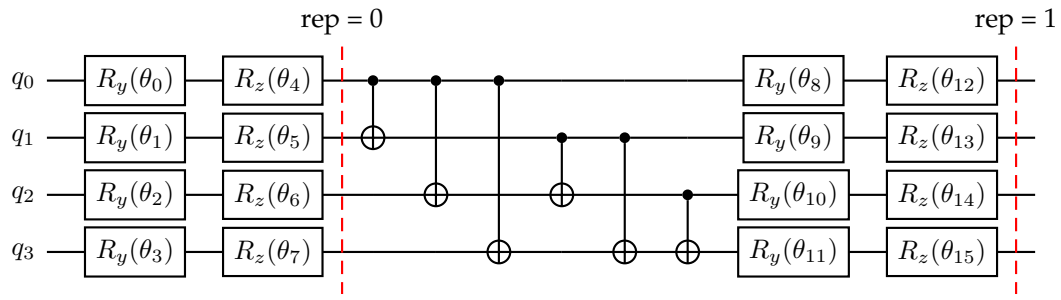


Figure 5.7: Efficient SU2 ansatz of with full entanglement for 4 qubits for one circuit repetition

Notice that for the IC problem, a better estimate of the ground state is possible by expanding the SU2 ansatz with more repetitions of itself, that is, another block of CNOT gates followed by the single-qubit Y- and Z-rotations. However, since the number of optimization parameters is already much larger than that of the RH problem, the ansatz is only constructed for a single repetition to reduce the optimization time.

5.3.4. Simulation results

The result for the Ising chain VQE problem can be seen in Figure 5.8. The SU2 ansatz does attempt to get the lowest eigenvalue, but does not perform as well as the simpler ansatz for the random Hamiltonian problem. Lower error scores could be achieved by a different ansatz (for example, the U3 ansatz, as shown in Section 3.3.1), increasing the number of circuit iterations in the SU2 ansatz, or by changing the classical optimizer. However, for the QPack benchmark, this does not form a problem, as the performance of this ansatz can be used as long as a baseline value is used as a reference point instead of the exact solution.

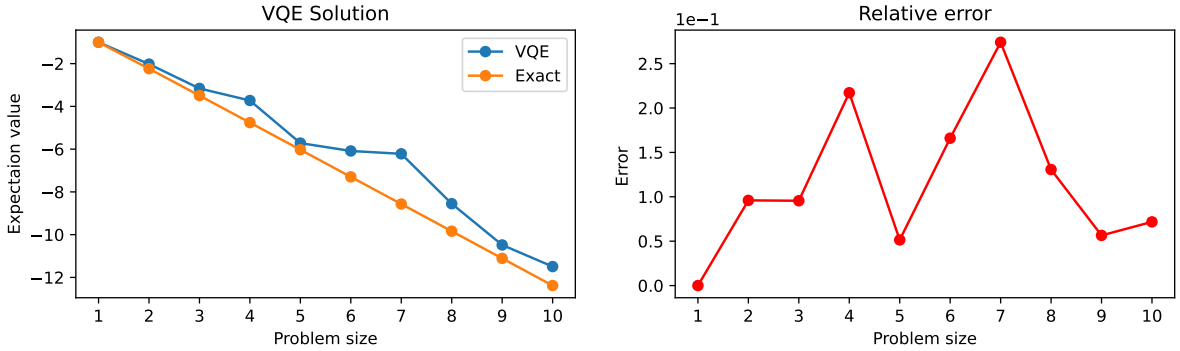


Figure 5.8: Results for the TFIM Hamiltonian. Quantum circuits were run on the Qiskit Aer simulator with 4096 shots

5.4. Circuit execution reduction

As the problem size grows, so does the number of partial Hamiltonians K as stated in Section 5.1. The number of optimization parameters already increases the total runtime of the VQE execution, as it takes

the optimizer longer to find the best set of parameters. For a given problem size N, the number of circuits K to evaluate for the RH problem is N, while for the IC problem $K = 2N - 1$. If we take a closer look at the Hamiltonians that are evaluated in the VQE problems, it can be seen that there are only two bases being measured, the X- and Z-basis, for different subsets as the circuits are measured. If the ansatz was measured completely in the X- or Z-basis, all information on these subsets would actually be encoded in the complete measured set. For the RH problem, for example, instead of evaluating 2 circuits for a problem size of 2, all the information from the Z-basis measurements can be done at once, see Figure 5.9.

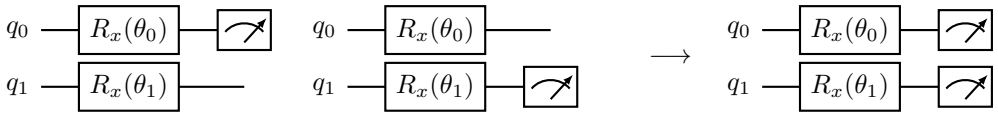


Figure 5.9: Z-basis circuit reduction for RH problem (problem size 2)

To extract the data, we need to reduce the complete histogram of the measured states, the subset of Pauli terms of each partial Hamiltonian. In the example in Figure 5.9, for the left circuit, a possible output state (for 1000 shots) could be

```
Histogram: [00: 450, 01: 550, 10: 0, 11: 0]
```

and for the right circuit:

```
Histogram: [00: 100, 01: 0, 10: 900, 11: 0]
```

However, when both qubits are measured together, the output histogram would look like this:

```
Histogram: [00: 50, 01: 50, 10: 400, 11: 500]
```

Looking at these state distributions, it becomes apparent that the measurement set of all qubits can be reduced to a subset of qubits, by simply summing each term of each subset together. For the above-mentioned example, to get the first state distribution, the states 00 and 10 are summed to get the 0-measurement and the states 10 and 11 are summed to get the 1-measurement. In the same way, the second state distribution can also be reconstructed from the combined state histogram. This way also works for subsets of multiple Pauli's, like the XX-term in the IC Hamiltonian.

Using this method, only one circuit needs to be evaluated for the RH problem and two circuits for the IC problem. This method only works as long as the partial term of the Hamiltonian contains the same Pauli operators, i.e., XX, YYY, Z, but not XYX, ZX, etc., as these terms are not encoded in the measurement of a complete basis. Fortunately, this does not form a problem in the current VQE implementation and can thus be implemented in QPack to significantly reduce the benchmarking runtime.

5.5. Resource comparison

As discussed previously in this chapter, the resources required to build the VQE ansatzes differ for the RH and IC problems. In Table 5.1, a comparison is made between the two VQE implementations for QPack, listing the number of qubits needed, single qubit gates, and multi-qubit gates. All high level gates are decomposed to standard single-qubit gates (rotation on a single qubit) or standard multi-qubit gates (controlled rotations, Toffoli). In addition, the number of optimization parameters is listed as well, to emphasize that VQE problems become more difficult to optimize as the problem size N increases.

	Qubits	Single-qubit gates	Multi-qubit gates	Parameters
RH	N	N	0	N
IC	N	$4N$	$\frac{N^2-N}{2}$	$4N$

Table 5.1: Required resources for each VQE problem

These results are visualized in Figure 5.10 to get an understanding of how these problems scale when N increases.

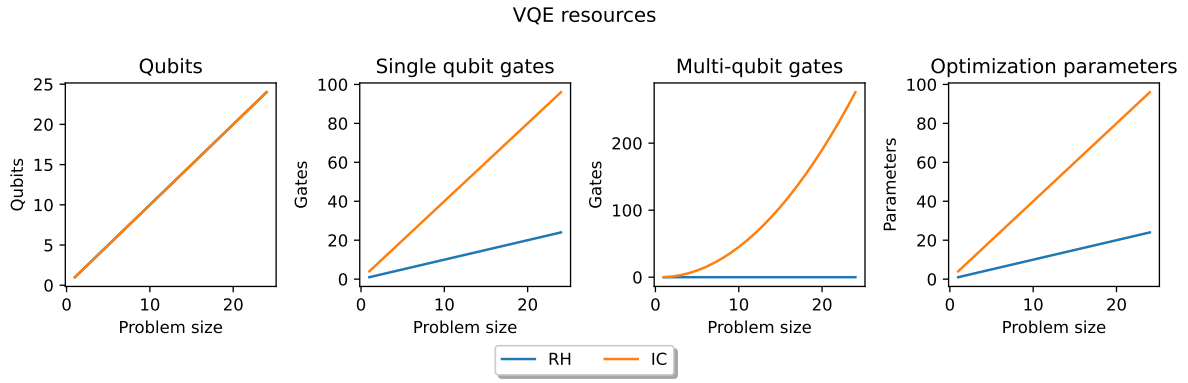


Figure 5.10: Scaling of VQE resources for each problem set

Because of the more complex ansatz, the Ising chain problem requires more resources than the random Hamiltonian problem. This should make the IC problem 'harder' to solve for a quantum computer and a classical optimizer, which makes it a good benchmarking application together with the RH problem and offers some variety in the QPack benchmarking application set. For both problems, the number of optimization parameters increases proportionally to the problem size. For NISQ-era quantum computers, this does not become a problem as typical circuit sizes will generally be small. However, when circuits become larger, the number of optimization parameters could increase benchmark runtime due to the more complex nature of the minimization problem. This could be counteracted by loosening optimization constraints, limiting optimizer iterations, choosing a different ansatz, or choosing a different classical optimizer altogether.

6

|Lib⟩ QAOA implementation

This chapter will focus on the implementation of four problems to solve using QAOA implemented in the |Lib⟩ library. All problems will be briefly described, after which a mapping to QAOA will be derived. A proof of implementation will be provided for each problem. All problems are graph-based NP-hard problems [118, 98]. For a recap on complexity theory, see Appendix A. The code that implements the QAOA functionality in |Lib⟩ can be found in the GitLab repository `https://gitlab.com/libket/qpack/-/stable/main/Qpack/include`, specifically the files `VQA.hpp`, `QAOA.hpp`, and `QAOA_loops.hpp`. All quantum gates that are used are defined in the |Lib⟩ documentation [23].

6.1. Graphs for QAOA problems

The network topology for a graph problem will be a 4-regular graph, i.e., a connected graph where all vertices have degree 4. To increase the problem size, the number of vertices in the regular graph can simply be increased. This will be useful later on, as the QPack benchmark will measure the performance of quantum systems by increasing the problem size. For graph sizes smaller than 5, a line, a triangular, and a square graph are used, as they cannot have a degree of 4. An illustration of such graphs is given in Figure 6.1. For all QAOA examples in this chapter, the 6-vertex regular graph will be used.

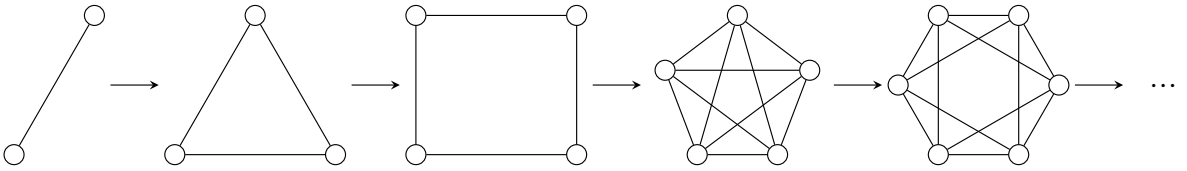


Figure 6.1: Procession of increasing graph sizes used for QAOA problems. From a graph size of 5, the graph is 4-regular

6.2. Maximum cut problem (MCP)

The maximum cut problem (MCP or MaxCut) is often used as a benchmark problem for QAOA. Although the problem is classified as NP-hard [118], the problem is simple and straightforward to explain. The MaxCut problem was also used as an example when Fahri et al. introduced QAOA [24] and was proven to work well for small graph sizes with low p iterations on NISQ-era quantum computers [119].

6.2.1. Problem description

The MaxCut problem is defined as follows. Given a graph $G = (V, E)$, with V vertices and E edges, find the subset S of V, such that the number of edges between S and $V \backslash S$ is maximized. In other words, how can one divide the vertices of the graph into two distinct sets such that if one would cut the edges between those sets, the amount of cut edges would be the largest possible. For the example graph, for instance, a possible solution (out of 18) is shown in Figure 6.2.

6.2.2. QAOA mapping

Encoding The set of vertices can be encoded in a single bitstring $x = x^0 x^1 ... x^{|V|-1}$ for V vertices, where $x^n \in [0, 1]$. If a vertex is in the first distinct set, its bit will be 0 and if it is in the other set, its bit

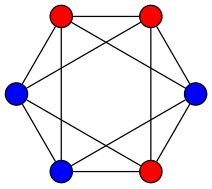


Figure 6.2: Solution of the MaxCut problem of a 6-vertex 4-regular graph. Out of 12 edges, there are 8 edges between the distinct subsets (red and blue vertices).

will be 1. The QAOA implementation will therefore require V qubits to encode the solution space of the MaxCut problem.

Initial State Since all possible permutations of bitstring x form valid potential solutions to the MaxCut problem, the initial state can be initialized to be an equal superposition of all states. Hence, the initial state $|s\rangle$ is:

$$|s\rangle = \prod_{i \in V} |+\rangle_i \tag{6.1}$$

Cost Hamiltonian The cost function will be the number of cuts a bitstring can make. So, for the set E of edges $\{u, v\}$, the cost Hamiltonian is:

$$C = \sum_{\{u,v\} \in E} \frac{1}{2}(1 - \sigma_u^z \sigma_v^z) \tag{6.2}$$

where σ_i^z denotes the Pauli-Z operator on qubit i (the identity operator is implied when no other Pauli operator is specified). Since the global phase term can be ignored, the cost Hamiltonian simplifies to:

$$C = \sum_{\{u,v\} \in E} \sigma_u^z \sigma_v^z \tag{6.3}$$

The cost unitary then becomes:

$$U_C(\gamma) = e^{-i\gamma C} = \prod_{\{u,v\} \in E} e^{-i\gamma \sigma_u^z \sigma_v^z} \tag{6.4}$$

This can be implemented with a ZZ-rotation by angle γ. This ZZ-gate is not a standard gate, but can be implemented using two CNOT gates and one $R_z(\gamma)$ gate. This is shown in Figure 6.3.

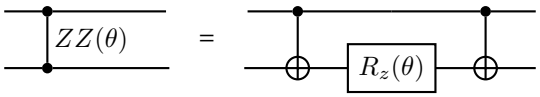


Figure 6.3: $R_{zz}(\theta)$ gate decomposition

Mixer Hamiltonian In order to mix up the solution space, simple Pauli-X gates can be applied to all qubits:

$$B = \sum_{i \in V} \sigma_i^x \tag{6.5}$$

Resulting in the unitary:

$$U_B(\beta) = e^{-i\beta B} = \prod_{i \in V} e^{-i\beta \sigma_i^x} \tag{6.6}$$

Which can simply be implemented with an $R_x(\beta)$ gate on each qubit.

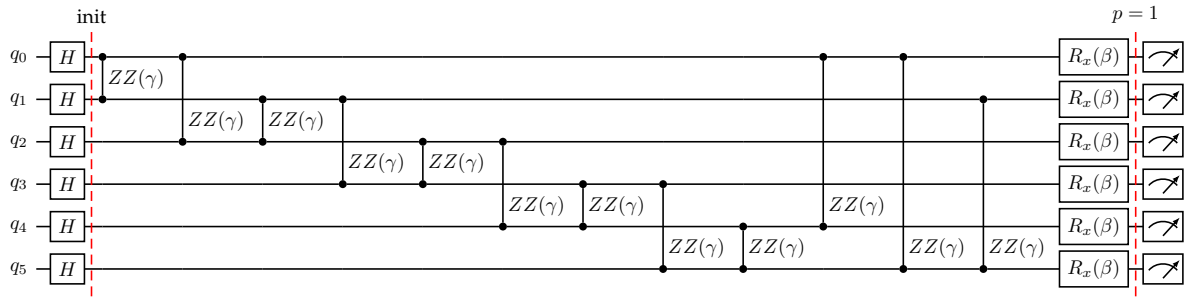


Figure 6.4: Circuit for the 6-vertex 4-regular graph example. The circuit shows the QAOA implementation for $p = 1$

Complete circuit Using the initial state together with the cost and mixer unitary yields the complete circuit for the MaxCut problem, as shown in Figure 6.4. Measuring all qubits of this circuit gives a state distribution of all possible solutions to the MaxCut problem, which can now be used to compute the expectation value.

6.2.3. Expectation Value

The expectation value of a given output distribution can be computed by taking the score of each individual state (bitstring) and multiplying it by the number of counts of that state.

$$F_p = -\frac{1}{\#\text{shots}} \sum_{i=0}^{2^{|V|}-1} C_i E_i^{\text{cut}} \tag{6.7}$$

where C_i is the count (number of shots) per state and E_i^{cut} is the number of edges between two distinct sets encoded in the bitstring. Notice that the expectation value is a negative value, due to the fact that the classical optimizer attempts to minimize the expectation value, i.e., the more negative the value, the better the output distribution encodes the solution to the problem.

6.2.4. Simulation results

To verify the functionality of the MaxCut QAOA circuit in $|\text{Lib}\rangle$, a simulation on the Qiskit Aer simulator [120] is run for 4096 shots. The resulting state spectrum is depicted in Figure 6.5. In this figure, peaks of the optimal solution can be observed, with the highest peak at the state 100101. This indeed corresponds to one of the 18 optimal solutions for the MaxCut problem for the example graph.

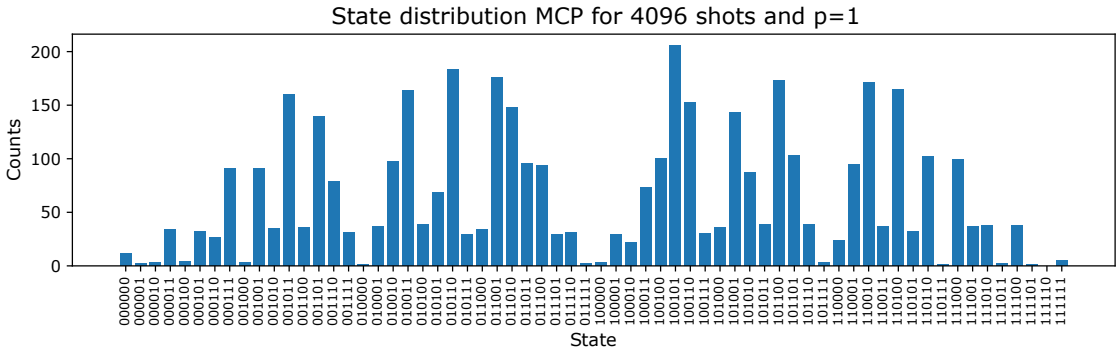


Figure 6.5: Simulation results for the MaxCut problem, 4096 shots on the Qiskit Aer simulator

The peaks of the optimal solutions become more distinguished as p increases, as shown in Figure 6.6. Non-optimal solutions are suppressed more and the overall score of the optimization result is improved in accordance with Equation 3.15.

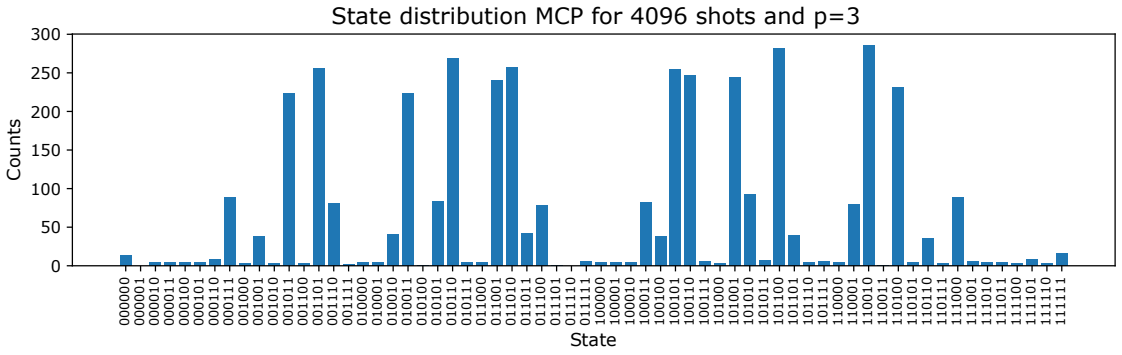


Figure 6.6: Simulation results for the MaxCut problem, 4096 shots on the Qiskit Aer simulator

6.3. Dominating set problem (DSP)

This problem was already part of QPack [21] and is henceforth also implemented in the |Lib⟩ version. The circuit of the dominating set problem (DSP) features a more complex cost Hamiltonian with high-level gates, requiring more gates and qubits than the MaxCut circuit. This nicely varies the type of benchmark application circuits used in QPack.

6.3.1. Problem description

A dominating set for a graph $G(V, E)$ is defined as the subset S of V such that each member of $V \backslash S$ is connected to at least one member of S. The dominating set problem aims to find the smallest set S that satisfies this condition. A common allegory is made with a network of surveillance vertices. Each surveillance vertex can survey itself and each vertex to which it is connected. The question then becomes: What is the smallest number of surveillance vertices needed to survey the whole network? For the example network, a solution (out of 15) to the dominating set problem is shown in Figure 6.7.

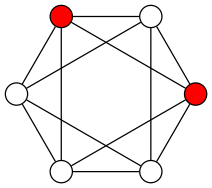


Figure 6.7: Solution of the dominating set problem of a 6-vertex 4-regular graph. The minimum number of vertices to survey all other vertices is 2, marked by the red vertices.

6.3.2. QAOA mapping

The mapping of the dominating set problem to QAOA was presented by Guerrero in 2020 [121]. The solution makes use of the high-level quantum OR-gate to encode the cost function using an ancilla qubit. For a complete graph, this approach requires $2V$ qubits. However, QPack uses the 4-regular graph as the problem input, this is reduced to only needing $V + 5$ qubits, which will become clear later in this section as the cost Hamiltonian is elaborated.

Encoding The solution is encoded in a bitstring $x = x^0 x^1 ... x^{|V|-1}$, where $x^n \in [0, 1]$. If a bit is in the subset D if it is 1, otherwise it is 0. All possible combinations of the bitstring x are valid (but not necessarily correct) solutions to the DSP.

Initial state Similar to the MaxCut problem, all possible combinations of bitstring x form valid potential solutions to the dominating set problem. Thus, the initial state can be initialized to be an equal superposition of all states. Hence, the initial state $|s\rangle$ is

$$|s\rangle = \prod_{i \in V} |+\rangle_i \tag{6.8}$$

Cost Hamiltonian Guerrero splits the cost function into two clauses, $T_k(x)$ and $D_k(x)$. The first clause measures the number of vertices surveyed (including itself) and is defined as

$$T_k(x) = \begin{cases} 1 & \text{if } x_k \text{ is connected to any } x_i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.9}$$

The second clause measures the number of surveillance vertices used, i.e., the number of ones in the string x:

$$D_k(x) = \begin{cases} 1 & \text{if } x_k = 0 \\ 0 & \text{if } x_k = 1 \end{cases} \tag{6.10}$$

Both clauses will be encoded using an ancilla qubit that sums the cost. Implementing these clauses like this will require an extra qubit along the V qubits that are necessary to encode the solution space.

To implement the $T_k(x)$ clause, a multi-OR-controlled quantum $R_z(\theta)$-gate is required. First, a three-qubit quantum OR-gate can be constructed out of a Toffoli gate and two CNOT gates, shown in Figure 6.8.

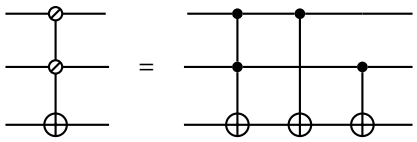


Figure 6.8: Quantum OR gate decomposition

The multi-OR-controlled gate is then simply a combination of multiple quantum OR-gates. For every extra control qubit, an additional ancilla qubit is needed as well. For example, a 3-control OR gate is decomposed as shown in Figure 6.9.

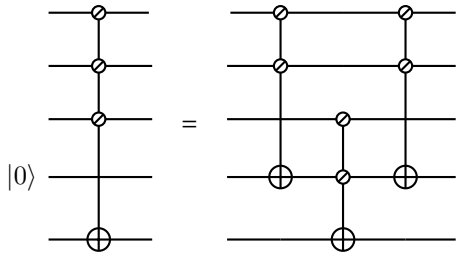


Figure 6.9: Multi-control Quantum OR gate decomposition

The final step is to perform a $R_z(\theta)$-operation on the cost ancilla qubit, controlled by multi-OR-controlled gates. With an extra ancilla qubit, this gate can be decomposed into two multi-OR-controlled gates and an $R_z(\theta)$-gate, as shown in Figure 6.10.

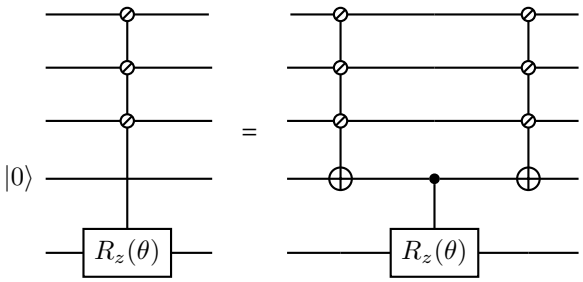


Figure 6.10: Multi-OR-controlled $R_z(\theta)$ gate decomposition

This multi-OR-controlled $R_z(\theta)$ can now be used to implement the $T_k(x)$ clause. For every vertex in the graph $G(V, E)$, such a gate is required, with the control qubits being the vertex itself and its neighbors, targeting the cost ancilla qubit. Since, for a regular graph, a vertex always has 4 neighbors, a 5-OR-controlled $R_z(\theta)$ gate is required, which uses 4 extra ancilla qubits. This makes the total number of qubits needed for the dominating set circuit equal to $V + 5$ for a 4-regular graph.

The $D_k(x)$ clause is simpler to implement and can be achieved by a bit flip of all bits in the bitstring. This can be implemented by inverted controlled $R_z(\theta)$-gates on the cost ancilla qubit. This gate can simply be implemented by using two X-gates and a controlled $R_z(\theta)$-gate as shown in Figure 6.11.

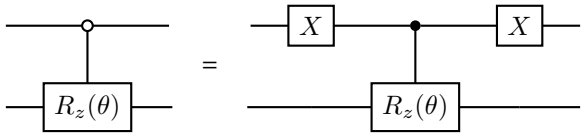


Figure 6.11: Inverted controlled $R_z(\theta)$ gate decomposition

Mixer Hamiltonian Similar to the MaxCut problem, the mixer Hamiltonian can be implemented with a $R_x(\beta)$-gate on each qubit that encodes the bitstring x, see Equation 6.6.

Complete circuit Combining the initial state generation, the cost and the mixer unitaries give us the building blocks for the QAOA circuit. For the example graph, the $p = 1$ circuit is shown in Figure 6.12.

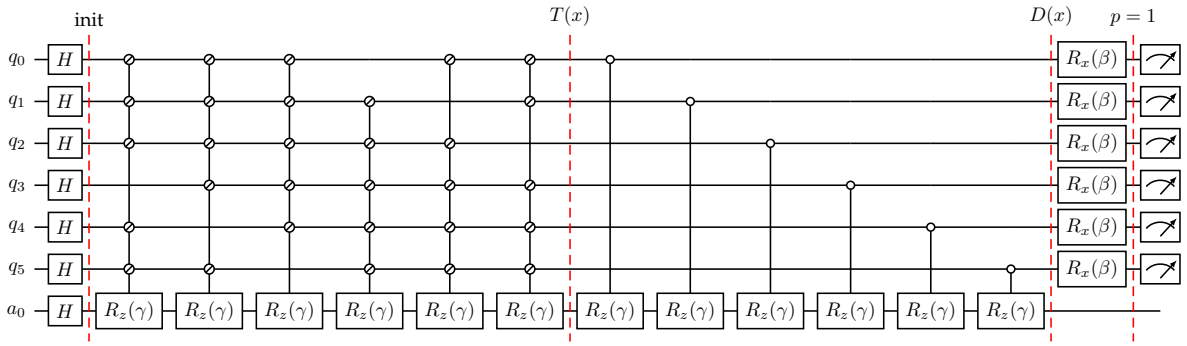


Figure 6.12: Circuit for the dominating set problem for the 6-vertex 4-regular graph example. The circuit shows the QAOA implementation for $p = 1$. Ancilla qubits for the multi-control OR gates are left out (4 in total)

Although the cost function is encoded in the ancilla qubit, only the first six qubits need to be measured to encode the solution space. This state distribution can then be used to compute the expectation value for the dominating set problem.

6.3.3. Expectation Value

Using the D- and T-clause functions, the expectation value for the dominating set problem can be computed from the measured output state distribution.

$$F_p = -\frac{1}{\#\text{shots}} \sum_{i=0}^{2^{|V|}-1} \left(C_i m(x_i) \sum_{j \in V} D_j(x_i) \right) \tag{6.11}$$

where x_i is the bitstring of the i-th state of the output distribution, C_i is the count (number of shots) per state, $\sum D_j(x_i)$ (see Equation 6.10) sums the number of surveillance vertices, and $m(x_i)$ checks if the solution monitors all vertices. This is the result of a single T_k bit being 0 (not surveyed) and can be expressed as:

$$m(x_i) = \prod_{j \in V} T_j(x_i) \tag{6.12}$$

6.3.4. Simulation results

Verifying the functionality of the dominating set QAOA implementation is again done by simulation on the Qiskit Aer simulator [120]. The results for $p = 1$ can be found in Figure 6.13. Here, a spectrum of roughly uniform distribution can be observed and no distinguishable solution can be found, in contrast to the MaxCut problem, where a rough estimate could be found for the same example graph for $p = 1$. This can be explained by the fact that the DSP circuit has a larger circuit width and depth, making it more difficult for the optimizer to find an optimal solution.

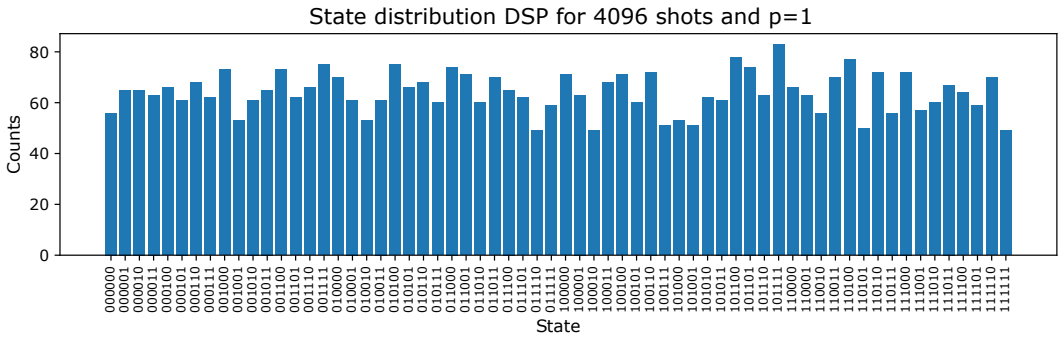


Figure 6.13: Simulation results for the dominating set problem, 4096 shots on the Qiskit Aer simulator

Increasing p by one gives the results shown in Figure 6.14. Here, more distinguishable peaks are observed, which indeed encode a solution to the dominating set problem. The highest peak can be found at string 010010, which is one of the optimal solutions for the 6-vertex example graph (for the 4-regular example graph, any string with 2 ones and 4 zeros is an optimal solution).

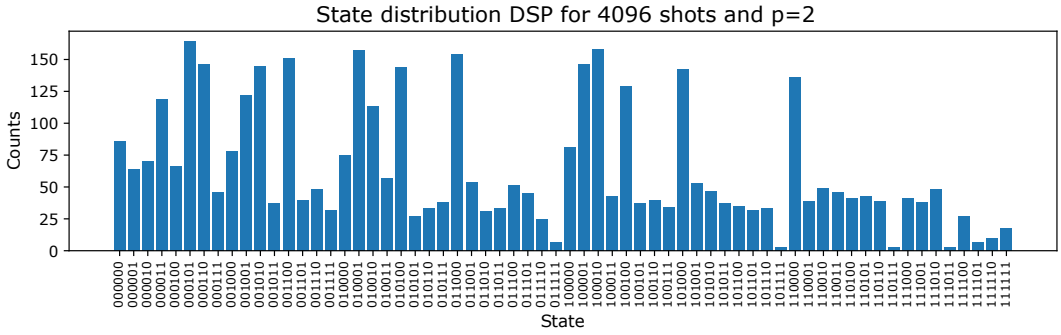


Figure 6.14: Simulation results for the dominating set problem, 4096 shots on the Qiskit Aer simulator

Increasing p even further (Figure 6.15) gives an even more distinct solution, which is of course expected behavior as p increases.

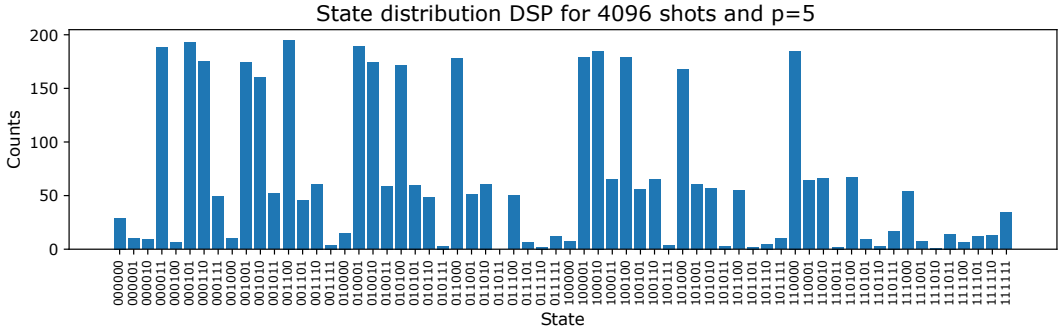


Figure 6.15: Simulation results for the dominating set problem, 4096 shots on the Qiskit Aer simulator

This gives a good insight into how p needs to be selected for more complex circuits, which will affect the accuracy and capacity metric of the QPack benchmark scores later on.

6.4. Maximal independent set problem (MIS)

The next problem implemented in QPack uses a more complex mixer family than the previous two problems. Instead of using simple single-qubit X-rotations, the solution space for the maximal independent set (MIS) problem is mixed using controlled mixers.

6.4.1. Problem description

Consider a graph $G(V, E)$, the subset $S \in V$ is the set of vertices that are not connected to each other. Find the set of vertices that maximizes the size of S. For the 6-vertex example graph, there are three solutions. One is shown in Figure 6.16. The other two are similar, but are the other two sets of opposite vertices in this ring shape. This problem has a similar feel to the dominating set problem. A solution to the maximal independent set problem is also a solution to the dominating set problem for regular graphs, but not vice versa.

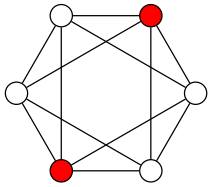


Figure 6.16: Solution of the maximal independent set problem of a 6-vertex 4-regular graph. The maximal independent set has two vertices

6.4.2. QAOA mapping

Encoding The vertices are encoded with bitstring $x = x^0x^1...x^{|V|-1}$, where a vertex is in V if $x^i = 1$, otherwise $x^i = 0$.

Initial state The initial qubit state can be any arbitrary state, since all possible states are possible solutions to the MIS problem for any graph. Other than with the MCP and DSP, all qubits are initialized as the empty set, that is, all qubits are initialized as $|0\rangle$.

$$|s\rangle = \prod_{i \in V} |0\rangle_i \tag{6.13}$$

Cost Hamiltonian The cost function is the size of the independent set, which is simply the sum of all bits x_i in the bitstring: $\sum_{i \in V} x_i$. Using a transformation of the objective function [66], this function can be translated to cost Hamiltonian

$$C = \sum_{i \in V} \sigma_i^z \tag{6.14}$$

Which can be implemented with a depth-1 circuit of $R_z(\gamma)$-gates for each qubit.

Mixer Hamiltonian The maximal independent set QAOA implementation provides a more complex family of mixers compared to the MaxCut or dominating set problems. This is a set of controlled mixer operations. For the MIS problem, we only want to mix the solution space (adding or removing a vertex from the set S) based on its neighboring vertices and whether they are already in the solution space or not. The mixer Hamiltonian is thus given by

$$B_i = \sigma_i^x H_{\text{NOR}(x_{\text{neighbor}(i)})} \tag{6.15}$$

where H_{NOR} is the NOR-controller as described in [66]. To build this Hamiltonian, a set of partitioned controlled mixers is used [66], where

$$U_{B,i}(\beta) = \Lambda_{\text{NOR}(x_{\text{neighbor}(i)})}(e^{-i\beta\sigma_i^x}) \tag{6.16}$$

where the Λ_{NOR} function denotes the NOR-controlled parameterized X-rotation. This is implemented using a quantum NOR-controlled $R_x(\theta)$ operation, described in Figure 6.17. This is the quantum OR circuit as elaborated in the dominating set problem section, but with two added X gates that function as a NOT gate.

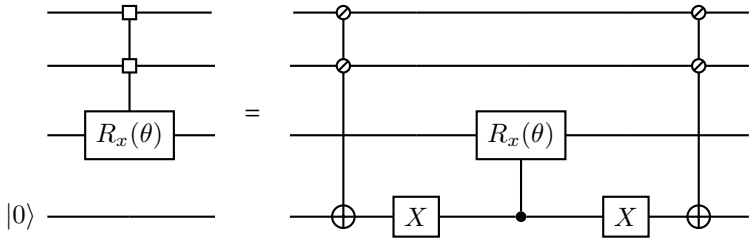


Figure 6.17: NOR-controlled $R_x(\theta)$ gate decomposition

It becomes clear that this circuit needs an extra ancilla qubit, as well as two more ancilla qubits for the multi-controlled quantum NOR gates since the maximum number of neighbors in the 4-regular graph is 4, similar to the multi-controlled quantum OR gate in Figure 6.10. This requires the MIS circuit to have $V + 3$ qubits.

Complete circuit The complete circuit can now be constructed with the aforementioned circuit building blocks, shown in Figure 6.18. Ancilla qubits are left out, but it should be noted that three more qubits are used in the decomposed circuit. In order for the solution states of the outcome to be more balanced, the order of the partial mixers is shifted per p iteration.

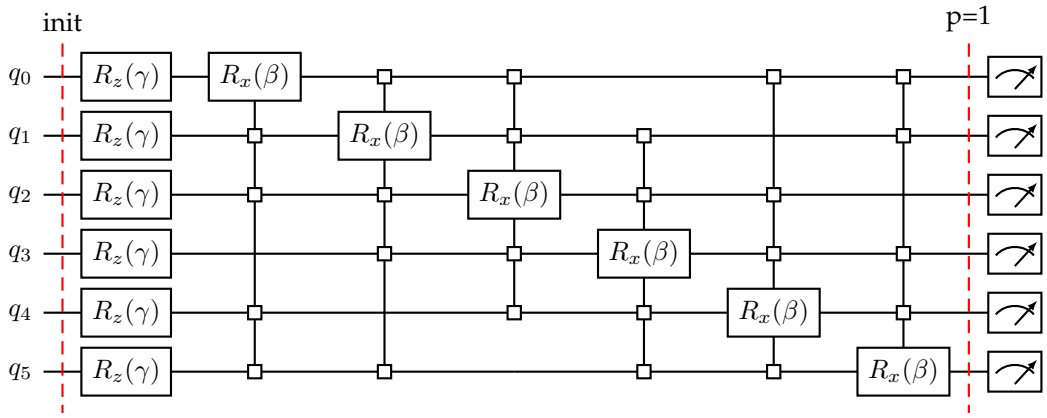


Figure 6.18: Circuit for the maximal independent set problem for the 6-vertex 4-regular graph example for $p = 1$. Three ancilla qubits used by the NOR-controlled gates are not drawn.

6.4.3. Expectation Value

The expectation value for the MIS problem can be calculated by checking each output state to see if it is a valid set and then giving that set a value of the number of vertices in that set.

$$F_p = -\frac{1}{\#\text{shots}} \sum_{i=0}^{2^{|V|}-1} \left(v(x_i) C_i \sum_{j \in V} x_i^j \right) \tag{6.17}$$

where x_i is the i-th state, x_i^j is the j-th bit of the i-th state, C_i is the count (number of shots) and $v(x_i)$ checks if the bitstring x_i is a valid independent set, that is, $v(x_i) = 1$ if the set is an independent set and $v(x_i) = 0$ otherwise.

6.4.4. Simulation results

The verification of the MIS problem is again done by executing the QAOA on the Qiskit Aer simulator for 4096 shots. The results for $p = 1$ are shown in Figure 6.19. Here, it can be seen that, in contrast to the MCP and DSP implementations, the state distribution is very sparse. This is due to the fact that the initial state started with the empty set. Although the three solutions (001001, 010010 and 100100) can be clearly distinguished, other nonsolutions are still a prominent part of the solution space.

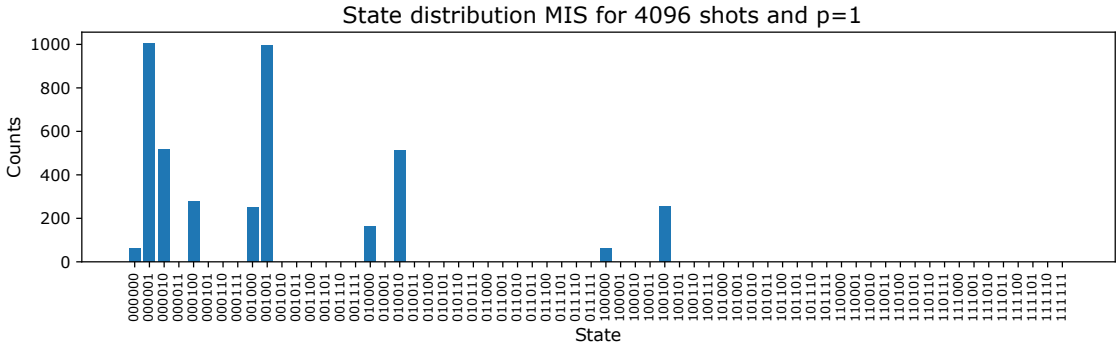


Figure 6.19: Simulation results for the Max independent set problem, 4096 shots on the Qiskit Aer simulator

If we increase p, these nonsolutions are suppressed, resulting in a cleaner state distribution and a higher expectation value, see Figure 6.20

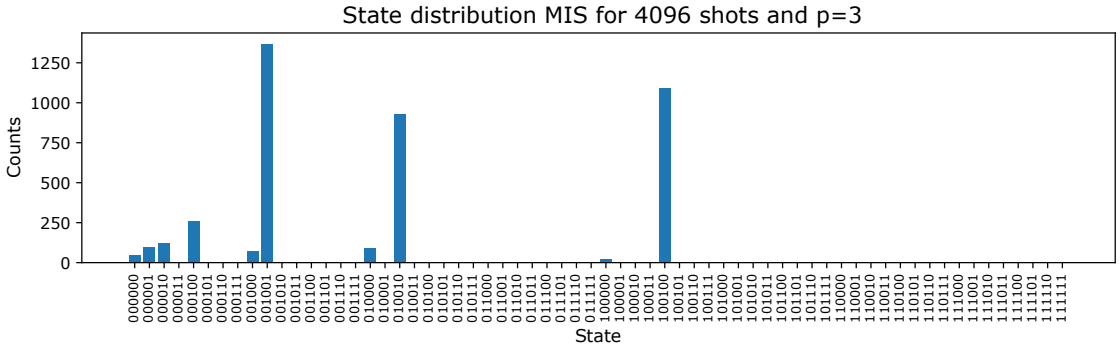


Figure 6.20: Simulation results for the Max independent set problem, 4096 shots on the Qiskit Aer simulator

6.5. Traveling salesperson problem (TSP)

The last QAOA problem implemented in QPack differs mainly in the encoding scheme compared to the previous three problems. Instead of encoding each vertex with a qubit, the adjacency matrix of the graph is mapped onto the qubits, which allows for encoding paths through network topologies.

6.5.1. Problem description

For a graph $G(V, E)$ and distances $d : [V]^2 \to \mathbb{R}_+$, find an ordering of the vertices that minimizes the total distance traveled for the corresponding tour. A tour visits each vertex once and returns from the last vertex to the first [66]. In other words, find the Hamiltonian cycle with the shortest distance.

Since the number of qubits scales quadratic with the size of the problem, due to the encoding of the problem, the example used for the traveling salesperson problem (TSP) will be a $G(3, 3)$ graph, shown in Figure 6.21, instead of the previously used 6-vertex 4-regular graph. The solution here is trivial, since there is only one possible path. For the purpose of scaling the problem size in the future, all edges will have weight 1. If there is no edge, the connection has weight 10.

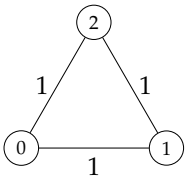


Figure 6.21: Reduced graph size for the traveling salesperson problem example. Vertices are labeled 0 to 2 and edges all have weight 1

6.5.2. QAOA mapping

The starting point for the implementation of the traveling salesperson problem was presented in a blog by Ceroni [122]. From his work, the problem encoding, cost Hamiltonian and mixer Hamiltonian have been implemented in the QPack TSP algorithm, with some minor tweaks to the original code. An improved state initialization using a Dicke state has been implemented for a more uniform distribution of the solution space, as presented by Mesman et al. [20, 21].

Encoding In order to find a path, the graph will be represented in an adjacency matrix and a corresponding weight matrix, which contains the weights (distances) of each vertex to each other.

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}, \qquad W = \begin{pmatrix} 20 & 1 & 1 \\ 1 & 20 & 1 \\ 1 & 1 & 20 \end{pmatrix}$$

where a vertex itself has weight 20, to discourage the path to travel to itself. The adjacency matrix is then mapped onto a qubit string by the matrix index. For instance, the example adjacency matrix can be encoded into the string $x = 011101110$.

Initial state Since not all string values are a valid solution to the problem (e.g., only strings that encode a valid adjacency matrix), initializing all qubits in a uniform superposition over all states would not be efficient. Instead, all qubits that encode a row of the adjacency matrix are set to a superposition of states with Hamming weight 2, known as a Dicke state $|D_k^n\rangle$ [123], defined as the equal superposition of all n-qubit states $|x\rangle$ with Hamming weight $wt(x) = k$. For the example graph in Figure 6.21, a row would be encoded as

$$\left|D_2^3\right\rangle = \frac{1}{\sqrt{3}}(|011\rangle + |101\rangle + |110\rangle) \tag{6.18}$$

The quantum circuit to encode the row in this way is presented by Mukherjee et al. [124]. The resulting initialization circuit for each row of the TSP example adjacency matrix will thus be

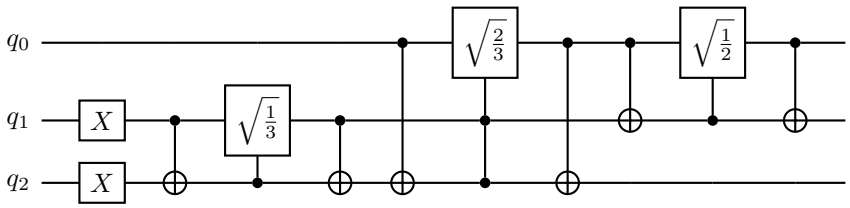


Figure 6.22: Row initialization for the traveling salesperson problem

where the controlled square root $\sqrt{\frac{l}{n}}$ denotes a controlled $R_y(2\text{acos}(\sqrt{\frac{l}{n}}))$ gate [124].

Cost Hamiltonian The cost Hamiltonian for the Traveling Salesperson problem consists of two parts, a soft and a hard constraint. The soft constraint is defined as

$$C_{\text{soft}} = \sum_{i,j} \frac{W_{ij}}{2}(1 - \sigma_{ij}^z) \tag{6.19}$$

where W_{ij} is the distance between two vertices i and j, defined by the distance matrix. Dropping the global phase term results in

$$C_{\text{soft}} = -\frac{1}{2}\sum_{i\in V}\sum_{j\in V} W_{ij}\sigma_{ij}^{z} \tag{6.20}$$

This constraint will force the result to provide a solution that encodes the shortest path. The hard constraint is defined as

$$H_{\text{hard}} = -\sum_{\{a,b\}\in\mathcal{D}} \sigma_a^z\sigma_b^z \tag{6.21}$$

where $a = A_{ij}$, $b = A_{ji}$ and $\mathcal{D}$ is the set of bit pairs that are a reflection around the diagonal of the adjacency matrix of eachother. This results in ZZ-rotations for every two qubit pairs that represent the reflection across the diagonal of the adjacency matrix. This essentially forces the resulting bitstring to produce a symmetric matrix. The final cost Hamiltonian is then:

$$C = C_{\text{soft}} + \omega C_{\text{hard}} = -\frac{1}{2}\sum_{i\in V}\sum_{j\in V} W_{ij}\sigma_{ij}^{z} - \omega\sum_{\{a,b\}\in\mathcal{D}} \sigma_a^z\sigma_b^z \tag{6.22}$$

where ω must be chosen such that it is relatively large to the soft constraint Hamiltonian. According to Ceroni, $\omega = 5$ is sufficient [122]. This cost Hamiltonian can then be mapped to a qubit circuit, as seen in Figure 6.23.

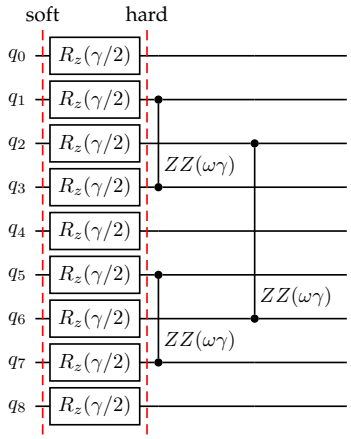


Figure 6.23: Cost unitary for the traveling salesperson example network

Mixer Hamiltonian Finally the solution space can be mixed with the mixer Hamiltonian, which can be achieved with SWAP gates between the adjacent qubits of a row. The SWAP operations conserve the Hamming weight and thus the solution space is preserved. The mixer Hamiltonian then is

$$B = \sum_{i,j}\text{SWAP}_{i,j} = \frac{1}{2}\sum_{i,j}(\sigma_i^x\sigma_j^x + \sigma_i^y\sigma_j^y) \tag{6.23}$$

which is an XX- and YY-rotation for each adjacent qubits pair encoding a row. For the three-vertex example, the circuit for a row would then look like the one shown in Figure 6.24.

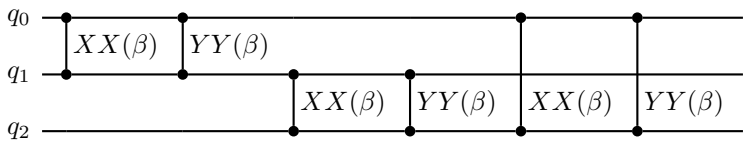


Figure 6.24: Mixer unitary for the traveling salesperson example network for one row

Complete circuit The complete circuit for the example network is shown in Figure 6.25 for $p = 1$. For each row, the Dicke initialization ensures that every row has a Hamming weight of 2 in complete superposition, resulting in a total of 3^3 nonzero states. The cost function then applies the soft and hard constraint, after which the mixer unitary is applied to mix up each row of the solution space. All qubits are measured and a distribution of $2^{|V|^2}$ states is obtained.

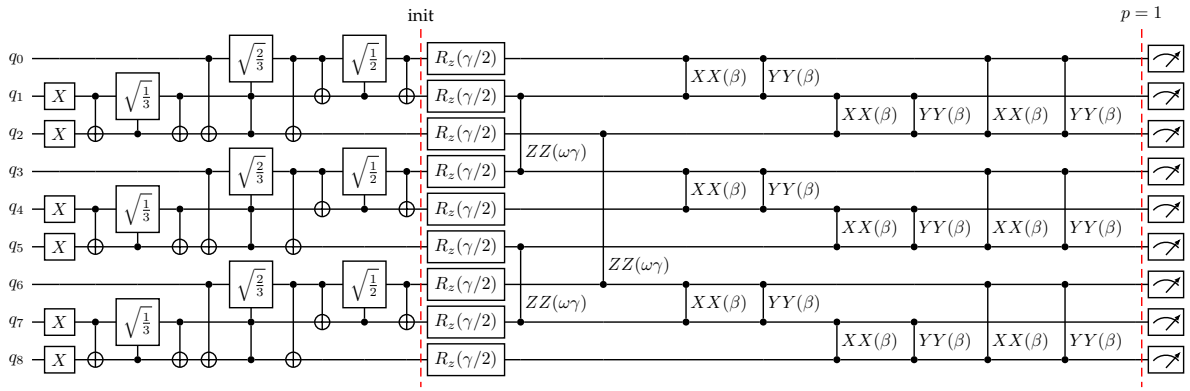


Figure 6.25: Complete circuit for the traveling salesperson example network

6.5.3. Expectation Value

The expectation value for the TSP problem would normally be the sum of the edges in the path (soft constraint). However, since we want to push the solution space towards valid adjacency matrices, the hard constraint is also taken into consideration. Then, the following expectation function is obtained:

$$F_p = \frac{1}{\#\text{shots}} \sum_{k=0}^{2^{|V|}-1} C_k \left(\sum_{i \in V} \sum_{j \in V} W_{ij} x_k^{ij} - \omega \sum_{\{a,b\} \in \mathcal{D}} x_k^{ab} \right) \tag{6.24}$$

where x_k^{ij} is a bit of the i-th row and the j-th column of the adjacency matrix A that is encoded by the k-th state, C_k is the count (number of shots), and W_{jk} the distance matrix.

6.5.4. Simulation results

Before checking the results of the QAOA iterations for different values of p, the Dicke initialization is evaluated first. This can be seen in Figure 6.26. Because of the large number of states (2^{3^2}), only the states that have more than 0 counts are shown in this and subsequent TSP figures. As expected, all 27 initial states are represented in the output state distribution at roughly the same amplitudes.

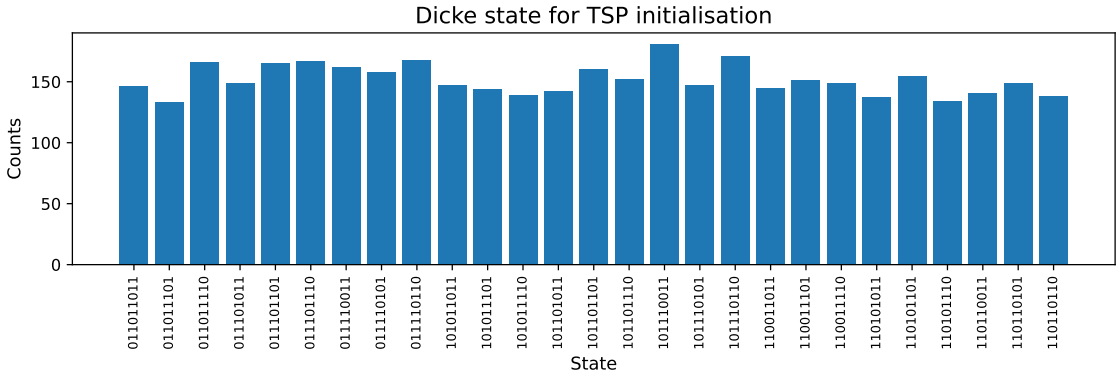


Figure 6.26: Simulation results Dicke state initialization, 4096 shots on the Qiskit Aer simulator (zero-count states are left out)

Running the Traveling Salesperson QAOA implementation on the Qiskit Aer simulator for $p = 1$ already gives useful results, see Figure 6.27. Here, a clear peak at bitstring 011101110 can be observed, which

indeed encodes the adjacency matrix that is the solution for the triangular graph. Although it is the only valid adjacency matrix, the QAOA algorithm can clearly find it.

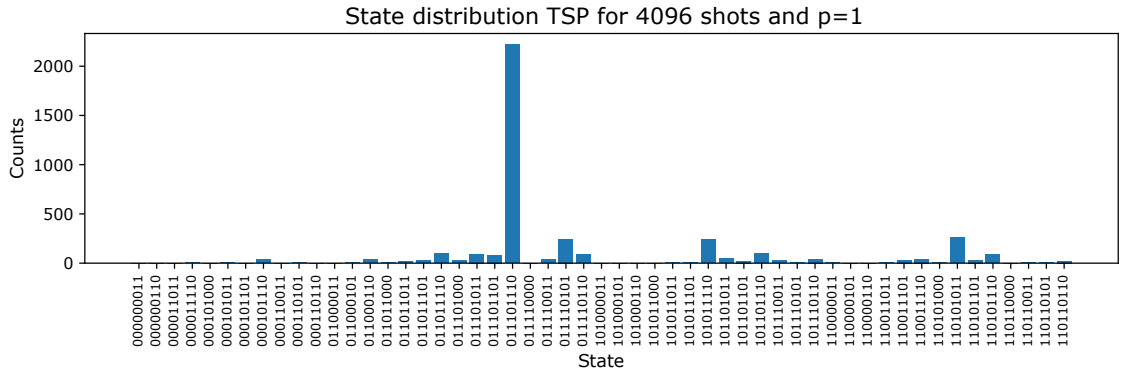


Figure 6.27: Simulation results for the traveling salesperson problem, 4096 shots on the Qiskit Aer simulator

Higher expectation values can be obtained when increasing p further, which reduces the amplitude of nonsolutions further, see Figures 6.28 and 6.29.

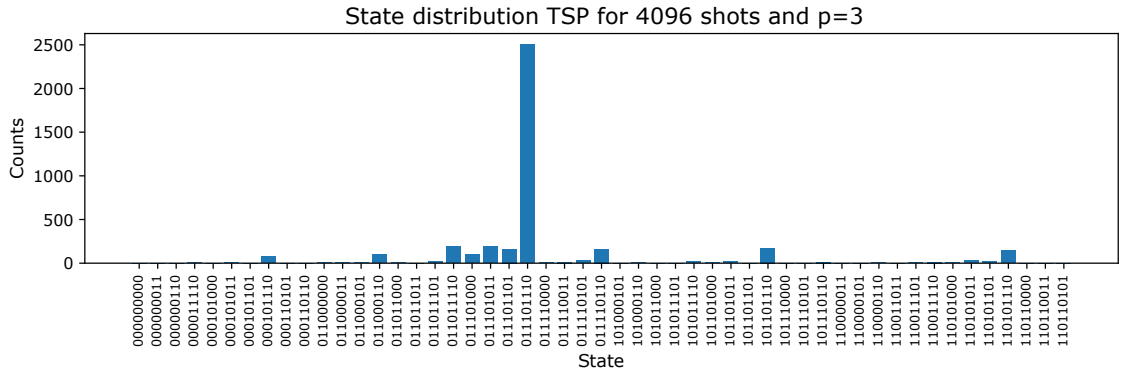


Figure 6.28: Simulation results for the traveling salesperson problem, 4096 shots on the Qiskit Aer simulator

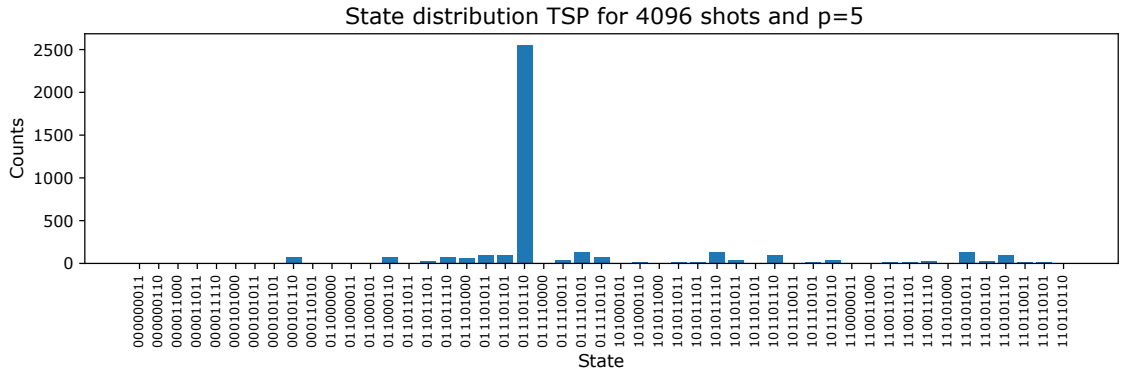


Figure 6.29: Simulation results for the traveling salesperson problem, 4096 shots on the Qiskit Aer simulator

6.6. Resource comparison

As mentioned previously in this chapter, some QAOA problems require more complex quantum circuits and thus require more resources (qubits and quantum gates). In Table 6.1, a comparison is made between all QAOA implementations for QPack, listing the number of qubits needed, single-qubit gates,

and multi-qubit gates. All high level gates are decomposed to standard single-qubit gates (rotation on a single qubit) or standard multi-qubit gates (controlled rotations, Toffoli). Resources are a function of the 4-regular graph size N and the QAOA iterations p.

	Qubits	Single-qubit gates	Multi-qubit gates
MCP	N	$N(3p+1)$	$4Np$
DSP	$N+5$	$N(3p+1)$	$44Np$
MIS	$N+3$	$3Np$	$31Np$
TSP	N^2	$2N^2 + p\frac{23N^2-N}{2}$	$4N^2 - 6N + p(5N^2 - N)$

Table 6.1: Required resources for each QAOA problem

These results are visualized in Figure 6.30 for a better insight on how these problems scale when N increases.

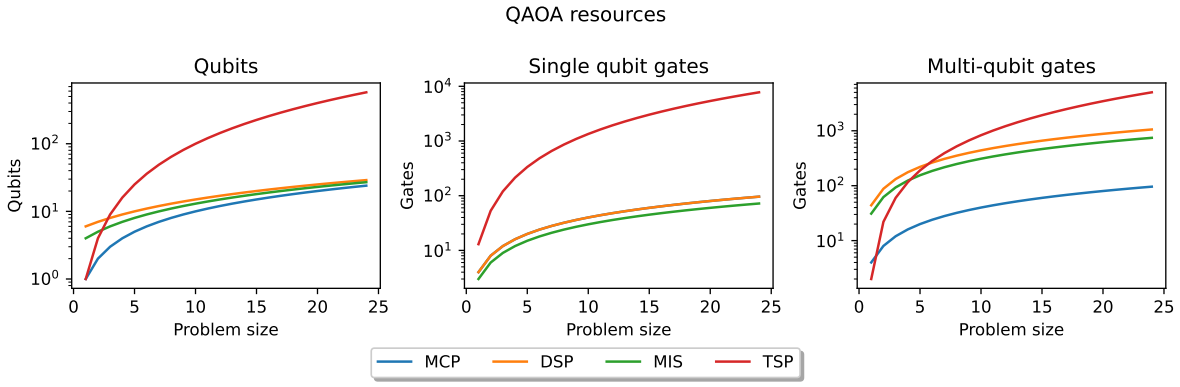


Figure 6.30: Scaling of QAOA resources for each problem set ($p = 1$)

As expected, the resources of the traveling salesperson implementation scale significantly more rapid than the MCP and DSP implementations, since the TSP qubit encoding scheme scales quadratic with the problem size, in contrast to the linear relation of the MCP, DSP, and MIS qubit encoding. The MCP and DSP circuits scale with the same number of single-qubit gates, but the DSP increases more ten time times faster in multi-qubit gates. The MIS implementation scales similarly in single-qubit gates, but with an offset of N.

This resource comparison shows that the implemented QAOA programs not only have varied cost and mixer unitaries, but also have a varied distribution of number of single- and multi-qubit gates. This variation will provide a solid basis to use QAOA as scalable quantum benchmarking applications.

7

Benchmark scores

Using the measured quantum execution data and benchmark criteria mentioned in Chapter 4, actual benchmark scores can now be defined. Taking inspiration from BAPCo [34, 35], an overall benchmark score can be decomposed into multiple subscores. These subscores and the connection between their quantum execution data (Table 4.1) can be seen in Figure 7.1. The overall score is divided into four subcategories: *runtime*, *accuracy*, *scalability* and *capacity*. Runtime will evaluate the time the quantum computer needs to execute a given circuit. Accuracy reflects the ability of the classical optimizer to find the optimal solution. Scalability evaluates the capability of the quantum computer to execute larger quantum circuit sizes. The capacity subscore will reflect the number of qubits of a quantum computer for which the classical optimizer is able to find an optimal value below a predefined threshold. The script for data processing can be found in `https://gitlab.com/libket/qpack/-/blob/stable/Qpack/data_processing/process_benchmark_data.py`

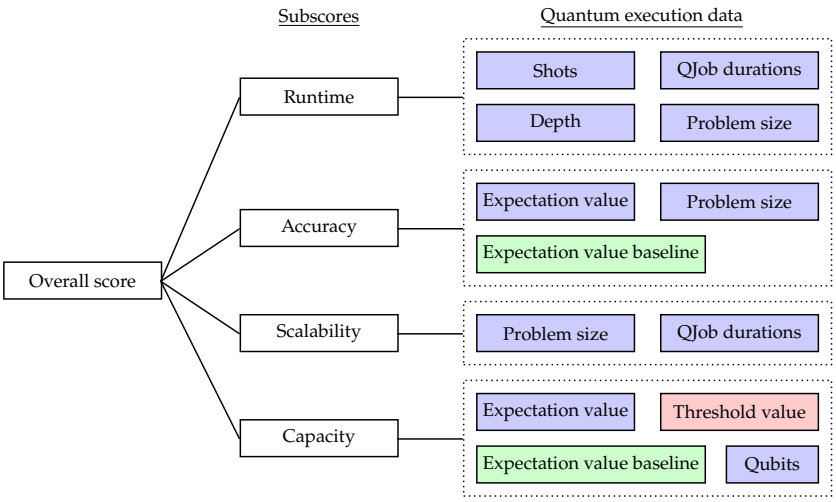


Figure 7.1: Benchmark score decomposition. The overall score is a combination of four subscores, which each are connected to their relevant quantum execution data. Blue boxes indicate that data is acquired during QPack execution, green boxed indicate values that have been collected from a separate benchmark run and the red box indicates a static value for all benchmark runs.

For the benchmark subscores, a distinction is made between pure scores S^{pure} and mapped scores S^{mapped}. The pure scores are the result of the transformed measured data, which provide quantitative score metrics. The mapped scores are obtained by taking these pure scores as input and mapping them to be proportional to performance and balanced relative to the other subscores.

7.1. Runtime

Perhaps the most straightforward metric to use is the time it takes to execute a quantum circuit, the quantum computer runtime. After all, quantum computing promises increased runtime performance of classical computers, so it is a valuable metric to include in the benchmark scores. For the runtime score, it is assumed that quantum computers can execute gates in parallel where possible. Then, to get

a fair runtime score for different depths and shots, a score can be defined as the number of gates per second that a quantum computer is able to execute. The number of gates g to execute a single circuit depends on the circuit depth and the number of shots, as described by Equation 7.1.

$$g_{P,N} = D_{P,N} S_{P,N} \tag{7.1}$$

where $D_{P,N}$ and $S_{P,N}$ are the depth and number of shots for a given VQA problem P and problem size N, respectively. Here, $D_{P,N}$ is the depth of the untranspiled circuit, that is, the hardware-agnostic implementation of QPack using the full set of software-visible gates. To ensure fair evaluation, this depth is the same for every quantum computer. This depth here is defined as the length of the critical path of the untranspiled QPack circuit. Transpiling the circuit in an efficient manner to fit the qubit topology and the base gate set of a quantum computer is a job left to the quantum computer provider. Since this is an application-level benchmark, performance is evaluated on the quantum application that is executed, which will be reflected in the manner in which a circuit is transpiled by the quantum computer provider.

As seen in Chapter 4 Table 4.1, two types of quantum time durations are being measured, the circuit execution time T^{QE} and the QJob time T^{QJob}. They time different parts of the quantum computing stack. An overview of what such a stack looks like is shown in Figure 7.2, with T^{QE} and T^{QJob} specified.

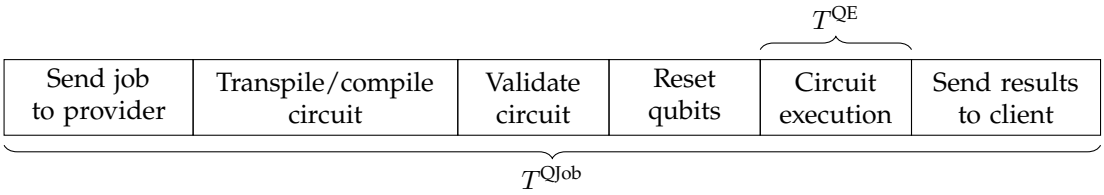


Figure 7.2: Typical quantum job timeline decomposition. Exact steps can vary per quantum provider

From earlier work on QPack [20, 21] and a recent paper on profiling the quantum control stack [125], it was found that in the current quantum computing era, T^{QE} is not the dominant contributor to the total QJob time. We can verify this by looking at the MaxCut benchmark data set that has been run on the IBMQ backend with the Quito quantum processor in Figure 7.3. Here, we see that T^{QJob} is much larger than T^{QE}, which is only 24.8 % of the total QJob runtime. This makes T^{QJob} more suitable for the current NISQ-era quantum computers, as overhead still plays an important role in efficient quantum runtime. Another problem with the T^{QE} duration is that exact definitions may differ per vendor of a quantum computer, or they are not provided at all. Since T^{QJob} is measured by QPack itself, it is a more reliable measurement to use for benchmarking.

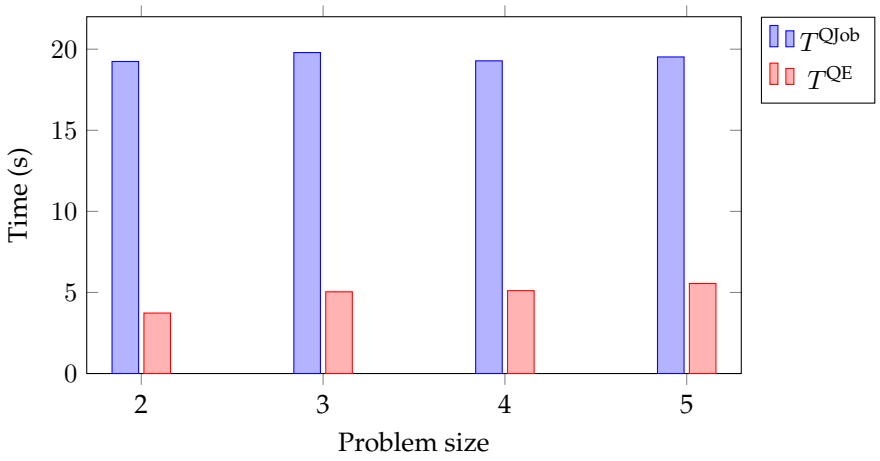


Figure 7.3: Quantum duration data of the IBMQ Quito quantum processor for the QAOA MaxCut problem.

For a given VQA problem, the runtime score can thus be computed as the average gates per second over all problem sizes, see Equation 7.2

$$S^{\text{pure}}_{\text{runtime},P} = \frac{1}{N_{P,e} - N_{P,s}} \sum_{N=N_s}^{N_e} \frac{g_{P,N}}{T^{\text{QJob}}_{P,N}} \tag{7.2}$$

where $T^{\text{QJob}}_{P,N}$ is the average time of quantum jobs for problem P and problem size N. $N_{P,s}$ and $N_{P,e}$ are the smallest and largest problem size for which the problem is evaluated.

7.2. Accuracy

Accuracy of the measured quantum state is an important aspect of quantum computers. Where fidelity is a common measure of the accuracy of single- or two- qubit gates, such low-level characteristics become more indistinguishable when the circuit size increases. The resulting output state deficiencies due to gate noise and qubit decoherence and relaxation can be compared to those of an ideal quantum simulator. For QPack, this entails comparing the performance of the classical optimizer on a perfect deterministic simulator to its performance on a noisy and nondeterministic quantum computer. In this case, the QuEST simulator [114] is used as an ideal simulator with a deterministic output state. This way, the classical optimizer can perform in a noiseless case and find the lowest possible solution for a given problem. The reason why the theoretical achievable score is not used as a reference is the fact that not all VQAs optimally encode the solution. For QAOA, the number of circuit iterations p limits the performance of QAOA. To avoid getting circuits that grow too large, p is static. In the case of VQE, the ansatz to find the ground-state energy of the Hamiltonian does not always encode the complete solution space and thus limits the accuracy of the VQE approximation.

The accuracy score is defined as the average relative error between the expectation value of the ideal simulator (QuEST) and the quantum computer under test. The relative error is simply the absolute error divided by the ideal expectation value.

$$S^{\text{pure}}_{\text{accuracy},P} = \frac{1}{N_{P,e} - N_{P,s}} \sum_{N=N_s}^{N_e} \frac{E^{\text{ideal}}_{P,N} - E^{Q}_{P,N}}{E^{\text{ideal}}_{P,N}} \tag{7.3}$$

where $E^{\text{ideal}}_{P,N}$ and $E^{Q}_{P,N}$ are the ideal and quantum computer's minimal optimizer expectation values over all execution cycles, respectively.

7.3. Scalability

Another performance characteristic is the scalability of quantum circuits. Although a quantum computer has a certain number of qubits to work with, its topology could make the evaluation of larger circuits more difficult as mapping and transpiling efficiently become more complex. Scalability looks at the runtime trend of a quantum computer for a growing circuit size.

To quantify the scalability, the exponential growth of the average quantum job time against the problem size will be evaluated. This is done by fitting the quantum job time T^{QJob} to function:

$$\tilde{T}^{\text{QJob}}(N) = N^a \tag{7.4}$$

where N is the problem size of the current problem. To fit this curve, the value of a needs to be determined. This is done by normalizing the input data and optimizing the value of a with a least-squares cost function using Nelder-Mead[105]. This way, the best fit for a can be found for the input data, as shown in Figure 7.4.

The value of a can now be used as a quantification of the scalability score, see Equation 7.5. For example, if a quantum job time scales linear with the problem size, $a = 1$ is expected. In the case that $a > 1$, the QJob time grows faster than the problem size and vice versa for $a < 1$.

$$S^{\text{pure}}_{\text{scalability},P} = a_P \tag{7.5}$$

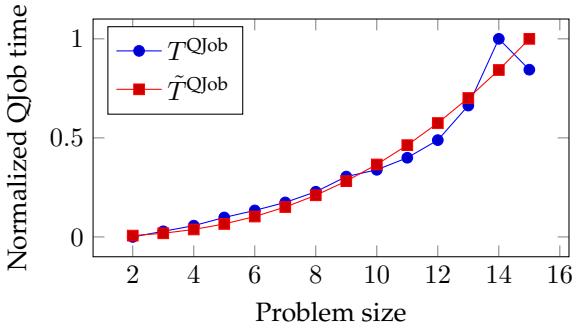


Figure 7.4: Normalized input data and fitted curve for the QAOA maximum cut problem on Qiskit Aer simulator

7.4. Capacity

The final score metric is the capacity of the quantum computer. This is the number of qubits that a quantum computer is able to run within a margin of the desired output accuracy. This is a similar approach to the Atos Q-score [14], but generalized over multiple quantum applications and using the QuEST simulator with the classical optimizer as a baseline. The capacity score is then the highest number of qubits $Q_{P,N}$ corresponding to problem P with problem size N for which the quantum computer can achieve a relative error within a set threshold accuracy A^*. The score is thus defined as:

$$S^{\text{pure}}_{\text{capacity},P} = \max\{Q_{P,N} \text{ where } \frac{E^{\text{ideal}}_{P,N} - E^{Q}_{P,N}}{E^{\text{ideal}}_{P,N}} \leq A^*\} \tag{7.6}$$

The main concern of this benchmark score is the value of A^*, which is arbitrarily chosen. Setting A^* to a good value is a tricky choice, as choosing this threshold value inherently defines the value at which the output of a quantum computer is considered acceptable, which could differ from application to application. By analyzing the obtained data from running the QPack benchmark on local quantum simulators (see Appendix D.1), it becomes clear that the relative accuracy differs per VQA result, where the lowest relative error rates are achieved by the RH benchmark and the highest error rates by the TSP benchmark. Since all simulators operate in a noiseless mode, they should all be able to achieve their maximum capacity score. This is why, currently, A^* is implemented as 25% relative accuracy.

7.5. Subscore mapping, balancing & combining

With all pure subscore metrics defined for each problem set, they need to be mapped and balanced to create scores that operate in a similar range to one another. Mapping transforms the pure score, such that a better performance in a score category leads to an increased benchmark score. This is then scaled up or down to get the subscore to be in a similar range relative to the other subscores, which we refer to as balancing the subscores. Combining then describes the method on which the subscores of different benchmark applications are merged to form a combined subscore.

For the runtime subscore, the decimal logarithm function is used to scale the pure subscore.

$$S^{\text{mapped}}_{\text{runtime},P} = \log_{10}(S^{\text{pure}}_{\text{runtime},P}) \tag{7.7}$$

which is able to map the whole range of the pure subscore, because $S^{\text{pure}}_{\text{runtime},P} > 0$.

Then, the accuracy score can be scaled to a range similar to the mapped runtime score. However, taking a log function is not applicable, since $S^{\text{pure}}_{\text{accuracy},P} \in \mathbb{R}$, e.g., the possibility that a simulator performs better than the baseline is allowed. Since a higher subscore should represent a better solution, a lower $S^{\text{pure}}_{\text{accuracy},P}$ should produce a higher score. This can be achieved with the mapping function.

$$f_{\text{map}}(x) = \frac{\pi}{2} - \arctan(x) \tag{7.8}$$

which maps the value x into the range $[0, \pi]$ as x decreases. Using this mapping function, we can map and balance the accuracy subscore to

$$S^{\text{mapped}}_{\text{accuracy},P} = c_0 f_{\text{map}}(c_1 S^{\text{pure}}_{\text{accuracy},P}) \tag{7.9}$$

where c_0 scales the mapping function and c_1 adjusts the sensitivity of the mapping function. By trial and error, $c_0 = 10$ and $c_1 = 5$ were found to be adequate. The same mapping function can be used for the scalability subscore, because again a lower value of $S^{\text{pure}}_{\text{scalabilty},P}$ should result in a higher score. Although $S^{\text{pure}}_{\text{scalabilty},P} \in \mathbb{R}$, values of $S^{\text{pure}}_{\text{scalabilty},P}$ are generally expected to be larger than one. To account for this, the mapping function is shifted by one:

$$S^{\text{mapped}}_{\text{scalability},P} = c_2 f_{\text{map}}(c_3 (S^{\text{pure}}_{\text{scalabilty},P} - 1)) \tag{7.10}$$

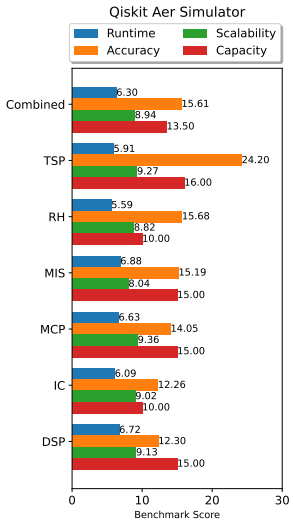
where c_2 scales the mapping function and c_3 adjusts the sensitivity of the mapping function. Using trial and error again, $c_2 = 10$ and $c_3 = 0.75$ were found to be adequate. The capacity score is found to already be in a similar range to the other performance metrics and reflects the performance of a quantum computer adequately. The capacity score is then kept as is:

$$S^{\text{mapped}}_{\text{capacity},P} = S^{\text{pure}}_{\text{capacity},P} \tag{7.11}$$

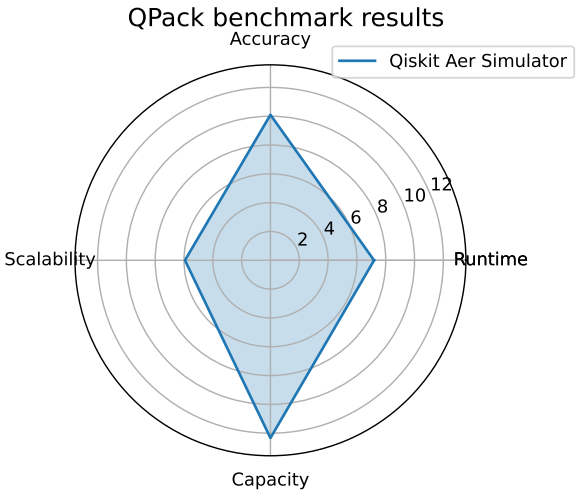
Now that a mapped and balanced subscore has been defined for each problem, the subscores for each problem need to be combined. The combined subscore S_C for each performance category $C \in \{\text{runtime}, \text{accuracy}, \text{scalability}, \text{capacity}\}$ is computed as the arithmetic mean over all problems:

$$S_C = \frac{1}{n_P} \sum_P S^{\text{mapped}}_{C,P} \tag{7.12}$$

where n_P is the number of problems for which the quantum computer is evaluated. For every quantum computer, the subscores for each problem in the set of problems are computed, after which the combined subscores are derived using Equation 7.12. An example of the scores obtained can be seen in Figure 7.5a. This figure lists all the subscores for each problem, as well as the combined subscore for each category. Presenting the results of the QPack benchmark in this way gives insight into the performance of the quantum computer under test for each type of application circuit.



(a) QPack result for the local Qiskit Aer simulator



(b) Combined score for the local Qiskit Aer simulator

Figure 7.5: QPack benchmark result for the local Qiskit Aer simulator

7.6. Overall score

Computing the subscores for a single quantum computer and displaying them like in Figure 7.5a gives a good insight of the performance of a single computer, but makes comparison between multiple quantum computers cumbersome. A better way to visualize performance differences between quantum

computers can be done with radar plots. Such a plot is shown in Figure 7.5b. Using this visualization, we can provide a single parameter to compare different computing platforms by taking the area of the four-sided region in Figure 7.5b. This defines the overall score as

$$S = \frac{1}{2}(S_{\text{runtime}} + S_{\text{scalability}})(S_{\text{accuracy}} + S_{\text{capacity}}) \tag{7.13}$$

This overall score fulfills the performance proportionality requirement and provides a simple way to compare between different quantum computers. Although this overall score is more abstract than the subscores, the advantages of it are its simplicity and general applicability.

7.7. Synthetic tests

Before using the benchmark scores on actual simulator or hardware data, the definition of the benchmark scores will first be applied to some synthetic datasets to verify that they indeed follow the desired behavior mentioned in Chapter 4. This will be done by considering two test cases, one set that has a linear runtime behavior and a set of differently scaled runtime behaviors.

The first case will analyze the behavior of the QPack scores with three synthetic datasets that scale linearly with quantum runtime as the problem size N grows ($T^{\text{QJob}} \propto N$), but differ in average quantum runtime, i.e., a comparison will be made between slower and faster quantum computers. The sets will also differ in expectation value. The first set, Linear1, will mirror the baseline expectation value and thus will always have a relative error of zero. The next set, Linear2, has a standard offset of 0.75 above the baseline expectation value, and the last set, Linear3, has an increasing relative error rate, which is something we can expect for NISQ-era quantum computers. The synthesized data of the sets can be found in Figure 7.6.

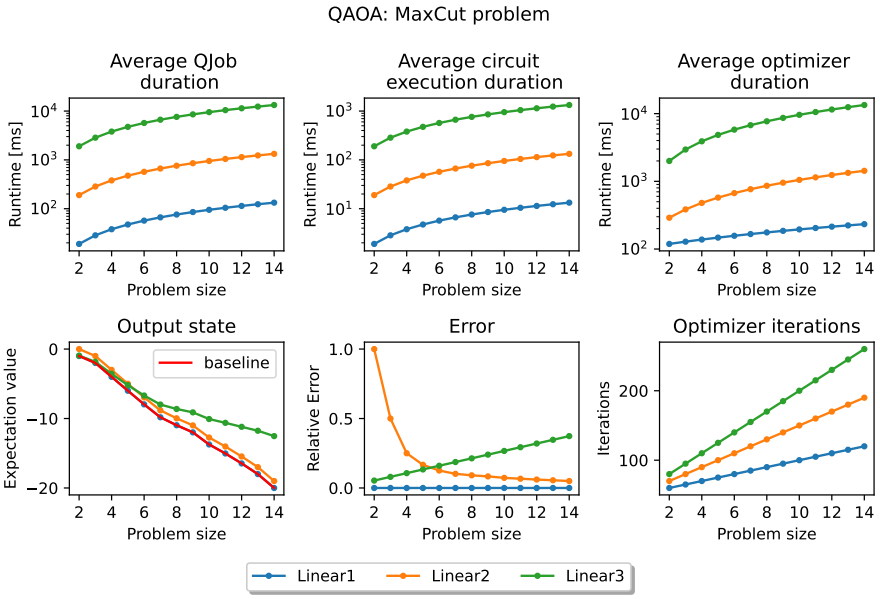


Figure 7.6: Data of the synthetic linear data set

In Figure 7.6, the distinction between the three linear data sets can be clearly seen. Each of them shows identical scaling behavior, offset by their different runtime magnitudes. For the QPack subscores, we thus expect the scalability subscore to be the same, while the runtime score will be the highest for Linear1 and the lowest for Linear3. Analyzing the relative error values for the expectation value, we see that the relative error of Linear2 decreases as the problem size grows, which is to be expected since the absolute error is constant. Linear3, on the other hand, has an increasing error rate as the problem size grows. From this it is expected that Linear3 will perform the worst in the accuracy and capacity subscores. To verify these expectations, let us look at the QPack benchmark results of this synthetic data set in Figure 7.7.

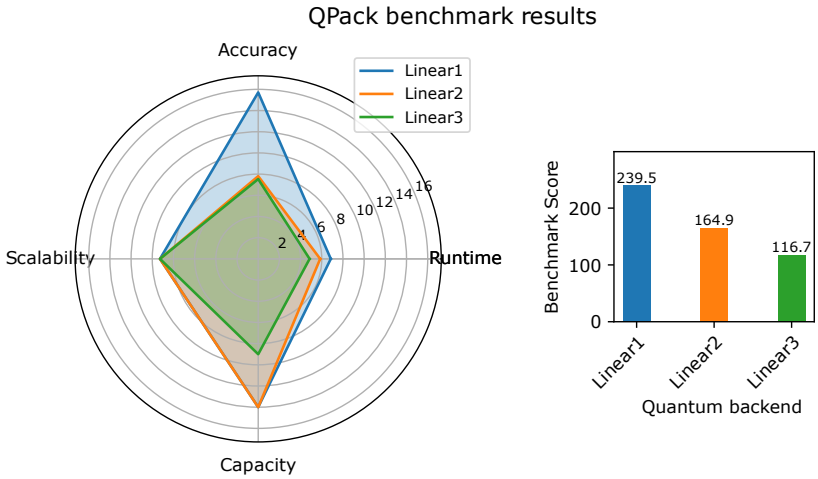


Figure 7.7: Benchmark result of the synthetic linear data set

From the radar plot in Figure 7.7, it can be seen that the Linear1 data set achieves the highest radar area and therefore the highest overall benchmark score. All expectations mentioned earlier are clearly visible in the radar plot, which gives insight into the performance of each subscore category. Note that the Linear2 and Linear3 sets have a similar accuracy score, while their error behavior is very different. One might come to the conclusion that both Linear2 and Linear3 are equally accurate, but when looking at the capacity subscore, it becomes obvious that Linear2 is able to be more accurate over larger problem sizes. This may indicate that Linear3 is more suited for small circuit sizes (5 qubits in this case), where its relative error is small than Linear2, as shown in Figure 7.6.

With data sets of the same scalability verified, we can move to the second test case, where data sets of different runtime scalability are compared. For this, the synthetic data sets Constant1, Quadratic1, and Exponential1 will be compared with the Linear1 data set. These data sets will only differ in the way their runtime scales for increasing problem sizes. The accuracy of these sets will be considered perfect, i.e., a relative error of 0. Constant1 will simulate a quantum computer with constant runtime ($T^{\text{QJob}} \propto C$), Quadratic1 will have a quadratic increase in runtime ($T^{\text{QJob}} \propto N^2$), and Exponential1 has an exponential increase in runtime ($T^{\text{QJob}} \propto 2^N$). Plots of these data sets can be found in Figure 7.8.

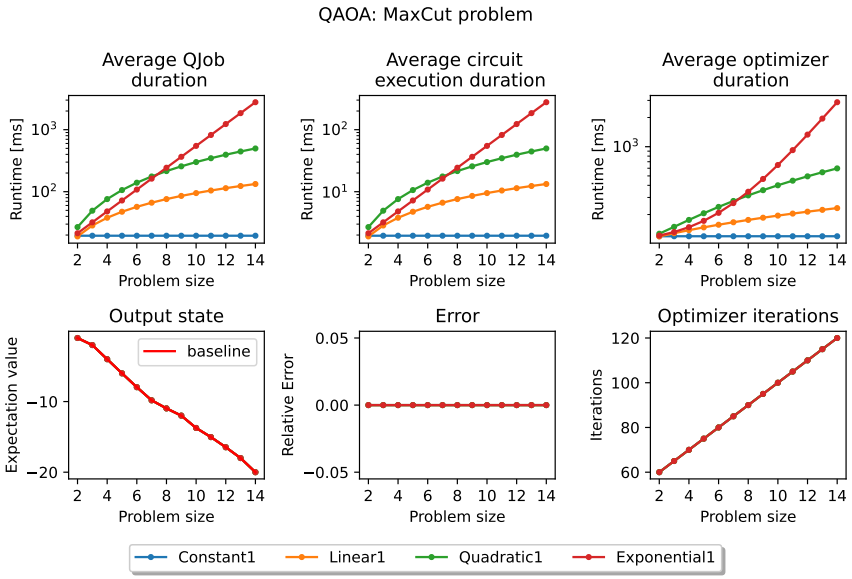


Figure 7.8: Data of the synthetic data sets

From this figure, we can indeed see that the runtimes are quite different from each other. Exponential1 seems like a reasonable choice over Quadratic1 for small problem sizes, but rapidly increases after a problem size of 8. For the benchmark scores, we expect all data sets to achieve the same accuracy and scalability score, as they all have the same expectation value as the baseline expectation value. For scalability, one would expect the Constant1 data set to be performing best, as it's runtime does not increase with the problem size. However, since the scalability score measures the curvature of the runtime trend, it is actually expected to have the same scalability score as the Linear1 data set. Exponential1 is expected to have the worst scalability performance, closely followed by the Quadratic1 data set. For the runtime subscore, Constant1 is expected to have the highest score, and the Linear1 data set is expected to have the second highest score. The difference in the runtime subscore between the Quadratic1 and Exponential1 should be minor, as they overlap at about halfway in the problem size set. The results of the QPack benchmark for these four synthetic sets are shown in Figure 7.9.

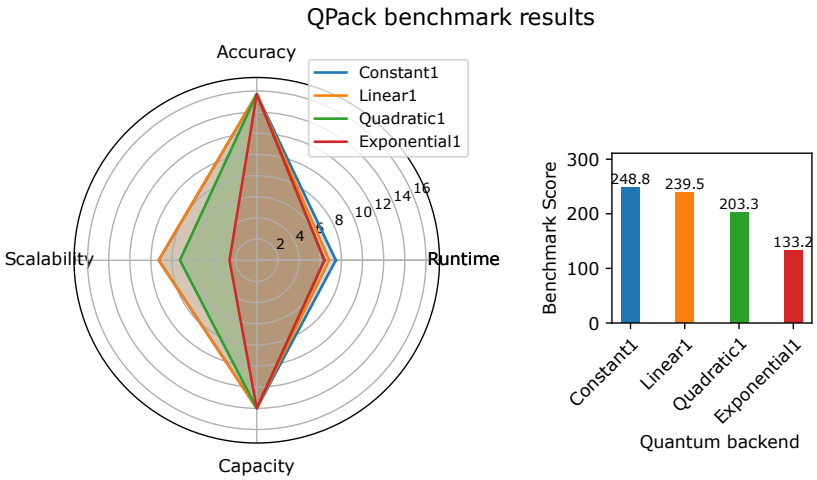


Figure 7.9: Benchmark result of the synthetic data sets

Again, expectations of the second test case are confirmed by the QPack benchmark result. Constant1 outperforms Linear1 solely on its runtime subscore. Quadratic1 and Exponential1 have a similar runtime score, but Exponential1 has a significantly lower scalability subscore, thus giving it an overall lower performance score than all other data sets.

With the QPack scoring system behaving as expected on these synthetic data sets, it can now be tested on some real quantum execution data from a selection of simulators and actual quantum hardware.

8

Benchmark results

The proposed benchmark scores can now be applied to various quantum simulators and actual quantum hardware. This section shows the results for simulators that can be installed on a local machine, cloud-accessible simulators, and cloud-accessible quantum hardware. The final subscores and overall scores of the backends under test are shown. A complete overview of the quantum backends that have been tested can be found in Appendix C and their benchmarking data is listed in Appendix D.

8.1. Local simulators

The first real test case for the QPack benchmark in |Lib⟩ and the QPack scores are some commonly available quantum computer simulators. These simulators are publicly available and can simply be installed on client desktop computers. This allows anyone to experiment with quantum computation on their own machine and provides opportunities for general quantum computing exploration.

8.1.1. Backends under test

The local simulators tested in this work are the Qiskit Aer [120], Cirq [126], Rigetti QVM [127] and QuEST [114] simulators. Simulators are run on a Windows 10 desktop computer, using an AMD Ryzen 5 3600 6-core CPU [128] with 16 GB RAM in an Ubuntu Windows Subsystem for Linux environment.

Qiskit Aer

The Qiskit Aer simulator [120] is part of the Python-based Qiskit project [129], superseding their previous QASM simulator. Aer is a high-performance simulator for quantum circuits that includes noise models to simulate IBMQ backends of different quantum processors and allows GPU-accelerated computing. For this test case, the Aer simulator (version 0.10.2) is working in a basic setup, i.e., no noise models are applied and GPU-acceleration is disabled.

Cirq

The Cirq Python library [130] comes with its own simulators, a pure state simulator, and a mixed state simulator, as well as an interface for external simulators, such as the qsim [131] and qFlex [132] simulators. For these results, the pure state simulator is used without noise models from version 0.14.0.

Rigetti QVM

The Rigetti Quantum Virtual Machine (QVM) [127, 133] is part of the PyQuil Python library [134, 135], for quantum programming using Quil [133]. The QVM is a classical implementation of the Quantum Abstract Machine (QAM), which is Rigetti's general-purpose representation of a quantum computer. To run a quantum program on the QVM, the program needs to be compiled to fit a specified QVM architecture using the Quilc compiler [133]. After compilation, the quantum program can be executed on the QVM. The Quilc compiler and QVM can either be run as local servers or a remote QVM can be accessed using Rigetti's Quantum Cloud Services (QCS) [136]. For this test case, both Quilc (version 1.23.0) and QVM (version 1.17.1) are run in local mode as a pure state simulator.

QuEST

The Quantum Exact Simulation Toolkit (QuEST) is a high-performance C++ based simulator of quantum circuits, state vectors and density matrices. QuEST promises high performance over competitive

quantum computer simulators by using multithreading and GPU acceleration to run lightning fast on laptops, desktops, and networked supercomputers [114, 137]. To compare local simulators fairly, no multithreading or GPU acceleration is enabled. We use version 3.5.0 in this test case.

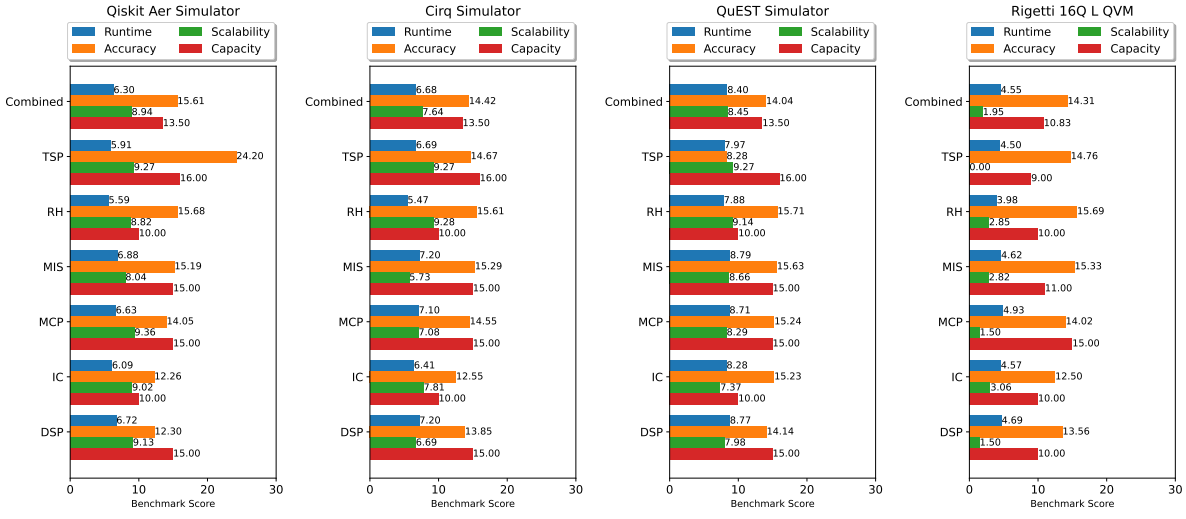
8.1.2. Backend analysis

Before analyzing the QPack benchmark results of each of the tested local simulators, the set of QPack applications is evaluated. In Table 8.1, an overview of all problem sizes is given for which benchmark data is collected. Due to a bug in |Lib⟩, most circuits were unable to be evaluated beyond 15 qubits. This is not that much of an issue, as we can still make an analysis for this limited data set. Moreover, note that the Rigetti QVM was not always able to reach a problem size as large as the other simulators. This is because the compiler and simulator had a timeout period of 5 minutes, which caused the QVM to not be able to complete higher problem sizes. Appendix D.1 lists the complete collected data sets.

	MCP		DSP		MIS		TSP		RH		IC	
Problem size	Start	End	Start	End	Start	End	Start	End	Start	End	Start	End
Qiskit Aer simulator	2	15	2	10	3	12	3	4	1	10	1	10
Cirq simulator	2	15	2	10	3	12	3	4	1	10	1	10
QuEST simulator	2	15	2	10	3	12	3	4	1	10	1	10
Rigetti QVM	2	15	2	5	3	8	3	3	1	10	1	10

Table 8.1: Overview of collected data sets for each tested local simulator

With these problem sizes in mind, we can now put the benchmark results in a better perspective. The QPack results of the local simulators can be found in Figure 8.1. Using these figures, we can see how well each simulator performed on an application, without having to compare all individual data sets.



(a) Qiskit Aer simulator benchmarking scores

(b) Cirq simulator benchmarking scores

(c) QuEST simulator benchmarking scores

(d) Rigetti 16-qubit QVM benchmarking scores

Figure 8.1: QPack benchmarking scores for tested local simulators

All simulators seem to have a similar runtime subscore behavior for each problem, where the TSP, RH, and IC typically achieve lower scores and the MCP, DSP, and MIS problems score on the high end. This is to be expected, of course, since all simulators can perform all common quantum operations between all qubits, i.e. they are not limited to working within a basic gate set and do not need to swap around qubits to perform the necessary quantum operations. All simulators seem to achieve the lowest runtime score for the RH problem. This could be explained by the fact that this is the smallest quantum circuit that is evaluated and is thus sensitive to any overhead that may exist within the simulation. For all simulators, they achieve a fairly similar score for each problem, except for the TSP problem, where the Aer simulator achieves its highest accuracy score and the QuEST simulator achieves its lowest. Since all simulators operate in a noiseless mode, these scores better reflect the ability of the classical optimizer

to find the VQA solution using the simulated quantum output than the noise in the system. Scalability behaves fairly constant for the Aer and QuEST simulators, as the scalability subscores for each problem do not deviate much from their combined scores. For the Cirq simulator, the RH and TSP scale better than the other problems. The Rigetti QVM shows some interesting scalability results. As it was only able to evaluate TSP for one data point, it achieves a scalability score of zero. Also, MCP and DSP do not scale as well as the other problems for the QVM. As expected, all simulators are able to achieve their maximum capacity scores, as the capacity threshold value was set to give this result by design.

8.1.3. Backend comparison

Using the radar plot in Figure 8.2, a comparison between the tested local simulators can be performed. The bar graph on the right shows the overall score for each simulator. Here, we see that QPack ranks the QuEST simulator as the best performing simulator, and the Rigetti QVM as the worst simulator.

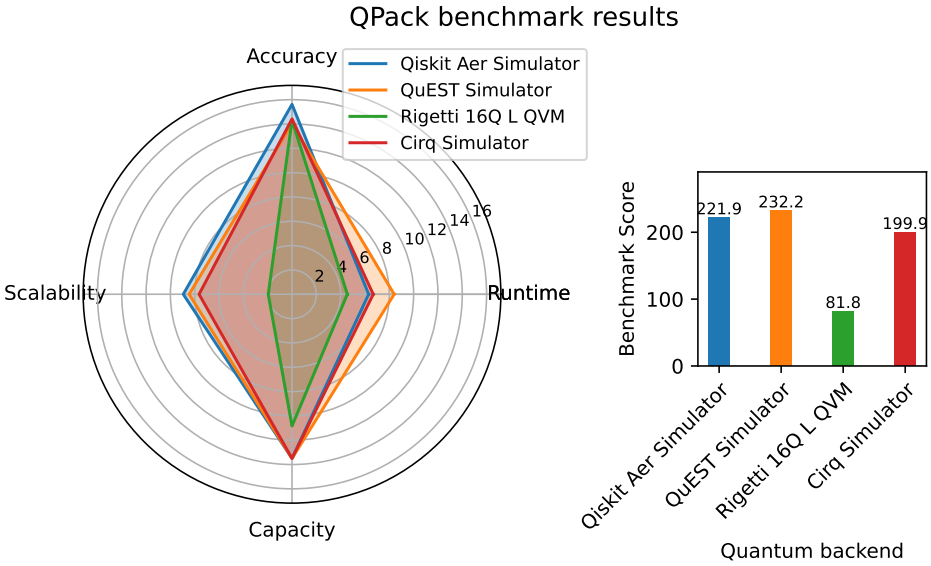


Figure 8.2: QPack benchmark result for local simulators

For the runtime subscore, the QuEST performance is the best, which was expected as it is a C++ based simulator. Aer and Cirq have a very similar runtime behavior, which could be explained because they both use a Python based internal simulator in their software. The Rigetti QVM is the slowest simulator of the tested units, indicating that running the Quilc and QVM software in a standalone server mode on a client may not be an efficient setup or that the programs are inherently inefficient. Accuracy scores are very similar for all simulators, which can be expected in noiseless operation. The Aer simulator was able to achieve the highest accuracy subscore, mainly due to the fact that it scored very well on accuracy for the TSP application. The scalability subscore is more interesting to analyze, as this reflects the simulator's efficiency at which it can simulate larger quantum systems. This makes it seem that the Aer simulator would be more suited for larger quantum systems, followed closely by the QuEST and Cirq simulators. Although the capacity score for a noiseless simulator would initially not be very interesting to evaluate, here we see that there is a difference between the Rigetti QVM and the other simulators. This has not so much to do with the output of the simulator, but rather with the ability to simulate a quantum circuit of a certain capacity. For the Rigetti, this is around 10 qubits (limited by time-outs), while for the others, this is around 14.

8.2. Cloud-accessible simulators

If local quantum simulation is not preferred, many providers allow cloud-access to their simulators, such as IBMQ, IonQ, and Rigetti. These simulators are often designed to test quantum applications before executing them on real hardware, as the use of quantum hardware is usually charged per time slot or per number of gates and shots. They can also be an option if a large simulation is required, as these simulators usually operate on high-performance computers rather than a personal computer.

8.2.1. Backends under test

In this work, a comparison is made between the remote IBMQ QASM simulator [49], the IonQ simulator [138], and the Rigetti QVM [127, 133].

IBMQ QASM simulator

The remote IBMQ QASM simulator (version 0.1.547) is a general-purpose simulator that offers both ideal and noisy simulations. It is able to simulate up to 32 qubits and supports a wide set of quantum operations [49]. For this test case, the QASM simulator is run without noise models. IBMQ uses a fair-share policy, which means that jobs are queued and have to wait for other clients to finish their jobs. To obtain a fair measurement of T^{QJob}, the queue time is subtracted from the quantum job time.

IonQ simulator

IonQ offers a free-access quantum simulator that is accessible through its API (version 0.3) [138]. The simulator supports circuits of up to 29 qubits and utilizes the same set of gates that are available on the IonQ trapped ion hardware [139], but without noise models [140].

Rigetti QVM

The Rigetti QVM is the same simulator as described in Section 8.1, but rather than running services locally, QVM (version 1.17.2) and Quilc (version 1.26.0) are run on the Rigetti server in pure state mode.

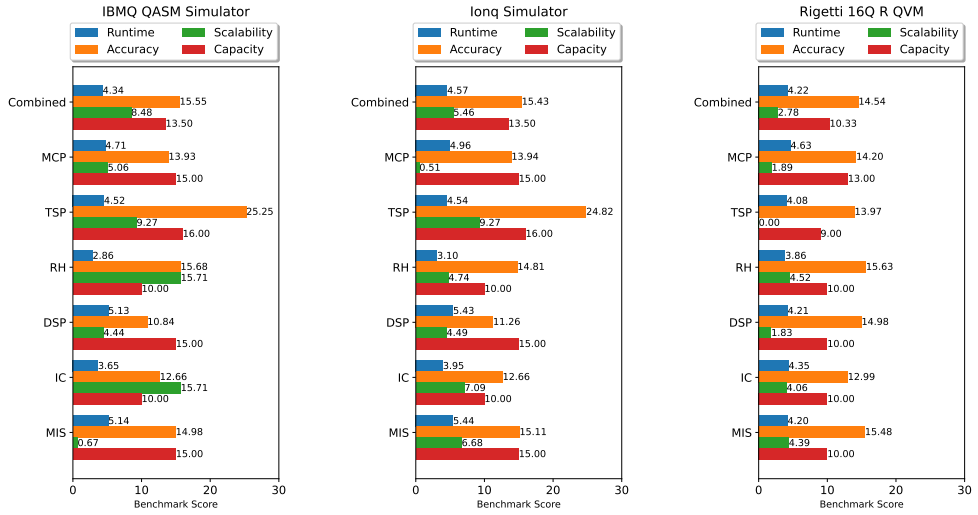
8.2.2. Backend analysis

For the remote simulators, a set of problem sizes similar to those of local simulators was used for benchmarking. In Table 8.2, an overview of all problem sizes is given for which benchmarking data is collected (see Appendix D.2). Due to similar reasons as for the local simulators, the Rigetti QVM was not able to achieve problem sizes as large as the other simulators, due to time-out constraints.

	MCP		DSP		MIS		TSP		RH		IC	
Problem size	Start	End	Start	End	Start	End	Start	End	Start	End	Start	End
IBMQ QASM simulator	2	15	2	10	3	12	3	4	1	10	1	10
IonQ simulator	2	15	2	10	3	12	3	4	1	10	1	10
Rigetti QVM	2	13	2	5	3	7	3	3	1	10	1	10

Table 8.2: Overview of collected data sets for each remote simulator under test for each QPack problem

With this setup, the QPack benchmark evaluated each remote simulator; see the results in Figure 8.3.



(a) IBMQ QASM simulator benchmarking scores

(b) IonQ simulator benchmarking scores

(c) Rigetti QVM remote simulator benchmarking scores

Figure 8.3: QPack benchmarking scores for tested remote simulators

The IBMQ QASM simulator seems to achieve higher runtime subscores for applications with large circuits (DSP, MIS) and low scores for smaller circuits (RH, IC), but its scalability subscores show the opposite behavior. This indicates that the QASM simulator has a lot of overhead when running small VQE circuits, but promises good runtime behavior as these problems grow. The larger QAOA circuits score much lower in scalability, indicating that the computational load to evaluate the circuit is a more dominant factor in runtime. The IonQ simulator also has better runtime scores for the larger QAOA circuits than the VQE problems, but its scalability scores do not seem to have a particular correlation with larger or smaller circuits. The Rigetti remote QVM has a fairly constant runtime score for all problems, but seems to be better suited for smaller circuits based on its scalability score. Notice that for the TSP it has a scalability score of 0, due to the fact that it was only able to perform the problems for problem size 3. For the accuracy subscore, all remote simulators perform similarly, which is to be expected for noiseless simulators. An interesting peak is the accuracy subscore of the IBMQ and IonQ simulators for the TSP. They were able to achieve an expectation score lower than the baseline value, obtaining a high score. All simulators were also able to achieve their maximum obtainable capacity score. Although this could be expected for noiseless simulators, keep in mind that the threshold value A^* was only set based on the results of local simulators. The fact that all remote simulators were able to stay within this threshold is a sign that the current A^* is not too strict.

8.2.3. Backend comparison

With the individual analysis of the remote simulators completed, a comparison between the tested backends can be made. To do so, we will use the QPack results in Figure 8.4. For the bar chart, we see that QPack ranks the IBMQ QASM simulator as the best performing simulator, followed by the IonQ simulator, and ranks the remote Rigetti QVM as the worst performing simulator.

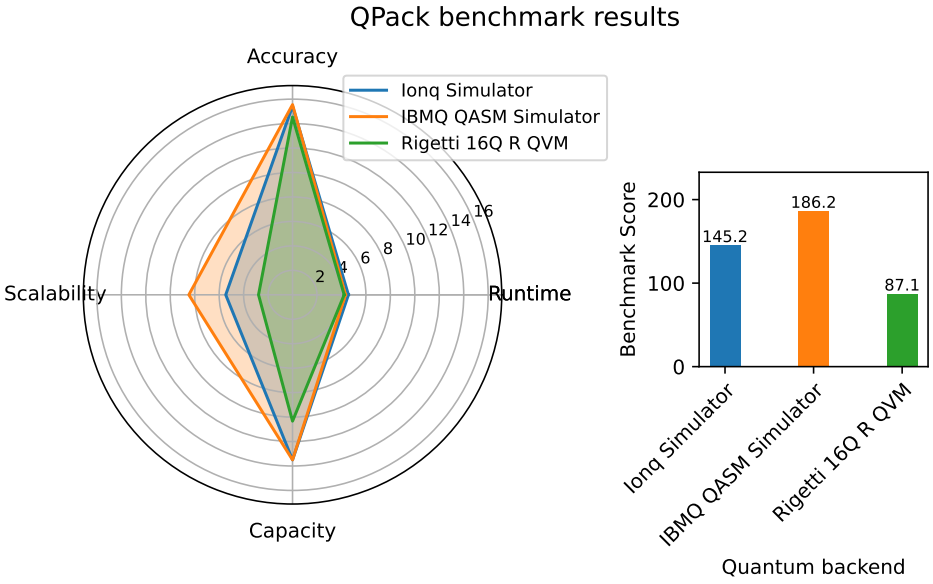


Figure 8.4: QPack benchmark result for remote simulators

From the radar chart, it can be seen that all three simulators score similarly in runtime and accuracy. That the simulators score similarly in accuracy is expected, as they all operate in a noiseless mode. The runtime scores, on the other hand, seem to be very close to each other as well, although during benchmarking runs, differences in runtime were clearly noticeable (see Appendix D.2). This can be explained by looking at the scalability score, where the scores have a more noticeable distinction. The IBMQ QASM simulator is the best scalable simulator of the three and the Rigetti QVM the worst. Similarly to what has been observed in the synthetic tests in Section 7.7, we can conclude that it has better runtime performance for small problem sizes, but deteriorates more rapid than the other simulators as the problem size grows. In terms of capacity, all simulators were able to reach their maximum available capacity score. The Rigetti QVM could not score as high, since it was unable to complete larger problem sets. In conclusion, the IBMQ QASM simulator would be the simulator of choice for all-purpose

quantum circuits. However, the IonQ simulator could be a faster option for circuits with of modes size ($\pm$15 qubits). For even smaller circuit sizes ($\pm$ 5 qubits) or very shallow circuits, the Rigetti QVM is still a valid option.

8.3. Cloud-accessible hardware

With the QPack benchmark tested on quantum computer simulators, we now proceed to evaluate quantum hardware using the QPack benchmark. Evaluating the performance of these hardware quantum computers will not only allow for a comparison between tested quantum computers, but will also shed light on the current state of quantum computing.

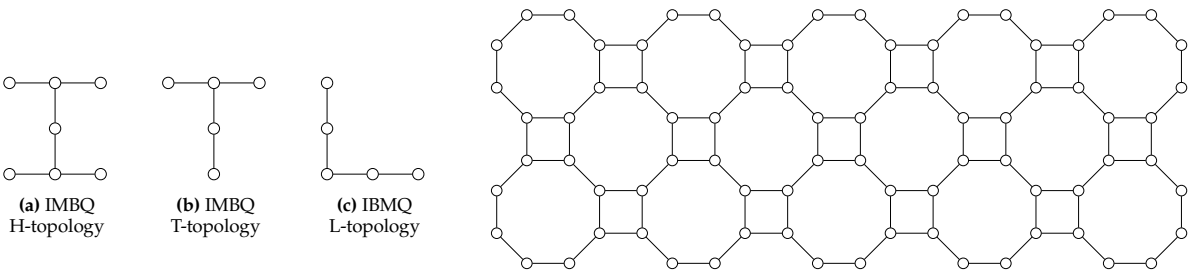
8.3.1. Backends under test

We will evaluate quantum hardware from the IBMQ family [141] and Rigetti [9]. In contrast to the benchmarks run on simulators, hardware-based quantum computers need to transpile the provided QPack circuit to fit to their native gate set and qubit topology. This becomes an important factor for all subscores, such as gate decomposition, state swapping, and scheduling, increases the total number of qubit operations and, therefore, introduces more noise in the system. An overview of the backends under test can be found in Table 8.3, listing some of their low-level characteristics. A visualization of the qubit topologies of these backends are shown in Figure 8.5.

Hardware	Qubits	Ave T1 [μs]	Ave T2 [μs]	Type	Topology	Native gates
IBMQ Quito	5	121.4	111.3	Superconducting	T (Fig 8.5b)	CX, ID, RZ, SX, X
IBMQ Manila	5	181.6	57.0	Superconducting	L (Fig 8.5c)	CX, ID, RZ, SX, X
IBMQ Nairobi	7	133.9	94.9	Superconducting	H (Fig 8.5a)	CX, ID, RZ, SX, X
IBMQ Perth	7	171.2	128.2	Superconducting	H (Fig 8.5a)	CX, ID, RZ, SX, X
IBMQ Lagos	7	139.2	89.6	Superconducting	H (Fig 8.5a)	CX, ID, RZ, SX, X
IBMQ Jakarta	7	148.5	41.4	Superconducting	H (Fig 8.5a)	CX, ID, RZ, SX, X
Rigetti Aspen-M-1	80	31.2	23.1	Superconducting	Octagonal (Fig 8.5d)	RX, RZ, CPHASE, CZ, XY

Table 8.3: Hardware-based quantum computers [50, 136] tested by QPack

As becomes clear from Table 8.3, all quantum computers tested are built using superconducting qubit technology. Superconducting qubits use supercooled Josephson junctions [142, 143] to create a quantum nonharmonic oscillator which is used as the qubit. Interestingly, the relaxation (T1) and decoherence (T2) times seem to be much lower for the Rigetti system than for the IBMQ systems. This could mean that the Rigetti computer is not able to perform well on quantum circuits with a large depth, without a type of error-correction scheme. However, the Aspen-M-1 has a lot more qubits to operate on, with its 80 available qubits compared to the 5- and 7-qubit IBMQ systems. Accessing the Rigetti systems is done on a pay-per-use basis, and we thank Oak Ridge for the funds to run our experiments on this state-of-the-art quantum computer. It should be noted that IBMQ does offer larger quantum computers as well, with its 127-qubit Eagle processor being their current flagship quantum computer. These systems are generally not accessible by the public, so we also express our gratitude to IBMQ for giving free access to their 5- and 7-qubit quantum backends.



(a) IMBQ H-topology **(b)** IMBQ T-topology **(c)** IBMQ L-topology

(d) Rigetti Aspen-M-1 topology

Figure 8.5: Qubit typologies of tested quantum hardwares

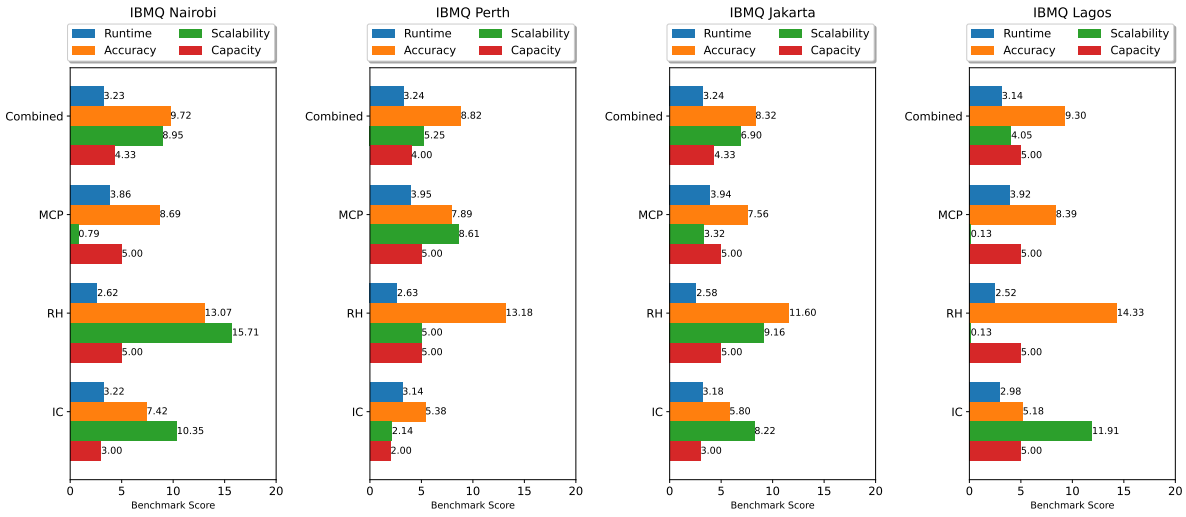
8.3.2. Backend analysis

The tested quantum hardware computers can be found in Table 8.4. Due to limited access to these backends, only the MCP, RH, and IC problems were evaluated, as they were able to run for small problem sizes in a relatively fast time compared to the other QPack problems. We also limit the benchmark execution to one VQA repetition per problem to save time. This is also why not all quantum computers have been evaluated to their maximum number of qubits. Especially the Rigetti Aspen-M-1 has a small data set, because the available funding allowed access for not more than one hour. The complete data sets are found in Appendix D.3. Analyzing all quantum computers at once can become a bit convoluted, so the analysis of the hardware is split into two parts. The first part covers the analysis of all 7-qubit IBMQ backends with an H-topology and is followed up in the second part with an analysis of the other hardwares with a different topology. After individual analysis, a comparison between H-topology quantum computer is made. The best scoring hardware in this set is then compared to the hardwares with different topologies.

	MCP		RH		IC	
Problem size	Start	End	Start	End	Start	End
IBMQ Manila (5-qubits)	2	5	1	2	-	-
IBMQ Quito (5-qubits)	2	5	1	5	1	4
IBMQ Nairobi (7-qubits)	2	7	1	7	1	7
IBMQ Perth (7-qubits)	2	5	1	5	1	5
IBMQ Jakarta (7-qubits)	2	5	1	5	1	5
IBMQ Lagos (7-qubits)	2	5	1	5	1	5
Rigetti Aspen-M-1 (80-qubits)	-	-	1	4	-	-

Table 8.4: Overiew of collected data sets for each backend under test for each QPack problem

The backends with the H-topology are those who use the Nairobi, Perth, Jakarta, and Lagos quantum processors from the IBMQ aviary. Their individual results are depicted in Figure 8.6. Due to limited access to IBMQ backends, not all backends have been benchmarked to their maximum qubit capacity. This is why, for a fair comparison, the Nairobi backend will only be evaluated to 5 qubits, to align it with the other backends.



(a) IBMQ Nairobi benchmarking scores

(b) IBMQ Perth benchmarking scores

(c) IBMQ Jakarta benchmarking scores

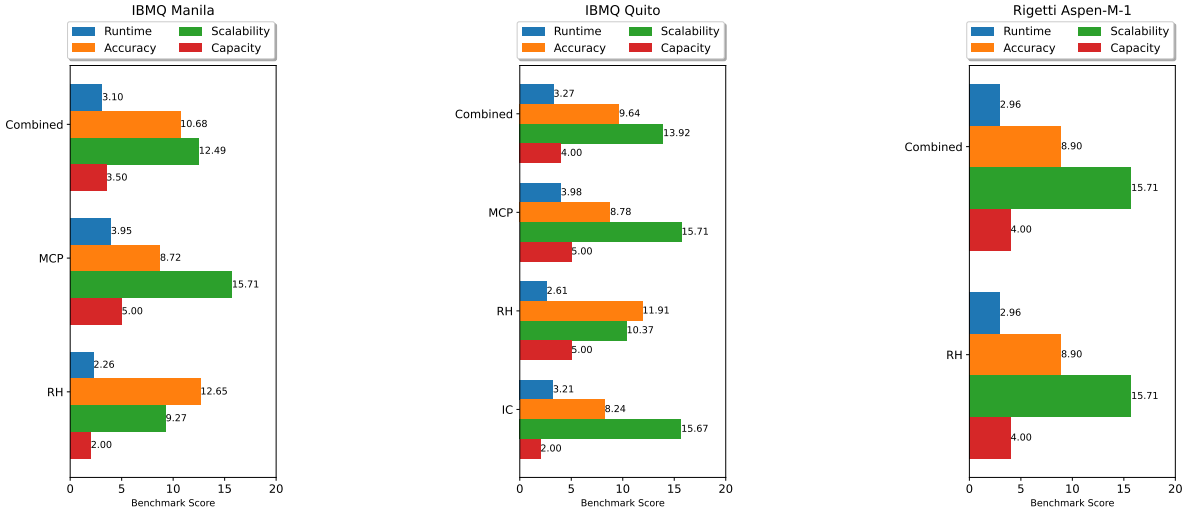
(d) IBMQ Lagos benchmarking scores

Figure 8.6: QPack benchmarking scores for tested remote quantum hardware with H-topology

Since these backends all use the same topology, we can see a lot of similarities in their benchmarking performance. They all see to behave similarly in runtime, achieving best runtime scores for MCP, then IC, and finally RH. As stated in Section 7.1, current quantum computers suffer from a large overhead factor in QJob time, and it again shows in the benchmark results, because the runtime score compensates for the depth of the circuit. Similar behavior is also observed in each quantum backend for the accuracy subscore, with RH ranking highest, MCP middle, and IC ranking lowest. Different behav-

ior in scalability can be observed, where RH is the highest scoring problem for the Nairobi and Jakarta backends, MCP scales best for the Perth backend and IC best for the Lagos backend. All backends reach their maximum capacity scores for all problems, apart from the IC problem, where the Perth backend is usable for two qubits, the Nairobi and Jakarta backends are usable for three qubits and the Lagos backend for five qubits.

Next, the backends with the IBMQ L- and T-topology and the Rigetti octagonal topology are analyzed. These are the IBMQ Quito, IMBQ Manila, and the Rigetti Aspen-M-1 backends.



(a) IBMQ Manila benchmarking scores

(b) IBMQ Quito benchmarking scores

(c) Rigetti Aspen-M-1 benchmarking scores

Figure 8.7: QPack benchmarking scores for tested remote quantum hardware with other topologies

Now for these backends, their performance estimate is very rough. Only the Quito backend has been evaluated for all three selected problems. Unfortunately, limited reservation times and long queue times only allowed to collect a small data set of RH data for the Aspen-M-1 and only part of the RH data set for the IBMQ Manila. The small amount of data can still give some indication on their performance in runtime, accuracy, and capacity, but the scalability subscore is not yet usable, as it will require more data points to fit this subscore metric. We see that the Manila and Quito backends obtain higher runtime scores for the large MCP than for the smaller RH problems, which could again be an indication of large overhead in the quantum computing stack. The RH application, however, is able to obtain a high accuracy subscore compared to the other problem applications. As with the H-topology backends, all backends are able to achieve their maximum capacity scores for all problems except for IC.

8.3.3. Backend comparison

With the insight into the individual benchmark results completed, the backends can be compared with each other. Again, this will be split into two parts, where the H-topology backends are compared, after which the best scoring one is compared to the other topologies. A comparison between the Nairobi, Perth, Lagos, and Jakarta backends can be found in Figure 8.8. Of the tested backends for the case of maximum 5 qubits, the IBMQ Nairobi is the best performing.

All backends perform similarly in terms of runtime, which is to be expected from the same topology backends with the same native gate set. The accuracy subscores show some differences, where the order of best to worst scoring backend is the Nairobi, Lagos, Perth, and finally Jakarta. These differences could partly be explained by the small data set and the performance of the classical optimizer, but could also rise from the fact that the backends have different decoherence times. For example, the Jakarta backend has an average T2 time of 41.4 μs, while the Nairobi has an average T2 time of 94.9 μs. This indicates that the Jakarta's qubits would trail off of their desired states faster than the Nairobi's qubits, resulting in a noisier output. However, the Perth backend has an average T2 time of 128.2 μs

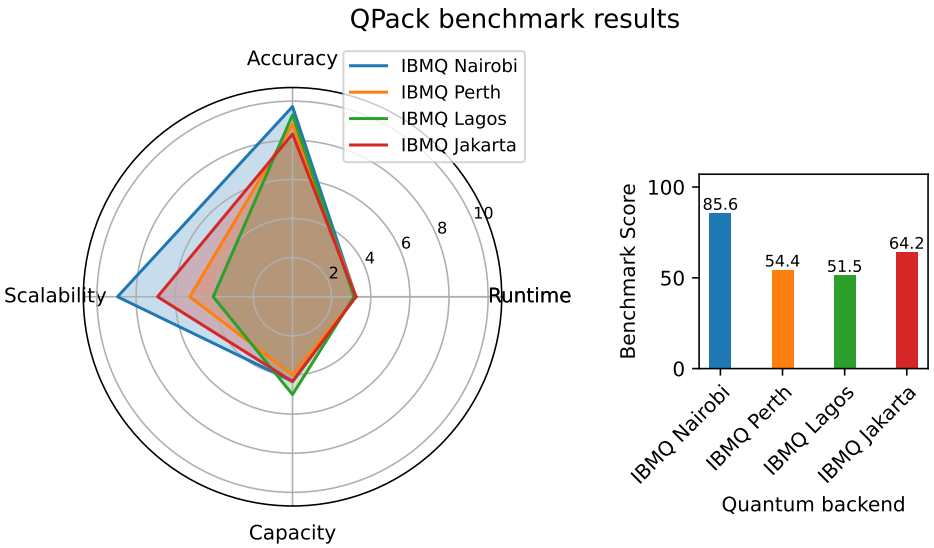


Figure 8.8: QPack benchmark result for IBMQ H-topology backends

and does not achieve the highest accuracy score. This highlights the importance of holistic benchmarking, as we observe that the behavior of a quantum computer cannot be properly understood only by its low-level performance metrics. Larger differences are observed for the scalability subscore, which is the main contributor to the differences in the overall score. For the limited 5-qubit case, the Nairobi backend has the best scalability, followed by the Jakarta, Perth, and Lagos backends, respectively. The capacity subscore is close for all backends as mentioned in their individual analysis, where the Nairobi and Jakarta backends are usable for 4 to 5 qubits and the Perth and Lagos backends for around 4 qubits, depending on the type of application.

Now we compare the other topology backends with the best performing H-topology backend; the IBMQ Nairobi. The overall benchmark results for the IBMQ Nairobi, IMBQ Manila, IBMQ Quito and Rigetti Aspen-M-1 are depicted in Figure 8.9.

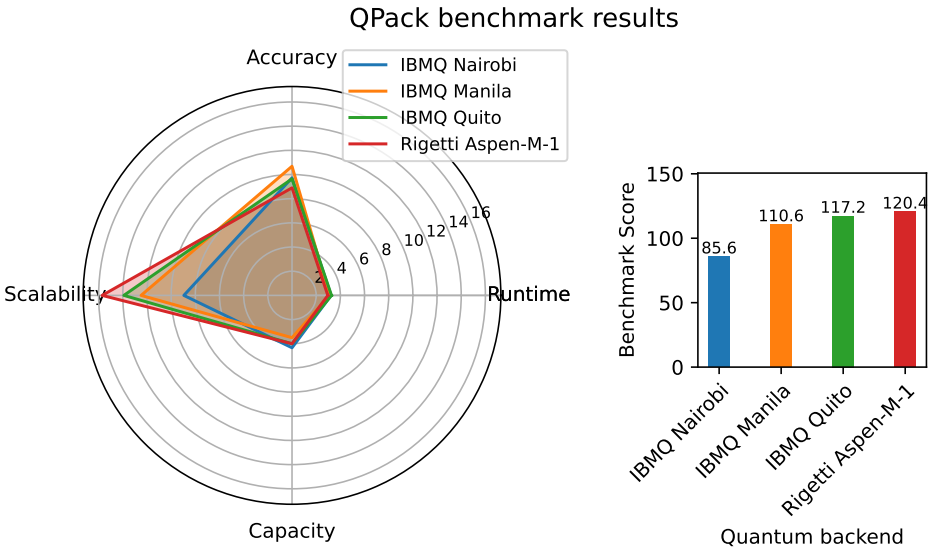


Figure 8.9: QPack benchmark result for quantum hardware with different topologies

Again, no large differences can be observed for the runtime subscores, where the IBMQ Quito obtains a slightly higher score over the other backends. The Aspen-M-1 could potentially be faster, as it achieved

the highest runtime score for the RH problem, but was not evaluated for the MCP and IC problem where higher runtime scores are usually obtained. The differences in accuracy are more apparent. However, only the Quito and Nairobi can be fairly compared, where the Nairobi backend takes the slight edge over the Quito. It looks like the Manila is more accurate than the other backends, but remember that it has not been evaluated for the IC problem, where lower accuracy scores are generally obtained by other backends. The Aspen-M-1 accuracy subscore is also interesting to analyze, since it has only been evaluated for the RH problem and yet it obtains the lowest accuracy subscore. Since backends usually score very well on the accuracy subscore on the RH problem, which would indicate that the Aspen-M-1 is less accurate in general compared to the IBMQ backends. Large differences in the scalability subscore are also clearly observable. The Aspen-M-1 has the highest scalability subscore, but keep in mind that this has only been evaluated for the RH problem. The IMBQ Nairobi performs worse than the other backends, which may indicate that the H-topology is not as scalable as the other available topologies. For the capacity subscore, the IMBQ Nairobi performs best, but it could be that the IMBQ Manila could achieve similar performance results as it has not been pushed to its maximum qubit capacity in the benchmark. The same goes for the Aspen-M-1, although it will probably not be able to achieve a maximum capacity score when evaluated for the MCP and IC problems, based on its accuracy score.

Based on the QPack benchmark result, the IMBQ Quito and Nairobi are good choices for circuits up to 5 qubits, as scalability is not a very important factor here since both backends can only run small circuits. For circuits of 6 and 7 qubits, the IBMQ Nairobi would be best to use. The IBMQ Manila also achieves high scores, but it is not sure if these scores remain as high if the complete data set has been run. The same can be said for the Rigetti Aspen-M-1, as it theoretically would be capable of running circuits much larger than the other tested backends. The question remains whether it is capable of producing usable results, as its accuracy performance on the small RH application did not perform as well as the other backends. To answer this question, more access to quantum computers is needed so that we can run more benchmark applications for larger problem sizes and take multiple measurements.

8.3.4. Noisy hardware simulation

As mentioned in the beginning of this section, only a small data set of quantum hardware could be collected for a single VQA repetition. In order to collect more data on quantum computers, more access is required from quantum computer providers. However, until this is provided, one might be inclined not to use the actual quantum hardware at all, but rather to use a simulator with a noise model and base gate set of the desired quantum computer to get an indication of performance. To see whether this approach is viable, a comparison is made between the noiseless Aer local simulator, the Qiskit Nairobi local simulator, and the actual Nairobi Quantum computer for the MCP, RH, and IC problems, for a maximum problem size of 7. The QPack benchmark result can be found in Figure 8.10.

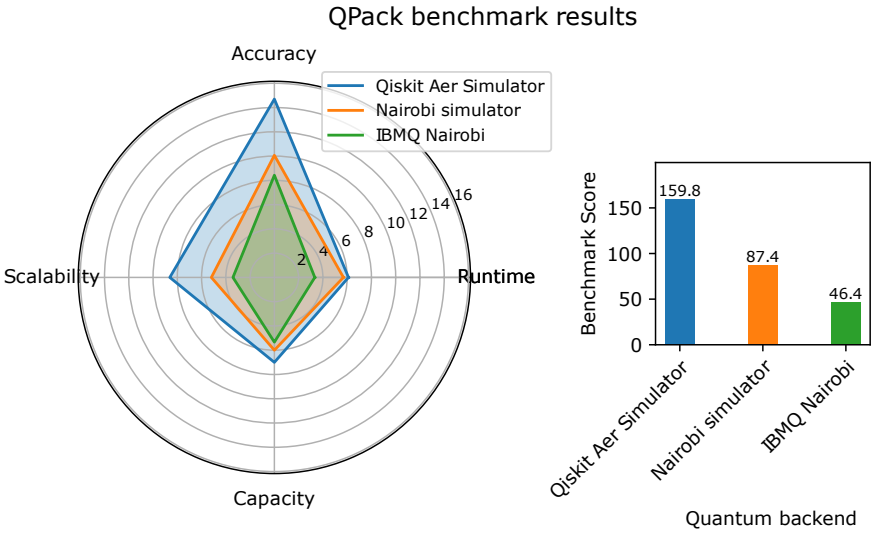


Figure 8.10: Comparison between noiseless simulation, simulation with Nairobi noise model and hardware Nairobi backend

The Aer simulator is obviously expected to have the best performance of the three, but will serve as a reference for the noisy Nairobi simulator. Even tough simulation of quantum hardware is meant as a tool for circuit debugging, we can observe significant differences between a simulator and the hardware implementation.

Instead analyzing each backend sequentially, let us directly go to the comparison between the simulators and the Nairobi backend. As seen in Figure 8.10, the Aer simulator easily outperforms both Nairobi implementations. The Nairobi simulator has a runtime score close to that of the Aer simulator, which is not that surprising, as it is running on the same classical hardware and needs to evaluate a slightly more complex circuit. More interesting is the comparison between the simulated and physical implementation of the Nairobi processor. It can be seen that the simulator outperforms the hardware in this case in all subscores. This could be expected for runtime, of course, as simulation of quantum systems is often faster than running quantum hardware itself (especially in the NISQ-era). However, the Nairobi simulator is more accurate than the real Nairobi backend. This indicates that simple noise models implemented on today's quantum simulators cannot accurately reflect the actual noise behavior of a real hardware quantum computer. This also causes the Nairobi simulator to have a higher capacity subscore, though they should model the same behavior. The discrepancy between the simulated noisy model and the actual noise of a quantum computer is known, as it is difficult to simulate noisy quantum many-body dynamics [144]. However, it is good to verify this in the QPack context. Also, for this small data set, the Nairobi simulator has a better scalability subscore than the physical Nairobi implementation. For the overall score, QPack ranks the Nairobi simulator almost twice has high. This indicates that, while hardware simulators may be useful for debugging, they say little about the performance for any of the QPack subscores.

Results for all noisy simulators of the tested quantum hardwares in this thesis can be found in Appendix D.4

9 Discussion

This chapter covers the discussion of some important choices that have been made in the design of the QPack benchmark. First, several points of discussion about the QPack scores are highlighted, as well as some scalability limitations. After this, the use of the |Lib⟩ library is reviewed. This is followed up by a comparison of the obtained QPack benchmark results with the QV and CLOPS metric for the tested IBMQ backends. The chapter ends with a reflection on the design criteria of the QPack benchmark.

9.1. QPack scores

The presented results indicate that the QPack benchmark is capable of providing an easy and insightful comparison between quantum computers. However, it should be noted that these scores use some arbitrary values. One of these, for example, is the threshold value A^*, which could be chosen to be more or less tolerant to the achieve relative error. In this implementation, the value of A^* was chosen such that all local simulators without noise models would achieve a maximum capacity score for all benchmarked problem sets to compensate for their nondeterministic behavior. This sets the bar at such a level that a real quantum computer needs to achieve output state close to that of its simulated noiseless counterparts for all benchmark applications. This may be too strict for NISQ-era hardware and more applicable to post-NISQ-era quantum computers. As this threshold is the same for all applications, it does not necessarily reflect the usability of an output state for each specific problem. Another approach could be used, where different threshold levels per application are applied and can be set such that the threshold is the maximum relative error at which the output of the benchmark application is still usable.

The balancing parameters c_0, c_1, c_2, c_3 are also subject to discussion, as changing their values could significantly change the accuracy and scalability subscores. They were chosen such that different scores of the performance categories fall within a similar range, making comparison and visualization more practical. Scores are thus not absolute values, but can give a relative difference between two quantum computers, as long as parameter choices are consistent. Related to this is the combination of the subscores to the overall score. This, again, is a practicality to support the ability of QPack to compare quantum hardware with a single-number performance metric. Here, consistency is the key to a fair comparison between quantum computers.

When scaling up benchmark application sizes, comparing the expectation value of a quantum computer to an ideal simulator becomes increasingly difficult. Currently, the QuEST simulator is used to determine the reference output state used in the accuracy and capacity subscores. From its documentation [137], it is stated that the memory used increased exponentially with the number of qubits, as

$$\text{memory} = b \cdot 2^{\text{qubits}-29} \quad [\text{GiB}] \tag{9.1}$$

where b is the number of bytes used to represent a number, typically $b = 8$ for double precision. This means that to simulate 40 qubits, around 16 TiB of memory is needed! To negate this, other references can be taken to use as a baseline. One option is the theoretical optimal solution for all VQA problems, which can be easily computed, since both the QAOA and the VQE solutions have a well-defined relation with the size of the problem. However, this requires the VQA implementation to be able to find this solution to the problem, which, as seen in this work, is not always the case. This is caused by the

fact that QAOA only approximates valid solutions, and, in a similar manner, the VQE solution depends on the type of ansatz that is used. Another approach is not to use a simulator as a baseline at all, but to use another quantum computer to get a baseline value, and all other benchmarked units achieve a relative score to this baseline quantum computer. This would require a very low-noise quantum computer, and this likely needs to be updated frequently as quantum computers grow in size and this baseline quantum computer becomes redundant over time.

For the runtime and capacity subscores, T^{QJob} is chosen to measure performance rather than the actual execution time of the circuit T^{QE}. As mentioned in Section 7, overhead is currently a major factor and, as such, a more pristine target for quantum job time optimization. However, as quantum computers leave the NISQ era, T^{QE} could change to be the dominant factor in quantum job time. Nevertheless, T^{QJOb} would still be the time duration of choice, as this gives a more holistic view of quantum computing systems and is not dependent on data from quantum providers, but can be timed by QPack itself.

Another impact on the benchmark scores is the set of collected problem data. For example, when comparing a 5- and 7-qubit quantum computer, a larger problem set can be run on the 7-qubit quantum computer. For a fair comparison, both quantum computers should be evaluated on the same problems for the same problem sizes. However, this could punish the 7-qubit quantum computer, as it is theoretically capable of achieving a higher capacity score, but as a result its overall accuracy score possibly decreases. This may be a problem, especially for NISQ-era quantum computers, as differences between running problem sizes that only differ by a small amount can have a notable impact on the benchmark performance. As quantum computers become larger, this problem will probably become less prominent.

9.2. |Lib⟩ limitations

Using |Lib⟩ as a cross-platform library has an obvious benefit when benchmarking different quantum computers. However, there are some downsides to the current |Lib⟩ version. For example, a bug persists that does not allow for the creation of large quantum circuits. That is, with a certain number of qubits and/or quantum gates, some memory issues arise when running |Lib⟩. This has led quantum simulators to only evaluate VQAs to a maximum of around 14 to 16 qubits. Due to time and qubit constraints on the tested quantum hardware, this did not cause any limits on performance evaluation. However, as quantum hardware with a larger number of qubits becomes more available, the current version of QPack is not able to evaluate more than 14-16 qubits until this issue has been resolved. Because QPack is built on top of the |Lib⟩ library, in order to benchmark new quantum computers, they first have to be included in |Lib⟩. This means that in order for QPack to be updated, |Lib⟩ has to be updated first. Although this is not a huge barrier to overcome, it does take some extra effort to benchmark new quantum computers.

9.3. Comparison to other benchmarks

It is interesting to see how the QPack benchmark compares with other commonly used benchmarks. For this, we will compare the QPack benchmark scores for the tested IBMQ quantum computer with their own benchmark metrics, the Quantum Volume (QV) and CLOPS metric. The QV relates most to the accuracy and capacity subscores, as the QV evaluates the output of the quantum computer, while the CLOPS metric is more relatable to the runtime subscore. Their scores can be found in Table 9.1. The IMBQ Manila is not included due to limited acquired data.

Table 9.1: Score metrics for the QPack, QV and CLOPS benchmarks on tested IMBQ backends.

	QPack				IBM	
Hardware	Runtime	Accuracy	Scalability	Capacity	QV	CLOPS
IBMQ Quito	3.27	9.64	13.92	4.00	16	2.5K
IBMQ Nairobi	3.23	9.72	8.95	4.33	32	2.6K
IBMQ Perth	3.24	8.82	5.25	4.00	32	2.9K
IBMQ Lagos	3.14	9.30	4.05	5.00	32	2.7K
IBMQ Jakarta	3.24	8.32	6.90	4.33	16	2.4K

From the table, we would expect that the Nairobi, Perth, and Lagos backends achieve the highest accuracy and capacity scores, as their QV is the highest. However, the Quito achieves the second highest accuracy score, even though it only has a QV of 16. The same goes for the runtime score, where the Perth backend achieves the highest CLOPS score and also the second highest QPack runtime score, but the Jakarta gets the lowest CLOPS score and the second highest runtime score in QPack. These differences again highlight the difference between randomized and application-oriented benchmarking. Some correlation is present, but the QPack benchmark is able to give an insight of quantum performance for actual applications.

9.4. Score criteria

A reflection can be made on the score criteria as defined in Section 4.3 can be made. Recall that the criteria are as follows.

1. Benchmark score reflects application-level performance of quantum computers (simulators and hardware implementations)
2. Benchmark score is a composite of measurement data of multiple quantum applications
3. Benchmark score is a single number (but may be split up into sub-scores)
4. Benchmark score is proportional to performance, i.e., a higher score means higher performance
5. Benchmark score are scalable, i.e., score has no upper limit
6. Benchmark score does not become too abstract from the data it is based on
7. Sub-scores should be balanced, such that one sub-score does not become dominant in the overall score

Criterion one ensures application-oriented benchmarking, which is achieved by using VQE and QAOA implementations that have a real-life purpose, either solving a graph problem or finding the lowest eigenvalue of a quantum system. The second criterion is ensured by combining the subscores of all individual problems by taking an arithmetic mean. Combining the subscores to an overall score by taking the area of the four-sided radar plot figure satisfies the third criterion, which enforces a single-number performance score. By mapping the pure subscores, criteria four is also met, which ensures proportional performance scores. The fifth criterion is met as well, but is somewhat open to discussion for the accuracy subscore. It is indeed the case that the score has no upper limit (as the smaller the relative error gets, the higher the score becomes), but as quantum computers become less prone to noise, this value will likely saturate as we leave the NISQ-era. Criterion six is achieved by using subscores instead of solely using the overall score. Using this method, performance analysis of different aspects of quantum computers can be achieved. Lastly, balancing the subscores is achieved by tuning the coefficients c_0, c_1, c_2, and c_3. It could be said that the capacity subscore could grow faster than the other subscores for post-NISQ-era quantum computers. If this becomes the case, the balancing parameters could be tuned, or the capacity subscore could be scaled as well. The capacity subscore is not currently scaled up or down, as the score now nicely reflects the average number of usable qubits on a quantum computer. With all the criteria covered, it can be concluded that the current version of the QPack benchmark adheres to all points.

10

Conclusion & recommendations

This thesis has presented a new method to evaluate a quantum computer's performance using an application-oriented benchmark approach for holistic performance evaluation for NISQ-era quantum computers. Running these benchmarking applications on different quantum computers using |Lib⟩ allows the collection of quantum execution data to be used as a quantum performance indicator.

After an overview of the current state of quantum benchmarking, opportunities were made clear for a more holistic and versatile benchmarking approach, using VQAs as benchmark applications to create a variety of scalable benchmarking circuits using the cross-platform quantum library |Lib⟩.

With this in mind, the QPack benchmarking outline was presented to see what quantum execution data could be obtained when running these VQA-applications. From this execution data, differences in some performance could already be observed, which required to be transformed into actual performance scores. This led to the definition of the benchmark score criteria, which served as a guide to form the later defined benchmark scores.

The ability to execute the benchmark applications at all depends on the implementation of the chosen VQE and QAOA problems in |Lib⟩. For each of the 6 implemented VQA problems, a detailed step-by-step implementation was given, as well as a proof of implementation by showing some results for each |Lib⟩ implementation.

The transformation of the raw quantum execution data into the QPack scores provides four performance subscores based on runtime, accuracy, scalability, and capacity. Runtime evaluates how fast a quantum computer can execute a quantum circuit on average. Accuracy then checks how well the output of a quantum computer performs compared to that of an ideal quantum computer simulator. Scalability tells us something about how well the quantum computer is able to handle increasing circuit sizes. Finally, the capacity subscore indicates what the usable number of qubits on a quantum processor actually is, regardless of how many qubits a quantum computer can provide. These subscores were then evaluated on a set of synthetic data sets to verify that they behave as intended.

A proof-of-concept has been provided by running QPack on a set of local and remote quantum computer simulators, as well as a set of available quantum computer hardware. Local simulators from Cirq, Qiskit, Rigetti and QuEST have been compared to one another, where it was shown that the QuEST simulator has the best overall performance. In the evaluation of the remote quantum computer simulators provided by IBMQ, IonQ, and Rigetti, it became clear that the IBMQ QASM simulator had the best overall performance, but the IonQ and Rigetti simulators could be appropriate choice to use for certain circuit sizes. A selection of quantum hardware from the IBMQ aviary and the Rigetti Aspen-M-1 have also been benchmarked. This showed that the IBMQ Quito and Nairobi are the better performing quantum computers of the tested backends. Due to access limits from remote providers, only a small set of hardware data has been collected. In order to get a better insight into the performance of state-of-the-art quantum computers, more access time to these backends is required. For this thesis, a total of 7 simulators and 7 hardware quantum computers have been benchmarked.

In conclusion, the QPack benchmark is able to make a quantitative performance analysis between different quantum computers, be it a physical realization or a simulator. The code for the complete version of QPack in |Lib⟩ can be found in `https://gitlab.com/libket/qpack/-/tree/stable`.

There are of course still some improvements to make to the QPack benchmark. Besides the discussed points in Chapter 9, there are still plenty of improvement opportunities. For example, as shown in Section 4.2, only a few of the quantum execution data types presented are used to compute the QPack scores. This indicates that there are possibilities to create more subscores. For example, a comparison between classical and quantum runtimes could be made or the serviceability of a remote quantum computer can be quantified in a new sub-score.

An optimization in total benchmark duration is also something that can be a great improvement for the QPack benchmark. Currently, benchmarking quantum computers or simulators can take a long time, especially for remote services where resources are shared on a scheduled basis. This currently happens because QPack sends a circuit to the provider, waits for the results to come back, has the optimizer tweak the circuit parameters, and sends a new circuit back to the provider. This repeats until the optimizer has found the optimal parameters. This can be improved by sending the optimization program as a whole to the quantum provider, which IBM allows by using Qiskit Runtime [52] for example. The current version of |Lib⟩ does not yet offer the ability to send complete quantum programs to a provider. This way of submitting quantum jobs would offer a speedup of benchmarking quantum computers, as there is no back-and-forth communication between the client and the provider. When |Lib⟩ supports this function, it should be a high priority to implement it in QPack.

Further improvements to QPack could be made by implementing more problem sets of different VQAs or non-VQAs. Algorithms such as VQLS or VQS could be a good addition to the current QPack application set. But also non-VQA applications such as HHL [69], hydrogen simulation [115] or Shor's algorithm [60] could offer viable (post-)NISQ-era quantum applications to use in the further development of QPack, allowing for an even more varied set of quantum circuits to evaluate quantum performance. In order to select new applications for QPack, benchmark application circuits should be analyzed on their characteristics, to find what types of circuit are missing in the current set. The SupermarQ [19] benchmark, for example, indexes its application circuits based on *program communication*, *critical-depth*, *entanglement-ratio*, *parallelism*, *liveness* and *measurement*. Such an approach could also be interesting for QPack, so it can reliably offer a varied set of quantum benchmarking applications.

With this in mind, the question arises as to what quantum applications are actually suitable for benchmarking in the post-NISQ-era. While VQAs are a good solution for the current state of quantum computing, they may not be as usable for large quantum circuits, as they would likely need a large number of optimization parameters, which lays more computational stress on the classical optimizer. As observed for the VQE IC problem, a large number of optimization parameters generally cause the optimizer to need many optimization iterations (± 1000 for $N = 10$) to find the optimal parameters. It would therefore be a good effort to find some pure quantum applications for benchmarking that can be used as we leave the NISQ-era and the QPack benchmark can remain relevant.

This work has presented results for a small set of quantum computers from Rigetti and IBMQ quantum, but there are many more quantum computers that can be benchmarked. For instance, IonQ [139], Rigetti [136], IBMQ [50], Honeywell [145], and Quantum Inspire [146] offer a multitude of quantum computers. Evaluating all of these backends should give an insightful comparison between quantum computers of different vendors and could show what the current state of quantum technology is at this time.

Bibliography

[1] D. Wineland, J. Bergquist, J. Bollinger, and W. Itano, "Quantum effects in measurements on trapped ions," *Physica Scripta*, 1995-01-01 1995.

[2] D. P. DiVincenzo, "The physical implementation of quantum computation," *Fortschritte der Physik*, vol. 48, p. 771–783, Sep 2000. http://dx.doi.org/10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E.

[3] I. L. Chuang, N. Gershenfeld, and M. Kubinec, "Experimental implementation of fast quantum searching," *Phys. Rev. Lett.*, vol. 80, pp. 3408–3411, Apr 1998. https://link.aps.org/doi/10.1103/PhysRevLett.80.3408.

[4] W. Pfaff, B. J. Hensen, H. Bernien, S. B. van Dam, M. S. Blok, T. H. Taminiau, M. J. Tiggelman, R. N. Schouten, M. Markham, D. J. Twitchen, and R. Hanson, "Unconditional quantum teleportation between distant solid-state quantum bits," *Science*, vol. 345, no. 6196, pp. 532–535, 2014. https://www.science.org/doi/abs/10.1126/science.1253512.

[5] J. Preskill, "Quantum Computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, Aug. 2018. https://doi.org/10.22331/q-2018-08-06-79.

[6] F. Arute, K. Arya, Babbush, and R. et al, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, p. 505–510, 2019.

[7] E. Pednault, J. Gunnels, D. Maslov, and J. Gambetta, "On "quantum supremacy"," Oct 2019. https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/.

[8] J. Kelley, "A preview of bristlecone, google's new quantum processor," Mar 2018. https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html.

[9] Rigetti Computing, "Aspen-m-1 quantum processor," 2020. https://qcs.rigetti.com/qpus.

[10] H. Collins, "Ibm unveils breakthrough 127-qubit quantum processor," Nov 2021. https://newsroom.ibm.com/2021-11-16-IBM-Unveils-Breakthrough-127-Qubit-Quantum-Processor.

[11] D-Wave Systems Inc., "The advantage quantum computer," 2022. https://www.dwavesys.com/solutions-and-products/systems/.

[12] Challenge Institute for Quantum Information, "Scaling quantum technologies." https://ciqc.berkeley.edu/scaling-quantum-technologies.

[13] R. Blume-Kohout and K. C. Young, "A volumetric framework for quantum computer benchmarks," *Quantum*, vol. 4, p. 362, Nov 2020. http://dx.doi.org/10.22331/q-2020-11-15-362.

[14] S. Martiel, T. Ayral, and C. Allouche, "Benchmarking quantum coprocessors in an application-centric, hardware-agnostic, and scalable way," *IEEE Transactions on Quantum Engineering*, vol. 2, p. 1–11, 2021. http://dx.doi.org/10.1109/TQE.2021.3090207.

[15] T. Lubinski, S. Johri, P. Varosy, J. Coleman, L. Zhao, J. Necaise, C. H. Baldwin, K. Mayer, and T. Proctor, "Application-oriented performance benchmarks for quantum computing," 2021.

[16] IonQ, Inc, "Algorithmic qubits: A better single-number metric," Feb 2022. https://ionq.com/posts/february-23-2022-algorithmic-qubits.

[17] P.-L. Dallaire-Demers, M. Stechly, J. F. Gonthier, N. T. Bashige, J. Romero, and Y. Cao, "An application benchmark for fermionic quantum simulations," 2020.

[18] A. Cornelissen, J. Bausch, and A. Gilyén, "Scalable benchmarks for gate-based quantum computers," 2021. https://arxiv.org/abs/2104.10698.

[19] T. Tomesh, P. Gokhale, V. Omole, G. S. Ravi, K. N. Smith, J. Viszlai, X.-C. Wu, N. Hardavellas, M. R. Martonosi, and F. T. Chong, "Supermarq: A scalable quantum benchmark suite," 2022. https://arxiv.org/abs/2202.11045.

[20] K. Mesman, *QPack: QAOA as scalable application-level quantum benchmark*. PhD thesis, Delft University of Technology, 2021. http://resolver.tudelft.nl/uuid:cc8d7440-928d-4518-9a91-14f8770b31e9.

[21] K. Mesman, H. Donkers, Z. Al-Ars, and M. Möller, "Qpack: Quantum approximate optimization algorithms as universal benchmark for quantum computers," 2021. https://arxiv.org/abs/2103.17193.

[22] M. Möller and M. Schalkers, "Libket: A cross-platform programming framework for quantum-accelerated scientific computing," in *Computational Science – ICCS 2020* (V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, eds.), (Cham), pp. 451–464, Springer International Publishing, 2020.

[23] M. Möller, "Libket," 2021. https://libket.readthedocs.io/en/latest/.

[24] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," 2014.

[25] A. Peruzzo, J. Mcclean, P. Shadbolt, M. H. Yung, X. Zhou, P. Love, A. Aspuru-Guzik, and J. O'Brien, "A variational eigenvalue solver on a quantum processor," *Nature communications*, vol. 5, 04 2013.

[26] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 6th ed., 2017.

[27] J. von Kistowski, J. Arnold, K. Huppler, K.-D. Lange, J. Henning, and P. Cao, "How to build a benchmark," in *ICPE 2015 - Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 02 2015.

[28] W. Dai and D. Berleant, "Benchmarking contemporary deep learning hardware and frameworks: A survey of qualitative metrics," *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, Dec 2019. http://dx.doi.org/10.1109/CogMI48466.2019.00029.

[29] R. Weicker, "An overview of common benchmarks," *Computer*, vol. 23, no. 12, pp. 65–75, 1990.

[30] J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 803–820, 08 2003.

[31] J. Dongarra, *Report on the Sunway TaihuLight System*. University of Tennessee, 2016.

[32] J. Dongarra and T. Haigh, "Oral history interview," Apr 2005. http://history.siam.org/oralhistories/dongarra.htm.

[33] "Standard performance evaluation corporation," Oct 2021. http://www.spec.org/.

[34] BAPCo, "Business applications performance corporation," Oct 2021. https://bapco.com/.

[35] BAPCo, "Sysmark25 white paper," Jul 2020. https://bapco.com/products/sysmark-25/attachment/sysmark25whitepaper/.

[36] EEMBC, "Embedded microprocessor benchmark consortium." https://www.eembc.org/.

[37] S. Gal-on and M. Levy, "Exploring coremark™ – a benchmark maximizing simplicity and efficacy," whitepaper, EEMBC, May 2012. https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf.

[38] D. P. DiVincenzo, "The physical implementation of quantum computation," *Fortschritte der Physik*, vol. 48, p. 771–783, Sep 2000. http://dx.doi.org/10.1002/1521-3978(200009)48:9/11<771::AID-PROP771>3.0.CO;2-E.

[39] M. L. Dahlhauser, *Characterization and Benchmarking of Quantum Computers*. PhD thesis, University of Tennessee, 2021.

[40] P. Bardroff, C. Leichtle, G. Schrade, and W. Schleich, "Quantum state preparation and measurement," in *Conference Proceedings LEOS'96 9th Annual Meeting IEEE Lasers and Electro-Optics Society*, vol. 2, pp. 347–348 vol.2, 1996.

[41] E. Knill, D. Leibfried, R. Reichle, J. Britton, R. B. Blakestad, J. D. Jost, C. Langer, R. Ozeri, S. Seidelin, and D. J. Wineland, "Randomized benchmarking of quantum gates," *Physical Review A*, vol. 77, jan 2008. https://doi.org/10.1103%2Fphysreva.77.012307.

[42] E. Nielsen, J. K. Gamble, K. Rudinger, T. Scholten, K. Young, and R. Blume-Kohout, "Gate set tomography," *Quantum*, vol. 5, p. 557, oct 2021. https://doi.org/10.22331%2Fq-2021-10-05-557.

[43] A. W. Cross, L. S. Bishop, S. Sheldon, P. D. Nation, and J. M. Gambetta, "Validating quantum computers using randomized model circuits," *Physical Review A*, vol. 100, Sep 2019. http://dx.doi.org/10.1103/PhysRevA.100.032328.

[44] E. Magesan, J. M. Gambetta, and J. Emerson, "Scalable and robust randomized benchmarking of quantum processes," *Physical Review Letters*, vol. 106, may 2011. https://doi.org/10.1103%2Fphysrevlett.106.180504.

[45] S. Boixo, S. V. Isakov, V. N. Smelyanskiy, R. Babbush, N. Ding, Z. Jiang, M. J. Bremner, J. M. Martinis, and H. Neven, "Characterizing quantum supremacy in near-term devices," *Nature Physics*, vol. 14, pp. 595–600, apr 2018. https://doi.org/10.1038%2Fs41567-018-0124-x.

[46] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, (New York, NY, USA), p. 212–219, Association for Computing Machinery, 1996. https://doi.org/10.1145/237814.237866.

[47] H. Bethe, "Zur theorie der metalle," *Zeitschrift für Physik*, vol. 71, pp. 205–226, Mar 1931. https://doi.org/10.1007/BF01341708.

[48] P. Erdös and A. Rényi, "On random graphs i," *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.

[49] IBM Quantum Services, "Ibm quantum simulators," 2022. https://quantum-computing.ibm.com/services/docs/services/manage/simulator/#qasm.

[50] IBM Quantum Services, "Ibm quantum services," 2022. https://quantum-computing.ibm.com/services?services=systems.

[51] A. Wack, H. Paik, A. Javadi-Abhari, P. Jurcevic, I. Faro, J. M. Gambetta, and B. R. Johnson, "Quality, speed, and scale: three key attributes to measure the performance of near-term quantum computers," 2021. https://arxiv.org/abs/2110.14108.

[52] Qiskit Development Team, "Qiskit runtime overview," May 2022. https://qiskit.org/documentation/partners/qiskit_ibm_runtime/index.html.

[53] Super.tech Labs, "Superstaq," Sep 2021. https://www.super.tech/about-superstaq/.

[54] T. Lubinski, S. Johri, P. Varosy, J. Coleman, L. Zhao, J. Necaise, C. Baldwin, K. Mayer, and T. Proctor, "Application-Oriented Performance Benchmarks for Quantum Computing," 10 2021. https://arxiv.org/abs/2110.03137.

[55] A. J. McCaskey, D. I. Lyakh, E. F. Dumitrescu, S. S. Powers, and T. S. Humble, "Xacc: A system-level software infrastructure for heterogeneous quantum-classical computing," 2019.

[56] A. J. McCaskey, "Xacc: Quantum framework," 2019. https://xacc.readthedocs.io/en/latest/.

[57] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020.

[58] super.tech, "Super.tech announces quantum software platform superstaq," *Super.tech*, Aug 2021. https://www.super.tech/super-tech-announces-quantum-software-platform-superstaq/.

[59] A. Y. Kitaev, "Quantum measurements and the abelian stabilizer problem," *Electron. Colloquium Comput. Complex.*, vol. 3, 1996.

[60] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, 1994.

[61] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio, and et al., "Variational quantum algorithms," *Nature Reviews Physics*, vol. 3, p. 625–644, Aug 2021. http://dx.doi.org/10.1038/s42254-021-00348-9.

[62] N. Hatano and M. Suzuki, "Finding exponential product formulas of higher orders," 2005.

[63] A. J., A. Adedoyin, J. Ambrosiano, P. Anisimov, A. Bärtschi, W. Casper, G. Chennupati, C. Coffrin, H. Djidjev, D. Gunter, S. Karra, N. Lemons, S. Lin, A. Malyzhenkov, D. Mascarenas, S. Mniszewski, B. Nadiga, D. O'Malley, D. Oyen, S. Pakin, L. Prasad, R. Roberts, P. Romero, N. Santhi, N. Sinitsyn, P. J. Swart, J. G. Wendelberger, B. Yoon, R. Zamora, W. Zhu, S. Eidenbenz, P. J. Coles, M. Vuffray, and A. Y. Lokhov, "Quantum algorithm implementations for beginners," 2018. https://arxiv.org/abs/1804.03719.

[64] A. Anand, P. Schleich, S. Alperin-Lea, P. W. K. Jensen, S. Sim, M. Dí az-Tinoco, J. S. Kottmann, M. Degroote, A. F. Izmaylov, and A. Aspuru-Guzik, "A quantum computing view on unitary coupled cluster theory," *Chemical Society Reviews*, vol. 51, no. 5, pp. 1659–1684, 2022. https://doi.org/10.1039%2Fd1cs00932j.

[65] Qiskit Development Team, "Efficientsu2," Feb 2022. https://qiskit.org/documentation/stubs/qiskit.circuit.library.EfficientSU2.html.

[66] S. Hadfield, Z. Wang, B. O'Gorman, E. Rieffel, D. Venturelli, and R. Biswas, "From the quantum approximate optimization algorithm to a quantum alternating operator ansatz," *Algorithms*, vol. 12, p. 34, Feb 2019. http://dx.doi.org/10.3390/a12020034.

[67] C. Bravo-Prieto, R. LaRose, M. Cerezo, Y. Subasi, L. Cincio, and P. J. Coles, "Variational quantum linear solver," 2020.

[68] C.-C. Chen, S.-Y. Shiau, M.-F. Wu, and Y.-R. Wu, "Hybrid classical-quantum linear solver using noisy intermediate-scale quantum machines," *Scientific Reports*, vol. 9, nov 2019.

[69] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Phys. Rev. Lett.*, vol. 103, p. 150502, Oct 2009. https://link.aps.org/doi/10.1103/PhysRevLett.103.150502.

[70] A. Luongo, "Quantum algorithms for data analysis," Apr 2022. https://quantumalgorithms.org/chapter-intro.html#hadamard-test.

[71] E. Farhi and H. Neven, "Classification with quantum neural networks on near term processors," 2018. https://arxiv.org/abs/1802.06002.

[72] M. Schuld, A. Bocharov, K. M. Svore, and N. Wiebe, "Circuit-centric quantum classifiers," *Physical Review A*, vol. 101, mar 2020. https://doi.org/10.1103%2Fphysreva.101.032308.

[73] V. Havlíček, A. D. Córcoles, K. Temme, A. W. Harrow, A. Kandala, J. M. Chow, and J. M. Gambetta, "Supervised learning with quantum-enhanced feature spaces," *Nature*, vol. 567, pp. 209–212, mar 2019. https://doi.org/10.1038%2Fs41586-019-0980-2.

[74] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[75] R. A. FISHER, "The use of multiple measurements in taxonomic problems," *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936.

[76] A. M. Horst, A. P. Hill, and K. B. Gorman, "palmerpenguins: Palmer archipelago (antarctica) penguin data," 2020. R package version 0.1.0.

[77] N. Killoran, T. R. Bromley, J. M. Arrazola, M. Schuld, N. Quesada, and S. Lloyd, "Continuous-variable quantum neural networks," *Physical Review Research*, vol. 1, oct 2019. https://doi.org/10.1103%2Fphysrevresearch.1.033063.

[78] G. Verdon, J. Marks, S. Nanda, S. Leichenauer, and J. Hidary, "Quantum hamiltonian-based models and the variational quantum thermalizer algorithm," 2019. https://arxiv.org/abs/1910.02071.

[79] E. R. Anschuetz, J. P. Olson, A. Aspuru-Guzik, and Y. Cao, "Variational quantum factoring," 2018. https://arxiv.org/abs/1808.08927.

[80] A. Matsuo, Y. Suzuki, and S. Yamashita, "Problem-specific parameterized quantum circuits of the vqe algorithm for optimization problems," 2020.

[81] M. Lavrov, "Lecture 35: The traveling salesman problem," *Math 482: Linear Programming*, 05 2020. https://faculty.math.illinois.edu/ mlavrov/docs/482-spring-2020/lecture35.pdf.

[82] S. Narain, "Lecture notes cs:5350 minimum vertex cover: 2-approximation and lp-based views," 10 2019. http://homepage.cs.uiowa.edu/ sriram/5350/fall19/notes/10.31/10.31.pdf.

[83] C. Commander, "Maximum cut problem, max-cut," 01 2008.

[84] P. Sung, "Maximum satisfiability," 03 2006. https://math.mit.edu/goemans/18434S06/max-sat-phil.pdf.

[85] J. Hartmanis, "Computers and intractability: A guide to the theory of np-completeness (michael r. garey and david s. johnson)," *SIAM Review*, vol. 24, no. 1, pp. 90–91, 1982.

[86] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm applied to a bounded occurrence constraint problem," 2014.

[87] P. Vigoda, "Lecture notes on a parallel algorithm for generating a maximal independent set," 03 2010. https://faculty.cc.gatech.edu/ vigoda/7530-Spring10/MIS.pdf.

[88] P. Pardalos and J. Xue, "The maximum clique problem," *Journal of Global Optimization*, vol. 4, pp. 301–328, 04 1994.

[89] R. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, vol. 40, pp. 85–103, 01 1972.

[90] T. Stern, "Seminar in theoretical computer science," 09 2006. https://math.mit.edu/ goemans/18434S06/setcover-tamara.pdf.

[91] R. Sotirov, O. Kuryatnikova, and J. Vera, "The maximum k-colorable subgraph problem and related problems," 2020.

[92] R. Krauthgamer and M.-Y. Kao, *Minimum Bisection*, pp. 1294–1297. New York, NY: Springer New York, 2016.

[93] F. Della Croce and V. Paschos, "On the max k-vertex cover problem," 03 2011.

[94] R. Sotirov, O. Kuryatnikova, and J. Vera, "The maximum k-colorable subgraph problem and related problems," 2020.

[95] A. Mittal, P. Jain, S. Mathur, and P. Bhatt, "Graph coloring with minimum colors: An easy approach," *Proceedings - 2011 International Conference on Communication Systems and Network Technologies, CSNT 2011*, 06 2011.

[96] E. L. Lawler, J. K. Lenstra, A. H. Rinnooy Kan, and D. B. Shmoys, "Chapter 9 sequencing and scheduling: Algorithms and complexity," in *Logistics of Production and Inventory*, vol. 4 of *Handbooks in Operations Research and Management Science*, pp. 445–522, Elsevier, 1993.

[97] T. Gobel, *On the physics of Trotterization*. PhD thesis, Faculty of Science, 2022.

[98] L. Bittel and M. Kliesch, "Training variational quantum algorithms is np-hard," *Physical Review Letters*, vol. 127, Sep 2021. http://dx.doi.org/10.1103/PhysRevLett.127.120502.

[99] J. Brownlee, "Local optimization versus global optimization," Oct 2021.

[100] Y. Xiang, S. Gubian, B. Suomela, and J. Hoeng, "Generalized simulated annealing for global optimization: The gensa package," *The R Journal Volume 5(1):13-29, June 2013*, vol. 5, 06 2013.

[101] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[102] H. Szu and R. Hartley, "Fast simulated annealing," *Physics Letters A*, vol. 122, no. 3, pp. 157–162, 1987.

[103] J. Brownlee, "Dual annealing optimization with python," Oct 2021. https://machinelearningmastery.com/dual-annealing-optimization-with-python/.

[104] The SciPy community, "scipy.optimize.dual_annealing," 2022. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.dual_annealing.html.

[105] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[106] S. Joglekar, "Nelder-mead optimization," Jan 2016. https://codesachin.wordpress.com/2016/01/16/nelder-mead-optimization/.

[107] D. F. Shanno, "Conditioning of quasi-newton methods for function minimization," *Mathematics of Computation*, vol. 24, pp. 647–656, 1970.

[108] J. Brownlee, "A gentle introduction to the bfgs optimization algorithm," Oct 2021. https://machinelearningmastery.com/bfgs-optimization-in-python/.

[109] M. Powell, "A view of algorithms for optimization without derivatives," *Mathematics TODAY*, vol. 43, 01 2007.

[110] S. G. Johnson, "The nlopt nonlinear-optimization package," Dec 2021. http://github.com/stevengj/nlopt.

[111] M. Powell, "The bobyqa algorithm for bound constrained optimization without derivatives," *Technical Report, Department of Applied Mathematics and Theoretical Physics*, 01 2009.

[112] S. C. Endres, C. Sandrock, and W. W. Focke, "A simplicial homology algorithm for lipschitz optimisation," *Journal of Global Optimization*, vol. 72, pp. 181–217, Oct 2018.

[113] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[114] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "Quest and high performance simulation of quantum computers," *Scientific Reports*, vol. 9, no. 1, 2019.

[115] P. O'Malley, R. Babbush, I. Kivlichan, J. Romero, J. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, and et al., "Scalable quantum simulation of molecular energies," *Physical Review X*, vol. 6, Jul 2016. http://dx.doi.org/10.1103/PhysRevX.6.031007.

[116] Bradben, "Single- and multi-qubit pauli measurement operations," Jul 2021. https://docs.microsoft.com/en-us/azure/quantum/concepts-pauli-measurements.

[117] K. Zhang and Z. Song, "Quantum phase transition in a quantum ising chain at nonzero temperatures," *Physical Review Letters*, vol. 126, mar 2021. https://doi.org/10.1103%2Fphysrevlett.126.116401.

[118] M. Pokharel, "Computational complexity theory(p,np,np-complete and np-hard problems)," *ResearchGate*, 06 2020.

[119] L. Zhou, S.-T. Wang, S. Choi, H. Pichler, and M. D. Lukin, "Quantum approximate optimization algorithm: Performance, mechanism, and implementation on near-term devices," *Physical Review X*, vol. 10, jun 2020. https://doi.org/10.1103%2Fphysrevx.10.021067.

[120] Qiskit Development Team, "Aer simulator," Dec 2021. https://qiskit.org/documentation/stubs/qiskit.providers.aer.AerSimulator.html.

[121] N. J. Guerrero, *Solving Combinatorial Optimization Problems using the Quantum Approximation Optimization Algorithm*. PhD thesis, Air Force Institute of Technology, Mar 2020. https://scholar.afit.edu/etd/3263.

[122] J. Ceroni, "Fun with graphs and qaoa," Mar 2020. https://lucaman99.github.io/new_blog/2020/mar16.html.

[123] A. Bärtschi and S. Eidenbenz, "Deterministic preparation of dicke states," *Lecture Notes in Computer Science*, p. 126–139, 2019. http://dx.doi.org/10.1007/978-3-030-25027-0_9.

[124] C. S. Mukherjee, S. Maitra, V. Gaurav, and D. Roy, "On actual preparation of dicke state on a quantum computer," 2020.

[125] K. Mesman, F. Battistel, M. Tiggelman, J. Gloudemans, J. van Oven, and C. C. Bultink, "Benchmarking and profiling quantum control stacks," *Under review*, 05 2022.

[126] Cirq Development Team, "Simulation," Apr 2022. https://quantumai.google/cirq/simulation.

[127] Rigetti Computing, "The quantum virtual machine (qvm)," 2021. https://pyquil-docs.rigetti.com/en/latest/qvm.html.

[128] Advanced Micro Devices, Inc, "Amd ryzen™ 5 3600," *AMD.com*, 2022. https://www.amd.com/en/product/8456.

[129] M. Treinish and et. al., "Qiskit: An open-source framework for quantum computing," 2021.

[130] Cirq Developers, "Cirq," Aug. 2021. https://doi.org/10.5281/zenodo.5182845.

[131] Quantum AI team and collaborators, "qsim," Sept. 2020. https://doi.org/10.5281/zenodo.4023103.

[132] B. Villalonga, S. Boixo, B. Nelson, C. Henze, E. Rieffel, R. Biswas, and S. Mandrà, "A flexible high-performance simulator for verifying and benchmarking quantum circuits implemented on real hardware," *npj Quantum Information*, vol. 5, oct 2019. https://doi.org/10.1038%2Fs41534-019-0196-1.

[133] R. S. Smith, M. J. Curtis, and W. J. Zeng, "A practical quantum instruction set architecture," 2016.

[134] Rigetti Computing, "Pyquil: Quantum programming in python." `https://github.com/rigetti/pyquil`, 2022.

[135] Rigetti Computing, "Welcome to the docs for pyquil!," 2021. https://pyquil-docs.rigetti.com/en/latest/.

[136] P. J. Karalekas, N. A. Tezak, E. C. Peterson, C. A. Ryan, M. P. da Silva, and R. S. Smith, "A quantum-classical cloud platform optimized for variational hybrid algorithms," *Quantum Science and Technology*, vol. 5, p. 024003, apr 2020. https://doi.org/10.1088%2F2058-9565%2Fab7559.

[137] T. Jones, B. Koczor, R. Meister, and S. Benjamin, "Quantum exact simulation toolkit," Jul 2019. https://quest.qtechtheory.org/.

[138] IonQ, Inc, "Documentation," 2022. https://ionq.com/docs.

[139] K. Wright, K. M. Beck, S. Debnath, J. M. Amini, Y. Nam, N. Grzesiak, J.-S. Chen, N. C. Pisenti, M. Chmielewski, C. Collins, K. M. Hudek, J. Mizrahi, J. D. Wong-Campos, S. Allen, J. Apisdorf, P. Solomon, M. Williams, A. M. Ducore, A. Blinov, S. M. Kreikemeier, V. Chaplin, M. Keesan, C. Monroe, and J. Kim, "Benchmarking an 11-qubit quantum computer," *Nature Communications*, vol. 10, p. 5464, Nov 2019. https://doi.org/10.1038/s41467-019-13534-2.

[140] S. L. Bravo, "Ionq provider - azure quantum," 2022. https://docs.microsoft.com/en-us/azure/quantum/provider-ionq.

[141] IBM Quantum, "Ibm quantum," 2021. https://quantum-computing.ibm.com/.

[142] B. Josephson, "Possible new effects in superconductive tunnelling," *Physics Letters*, vol. 1, no. 7, pp. 251–253, 1962.

[143] P. W. Anderson and J. M. Rowell, "Probable observation of the josephson superconducting tunneling effect," *Phys. Rev. Lett.*, vol. 10, pp. 230–232, Mar 1963.

[144] X. Gao and L. Duan, "Efficient classical simulation of noisy quantum computation," 2018.

[145] Honeywell International Inc., "Honeywell system model h1," 2022. https://www.honeywell.com/us/en/company/quantum/quantum-computer.

[146] Quantum Inspire, "Available quantum processors," 2022.

[147] W. Dean, "Computational Complexity Theory," in *The Stanford Encyclopedia of Philosophy* (E. N. Zalta, ed.), Metaphysics Research Lab, Stanford University, Fall 2021 ed., 2021.

[148] S. Huang, "What is big o notation explained: Space and time complexity," Jun 2021. https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/.

[149] E. Roswell, "Know thy complexities!." https://www.bigocheatsheet.com/.

[150] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser, "Quantum computation by adiabatic evolution," *arXiv: Quantum Physics*, 02 2000.

[151] M. Born and V. Fock, "Beweis des Adiabatensatzes," *Zeitschrift fur Physik*, vol. 51, pp. 165–180, Mar. 1928.

[152] E. Crosson, E. Farhi, C. Y.-Y. Lin, H.-H. Lin, and P. Shor, "Different strategies for optimization using the quantum adiabatic algorithm," 2014.

[153] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

A

Complexity theory overview

Computational complexity theory is a field of computer science that classifies and compares the difficulty of solving problems [147]. A problem X is considered more complex or harder than problem Y if Y possesses a more efficient decision algorithm than the most efficient algorithm to solve X. A more decisive definition to compare these problems can be made with computational conventions, such as the Big-O notation.

Big-O Notation
The Big-O notation is a measure of how quickly an algorithm solves a problem and gives an upper bound to the input [148, 118]. The following Big-O values can be found commonly in the literature:

- $O(1)$: Constant-time - Time to solve is independent on problem size
- $O(log(n))$: Logarithmic-time - Good complexity, less complex than $O(\sqrt(n))$
- $O(n)$: Linear-time - Time to solve scales linearly with problem size
- $O(n^2)$: Quadratic-time - Time to solve scales quadratic with problem size
- $O(n^k)$: Polynomial-time - Same idea, $O(n^4)$ is less complex than $O(n^5)$
- $O(k^n)$: Exponential-time - More complex than Polynomial time
- $O(n!)$: Factorial-time - Even more complex than exponential time

In Figure A.1, a comparison of different complexity classes can be found, as indexed by Roswell [149].

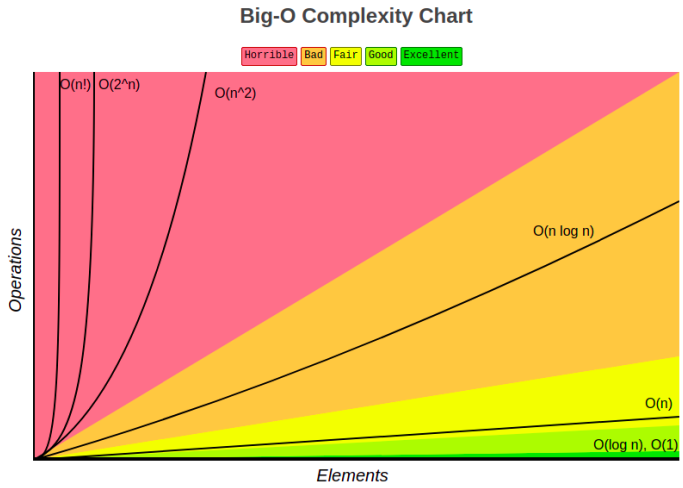


Figure A.1: Big-O notation efficiency indexing [149]

Problem classification
Computable problems (unlike their suggested counterpart non-computable problems) have an existing algorithm that can solve that problem, e.g., there exists a Turing machine that can compute the answer to that problem [118]. From easy to hardest, they are classified into separate classes:

P: Polynomial problems
The set of problems that can be solved in polynomial time, i.e., all problems have:

$$T(n) = O(C * n^k)$$

where constants $C, k > 0$. Examples of P problems are mathematical operations, sorting problems, shortest path problems and table lookups.

NP: Non-deterministic Polynomial problems
This class of problems cannot be solved in polynomial time, but can be verified in polynomial time. These algorithms have an expected exponential or factorial time complexity, defined as:

$$T(n) = O(C_1 * K^{C_2 * n})$$

where constants $C_1, C_2, k > 0$. $T(n)$ is of exponential time if $C_1 = C_2 = 1$.

NP-Complete:
This is a subset of NP, with the additional property that they are complete. This means that there exists a polynomial-time algorithm that can reduce a certain problem into another NP-complete problem. Examples of these problems are the traveling salesman, the knapsack, and the graph coloring problem.

NP-Hard:
The hardest class of problems to solve. They cannot be solved in polynomial time and cannot be verified in polynomial time. However, they can be reduced to any other NP problem.

The relation of these problem classes can be found in Figure A.2, assuming P $\neq$ NP.

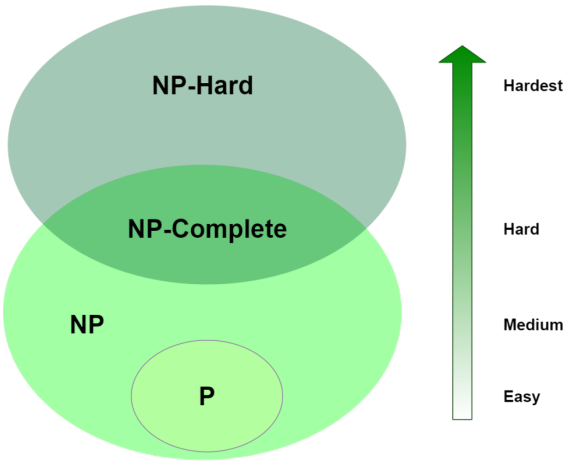


Figure A.2: Complexity classes [118]

B

Various quantum algorithms

These sets of algorithms consist of pure quantum algorithms, which, in the NISQ-era, may not be as effective as VQAs for larger problem sets, but are interesting to list as benchmark opportunities as well. Some of the more well-known quantum algorithms are briefly described in this subsection.

Quantum Adiabatic Algorithm
In 2000, the Quantum Adiabatic Algorithm (QAA) was proposed by Fahri et al. [150], using the adiabatic theorem [151] to find the eigenstate of a Hamiltonian that encodes the solution for a given problem. The algorithm starts with a simple Hamiltonian H_B, of which the ground state is known. This Hamiltonian is then slowly evolved over time to the problem encoded Hamiltonian H_P according to:

$$H(t) = (1 - t/T)H_B + (t/T)H_P \tag{B.1}$$

If one then starts to evolve the system at $t = 0$, i.e. $H(0) = H_B$, and stops at $t = T$, then the ground state $|\psi(T)\rangle$ will be very close to the ground state of H_P. One should be careful to pick a long enough time T as a too fast evolution will cause the ground state to cross with higher level energy states and the resulting eigenstate will no longer be the ground state.

Strategies for more efficient use of QAA are possible, such as decreasing the Hamiltonian evolution time, using initial excited states instead of ground states, and adding a random local Hamiltonian as an additional step [152]. These strategies have the potential to increase the success probability of QAA, but have only been tested with a small number of qubits in simulations.

Quantum Phase Estimation
The Quantum Phase Estimation introduced by Kitaev [59] is a key algorithm that finds the phase of an eigenvalue of a unitary operator U. This unitary has an eigenvector $|u\rangle$, with eigenvalue $e^{2\pi i\phi}$. The estimation is performed by using so-called black boxes (oracles) capable of preparing state $|u\rangle$ and performing controlled U^{2^j} operations [153]. By applying these a qubit register with t qubits as a control to these unitaries, the value of ϕ can be estimated. A higher number t will give a more accurate result. The QPE circuit can be found in figure B.1.

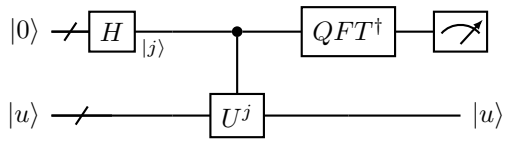


Figure B.1: Quantum Phase Estimation Circuit

Shor
A famous algorithm that uses QPE as a subroutine is the quantum factor finding algorithm, better known as Shor's algorithm, who first proposed it in 1994 [60]. Although, many others have improved the algorithm since [153]. The algorithm promises speedup over its classical counterpart to find the factors of an integer N, by finding the power r using an oracle function for an input value x. If

$x^{r/2} \neq -1 (\text{mod } N)$ and $\text{gcd}(x^{r/2} \pm 1, N)$ are non-trivial factors, then the factors of N can be found. This problem on a quantum computer takes $O(b^3)$ time, compared to the classic $O(b^k)$ time, where no algorithm is known to solve these complex problems in polynomial time.

Grover
The Grover search algorithm is used to find a certain element in an array of items [46]. The procedure classically takes $O(N)$ operations, but the quantum search algorithm requires only $O(\sqrt{N})$ [46, 153], which is a substantial speedup, certainly for large N. The algorithm consists of multiple Grover iterations. A Grover iteration can be broken down into four steps:

1. Apply the oracle function, which 'tags' the correct element in the list by shifting its phase.
2. Apply a Hadamard gate to each of the n input qubits
3. Perform a coniditonal phase shift: $|0\rangle \rightarrow |0\rangle$ and $|x\rangle \rightarrow -|x\rangle$
4. Apply a Hadamard gate again to each of the n input qubits

The number of these Grover operations scales with $O(\sqrt{N})$. The final output is then measured, where the desired element (bit-string) has the highest probability amplitude.

C

Overview of benchmarked quantum backends

Table C.1: Overview of collected data sets for each quantum computer under test for each QPack problem

		MCP		DSP		MIS		TSP		RH		IC	
	Problem size	Start	End	Start	End	Start	End	Start	End	Start	End	Start	End
local simulators	Qiskit Aer simulator	2	15	2	10	3	12	3	4	1	10	1	10
	Cirq simulator	2	15	2	10	3	12	3	4	1	10	1	10
	QuEST simulator	2	15	2	10	3	12	3	4	1	10	1	10
	Rigetti QVM	2	15	2	5	3	8	3	3	1	10	1	10
remote simulators	IBMQ QASM simulator	2	10	2	10	3	10	3	4	1	10	1	10
	IonQ simulator	2	10	2	10	3	10	3	4	1	10	1	10
	Rigetti QVM	2	13	2	5	3	7	3	3	1	10	1	10
remote hardware	IBMQ Manila (5-qubits)	2	5	-	-	-	-	-	-	1	2	-	-
	IBMQ Quito (5-qubits)	2	5	-	-	-	-	-	-	1	5	1	4
	IBMQ Nairobi (7-qubits)	2	7	-	-	-	-	-	-	1	7	1	7
	IBMQ Perth (7-qubits)	2	5	-	-	-	-	-	-	1	5	1	5
	IBMQ Jakarta (7-qubits)	2	5	-	-	-	-	-	-	1	5	1	5
	IBMQ Lagos (7-qubits)	2	5	-	-	-	-	-	-	1	5	1	5
	Rigetti Aspen (80-qubits)	-	-	-	-	-	-	-	-	1	4	-	-

D

Collected benchmark data

D.1. Local quantum simulators

The local simulators tested in this work are the Qiskit Aer [120], Cirq [126], Rigetti QVM [127] and QuEST [114] simulators. Data was collected using a Windows 10 desktop computer, utilizing an AMD Ryzen 5 3600 6-core CPU [128] with 16 GB RAM in an Ubuntu Windows Subsystem for Linux environment.

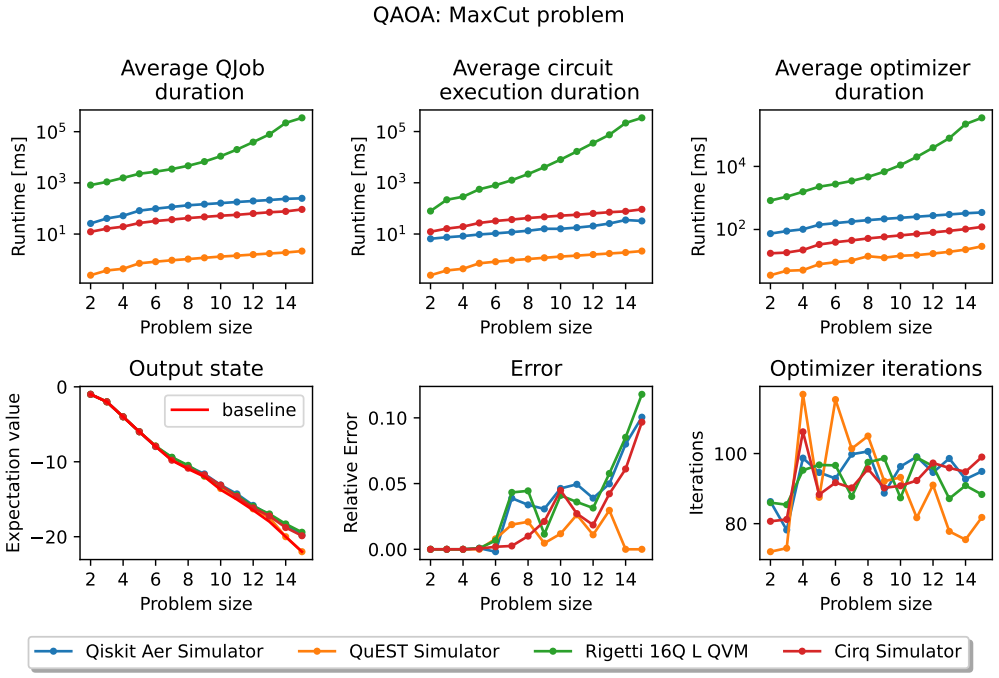


Figure D.1: Result of the MCP benchmark on local simulators

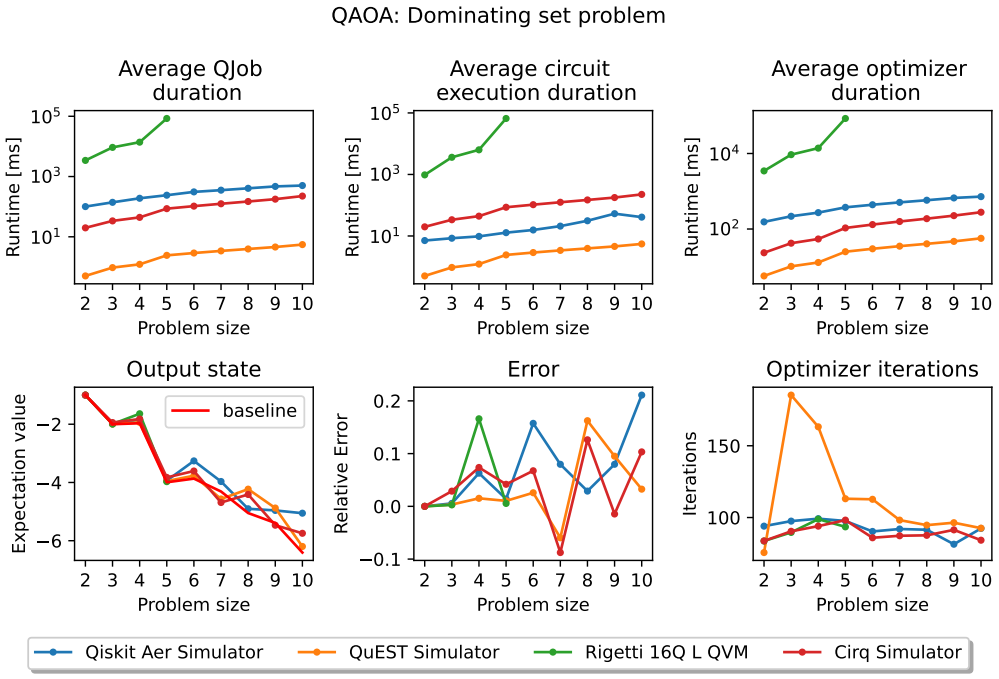


Figure D.2: Result of the DSP benchmark on local simulators

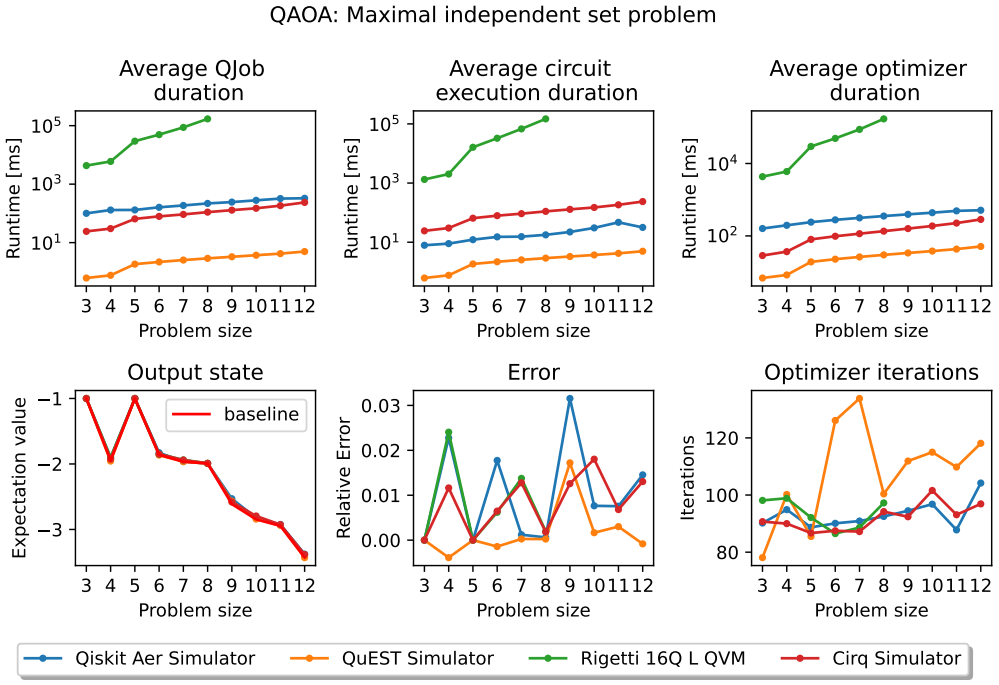


Figure D.3: Result of the MIS benchmark on local simulators

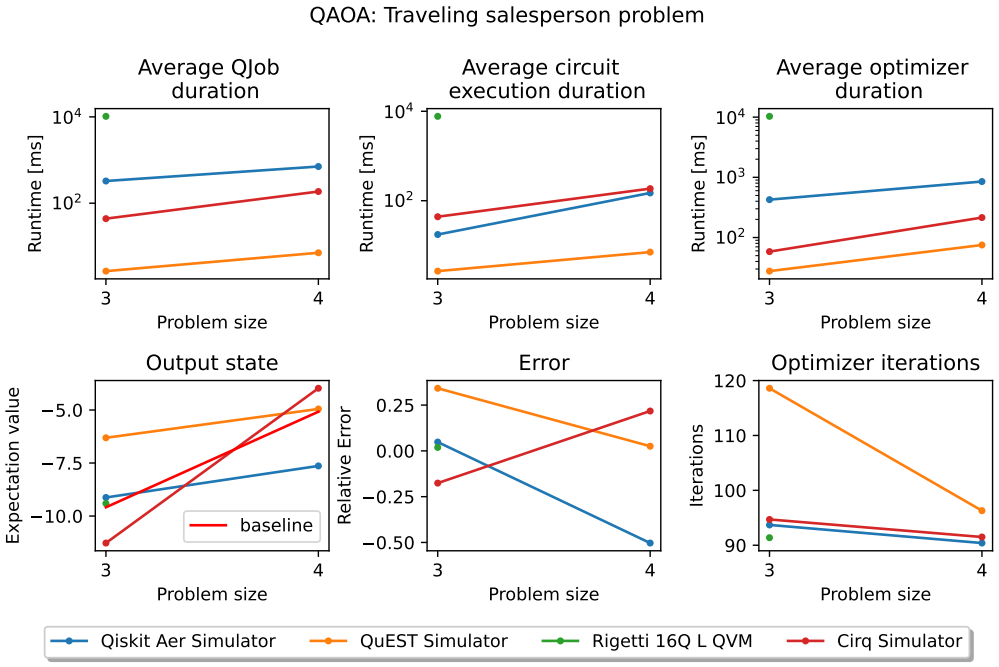


Figure D.4: Result of the TSP benchmark on local simulators

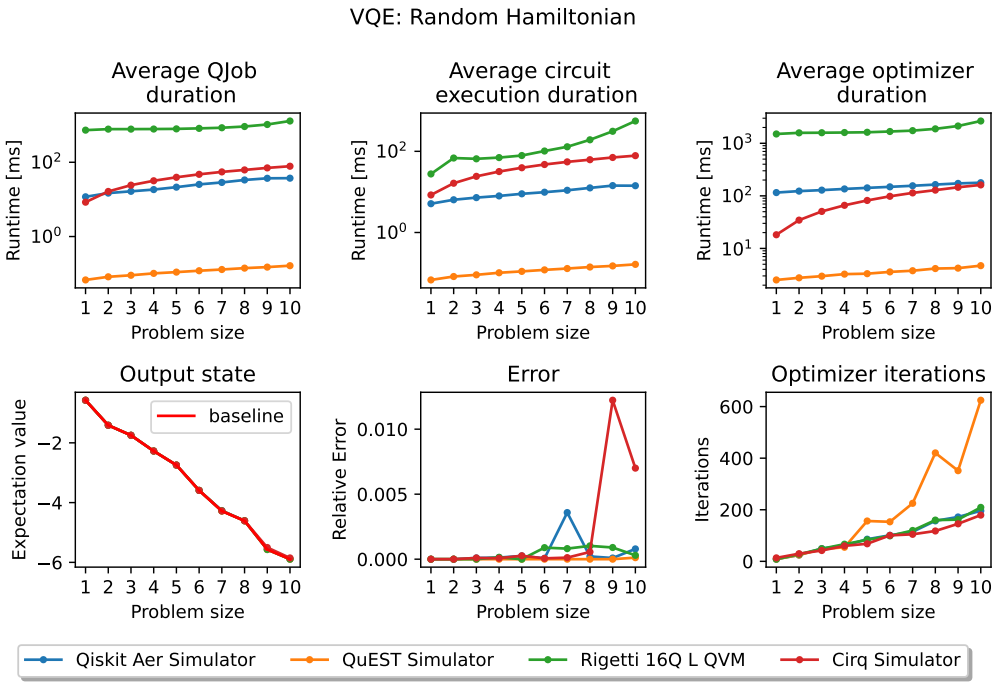


Figure D.5: Result of the RH benchmark on local simulators

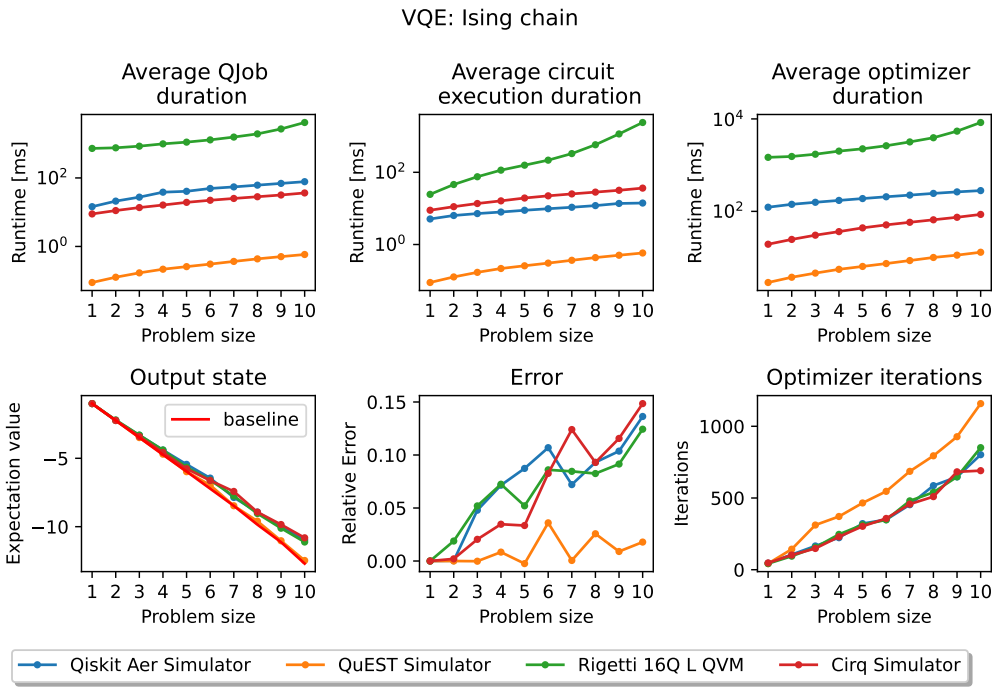


Figure D.6: Result of the IC benchmark on local simulators

D.2. Remote quantum simulators

Benchmark results for the IBMQ QASM simulator [49], IonQ simulator [138] and Rigetti QVM [127, 133].

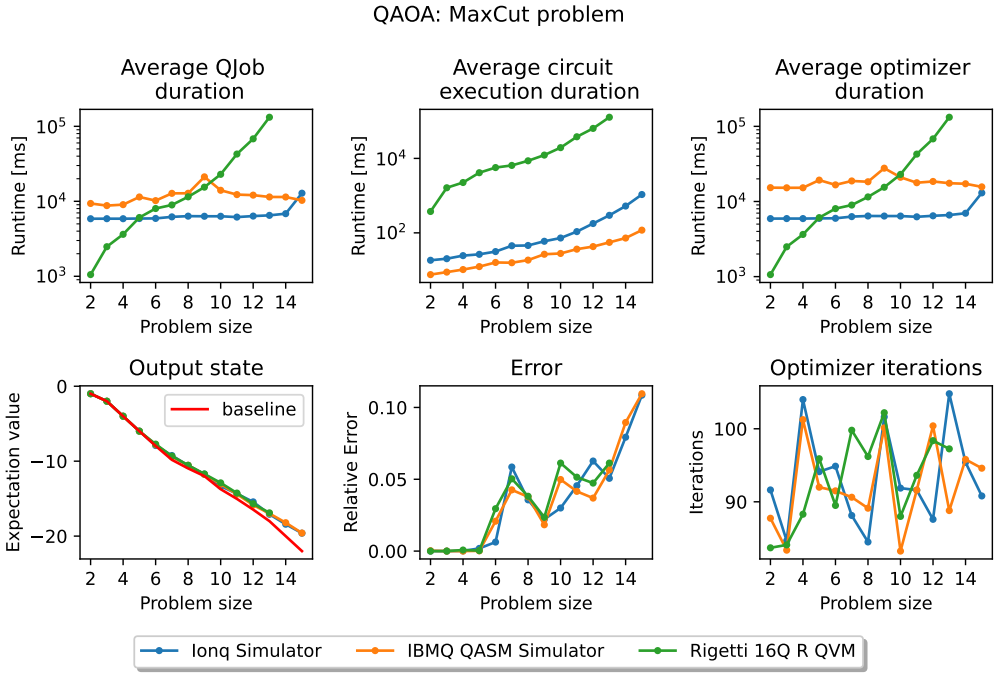


Figure D.7: Result of the MCP benchmark on remote simulators

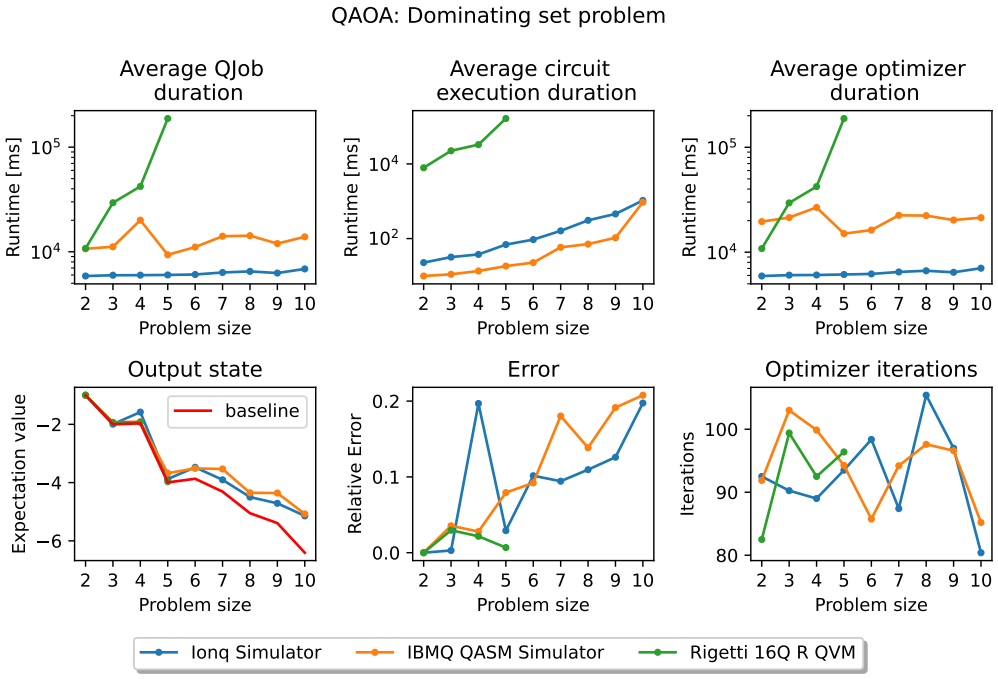


Figure D.8: Result of the DSP benchmark on remote simulators

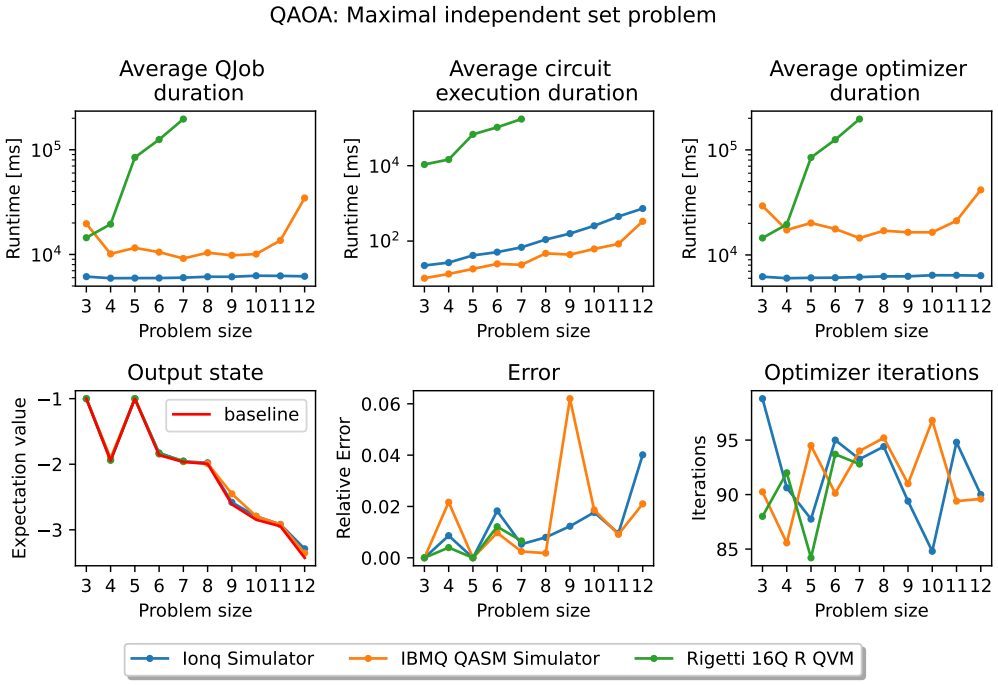


Figure D.9: Result of the MIS benchmark on remote simulators

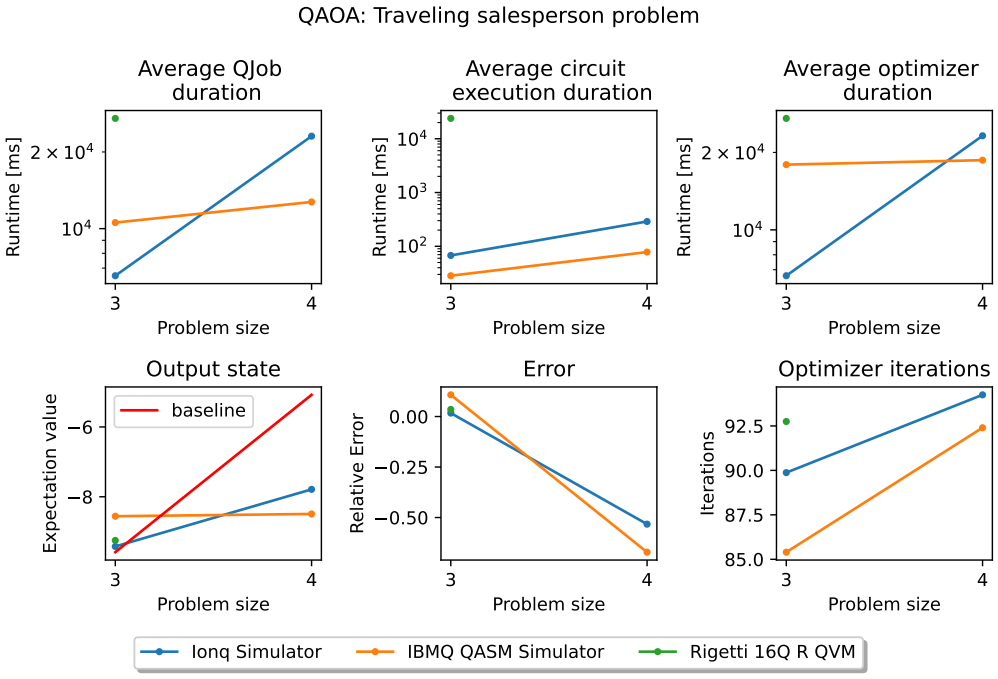


Figure D.10: Result of the TSP benchmark on remote simulators

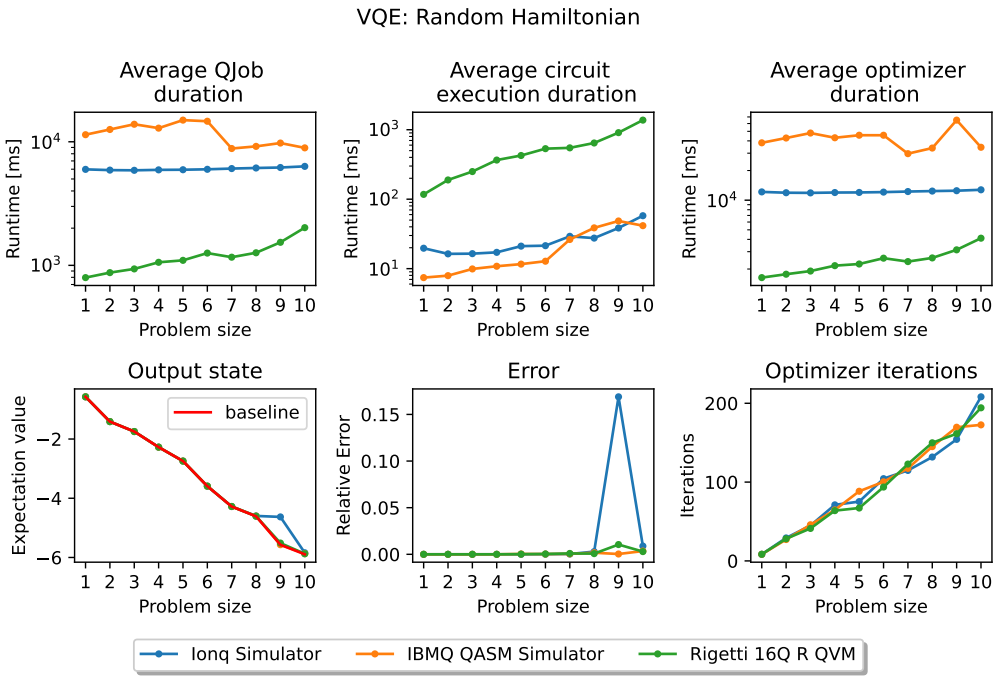


Figure D.11: Result of the RH benchmark on remote simulators

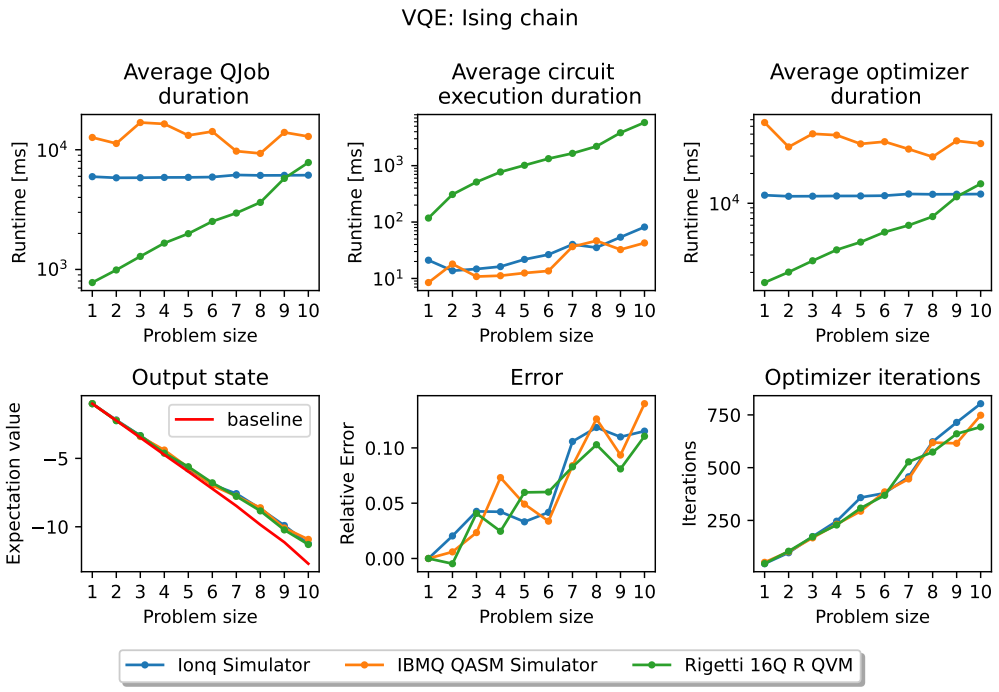


Figure D.12: Result of the IC benchmark on remote simulators

D.3. Remote quantum hardware

This section lists the results of quantum hardware from the IBMQ family [141, 50] and Rigetti [9].

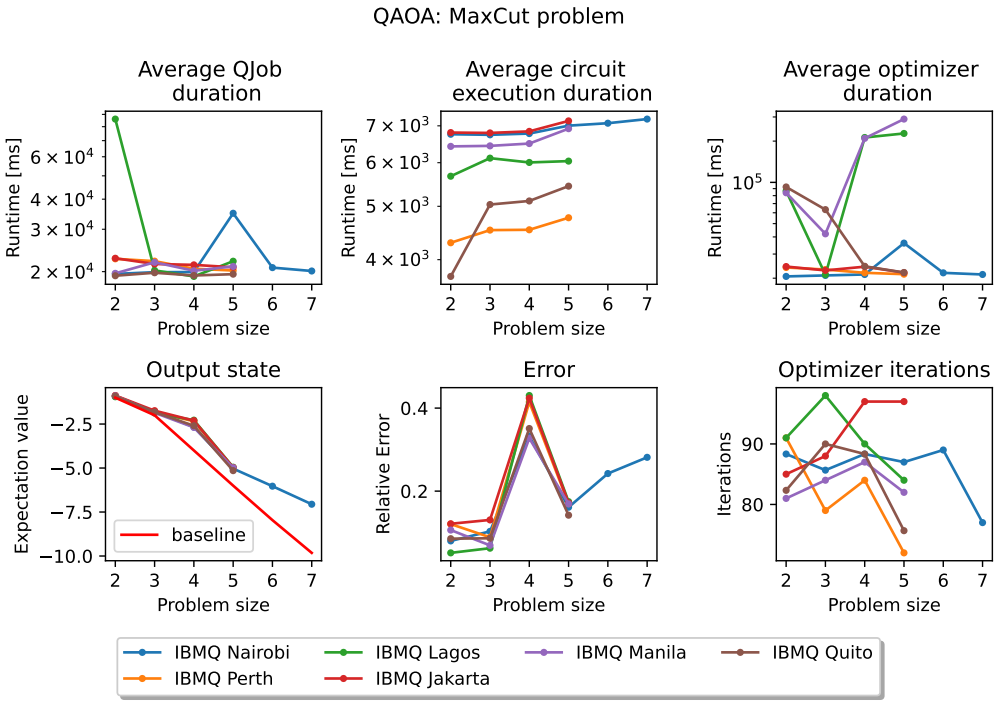


Figure D.13: Result of the MCP benchmark on remote hardware

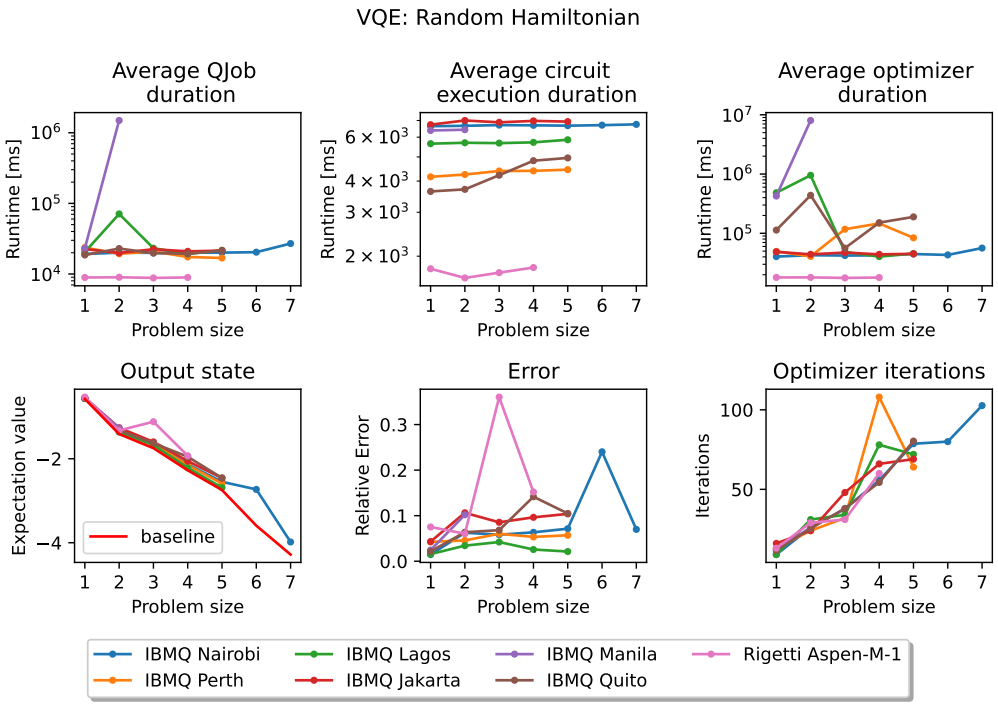


Figure D.14: Result of the RH benchmark on remote hardware

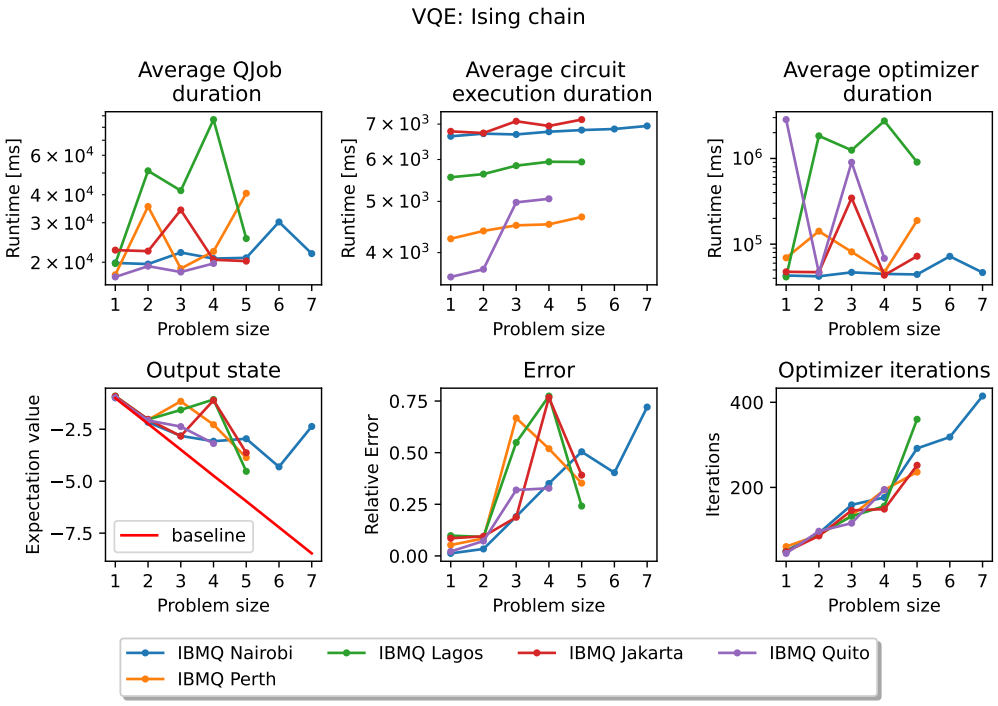


Figure D.15: Result of the IC benchmark on remote hardware

D.4. Noisy local simulators

This section lists the noisy simulation of the IMBQ Quito, Manila, Perth, Nairobi, Jakarta and, Lagos backends and the Rigetti Aspen-M-1.

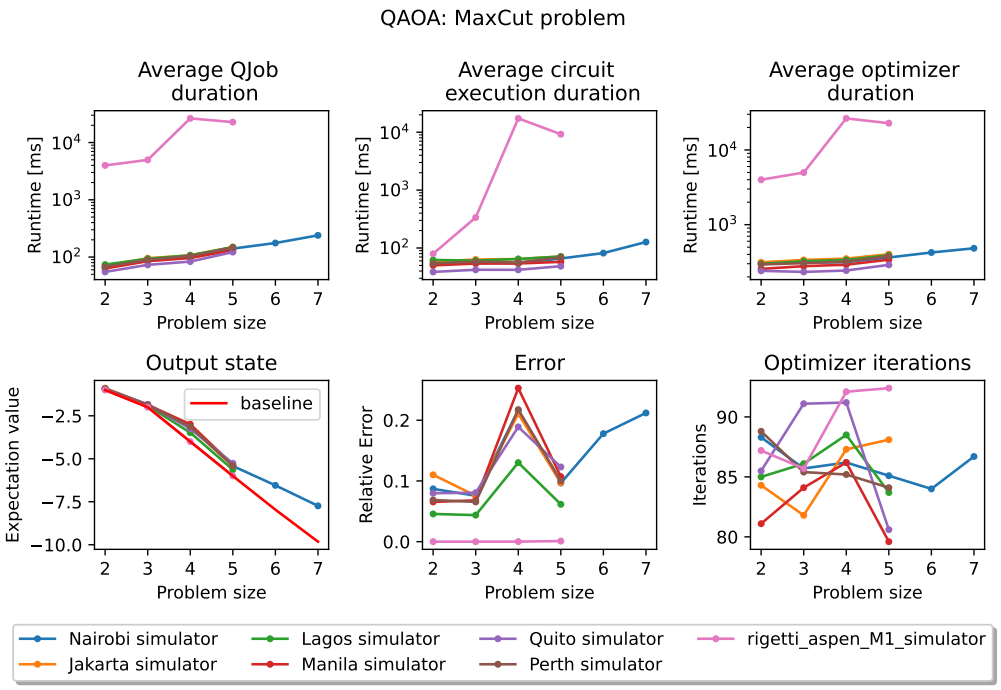


Figure D.16: Result of the MCP benchmark on simulated hardware

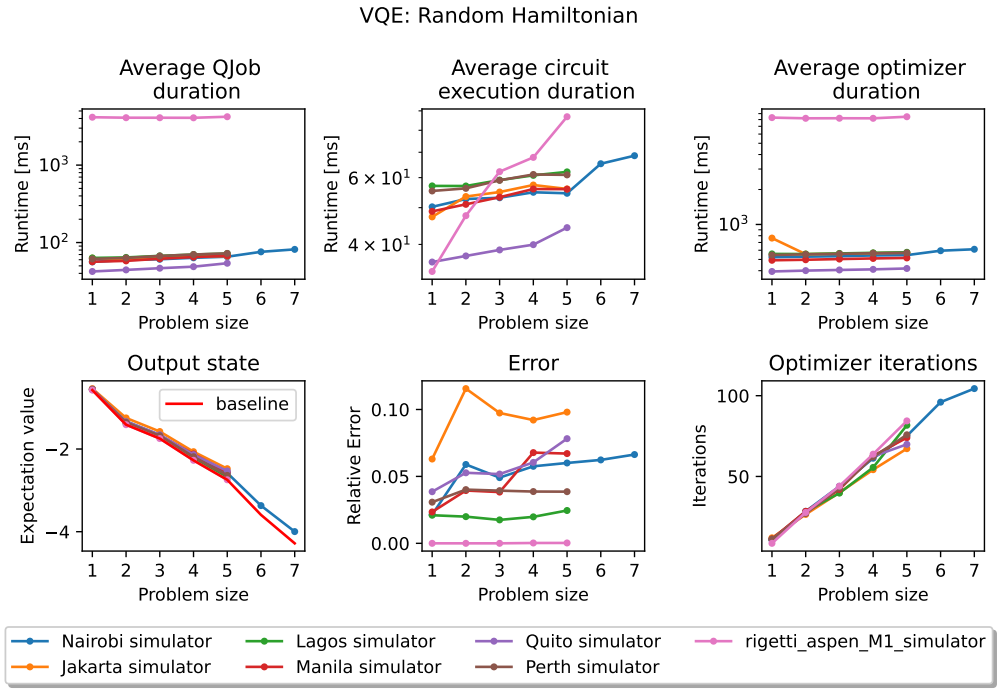


Figure D.17: Result of the RH benchmark on simulated hardware

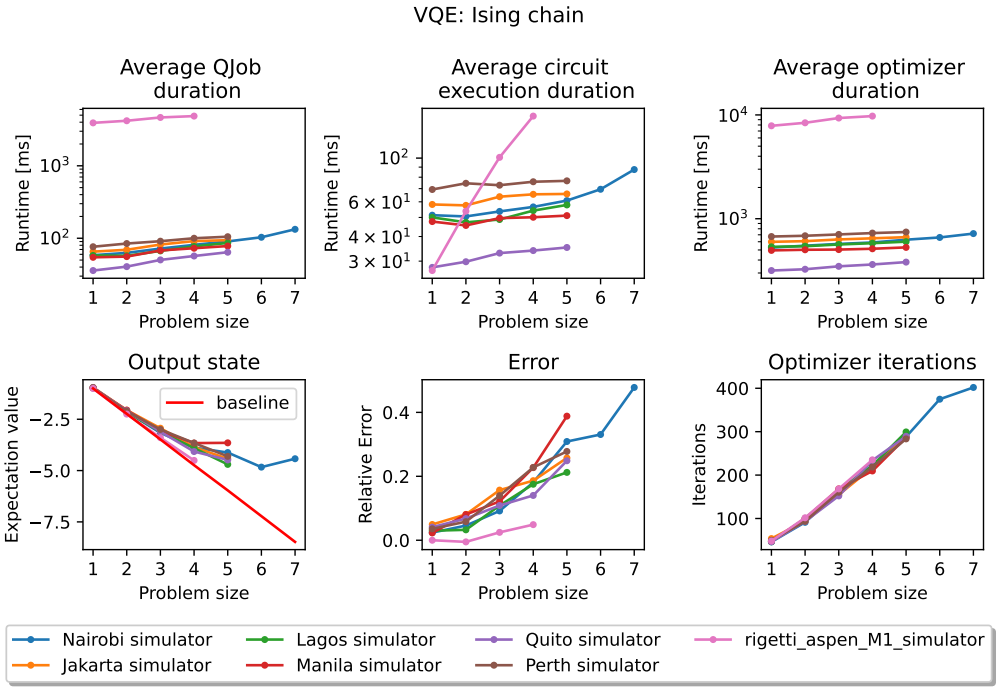


Figure D.18: Result of the IC benchmark on simulated hardware

E

|Lib⟩ Code: Data collection

```
/** @file Qpack/src/main.cpp

    @brief Qpack main source file

    @author Huub Donkers

    @defgroup qpack libket
 */

//#define OUTPUT_DIR  "/qpack/qpack_output/baseline_new/"
#define OUTPUT_DIR  "/qpack/qpack_output/benchmark_new/"
#define QUEST_ENABLE false

#include <iostream>
#include <LibKet.hpp>
#include "../include/Optimizers.hpp"
#include "../include/QAOA.hpp"
#include "../include/VQE.hpp"
#include "../include/BenchmarkLoops.hpp"

using namespace LibKet;
using namespace LibKet::circuits;
using namespace LibKet::filters;
using namespace LibKet::gates;


//Pack benchmark data
struct AlgData{

    static constexpr int reps = 10;      //Set number of repetions for VQA to run
    static constexpr int P = 3;         //Set QAOA iterations
    static constexpr int shots = 4096;  //Set QPU shots
    static constexpr vqa_problems problems[] = {MCP, DSP, MIS, TSP, RH, IC};      //Set problem sets to be evaluated
    static constexpr QDeviceType qpuIDs[] = {//QDeviceType::ibmq_qasm_simulator   //Select units under test
                                             QDeviceType::qiskit_aer_simulator
                                             //QDeviceType::ionq_simulator
                                             //QDeviceType::quest
                                             //QDeviceType::cirq_simulator
                                             //QDeviceType::rigetti_16q_simulator
                                             //QDeviceType::ibmq_nairobi,
                                             //QDeviceType::ibmq_quito
                                             //QDeviceType::ibmq_manila
                                             //QDeviceType::ibmq_jakarta
                                             //QDeviceType::ibmq_perth
                                             //QDeviceType::ibmq_lagos
                                            };

    static constexpr std::array<int, 3> graph_loop_params(vqa_problems problem){   //Select problem sizes for each problem
        std::array<int, 3> MCPparams = {2, 12, 1}; //Start, End, Step
        std::array<int, 3> MISparams = {3, 9, 1};
        std::array<int, 3> DSPparams = {2, 7, 1};
        std::array<int, 3> TSPparams = {3, 4, 1};
        std::array<int, 3> RHparams  = {1, 12, 1};
        std::array<int, 3> ICparams  = {1, 12, 1};
        std::array<int, 3> ones = {1, 1, 1};
        return problem == MCP ? MCPparams :
               problem == MIS ? MISparams :
               problem == DSP ? DSPparams :
               problem == TSP ? TSPparams :
               problem == RH ? RHparams :
```

```
                problem == IC ? ICparams : ones;
        };
} algData;


int main(int argc, char *argv[]) {


    //Determine problem sizes
    constexpr auto problemSize = sizeof(AlgData::problems)/sizeof(vqa_problems);

    //Run Qpack for selected QPUs
    int status = utils::static_for<0, problemSize-1, 1, benchmark_loops::problem_loop>(0, algData);

    //Finishing message
    std::cout << "==============================\n"
              << "                                \n"
              << " ─┤ Q ├─■─┤ Pack ├──────── \n"
              << "                                \n"
              << "                               \n"
              << " ─┤Benchmarking Completed├─! \n"
              << "                               \n"
              << "==============================\n"
              << std::flush;

    return 0;
}
```

Listing E.1: main.cpp code of the QPack benchmark in LibKet.

```
/** @file Qpack/include/BenchmarkLoops.hpp

 @brief Qpack BenchmarkLoops header file

 @author Huub Donkers

 @defgroup qpack libket
*/

#pragma once
#ifndef BENCHMARKLOOPS_HPP
#define BENCHMARKLOOPS_HPP

#include "../include/Optimizers.hpp"
#include "../include/QAOA.hpp"
#include "../include/VQE.hpp"


namespace benchmark_loops{

   //Struct to pass constexpressions in function parameters (indices in this case)
    template<int index>
    struct IndexPass{
     static constexpr const int getIndex(){return index;}
    };

    //Function to create multiple from path
    void mkdirs(const char* path){

     //Decompose path in a folder vector
     int index = 0;
     std::vector<int> dir_index = {};
     while(path[index] != '\0'){
      if(path[index] == '/'){
       dir_index.push_back(index);
    }
    index++;
     }

     //Loop over path, add subfolder each loop
    for(int i = 0; i < dir_index.size()-1; i++){
     char output_path[256] = "";
     char sub_path[256] = "";

     memcpy(sub_path, &path[0], dir_index[i+1]);

     std::strcat(output_path, std::getenv("HOME"));
     std::strcat(output_path, sub_path);

      if(mkdir(output_path, 0777) != 0){
         char buffer[ 256 ];
         char * errorMsg = strerror_r( errno, buffer, 256 );    // GNU-specific version, Linux default
         if(std::string(errorMsg).compare("File exists") != 0)
```

```
            std::cout << "Error: " << errorMsg << std::endl;   //return value has to be used since buffer might not be modified
     }
    }
   }

 //Repeat QAOA measurements
   template<typename Optimizer>
   void repeat_execution(Optimizer&& optimizer, int reps, const char* output_path){

    //Create ouput folder to save results
    mkdirs(output_path);

    //Execute VQA repeatedly
    for(int index=0; index < reps; index++){

     //Create JSON object to store results
     nlohmann::json result;

     //Generate output file path
     char output_file[256] = "";
     std::strcat(output_file, std::getenv("HOME"));
     std::strcat(output_file, output_path);
     std::strcat(output_file, "dataset");
     std::strcat(output_file, index < 10 ? ("0"+std::to_string(index)).c_str() : std::to_string(index).c_str());
     std::strcat(output_file, ".json");

     //Execution message
      std::string message = "(" + std::to_string(index + 1) + "/" + std::to_string(reps) + ")";
      std::cout << message << std::flush;
      for(int i = 0; i < message.size(); i++){
       std::cout << "\b";
      }

     //Check if data is already available
     if(access(output_file, F_OK) != 0){

      //Reset optimizer data
      optimizer.resetData();

      //Run optimizer
      arma::vec opt_params = optimizer.COBYLA();

      //Store optimizer results in JSON format
      result = optimizer.getData();

    //Save result in file
      std::ofstream output(output_file);
      output << std::setw(2) << result << std::endl;

     }
    }
   }


   //Loop over graph sizes
   template<index_t start, index_t end, index_t step, index_t index>
   struct graphSize_loop
   {
     template<typename ProblemIndex, typename QpuIndex, typename AlgData>
     inline constexpr auto operator()(int status, ProblemIndex, QpuIndex, AlgData) noexcept
     {

     //Setup QAOA object
    constexpr int problemSize = index;
    constexpr int P = AlgData::P;
    constexpr int shots = AlgData::shots;
    constexpr vqa_problems problem = AlgData::problems[ProblemIndex::getIndex()];
    constexpr QDeviceType qpuID = AlgData::qpuIDs[QpuIndex::getIndex()];
    constexpr int repetitions = AlgData::reps;

    //Execution message
    std::cout << "\t\tProblem Size: " << std::to_string(problemSize) << " " << std::flush;

    //Set name buffers for this problem, concatenate
    char qpu_buffer [16] = {};
    char size_buffer [8] = {};
    char p_buffer [8] = {};
    char shots_buffer [8] = {};
    std::sprintf(qpu_buffer, "%X", static_cast<int>(qpuID));
    std::sprintf(size_buffer, "%d", problemSize);
    std::sprintf(p_buffer, "%d", P);
    std::sprintf(shots_buffer, "%d", shots);
```

```
136
137     //Concat ouput path
138     char output_path[256] = "";
139     std::strcat(output_path, OUTPUT_DIR);
140     std::strcat(output_path, qpu_buffer);
141     std::strcat(output_path, "/");
142     std::strcat(output_path, vqa_names[problem].c_str());
143     std::strcat(output_path, "/N");
144     if(problemSize < 10){
145      std::strcat(output_path, "0");
146     }
147     std::strcat(output_path, size_buffer);
148     std::strcat(output_path, "/");
149
150     //Select QAOA or VQE problem optimizer and repeat execution
151     switch(problem){
152
153      //QAOA
154      case MCP :
155      case MIS :
156      case DSP :
157      case TSP :
158       {
159       QAOA<problemSize, problem, P, shots, qpuID> qaoa;
160        Optimizer<QAOA<problemSize, problem, P, shots, qpuID>> qaoa_optimizer(&qaoa);
161
162        //Repeat execution
163        repeat_execution(qaoa_optimizer, repetitions, output_path);
164
165         }
166       break;
167
168      //VQE
169      case RH :
170      case IC :
171       {
172     VQE<problem, problemSize, shots, qpuID> vqe;
173       Optimizer<VQE<problem, problemSize, shots, qpuID>> vqe_optimizer(&vqe);
174
175
176       //Repeat execution
177       repeat_execution(vqe_optimizer, repetitions, output_path);
178         }
179       break;
180
181      //Unknown problem selected
182      default:
183       std::cout << "QPack error: Unknown problem!" << std::endl;
184
185     }
186
187    //Message signalling all repetions have been completed
188     std::cout << "DONE          " << std::endl;
189
190     //Return status
191     return 1;
192
193      }
194    };
195
196    //Loop over graph sizes
197    template<index_t start, index_t end, index_t step, index_t index>
198    struct qpu_loop
199    {
200      template<typename ProblemIndex, typename AlgData>
201      inline constexpr auto operator()(int status, ProblemIndex, AlgData) noexcept
202      {
203       //Set current qpu loop index
204     IndexPass<index> qpuIndex;
205
206     //Print execution message
207     std::cout << "\tQPU: " << AlgData::qpuIDs[index] << std::endl;
208
209      //Get graph loop parameters bases on problem set
210      constexpr auto graph_loop_params = AlgData::graph_loop_params(AlgData::problems[ProblemIndex::
    getIndex()]);
211
212     //Loop over graph sizes
213     int graph_status = utils::static_for<graph_loop_params[0],
214                 graph_loop_params[1],
215                 graph_loop_params[2],
216                 benchmark_loops::graphSize_loop>(0, ProblemIndex{}, qpuIndex, AlgData{});
217
218     //Return status
```

```
    return 1;

     }
   };

   //Loop over graph sizes
   template<index_t start, index_t end, index_t step, index_t index>
   struct problem_loop
   {
     template<typename AlgData>
     inline constexpr auto operator()(int status, AlgData) noexcept
     {

     //Set current problem loop index
    IndexPass<index> problemIndex;

    //Starting message
    switch(AlgData::problems[index]){
     case MCP :
     case MIS :
     case DSP :
     case TSP :
      std::cout << "Running QAOA " << vqa_names[AlgData::problems[index]] << ":" << std::endl;
      break;

     case RH :
     case IC :
      std::cout << "Running VQE " << vqa_names[AlgData::problems[index]] << ":" << std::endl;
      break;

     default:
      std::cout << "Unknown Algorithm!" << std::endl;
    }


    //Loop over problem set
    constexpr auto qpuSize = sizeof(AlgData::qpuIDs)/sizeof(QDeviceType);
    int qpu_status = utils::static_for<0,
                qpuSize - 1,
                1,
                benchmark_loops::qpu_loop>(0, problemIndex, AlgData{});

    //Return status
    return 1;

     }
   };
}

#endif //BENCHMARKLOOPS_HPP
```

Listing E.2: BenchmarkLoops.hpp code of the QPack benchmark in LibKet.

```
/** @file Qpack/include/Optimizers.hpp

 @brief Qpack Optimizers header file

 @author Huub Donkers

 @defgroup qpack libket
*/

#pragma once
#ifndef OPTIMIZER_HPP
#define OPTIMIZER_HPP

#define OPTIM_ENABLE_ARMA_WRAPPERS
#include "../include/QAOA.hpp"
#include "../include/VQE.hpp"
#include <math.h>
#include <iomanip>
#include <iostream>
#include <vector>
#include <chrono>
#include <nlopt.hpp>

//Optimizer class which runs a VQA circuit and uses an optimizer to find the minimum score
template<typename vqaType>
class Optimizer{
private:
 double _minf;
 std::vector<double> _opt_params;
 double _totalDuration; //Total optimizer duration in seconds
```

```

 struct OptData{ //VQA
  vqaType* _algorithm_ptr;
  std::vector<double> _execDurations;
  std::vector<double> _queueDurations;
  std::vector<double> _jobDurations;
  std::vector<double> _optDurations;
  std::chrono::steady_clock::time_point _itTime;
 } optData;

//NLOpt optimization function
static double vqa_opt(const std::vector<double> &params, std::vector<double> &grad, void *func_data)
{
    if (!grad.empty()) {
        grad[0] = 0.0;
        grad[1] = 0.5 / sqrt(params[1]);
    }

 //Recast data
 OptData* optData = (OptData*) func_data;
 auto vqa = optData->_algorithm_ptr;

 //Execute quantum circuit
 vqa->run(params, false);

 //Retrieve expectation value
 double score = vqa->getExpectation();

 //Append Q-job time to data struct
 optData->_execDurations.push_back(vqa->getExecDuration()*1000); //Convert s to ms
 optData->_jobDurations.push_back(vqa->getJobDuration()*1000);   //Convert s to ms
 optData->_queueDurations.push_back(vqa->getQueueDuration()*1000);   //Convert s to ms

 //Append iteration ducatrion to data struct
 optData->_optDurations.push_back((double)std::chrono::duration_cast<std::chrono::microseconds>(std::
     chrono::steady_clock::now() -optData->_itTime).count()/1000); // us to ms
 optData->_itTime = std::chrono::steady_clock::now(); //Reset startime

 return score;
}

public:
 //Constructor
 template<typename algType>
 Optimizer(algType algorithm_ptr){
  optData._algorithm_ptr = algorithm_ptr;
 }

 //Reset data
 void resetData(){
  optData._jobDurations.clear();
  optData._optDurations.clear();
  optData._execDurations.clear();
  optData._queueDurations.clear();
  _totalDuration = 0.0;
  _minf = 0.0;
  _opt_params.clear();
 }

 //Return optimizer iterations
 unsigned long getIterations(){
  return optData._jobDurations.size();
 }

 //Return vector of circuit execution durations in ms
 std::vector<double> getQueueDurations(){
  return optData._queueDurations;
 }

 //Return vector of circuit execution durations in ms
 std::vector<double> getExecDurations(){
  return optData._execDurations;
 }

 //Return vector of qjob durations in ms
 std::vector<double> getJobDurations(){
  return optData._jobDurations;
 }

 //Return vector of optimizer durations in ms
 std::vector<double> getOptDurations(){
  return optData._optDurations;
 }

```

```
//Return total job duration in ms
double getTotalJobDuration(){
 double totalJobDuration = 0.0;
    for(double i : optData._jobDurations){
           totalJobDuration += i;
    }
 return totalJobDuration;
}

//Return optimal score
double getOptimalScore(){
 return optData._algorithm_ptr->getOptimalScore();
}

//Return average job duration in ms
double getAverageJobDuration(){
 return getTotalJobDuration()/optData._jobDurations.size();
}

//Return number of qubits
int getNumQBits(){
 return optData._algorithm_ptr->getNumQubits();
}

//Return (average) circuit depth
double getDepth(){
 return optData._algorithm_ptr->getCircuitDepth();
}


//Return all data
  nlohmann::json getData(){
   char qpu_buffer [16] = {};
    std::sprintf(qpu_buffer, "%X", static_cast<int>(optData._algorithm_ptr->getQpuID()));

    nlohmann::json data;
    data["QPU"] = qpu_buffer;
    data["Problem"] = vqa_names[optData._algorithm_ptr->getProblem()];
    data["Size"] = optData._algorithm_ptr->getSize();
    data["P"] = optData._algorithm_ptr->getP();
    data["Shots"] = optData._algorithm_ptr->getNumShots();
    data["Qubits"] = getNumQBits();
    data["Depth"] = getDepth();
    data["Optimizer iterations"] = getIterations();
    data["QJob durations [ms]"] = getJobDurations();
    data["Queue durations [ms]"] = getQueueDurations();
    data["Circuit execution durations [ms]"] = getExecDurations();
    data["Expectation Value"] = _minf;
    data["Optimal Expectation Value"] = getOptimalScore();
    data["Optimizer params"] = _opt_params;
    data["Optimizer durations [ms]"] = getOptDurations();
    data["Total Algorithm duration [s]"] = _totalDuration;
    data["Total Quantum duration [s]"] = getTotalJobDuration()/1000; // ms to s
    data["Total Classic duration [s]"] = _totalDuration - getTotalJobDuration()/1000; //ms to s

    return data;
  }

//Run VQA with COBYLA
arma::vec COBYLA(){

 //NLopt optimizer initialisation
 int numParams;
 std::vector<double> lb;
 std::vector<double> ub;
 std::vector<double> params;
 float tol;

 //Fill in parameters based on algorithm
 switch(optData._algorithm_ptr->id){
  case 0 : //QAOA
      numParams = optData._algorithm_ptr->getNumParams();
      for(int i=0; i < numParams; i++){
       lb.push_back(0.0);   //Set lower bounds to 0.0
       ub.push_back(1.0);   //Set upper bounds to 1.0
       float r = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
       params.push_back(r);  //Set random initial parameters
       tol = 1e-4;    //Set tolerance
      }
   break;

  case 1 : //VQE
   numParams = optData._algorithm_ptr->getNumParams();
      for(int i=0; i < numParams; i++){
```

```
        lb.push_back(0.0);   //Set lower bounds to 0.0
        ub.push_back(1.0);   //Set upper bounds to 1.0
        float r = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
        params.push_back(r);  //Set random initial parameters
        tol = 1e-3;    //Set tolerance
       }
    break;

   default :
    std::cout << "Algorithm not recognized" << std::endl;
  }

  //Set parameters
  nlopt::opt opt(nlopt::LN_COBYLA, numParams);
  opt.set_lower_bounds(lb);
  opt.set_upper_bounds(ub);
  opt.set_min_objective(vqa_opt, &optData);
  opt.set_xtol_rel(tol);


  try{
   //Set timer and begin optimization
   std::chrono::steady_clock::time_point startTime = std::chrono::steady_clock::now();
   optData._itTime = startTime;
      nlopt::result result = opt.optimize(params, _minf);
      _opt_params = params;
      _totalDuration = (double)std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::
     steady_clock::now() - startTime).count()/1000; //ms to s
  }
  catch(std::exception &e) {
      std::cout << "nlopt failed: " << e.what() << std::endl;
  }

     return params;
 }
};

#endif //OPTIMIZER_HPP
```

Listing E.3: Optimizers.hpp code of the QPack benchmark in LibKet.

```
/** @file Qpack/include/VQA.hpp

 @brief Qpack VQA header file

 @author Huub Donkers

 @defgroup qpack libket
*/

#pragma once
#ifndef VQA_HPP
#define VQA_HPP

/*
@brief: Base class for the QAOA and VQE classes. Holds shared variables and return fuctions
  for Variatonal Quantum Algorithms in Qpack
*/

/*
Declare VQA problems and their names:
 MaxCut problem (MCP): Find maximum cuts for bipartioned graph
 Dominating Set Problem (DSP): Find minium number of survailance nodes in a network
 Maximum Independent Set problem (MIS): Find maximum set of disconnected nodes
 Traveling Salesperson problem (TSP): Find shortest path in network
 Random Haltonian (RH): A diagonal hamiltonian with random coefficients
  Ising Chain (IC): Simulations of the 1-D Ising chain
*/
enum vqa_problems {MCP, DSP, TSP, MIS, RH, IC};
std::vector<std::string> vqa_names = {"MCP", "DSP", "TSP", "MIS", "RH", "IC"};

class VQA
{
protected:
 //Variables to store qpu results
 int _numParams;
 double _execDuration;
 double _queueDuration;
 double _jobDuration;
 double _expectation;

public:
 //Constants
```

```
 int id = 9999;
 std::string name = "Generic VQA";

 //Constructor
 VQA(){};

 //Placeholder functions
 void run(std::vector<double> params, bool printCircuit){};

 //Return number of parameters
 int getNumParams(){
  return _numParams;
 };

 auto getExecDuration(){
  return _execDuration;
 }

 //Return job duration
 auto getJobDuration(){
  return _jobDuration;
 }

 //Return job duration
 auto getQueueDuration(){
  return _queueDuration;
 }

 //Return expectation value for problem based on graph and qpu results
 auto getExpectation(){
     return _expectation;
 }
};

#endif //VQA_HPP
```

Listing E.4: VQA.hpp code of the QPack benchmark in LibKet.

```
/** @file Qpack/include/CircuitsQAOA.hpp

    @brief Qpack networks class header file

    @author Huub Donkers

    @defgroup qpack libket
 */

#pragma once
#ifndef QAOA_HPP
#define QAOA_HPP

#include<cmath>
#include <LibKet.hpp>
#include "../include/QAOA_Loops.hpp"
#include "../include/VQA.hpp"

using namespace LibKet;
using namespace LibKet::circuits;
using namespace LibKet::filters;
using namespace LibKet::gates;

/*
@brief: This class implements all necessary functionalities of the QAOA algorithm, such
        as determining the amount of qubits and statespace for a given problem. The class
        is able to generate QAOA circuits, execute these circuits and give an expectation
        value for each problem.
*/
template <int graphSize, vqa_problems problem, int P, int shots, QDeviceType qpuID>
class QAOA : public VQA{
public:

  //Return QPU ID
  static QDeviceType getQpuID(){
    return qpuID;
  }

  //Return problem
  static vqa_problems getProblem(){
    return problem;
  }

  //Return number of shots
  static int getNumShots(){
```

```
    return shots;
  }

  //Return P for QAOA
  static int getP(){
    return P;
  }

  //Return problem size
  static int getSize(){
    return graphSize;
  }


  //Returns the number of qubits based on selected problem
  static constexpr auto getNumQubits(){
  return problem == MCP ? graphSize   :
         problem == DSP ? (graphSize == 2 ? 4 :
                           graphSize == 3 ? 6 :
                           graphSize == 4 ? 7 :
                           graphSize+5
                          ) :
         problem == TSP ? graphSize*graphSize :
         problem == MIS ? (graphSize == 3 ? 4 :
                           graphSize == 4 ? 5 :
                           graphSize+3
                          ) :  0;
  }

private:

  //Returns the size of the histogram based on selected problem
  static constexpr auto getHistSize(){
  return problem == MCP ? 2 << (graphSize-1) :
         problem == DSP ? 2 << (graphSize-1) :
         problem == TSP ? 2 << (graphSize*graphSize-1) :
         problem == MIS ? 2 << (graphSize-1) : 1;
  }

  //Variable to store qpu result
  QArray<getHistSize(), unsigned long> _hist;

  //Function to generate maxcut QAOA circuit. Returns quantum expression
  template <typename graphType, typename paramType>
  auto genCircuitMCP(graphType graph, paramType params){

    //Set inital state to superposition
    auto s0 = h(range<0,graphSize-1>(init()));

    //QAOA iterations
    auto step = utils::static_for<0, P-1, 1, QAOAGates::maxcut_step>(tag<0>(s0), graph, params);

    return step;
  }

  //Function to generate DSP QAOA circuit. Returns quantum expression
  template <typename graphType, typename paramType>
  auto genCircuitDSP(graphType graph, paramType params){

    //Set inital state to superposition
    auto s0 = all(h(range<0,graphSize-1>(init())));

    //QAOA iterations
    auto step = utils::static_for<0, P-1, 1, QAOAGates::dsp_step>(tag<1>(s0), graph, params);

    return step;
  }

  //Static for loop over graph edges for DMatrix construction.
  template<index_t start, index_t end, index_t step, index_t index>
  struct D_Matrix_Loop
  {
    template<typename DMatrix, typename Graph>
    inline constexpr auto operator()(DMatrix&& dmatrix, Graph) noexcept
    {
      //Get link
      int i_index = Graph::template from<index>();
      int j_index = Graph::template to<index>();

      //Copy matrix information
      std::array<std::array<int, graphSize>, graphSize> new_matrix = {0};
      for(int i = 0; i < graphSize; i++){
       for(int j = 0; j < graphSize; j++){
        new_matrix[i][j] = dmatrix[i][j];
```

```
      }
    }

    //Set distance in matrix for connected links
    new_matrix[i_index][j_index] = 1;
    new_matrix[j_index][i_index] = 1;

    return new_matrix;
  }
};

//Function to generate distance matrix for TSP. Returns 2D array
template <typename graphType>
auto genDMatrix(graphType graph){
  std::array<std::array<int, graphSize>, graphSize> init_matrix;
  for(int i = 0; i < graphSize; i++){
    for(int j = 0; j < graphSize; j++){
      if(i == j){
        init_matrix[i][j] = 20;    //Diagonal elements have higher penalty
      }else{
        init_matrix[i][j] = 10;    //All items have default penalty
      }

    }
  }
  auto matrix = utils::static_for<0, graph.size()-1, 1, D_Matrix_Loop>(init_matrix, graph);

  return matrix;
}

//Function to generate TSP QAOA circuit. Returns quantum expression
template <typename graphType, typename paramType>
auto genCircuitTSP(graphType graph, paramType params){

  //Set inital state to superposition
  auto s0 = utils::static_for<0, graphSize*graphSize-1, graphSize, QAOAGates::tsp_init>(tag<2>(init()));

  //QAOA iterations
  auto fullGraph = utils::make_full_graph<graphSize>();
  std::array<std::array<int, graphSize>, graphSize> D_matrix = genDMatrix(graph);

  auto step = utils::static_for<0, P-1, 1, QAOAGates::tsp_step>(tag<2>(s0), graph, fullGraph, D_matrix,
   params);

  return step;
}

//Function to generate MIS QAOA circuit. Returns quantum expression
template <typename graphType, typename paramType>
auto genCircuitMIS(graphType graph, paramType params){

  //Set inital state to all zeros
  auto s0 = init();

  //QAOA iterations
  auto step = utils::static_for<0, P-1, 1, QAOAGates::mis_step>(tag<3>(s0), graph,  params);

  return step;
}

//Loop to determine number of shared edges
template<index_t start, index_t end, index_t step, index_t index>
struct shared_edge_loop
{
  template<typename Shared, typename Graph, typename BitString>
  inline constexpr auto operator()(Shared&& shared, Graph, BitString bitString) noexcept
  {
    if(bitString[Graph::template from<index>()] != bitString[Graph::template to<index>()]){
      shared -= 1;
    }

    return shared;
  }
};

//Computes expectation value for MCP problem
template <typename graphType, typename histType>
float expectationMCP(graphType graph, histType hist) {

  static constexpr const std::size_t numEdges = graph.size();
  static constexpr const std::size_t numStates = hist.size();
  static constexpr const std::size_t numBits = graphSize;

  int average = 0;
```

```
    int total_count = 0;
    //Loop over all output states
    for(int i = 0; i < numStates; i++){

      //Convert state number to bitstring
      std::string bitString = std::bitset<numBits>(i).to_string();

      //Loop over all egdes
      int shared = 0;
      int shared_edges = utils::static_for<0, numEdges-1, 1, shared_edge_loop>(shared, graph, bitString);

      //Sum shared edges with histogram weight
      average += shared_edges * hist[i];
      total_count += hist[i];
    }

    //Return average result
    return (float)average / (float)total_count;
  }

  //Static for loop for expectation function
  template<index_t start, index_t end, index_t step, index_t index>
  struct T_loop
  {
    template<typename TBits, typename Graph, typename Bits>
    inline constexpr auto operator()(TBits&& tBits, Graph, Bits bits) noexcept
    {
      int e0 = Graph::template from<index>();
      int e1 = Graph::template to<index>();
      int end_index = graphSize-1;

      //index reverse because of bitsting indexing
      if(bits[end_index - e0] == 1 or bits[end_index - e1] == 1){
        tBits[end_index - e0] = 1;
        tBits[end_index - e1] = 1;
      }

      return tBits;
    }
  };

  //Computes expectation value for DSP problem
  template <typename graphType, typename histType>
  float expectationDSP(graphType graph, histType hist) {

    static constexpr const std::size_t numEdges = graph.size();
    static constexpr const std::size_t numStates = hist.size();

    int average = 0;
    int total_count = 0;
    //Loop over all output states
    for(int i = 0; i < numStates; i++){

      //Convert state number to bitstring
      std::bitset<graphSize> bits(i);

      //Cost of this state
      int cost = 0;
      std::bitset<graphSize> tBits(0);
      tBits = utils::static_for<0, numEdges-1, 1, T_loop>(tBits, graph, bits);

      //Check if solution monitors all nodes
      bool valid = true;
      for(int j = 0; j < graphSize; j++){
        if(!tBits[j]){
          valid = false;
        }
      }

      //If solution is valid, compute solution cost
      if(valid){
        for(int j = 0; j < graphSize; j++){
          cost -= !bits[j];  //Subtract to get best score be most negative
        }
      }

      //Sum cost with histogram weight
      average += cost * hist[i];
      total_count += hist[i];
    }

    //Return average result
    return (float)average / (float)total_count;
  }
```

```
297
298 //Computes expectation value for TSP problem
299 template <typename graphType, typename histType>
300 float expectationTSP(graphType graph, histType hist) {
301
302   int average = 0;
303   int total_count = 0;
304   static constexpr const std::size_t numStates = hist.size();
305   std::array<std::array<int, graphSize>, graphSize> D_matrix = genDMatrix(graph);
306
307   for(int state = 0; state < numStates; state++){
308
309     //Create adjacency matrix from bitstring
310     std::bitset<graphSize*graphSize> bits(state);
311     auto A_matrix = [] (std::bitset<graphSize*graphSize> bits) {
312       std::array<std::array<int, graphSize>, graphSize> matrix {0};
313       for(int bit = 0; bit < graphSize*graphSize ; bit++){
314         matrix[bit/graphSize][bit%graphSize] = bits[graphSize*graphSize-bit-1];
315       }
316       return matrix;
317     }(bits);
318
319     //Compute score for valid adjacency matrices
320     float score = 0.0;
321       for (int i = 0; i < graphSize; i++){
322         for (int j = 0; j < graphSize; j++){
323           if(A_matrix[i][j] == 1){
324             score += D_matrix[i][j];
325           }
326           if((A_matrix[i][j]*A_matrix[j][i] == 1) && (i != j)){
327             score += -5.0;
328           }
329         }
330       }
331       score /= 2.0;
332
333     //Sum cost with histogram weight
334     average += score * hist[state];
335     total_count += hist[state];
336   }
337
338   //Return average result
339   return (float)average / (float)total_count;
340 }
341
342 //Static for loop for expectation function
343 template<index_t start, index_t end, index_t step, index_t index>
344 struct check_IS
345 {
346   template<typename Valid, typename Graph, typename Bits>
347   inline constexpr auto operator()(Valid&& valid, Graph, Bits bits) noexcept
348   {
349     int e0 = Graph::template from<index>();
350     int e1 = Graph::template to<index>();
351     int end_index = graphSize-1;
352
353     //index reverse because of bitsting indexing
354     if(bits[end_index - e0] == 1 and bits[end_index - e1] == 1){
355       valid = false;
356     }
357
358     return valid;
359   }
360 };
361
362 //Computes expectation value for MIS problem
363 template <typename graphType, typename histType>
364 float expectationMIS(graphType graph, histType hist) {
365
366   static constexpr const std::size_t numEdges = graph.size();
367   static constexpr const std::size_t numStates = hist.size();
368
369   int average = 0;
370   int total_count = 0;
371   //Loop over all output states
372   for(int i = 0; i < numStates; i++){
373
374     //Convert state number to bitstring
375     std::bitset<graphSize> bits(i);
376
377     //Cost of this state
378     int cost = 0;
379
380     //Check if bitstring is an independent set
```

```
      bool valid = utils::static_for<0, numEdges-1, 1, check_IS>(true, graph, bits);

      //If solution is valid, compute solution cost
      if(valid){
        for(int j = 0; j < graphSize; j++){
          cost -= bits[j];  //Subtract to get best score be most negative
        }
      }

      //Debug message
      //std::cout << "State:" << bits << ": valid=" << valid << ", Score: " << cost << std::endl;

      //Sum cost with histogram weight
      average += cost * hist[i];
      total_count += hist[i];
    }

    //Return average result
    return (float)average / (float)total_count;
  }

  //Functions to select which QAOA circuit to generate based on problem
  template<typename graphType, typename paramType>
  auto genCircuit(std::integral_constant<vqa_problems, MCP>, graphType graph, paramType params){
    return genCircuitMCP(graph, params);
  }

  template<typename graphType, typename paramType>
  auto genCircuit(std::integral_constant<vqa_problems, DSP>, graphType graph, paramType params){
    return genCircuitDSP(graph, params);
  }

  template<typename graphType, typename paramType>
  auto genCircuit(std::integral_constant<vqa_problems, TSP>, graphType graph, paramType params){
    return genCircuitTSP(graph, params);
  }

  template<typename graphType, typename paramType>
  auto genCircuit(std::integral_constant<vqa_problems, MIS>, graphType graph, paramType params){
    return genCircuitMIS(graph, params);
  }

  //Functions to select which expectation function to use based on problem
  template<typename graphType>
  auto expectation(std::integral_constant<vqa_problems, MCP>, graphType graph){
    return expectationMCP(graph, _hist);
  }

  template<typename graphType>
  auto expectation(std::integral_constant<vqa_problems, DSP>, graphType graph){
    return expectationDSP(graph, _hist);
  }

  template<typename graphType>
  auto expectation(std::integral_constant<vqa_problems, TSP>, graphType graph){
    return expectationTSP(graph, _hist);
  }

  //VQE problems TODO: FIND GENERAL DEFAULT
  template<typename graphType, typename paramType>
  auto genCircuit(std::integral_constant<vqa_problems, RH>, graphType graph, paramType params){
    return init();
  }

  template<typename graphType, typename paramType>
  auto genCircuit(std::integral_constant<vqa_problems, IC>, graphType graph, paramType params){
    return init();
  }

  template<typename graphType>
  auto expectation(std::integral_constant<vqa_problems, MIS>, graphType graph){
    return expectationMIS(graph, _hist);
  }

  template<typename graphType>
  auto expectation(std::integral_constant<vqa_problems, RH>, graphType graph){
    return 0.0;
  }

  template<typename graphType>
  auto expectation(std::integral_constant<vqa_problems, IC>, graphType graph){
    return 0.0;
  }

```

```
public:
  //Information constants
  const int id = 0; //QAOA
  const std::string name = "QAOA";

  //Constructor
  QAOA() {}

  //Run QAOA circuit
  template <typename paramType>
  void run(paramType params, bool printCircuit) {

    //Create Graph
    auto graph = utils::make_regular_graph<graphSize>();

    //Generate quantum circuit
    auto expr = genCircuit(std::integral_constant<vqa_problems, problem>{}, graph, params);

    //Execute circuit on quantum backend
    QDevice<qpuID, getNumQubits()> qpu;

    if(qpuID != QDeviceType::quest and
       qpuID != QDeviceType::qx){

      //Retrieve results
      qpu(measure(range<0, problem == TSP ? graphSize*graphSize -1 : graphSize-1>(expr)));
      auto job = qpu.execute(shots);
      utils::json result = job->get();

      //std::cout << result << std::endl;
      //std::cout << qpu.print_circuit() << std::endl;

      //Store results
      _execDuration = qpu.template get<QResultType::duration>(result).count();
      _jobDuration = qpu.template get<QResultType::jobDuration>(result).count();
      _queueDuration = qpu.template get<QResultType::queueDuration>(result).count();
      auto full_hist = qpu.template get<QResultType::histogram>(result);
      for(int i=0; i < getHistSize(); i++){ //Store only qubit measurents in histogram
        _hist[i] = full_hist[i];
      }

    } else{

      qpu(expr);
      auto job = qpu.execute(shots);

    #if QUEST_ENABLE
      _execDuration = qpu.duration_s();
      _jobDuration = qpu.duration_s();
      _queueDuration = 0.0;
      auto probs =  qpu.probabilities();


      for(int i=0; i < getHistSize(); i++){
        _hist[i] = 0;
      }

      for(int i=0; i < (1 << getNumQubits()); i++){

        _hist[i % getHistSize()] += shots*probs[i];
      }
    #else
      std::cout << "\nPlease enable QuEST macro. Exiting..." << std::endl;
      std::exit(0);
    #endif

    }
      //Print circuit if enabled (with Qiskit)
      if(printCircuit){
        QDevice<QDeviceType::qiskit_aer_simulator, getNumQubits()> qiskit;
        qiskit(expr);
        std::cout << qiskit.print_circuit() << std::endl;
      }

    _expectation = expectation(std::integral_constant<vqa_problems, problem>{}, graph);
  }

  //Return histogram results
  auto getHist(){
    return _hist;
  }

  //Returns number of parameters needed for optimization
  int getNumParams(){
```

```
549     return 2*P;
550   }
551
552   //Returns optimal score
553   int getOptimalScore(){
554     int MCP_scores[] = {0, 0, 1, 2, 4, 6, 8, 10, 12, 12, 14, 16, 18, 18, 20, 22, 24, 24, 26, 28, 30, 30,
       32, 34, 36, 36, 38, 40, 42, 42, 44, 46, 48, 48, 50, 52, 54, 54, 56, 58, 60, 60, 62, 64, 66, 66, 68,
       70, 72, 72};
555     return problem == MCP ? -MCP_scores[graphSize] :
556            problem == DSP ? -(graphSize-((graphSize-1)/5 + 1)) :
557            problem == TSP ? -5*graphSize*(graphSize-1)/2+graphSize :
558            problem == MIS ? -graphSize/3 :  0;
559   }
560
561   //Return circuit depth
562   double getCircuitDepth(){
563
564     //Get QPU
565     QDevice<QDeviceType::qiskit_aer_simulator, getNumQubits()> qiskit;
566
567     //Create Graph
568     auto graph = utils::make_regular_graph<graphSize>();
569
570     //Generate quantum circuit
571     arma::vec params = arma::ones<arma::vec>(2*P);
572     auto expr = genCircuit(std::integral_constant<vqa_problems, problem>{}, graph, params);
573
574     //Load expression into qiskit backend
575     qiskit(expr);
576
577     return std::stoi(qiskit.circuit_depth()) + 1;
578   }
579
580 };
581
582 #endif //QAOA_HPP
```

Listing E.5: QAOA.hpp code of the QPack benchmark in LibKet.

```
1  /** @file Qpack/include/QAOA_loops.hpp
2
3   @brief Qpack QAOA_loops header file
4
5   @author Huub Donkers
6
7   @defgroup qpack libket
8  */
9
10 #pragma once
11 #ifndef QAOA_LOOPS_HPP
12 #define QAOA_LOOPS_HPP
13
14 #include <LibKet.hpp>
15 #include <math.h>
16
17 using namespace LibKet;
18 using namespace LibKet::circuits;
19 using namespace LibKet::filters;
20 using namespace LibKet::gates;
21
22 #define PI  3.14159265358979323846  /* pi */
23
24 namespace ct_math{
25   //Function to take a compile time square root (only for integer squares)
26   static constexpr std::size_t ct_sqrt(std::size_t res, std::size_t l, std::size_t r){
27       if(l == r){
28           return r;
29       } else {
30           const auto mid = (r + l) / 2;
31
32           if(mid * mid >= res){
33               return ct_sqrt(res, l, mid);
34           } else {
35               return ct_sqrt(res, mid + 1, r);
36           }
37       }
38   }
39
40   static constexpr std::size_t ct_sqrt(std::size_t res){
41       return ct_sqrt(res, 1, res);
42   }
43 } //ct_math
44
```

```
namespace QAOAGates{

    //CNOT RZ CNOT Maxcut
    template<index_t start, index_t end, index_t step, index_t index>
    struct maxcut_cost_function
    {
      template<typename  Expr, typename Graph, typename Gamma>
      inline constexpr auto operator()(Expr&& expr, Graph, Gamma) noexcept
      {
        auto cnot1 = cnot(sel<Graph::template from<index>()>(gototag<0>()),
                          sel<Graph::template to<index>()>(gototag<0>(expr))
                          );

        auto rot_z_gamma = rz(Gamma{}, sel<1>(cnot1));

        auto cnot2 = cnot(sel<Graph::template from<index>()>(gototag<0>()),
                          sel<Graph::template to<index>()>(gototag<0>(rot_z_gamma))
                          );

        return gototag<0>(cnot2);
      }
    };

    //Maxcut p iteration
    template<index_t start, index_t end, index_t step, index_t index>
    struct maxcut_step
    {
      template<typename  Expr, typename Graph, typename Params>
      inline constexpr auto operator()(Expr&& expr, Graph, Params&& params) noexcept
      {

        //Set circuits parameters
        QVar_t<2*index,1> beta(2*PI*params[2*index]);
        QVar_t<2*index+1,1> gamma(2*PI*params[2*index+1]);

        //Cost unitaries
        auto cost = utils::static_for<0,Graph::size() -1,1,maxcut_cost_function>(tag<0>(expr), Graph{},
     gamma);

        //Mixer unitaries
        auto mixed = rx(beta, gototag<0>(cost));

        return gototag<0>(mixed);
      }
    };

    //Inverted CRZ DSP
    template<index_t start, index_t end, index_t step, index_t index>
    struct dsp_inv_cphase
    {
      template<typename Expr, typename A_filter, typename Gamma>
      inline constexpr auto operator()(Expr&& expr, A_filter&& a_filter, Gamma) noexcept
      {
        auto x1 = x(sel<index>(gototag<1>(expr)));
        auto crz_gate = crz(Gamma{},
                          sel<index>(),
                          a_filter(gototag<1>(x1)));
        auto x2 = x(sel<index>(gototag<1>(crz_gate)));

        return gototag<1>(x2);
      }
    };

    template<int index>
    struct data{
      static constexpr const int _index = index;
    };

    //Find neigbours
    template<index_t start, index_t end, index_t step, index_t index>
    struct get_neighbour_filter
    {
      template<int node, int e0, int e1>
      auto getNeighbour(std::false_type){
        return sel<>();
      }

      template<int node, int e0, int e1>
      auto getNeighbour(std::true_type){
        return sel<node == e0 ? e1 : e0>();
      }

      template<typename Q_filter, typename NodeIndex, typename Graph>
      inline constexpr auto operator()(Q_filter&& q_filter, NodeIndex, Graph) noexcept
```

```
    {
      constexpr auto e0 = Graph::template from<index>();
      constexpr auto e1 = Graph::template to<index>();
      constexpr bool hasNeighbour = NodeIndex::_index == e0 || NodeIndex::_index == e1;

      return q_filter<<getNeighbour<NodeIndex::_index, e0, e1>(std::integral_constant<bool, hasNeighbour
>{});

    }
  };

  //Arbitrary OR cphase for all connected nodes
  template<index_t start, index_t end, index_t step, index_t index>
  struct dsp_arb_OR_cphase
  {
    template<typename Expr, typename A_filter, typename AA_filter, typename Graph, typename Gamma>
    inline constexpr auto operator()(Expr&& expr, A_filter&& a_filter, AA_filter&& aa_filter, Graph,
 Gamma) noexcept
    {
      //Loop over graph to find connection for this node
      constexpr auto nodeIndex = data<index>();
      auto q_filter = utils::static_for<0, Graph::size()-1,1, get_neighbour_filter>(sel<index>(),
 nodeIndex, Graph{});

      //Reduce number of ancillas based on q_filter size
      constexpr auto size = q_filter.size();
      auto new_aa_filter = range<0,size-2>(aa_filter);

       //Apply OR cphase to selected qubits
      auto arb_OR_cphase = arb_OR<>(crz(Gamma{}),
                                    q_filter(),
                                    a_filter(),
                                    new_aa_filter(gototag<1>(expr))
                                    );

      return gototag<1>(arb_OR_cphase);
    }
  };

  //DSP p iteration
  template<index_t start, index_t end, index_t step, index_t index>
  struct dsp_step
  {
    template<typename  Expr, typename Graph, typename Params>
    inline constexpr auto operator()(Expr&& expr, Graph, Params&& params) noexcept
    {

      //Set circuits parameters
      QVar_t<2*index,1> beta(2*PI*params[2*index]);
      QVar_t<2*index+1,1> gamma(2*PI*params[2*index+1]);
      constexpr auto numQubits = Graph::size() == 1 ? 2 :
                                 Graph::size() == 3 ? 3 :
                                 Graph::size() == 4 ? 4 : Graph::size()/2; //For regular graphs

      //Set ancilla filters
      auto cost_ancilla = QFilterSelect<numQubits>();
      auto ctrl_anchillas = QFilterSelectRange<numQubits+1, numQubits+4>();

      //Inverted CRZ for each qubit on ancilla
      auto inv_cphase = utils::static_for<0,numQubits-1,1, dsp_inv_cphase>(tag<1>(expr), cost_ancilla,
 gamma);

      //Logical OR for each qubit on ancilla
      auto cost = utils::static_for<0,numQubits-1,1, dsp_arb_OR_cphase>(tag<1>(inv_cphase), cost_ancilla
, ctrl_anchillas, Graph{}, gamma);

      //Mixer unitaries
      auto mixed = rx(beta, range<0,numQubits-1>(gototag<1>(cost)));

      return gototag<1>(mixed);
    }
  };

  /* Implementation of CRY gate
  ──────■──────     ──────────■──────────
  ┌─────┴─────┐     ┌────────┴────────┐┌─┴─┐┌────────┴────────┐┌─┴─┐
  ┤ Ry(0.5) ├  =  ┤ Ry(0.25) ├┤ X ├┤ Ry(-0.25) ├┤ X ├
  └───────────┘     └─────────────────┘└───┘└─────────────────┘└───┘
  */
  template<typename AnglePos, typename AngleNeg, typename CFilter, typename TFilter, typename Expr>
  auto cry(AnglePos, AngleNeg, CFilter&& cFilter, TFilter&& tFilter, Expr&& expr){

    auto RyPos = ry(AnglePos{}, tFilter(all(expr)));
```

```
    auto cnot1 = cnot(cFilter(),
                      tFilter(all(RyPos)));

    auto RyNeg = ry(AngleNeg{}, tFilter(all(cnot1)));

    auto cnot2 = cnot(cFilter(),
                      tFilter(all(RyNeg)));

    return all(cnot2);
  }

  /* Implementation of CCRY gate
  ──■──   ──────■──────
    │       │
  ──■──  =  ──■──────■──
  ┌─┴─┐   ┌──┴──┐┌─┴─┐┌──┴──┐┌─┴─┐
  ┤Ry(0.5)├  ┤Ry(0.25)├┤ X ├┤Ry(-0.25)├┤ X ├
  └───────┘  └─────────┘└───┘└─────────┘└───┘
  */
  template<typename AnglePos, typename AngleNeg, typename C0Filter, typename C1Filter, typename TFilter,
    typename Expr>
  auto ccry(AnglePos, AngleNeg, C0Filter&& c0Filter, C1Filter&& c1Filter, TFilter&& tFilter, Expr&& expr
   ){

    auto RyPos = ry(AnglePos{}, tFilter(all(expr)));

    auto ccnot1 = ccnot(c0Filter(),
                        c1Filter(),
                        tFilter(all(RyPos)));

    auto RyNeg = ry(AngleNeg{}, tFilter(all(ccnot1)));

    auto ccnot2 = ccnot(c0Filter(),
                        c1Filter(),
                        tFilter(all(RyNeg)));

    return all(ccnot2);
  }

  //TSP inital state encoding: Dicke state with Hamming weight 2
  //Implemenation by Mukherjee et. al. (2020)
  template<index_t start, index_t end, index_t step, index_t index>
  struct tsp_scs
  {
    template<typename  Expr, typename RowFilter>
    inline constexpr auto operator()(Expr&& expr, RowFilter&& rowFilter) noexcept
    {
      //Setup filters for SCS iteration (three qubits for Hamming weight 2)
      auto q0filter = QFilterSelect<end-index>()(rowFilter);
      auto q1filter = QFilterSelect<end-index+1>()(rowFilter);
      auto q2filter = QFilterSelect<end-index+2>()(rowFilter);

      //Select angles for SCS iteration
      double angle0 = 2*std::acos(std::sqrt(1.0/(end+3-index)));
      double angle1 = 2*std::acos(std::sqrt(2.0/(end+3-index)));

      //Set angles to QVars with unique index (divided by 2 for CRY and CCRY gates)
      QVar_t<300+index*4+0> angle0pos(angle0/2);
      QVar_t<300+index*4+1> angle0neg(-angle0/2);
      QVar_t<300+index*4+2> angle1pos(angle1/2);
      QVar_t<300+index*4+3> angle1neg(-angle1/2);

      //Add mu block
      auto mu = gototag<2>(cnot(q1filter(),
                          q2filter(gototag<2>(cry(angle0pos,
                                                  angle0neg,
                                                  q2filter(),
                                                  q1filter(),
                                                  gototag<2>(cnot(q1filter(),
                                                                  q2filter(expr)
                                                                 )
                                                            )
                                                 )
                                              )
                                   )
                               )
                          );

      //Add M block
      auto M = gototag<2>(cnot(q0filter(),
                  q2filter(gototag<2>(ccry(angle1pos,
                                           angle1neg,
                                           q2filter(),
                                           q1filter(),
```

```
                                        q0filter(),
                                        gototag<2>(cnot(q0filter(),
                                                        q2filter(mu))
                                                   )

                                     )
                                  )
                               )
                            )
                  );

      return gototag<2>(M);
    }
  };

  //TSP inital state encoding
  template<index_t start, index_t end, index_t step, index_t index>
  struct tsp_init
  {
    template<typename  Expr>
    inline constexpr auto operator()(Expr&& expr) noexcept
    {

      auto rowFilter = QFilterSelectRange<index, index+step-1>();

      auto flipFilter = QFilterSelect<step-2, step-1>()(rowFilter);

      //Dicke state of k = 2: Flip bottom two qubits
      auto flips = gototag<2>(x(flipFilter(expr)));

      //Execute SCS blocks
      auto scs = utils::static_for<0, step-3, 1, tsp_scs>(flips, rowFilter);

      //Add final SCS block
      double angle = 2*std::acos(std::sqrt(1.0/2));
      QVar_t<400+index*2+0> anglepos(angle/2);
      QVar_t<400+index*2+1> angleneg(-angle/2);
      auto scs_final = gototag<2>(cnot(sel<0>(rowFilter()),
                                       sel<1>(rowFilter(gototag<2>(cry(anglepos,
                                                                       angleneg,
                                                                       sel<1>(rowFilter()),
                                                                       sel<0>(rowFilter()),
                                                                       gototag<2>(cnot(sel<0>(rowFilter
 ()),
                                                                                       sel<1>(rowFilter(
 scs))
                                                                                   )
                                                                    )
                                                               )
                                                          ))
                                                )
                                       );

      return gototag<2>(scs_final);
    }
  };

  //Struct to pass constexpressions in function parameters (indices in this case)
  template<int index>
  struct IndexPass{
    static constexpr const int getIndex(){return index;}
  };

  //TSP: loop over all qubits
  template<index_t start, index_t end, index_t step, index_t index>
  struct tsp_qubit_loop
  {
    template<typename  Expr, typename D_Matrix, typename NumCities, typename Gamma>
    inline constexpr auto operator()(Expr&& expr, D_Matrix&& D_matrix, NumCities&& numCities, Gamma&&
 gamma) noexcept
    {
      //Set Rz angle bases on D_matrix weights
      QVar_t<100+index,1> angle(gamma/(2*PI)*D_matrix[index/numCities][index%numCities]);

      return  gototag<2>(rz(angle, sel<index>(gototag<2>(expr))));
    }
  };

  //TSP: loop over all couplings
  template<index_t start, index_t end, index_t step, index_t index>
  struct tsp_coupling_loop
  {
    template<typename  Expr, typename FullGraph, typename Gamma>
```

```
370       inline constexpr auto operator()(Expr&& expr, FullGraph, Gamma&& gamma) noexcept
371       {
372         //RZZ angle is 5 gamma
373         QVar_t<200,1> angle(5*gamma/PI);
374
375         //Link all qubits that are symmetrically opposite of the matrix diagonal
376         constexpr int e0 = FullGraph::template from<index>();
377         constexpr int e1 = FullGraph::template to<index>();
378         constexpr int numCities = (1+ct_math::ct_sqrt(1+8*FullGraph::size()))/2;
379         constexpr int q0 = e0*numCities + e1;
380         constexpr int q1 = e1*numCities + e0;
381
382         return gototag<2>(rzz(angle, sel<q0>(gototag<2>()), sel<q1>(gototag<2>(expr))));
383       }
384     };
385
386     //TSP: rxx loop
387     template<index_t start, index_t end, index_t step, index_t index>
388     struct tsp_rxx_loop
389     {
390       template<typename  Expr, typename Beta>
391       inline constexpr auto operator()(Expr&& expr, Beta&& beta) noexcept
392       {
393         constexpr int N = end-start+1;
394         constexpr int q0 = index;
395         constexpr int q1 = index-start == N-1 ? start : index+1;
396         return rxx(beta,
397                    sel<q0>(gototag<2>()),
398                    sel<q1>(gototag<2>(expr)));
399
400       }
401     };
402
403     //TSP: ryy loop
404     template<index_t start, index_t end, index_t step, index_t index>
405     struct tsp_ryy_loop
406     {
407       template<typename  Expr, typename Beta>
408       inline constexpr auto operator()(Expr&& expr, Beta&& beta) noexcept
409       {
410         constexpr int N = end-start+1;
411         constexpr int q0 = index;
412         constexpr int q1 = index-start == N-1 ? start : index+1;
413         return ryy(beta,
414                    sel<q0>(gototag<2>()),
415                    sel<q1>(gototag<2>(expr)));
416       }
417     };
418
419
420     //TSP: loop over all nodes
421     template<index_t start, index_t end, index_t step, index_t index>
422     struct tsp_node_loop
423     {
424       template<typename  Expr, typename Beta>
425       inline constexpr auto operator()(Expr&& expr, Beta&& beta) noexcept
426       {
427         //Decomposed SWAP gate (SWAP_{ij} = e^{X_i X_j + Y_i Y_j})
428         constexpr int N = end+1;
429         auto rxx1 = utils::static_for<N*index, N*index+N-1, 1, tsp_rxx_loop>(gototag<2>(expr), beta);
430         auto ryy1 = utils::static_for<N*index, N*index+N-1, 1, tsp_ryy_loop>(gototag<2>(rxx1), beta);
431
432         return gototag<2>(ryy1);
433       }
434     };
435
436     //TSP p itersation
437     template<index_t start, index_t end, index_t step, index_t index>
438     struct tsp_step
439     {
440       template<typename  Expr, typename Graph, typename FullGraph, typename D_Matrix, typename Params>
441       inline constexpr auto operator()(Expr&& expr, Graph, FullGraph fullGraph, D_Matrix&& D_matrix,
     Params&& params) noexcept
442       {
443         //Set circuits parameters
444         QVar_t<2*index,1> beta(-(params[2*index]-0.5));///(end+1));
445         auto gamma = (2*PI*(params[2*index+1]-0.5));///(end+1));
446         constexpr auto numCities = Graph::size() == 3 ? 3 :
447                                    Graph::size() == 4 ? 4 :
448                                    Graph::size()/2; //For regular graphs
449
450         //Cost hamiltonians
451         //Soft constraint, enforcing minimal travel distance
452         auto cost_soft = utils::static_for<0,
```

```
                                        numCities*numCities-1,
                                        1,
                                        tsp_qubit_loop>(gototag<2>(expr), D_matrix, numCities, gamma);

        //Hard constraint, enforcing a symmetric adjacency matrix
        auto cost_hard = utils::static_for<0, FullGraph::size()-1, 1, tsp_coupling_loop>(gototag<2>(
    cost_soft), FullGraph{}, gamma);

        //Mixer hamiltonian: Mixes the solution space
        auto mixer = utils::static_for<0, numCities-1, 1, tsp_node_loop>(gototag<2>(cost_hard), beta);

        return gototag<2>(mixer);
      }
    };

    //Arbitrary NOR X-rotation for all connected nodes
    template<index_t start, index_t end, index_t step, index_t index>
    struct mis_NOR_rotX
    {
      template<typename Expr, typename Ancillas, typename Graph, typename Beta, typename P>
      inline constexpr auto operator()(Expr&& expr, Ancillas&& ancillas, Graph, Beta, P) noexcept
      {
        //Loop over graph to find connection for this node
        constexpr size_t current_node = (index + P::_index) % (end+1);
        constexpr auto nodeIndex = data<current_node>();
        auto c_filter = utils::static_for<0,Graph::size()-1,1, get_neighbour_filter>(sel<>(), nodeIndex,
    Graph{});
        auto t_filter = sel<current_node>();

        //Reduce number of ancillas based on q_filter size
        constexpr auto size = c_filter.size();
        auto new_ancillas = range<0,size-2>(ancillas);
        auto c_last_filter = sel<size-2>(new_ancillas);

        //Apply OR cphase to selected qubits
        auto x1 = x(c_last_filter(gototag<3>(expr)));
        auto arb_OR_crz= arb_OR<>(crx(Beta{}),
                                  c_filter(),
                                  t_filter(),
                                  new_ancillas(gototag<3>(x1))
                                  );
        auto x2 = x(c_last_filter(gototag<3>(arb_OR_crz)));

        return gototag<3>(x2);
      }
    };

    //MIS p iteration
    template<index_t start, index_t end, index_t step, index_t index>
    struct mis_step
    {
      template<typename  Expr, typename Graph, typename Params>
      inline constexpr auto operator()(Expr&& expr, Graph, Params&& params) noexcept
      {

        //Set circuits parameters
        QVar_t<2*index,1> beta(PI/2*params[2*index]);
        QVar_t<2*index+1,1> gamma(2*PI*params[2*index+1]);
        constexpr auto numQubits = Graph::size() == 1 ? 2 :
                                   Graph::size() == 3 ? 3 :
                                   Graph::size() == 4 ? 4 : Graph::size()/2; //For regular graphs

        //Set ancilla filters
        auto ancillas = QFilterSelectRange<numQubits, numQubits+3>();

        //Cost Hamiltonian
        auto cost = rz(gamma, range<0,numQubits-1>(gototag<3>(expr)));

        //Mixer unitaries
        constexpr auto P = data<index>();
        auto mixed = utils::static_for<0,
                                       numQubits-1,
                                       1,
                                       mis_NOR_rotX>(gototag<3>(cost), ancillas, Graph{}, beta, P);

        return gototag<3>(mixed);
      }
    };

}

#endif //QAOA_LOOPS_HPP
```

Listing E.6: QAOA_Loops.hpp code of the QPack benchmark in LibKet.

```cpp
/** @file Qpack/include/VQE.hpp

 @brief Qpack VQE header file

 @author Huub Donkers

 @defgroup qpack libket
*/

#pragma once
#ifndef VQE_HPP
#define VQE_HPP

#include <LibKet.hpp>
#include "../include/VQA.hpp"
#include "../include/VQE_Loops.hpp"
#include "../include/VQE_Hamiltonians.hpp"
#include "../include/VQE_Ansatzes.hpp"

using namespace LibKet;
using namespace LibKet::circuits;
using namespace LibKet::filters;
using namespace LibKet::gates;

#define PI  3.14159265358979323846  /* pi */


/*
@brief: This class implements all necessary functionalities of the VQE algorithm, such
        as determining the amount of qubits and statespace for a given problem. The class
        is able to generate VQE circuits, execute these circuits and give an expectation
        value for each problem.
*/
template <vqa_problems problem, int problem_size, int shots, QDeviceType qpuID>
class VQE : public VQA{

public:

  //Return QPU ID
  static QDeviceType getQpuID(){
    return qpuID;
  }

  //Return problem
  static vqa_problems getProblem(){
    return problem;
  }

  //Return number of shots
  static int getNumShots(){
    return shots;
  }

  //Return P for VQE (not applicable)
  static int getP(){
    return 0;
  }

  //Return problem size
  static int getSize(){
    return problem_size;
  }

  //Returns the number of qubits based on selected problem
  static constexpr auto getNumQubits(){
  return problem == RH ? problem_size   :
         problem == IC ? problem_size   :
         1;
  }

private:
  //Returns the size of the histogram based on selected problem
  static constexpr auto getHistSize(){
  return  2 << getNumQubits();
  }

  //Variable to store qpu result
  QArray<getHistSize(), unsigned long> _hist;
  int _numParams = 1;

  //Struct containing QPU info to pass as constexpr function parameter
  struct QPU_Info{
    static constexpr const int _qubits = getNumQubits();
    static constexpr const int _shots = shots;
```

```
  static constexpr const QDeviceType _qpuID = qpuID;
};

//Static for loop to retreive Hamiltonian terms
template<index_t start, index_t end, index_t step, index_t index>
struct term_loop
{
  template<typename Vec, typename H_mol>
  inline constexpr auto operator()(Vec&& vec, H_mol) noexcept
  {
    vec[index] = H_mol::_terms[index];
    return vec;
  }
};

//Converts a Qpack Hamiltonian to an armadillo hamiltonian.
//Returns real value of Hamiltonian.
template<typename H_mol>
arma::cx_mat arma_hamiltonian(H_mol h_mol){

  //Complex contants (0, i, 1)
 std::complex<double> c_0(0,0);
 std::complex<double> c_i(0,1);
  std::complex<double> c_1(1,0);

  //Pauli matrices
 arma::cx_mat I = {{c_1,  c_0}, {c_0,  c_1}};
 arma::cx_mat X = {{c_0,  c_1}, {c_1,  c_0}};
 arma::cx_mat Y = {{c_0, -c_i}, {c_i,  c_0}};
 arma::cx_mat Z = {{c_1,  c_0}, {c_0, -c_1}};

  //Create an char array that read the terms of the Qpack Hamiltonian
  char empty_vec[H_mol::_numTerms*H_mol::_qubits];
  auto P = utils::static_for<0, H_mol::_numTerms*H_mol::_qubits-1, 1, term_loop>(empty_vec, h_mol);

  //Loop oper all Hamiltonian terms and create armadillo hamiltonian with kronecker products
  int mat_size = 2<<(getNumQubits()-1);
  arma::cx_mat H(mat_size, mat_size);
  for(int t=0; t < H_mol::_numTerms; t++){

    arma::cx_mat kronecker;
    for(int q=0; q < H_mol::_qubits; q++){

      arma::cx_mat pauli;
      switch(P[t*H_mol::_qubits+q]){
        case 'I':
          pauli = I;
          break;
        case 'X':
          pauli = X;
          break;
        case 'Y':
          pauli = Y;
          break;
        case 'Z':
          pauli = Z;
          break;
      }
      if(q == 0){ //First product is the first pauli matrix
        kronecker = pauli;
      }
      else{ //Tensor with previous pauli
        kronecker = arma::kron(kronecker, pauli);
      }
    }

    //Scale by term coefficient and sum to total Hamiltonian
    H += h_mol._coeffs[t]*kronecker;
  }

 return H;

}

//Functions to return ansatz and Hamiltonian based on problem set
template< typename paramType>
auto genAnsatz(std::integral_constant<vqa_problems, RH>, paramType params){
  return Ansatzes::genRandomAnsatz<problem_size>(params);
}

template< typename paramType>
auto genAnsatz(std::integral_constant<vqa_problems, IC>, paramType params){
  return Ansatzes::genIsingAnsatz<problem_size>(params);
}
```

```
169
170   //Functions to select which Hamiltonian to use based on problem
171   auto genHamiltonian(std::integral_constant<vqa_problems, RH>){
172     return Hamiltonians::genDiagonalHamiltonian<problem_size>();
173   }
174
175   //Functions to select which Hamiltonian to use based on problem
176   auto genHamiltonian(std::integral_constant<vqa_problems, IC>){
177     return Hamiltonians::genIsingHamiltonian<problem_size>();
178   }
179
180   //QAOA functions (TODO: Fix general default)
181   template< typename paramType>
182   auto genAnsatz(std::integral_constant<vqa_problems, MCP>, paramType params){ return init(); }
183
184   template< typename paramType>
185   auto genAnsatz(std::integral_constant<vqa_problems, MIS>, paramType params){ return init(); }
186
187   template< typename paramType>
188   auto genAnsatz(std::integral_constant<vqa_problems, DSP>, paramType params){ return init(); }
189
190   template< typename paramType>
191   auto genAnsatz(std::integral_constant<vqa_problems, TSP>, paramType params){ return init(); }
192
193   auto genHamiltonian(std::integral_constant<vqa_problems, MCP>){ return Hamiltonians::
      genDiagonalHamiltonian<1>();}
194   auto genHamiltonian(std::integral_constant<vqa_problems, MIS>){ return Hamiltonians::
      genDiagonalHamiltonian<1>();}
195   auto genHamiltonian(std::integral_constant<vqa_problems, DSP>){ return Hamiltonians::
      genDiagonalHamiltonian<1>();}
196   auto genHamiltonian(std::integral_constant<vqa_problems, TSP>){ return Hamiltonians::
      genDiagonalHamiltonian<1>();}
197
198 public:
199    //Information constant
200    const int id = 1; //VQE ID
201    const std::string name = "VQE";
202
203    //Constructor
204    VQE(){}
205
206    //Run VQE circuits
207    template <typename paramType>
208    void run(paramType params, bool printCircuit) {
209
210      //Generate ansatz circuit
211      auto ansatz = genAnsatz(std::integral_constant<vqa_problems, problem>{}, params);
212
213      //H2 coefficients and Hamiltonian gates
214       auto H_mol = genHamiltonian(std::integral_constant<vqa_problems, problem>{});
215
216      //Run VQE circuits and get expecation value
217      constexpr auto qpu_info = QPU_Info();
218       std::vector<float> execDurations = {};
219       std::vector<float> jobDurations = {};
220       std::vector<float> queueDurations = {};
221
222       float expectation = utils::static_for<0, 0, 1, VQE_Loops::pauli_measurements>(0.0, ansatz, H_mol,
      qpu_info, &execDurations, &jobDurations, &queueDurations, printCircuit);
223
224       //Store measurements (average of all circuits)
225       _execDuration = 0.0;
226       _jobDuration = 0.0;
227       _queueDuration = 0.0;
228
229       for(int i=0; i < execDurations.size(); i++){
230         _execDuration += execDurations[i];
231       }
232       _execDuration /= execDurations.size();
233
234       for(int i=0; i < jobDurations.size(); i++){
235         _jobDuration += jobDurations[i];
236       }
237       _jobDuration /= jobDurations.size();
238
239       for(int i=0; i < queueDurations.size(); i++){
240         _queueDuration += queueDurations[i];
241       }
242       _queueDuration /= queueDurations.size();
243
244       _expectation = expectation;
245    }
246
247   //Compute classical solution
```

```
  float classicalExpectation(){

   //Check classical solution
   auto H_mol = genHamiltonian(std::integral_constant<vqa_problems, problem>{});
   arma::cx_mat H_BK = arma_hamiltonian(H_mol);
   arma::vec eigval = arma::sort(arma::eig_sym(real(H_BK)));

    //arma::cout << eigval << std::endl;

 return eigval[0];
  }

  //Return histogram results
  auto getHist(){
    return _hist;
  }

  //Return number of optimizer parameters
  int getNumParams(){
   return problem == RH ? problem_size   :
          problem == IC ? problem_size*4 :
          0;
  }

  float getOptimalScore(){
    return classicalExpectation();
  }

  //Return circuit depth
  double getCircuitDepth(){
    return problem == RH ? 2.0 :
          problem == IC ? 4.0 + 2*(problem_size-1) + 0.5:
          0;
  }
};

#endif //VQE_HPP
```

Listing E.7: VQE.hpp code of the QPack benchmark in LibKet.

```
/** @file Qpack/include/VQE_Loops.hpp

 @brief Qpack VQE_Loops header file

 @author Huub Donkers

 @defgroup qpack libket
*/

#pragma once
#ifndef VQE_LOOPS_HPP
#define VQE_LOOPS_HPP

#include <LibKet.hpp>
#include <math.h>

using namespace LibKet;
using namespace LibKet::circuits;
using namespace LibKet::filters;
using namespace LibKet::gates;

namespace VQE_Loops{

 //Struct to pass constexpr as function parameter
 template<int I>
    struct static_int{
     static constexpr const int _I = I;
    };

  //VQE Pauli measurement loop
    template<index_t start, index_t end, index_t step, index_t index>
    struct pauli_measurement_gates
    {

  //Functions to select which expectation function to use based on problem
  template<typename Expr>
  auto pauli_measure(std::integral_constant<char, 'I'>, Expr&& expr){
  return all(expr);
  }

  template<typename Expr>
  auto pauli_measure(std::integral_constant<char, 'Z'>, Expr&& expr){
  return all(expr);
```

```
}

template<typename Expr>
auto pauli_measure(std::integral_constant<char, 'Y'>, Expr&& expr){
return all(h(sdag(expr)));
}

template<typename Expr>
auto pauli_measure(std::integral_constant<char, 'X'>, Expr&& expr){
return all(h(expr));
}


    template<typename Expr, typename H_mol, typename TermIndex>
    inline constexpr auto operator()(Expr&& expr, H_mol, TermIndex) noexcept
    {
     constexpr char pauli = H_mol::_terms[TermIndex::_I*H_mol::_qubits + index];
      return pauli_measure(std::integral_constant<char, pauli>{}, sel<index>(expr));
    }
  };

  //Convert Histogram output to expectation value
  template<typename HistType>
  float histToExp(HistType hist){

   //Initialize expectation value an total number of shots to zero
   long expectation = 0;
   long shots = 0;

   //Loop over all states
   for(int state = 0; state < hist.size(); state++){
    std::bitset<ct_math::ct_sqrt(hist.size())> bits(state); //Get bitset based on hist index

    //Add expectations bases on bitset parity
    if(bits.count() % 2 == 0){ //Even
     expectation += (long)hist[state];
    }else{ //Odd
  expectation -= (long)hist[state];
    }
    shots += hist[state];
   }
   //Return expecation over all shots
   return (float)expectation/(float)shots;
  }

  template<typename HistType, typename H_mol>
  auto reduced_hist(int index, HistType hist, H_mol h_mol){

   //std::cout << "Terms: " << h_mol._numTerms << std::endl;
   //std::cout << "Qubits: " << h_mol._qubits << std::endl;

   std::vector<int> measure_base = {};
   QArray<1 << h_mol._qubits, unsigned long> reducedHist;

   for(int i=0; i < h_mol._qubits; i++){
    char pauli = h_mol._terms[index*h_mol._qubits + i];
    if(pauli != 'I'){
     measure_base.push_back(i);
    }
   }

   for(int i=0; i < hist.size(); i++){

    //Convert state number to bitstring
     std::bitset<H_mol::_qubits> bits(i);
     std::bitset<H_mol::_qubits> subset(0);

    for(int j=0; j < measure_base.size(); j++){
     std::bitset<H_mol::_qubits> mask(bits[measure_base[j]] << measure_base[j]);
     subset = subset | mask;
    }

    //std::cout << subset << std::endl;
    reducedHist[subset.to_ulong()] += hist[i];
   }

   return reducedHist;
  }

  //VQE Pauli measurement loop
  template<index_t start, index_t end, index_t step, index_t index>
  struct pauli_measurements
  {
    template<typename Expectation, typename Ansatz, typename H_mol, typename QPU_Info>
```

```
128      inline constexpr auto operator()(Expectation&& expectation,
129                        Ansatz&& ansatz,
130                        H_mol& h_mol,
131                        QPU_Info,
132                        std::vector<float>* execDurations,
133                        std::vector<float>* jobDurations,
134                        std::vector<float>* queueDurations,
135                        bool printCircuit) noexcept
136      {
137        //Append pauli measurement gates based on Hamiltonian
138        constexpr auto termIndex = static_int<index>();
139      auto expr = utils::static_for<0, H_mol::_qubits-1, 1, pauli_measurement_gates>(ansatz, H_mol{},
     termIndex);
140
141      //Execute circuit on quantum backend
142       QDevice<QPU_Info::_qpuID, QPU_Info::_qubits> qpu_z;
143       QDevice<QPU_Info::_qpuID, QPU_Info::_qubits> qpu_x;
144       QArray<1 << QPU_Info::_qubits, unsigned long> hist_z;
145       QArray<1 << QPU_Info::_qubits, unsigned long> hist_x;
146
147
148       if(QPU_Info::_qpuID != QDeviceType::quest and
149          QPU_Info::_qpuID != QDeviceType::qx){
150
151        //Retrieve Z-basis results
152         qpu_z(measure_z(ansatz));
153      auto job_z  = qpu_z.execute(QPU_Info::_shots);
154
155        //Store results
156        execDurations->push_back(qpu_z.template get<QResultType::duration>(job_z->get()).count());
157        jobDurations->push_back(qpu_z.template get<QResultType::jobDuration>(job_z->get()).count());
158        queueDurations->push_back(qpu_z.template get<QResultType::queueDuration>(job_z->get()).count());
159        auto _hist_z = qpu_z.template get<QResultType::histogram>(job_z->get());
160
161         for(int i=0; i < (1 << QPU_Info::_qubits); i++){
162           hist_z[i] = _hist_z[i];
163         }
164
165         //Retrieve X-basis results
166         qpu_x(measure_x(ansatz));
167      auto job_x  = qpu_x.execute(QPU_Info::_shots);
168
169        //Store results
170        execDurations->push_back(qpu_x.template get<QResultType::duration>(job_x->get()).count());
171        jobDurations->push_back(qpu_x.template get<QResultType::jobDuration>(job_x->get()).count());
172        queueDurations->push_back(qpu_x.template get<QResultType::queueDuration>(job_x->get()).count());
173        auto _hist_x = qpu_x.template get<QResultType::histogram>(job_x->get());
174
175         for(int i=0; i < (1 << QPU_Info::_qubits); i++){
176           hist_x[i] = _hist_x[i];
177         }
178
179
180       } else{
181
182         qpu_z(ansatz);
183         auto job_z = qpu_z.execute(QPU_Info::_shots);
184
185         qpu_x(h(ansatz));
186         auto job_x = qpu_x.execute(QPU_Info::_shots);
187
188       #if QUEST_ENABLE
189         execDurations->push_back(qpu_z.duration_s());
190         jobDurations->push_back(qpu_z.duration_s());
191         queueDurations->push_back(0.0);
192         auto probs_z =  qpu_z.probabilities();
193
194         for(int i=0; i < (1 << QPU_Info::_qubits); i++){
195           hist_z[i] = 0;
196         }
197
198         for(int i=0; i < (1 << QPU_Info::_qubits); i++){
199
200           hist_z[i] += QPU_Info::_shots*probs_z[i];
201         }
202
203         execDurations->push_back(qpu_x.duration_s());
204         jobDurations->push_back(qpu_x.duration_s());
205         queueDurations->push_back(0.0);
206         auto probs_x =  qpu_x.probabilities();
207
208         for(int i=0; i < (1 << QPU_Info::_qubits); i++){
209           hist_x[i] = 0;
210         }
```

```
211
212         for(int i=0; i < (1 << QPU_Info::_qubits); i++){
213
214           hist_x[i] += QPU_Info::_shots*probs_x[i];
215         }
216       #else
217         std::cout << "\nPlease enable QuEST macro. Exiting..." << std::endl;
218         std::exit(0);
219       #endif
220       }
221
222       //Print circuit if enabled (with Qiskit)
223        if(printCircuit){
224         QDevice<QDeviceType::qiskit_aer_simulator, QPU_Info::_qubits> qiskit;
225          qiskit(expr);
226          std::cout << qiskit.print_circuit() << std::endl;
227        }
228
229        //Loop over all Hamiltonian terms
230        for(int i = 0; i < H_mol::_numTerms; i++){
231
232         bool z_flag = false;
233         bool x_flag = false;
234
235         for(int j = 0; j < H_mol::_qubits; j++){
236          char term = H_mol::_terms[i*H_mol::_qubits + j];
237
238          if(term == 'Z'){
239           z_flag = true;
240          }
241
242          if(term == 'X'){
243           x_flag = true;
244          }
245         }
246
247         //Reduce histogram
248         QArray<1 << QPU_Info::_qubits, unsigned long> reducedHist;
249         if(z_flag){
250          reducedHist = reduced_hist(i, hist_z, h_mol);
251         } else if(x_flag){
252          reducedHist = reduced_hist(i, hist_x, h_mol);
253         }
254
255        //Add partial expectation to total
256        expectation += h_mol._coeffs[i]*histToExp(reducedHist);
257        }
258
259       //Debugging
260       //std::cout <<"Histogram: " << hist << std::endl;
261       //std::cout <<"Reduced histogram: " << reduced_hist<index>(hist, h_mol) << std::endl;
262       //std::cout << "Partial expectation: " << h_mol._coeffs[index]*histToExp(hist) << std::endl;
263
264       return expectation;
265       }
266     };
267 }; //namespace VQE_LOOPS
268
269 #endif //VQE_LOOPS_HPP
```

Listing E.8: VQE_Loops.hpp code of the QPack benchmark in LibKet.

```
1  /** @file Qpack/include/VQE_Ansatzes.hpp
2
3   @brief Qpack VQE_Ansatzes header file
4
5   @author Huub Donkers
6
7   @defgroup qpack libket
8  */
9
10 #pragma once
11 #ifndef VQE_ANSATZES_HPP
12 #define VQE_ANSATZES_HPP
13
14 namespace Ansatzes{
15
16 ///////Random Anzats//////////////////////////////
17
18   //Applies Rx gate for all indices with indexed parameter values
19   template<index_t start, index_t end, index_t step, index_t index>
20   struct rotX_loop
21   {
```

```
    template<typename Expr, typename Params>
  inline constexpr auto operator()(Expr&& expr, Params&& params) noexcept
  {
      QVar_t<100+index, 1> angle(2*PI*params[index]);
        return all(rx(angle, sel<index>(expr)));
  }

  template<typename Expr, typename Params, typename Offset> //Overload function that adds an offset
  inline constexpr auto operator()(Expr&& expr, Params&& params, Offset) noexcept
  {
    QVar_t<200+index+Offset::_I, 1> angle(2*PI*params[index+Offset::_I]);
      return all(rx(angle, sel<index>(expr)));
  }
};

//Generate ansatz for random diagonal hamiltonian
template <int size, typename paramType>
constexpr auto genRandomAnsatz(paramType params){

 auto s0 = init();

  //RotX gates on all qubits for different parameters
 auto RotX = utils::static_for<0, size-1, 1, rotX_loop>(s0, params);

 return RotX;
};

///////Ising chain Ansatz//////////////////////////////

template<index_t start, index_t end, index_t step, index_t index>
struct rotY_loop
{
  template<typename Expr, typename Params>
  inline constexpr auto operator()(Expr&& expr, Params&& params) noexcept
  {
      QVar_t<300+index, 1> angle(2*PI*params[index]);
        return all(ry(angle, sel<index>(expr)));
  }

  template<typename Expr, typename Params, typename Offset>
  inline constexpr auto operator()(Expr&& expr, Params&& params, Offset) noexcept
  {
      QVar_t<400+index+Offset::_I, 1> angle(2*PI*params[index+Offset::_I]);
        return all(ry(angle, sel<index>(expr)));
  }
};

template<index_t start, index_t end, index_t step, index_t index>
struct rotZ_loop
{
  template<typename Expr, typename Params>
  inline constexpr auto operator()(Expr&& expr, Params&& params) noexcept
  {
      QVar_t<500+index, 1> angle(2*PI*params[index]);
        return all(rz(angle, sel<index>(expr)));
  }

  template<typename Expr, typename Params, typename Offset>
  inline constexpr auto operator()(Expr&& expr, Params&& params, Offset) noexcept
  {
      QVar_t<600+index+Offset::_I, 1> angle(2*PI*params[index+Offset::_I]);
        return all(rz(angle, sel<index>(expr)));
  }
};

template<index_t start, index_t end, index_t step, index_t index>
struct CNOT_inner_loop
{
  template<typename Expr>
  inline constexpr auto operator()(Expr&& expr) noexcept
  {
      return all(cnot(sel<start>(), sel<index+1>(expr)));
  }

};

template<index_t start, index_t end, index_t step, index_t index>
struct CNOT_outer_loop
{
  template<typename Expr>
  inline constexpr auto operator()(Expr&& expr) noexcept
  {
      return utils::static_for<index, end, 1, CNOT_inner_loop>(expr);
  }
```

```
  };

  template <int size, typename paramType>
  constexpr auto genIsingAnsatz(paramType params){

    //Initialize zeros
    auto s0 = init();

    //Parameter offsets
    auto offset1 = VQE_Loops::static_int<size>();
    auto offset2 = VQE_Loops::static_int<2*size>();
    auto offset3 = VQE_Loops::static_int<3*size>();

    //Linear efficient anzats (Qiskit)
    auto RotY1 = utils::static_for<0, size-1, 1, rotY_loop>(s0, params);
    auto RotZ1 = utils::static_for<0, size-1, 1, rotZ_loop>(RotY1, params, offset1);
    auto cnots = utils::static_for<0, size-2, 1, CNOT_outer_loop>(RotZ1);
    auto RotY2 = utils::static_for<0, size-1, 1, rotY_loop>(cnots, params, offset2);
    auto RotZ2 = utils::static_for<0, size-1, 1, rotZ_loop>(RotY2, params, offset3);

    return RotZ2;
  };

}; //namespace Anzatses

#endif //VQE_ANSATZES_HPP
```

Listing E.9: VQE_Ansatzes.hpp code of the QPack benchmark in LibKet.

```
/** @file Qpack/include/VQE_Hamiltonians.hpp

 @brief Qpack VQE_Hamiltonians header file

 @author Huub Donkers

 @defgroup qpack libket
*/

#pragma once
#ifndef VQE_HAMILTONIANS_HPP
#define VQE_HAMILTONIANS_HPP

namespace Hamiltonians{

  //Struct that containt Hamiltonian of Pauli matrices
  template<int numTerms, int numQubits, char... terms>
  struct Hamiltonian{
    static constexpr const int _numTerms = numTerms;
    static constexpr const int _qubits = numQubits;
    static constexpr const char _terms[] = {terms...};
    float _coeffs[numTerms] = {};

    // Creates a new Hamiltonian with additional pauli operator
   template<char pauli>
   using add_pauli = Hamiltonian<numTerms, numQubits, terms..., pauli>;

  };


  ////////////Random Hamiltonian//////////////////////////////////////

 //Loops over all qubits and assigns Z or I gate bases on qubit index and term index
  template<index_t start, index_t end, index_t step, index_t index>
  struct z_loop_inner
  {
    static constexpr auto pauli_mat(std::integral_constant<bool, false>){ return 'I';}
    static constexpr auto pauli_mat(std::integral_constant<bool, true>){  return 'Z';}

    template<typename H, typename TermIndex>
    inline constexpr auto operator()(H&& h, TermIndex) noexcept
    {
      //Get pauli char based on index
      constexpr char pauli = pauli_mat(std::integral_constant<bool,index==TermIndex::_I>{});

      //Return new Hamiltonian with added pauli gate
      return typename std::decay<H>::type::template add_pauli<pauli> {};
    }
  };

  //Loops over all Hamiltonian terms
  template<index_t start, index_t end, index_t step, index_t index>
  struct z_loop_outer
```

```
  {
   template<typename H, typename size>
    inline constexpr auto operator()(H&& h, size) noexcept
    {
      constexpr auto termIndex = VQE_Loops::static_int<index>();
      return utils::static_for<0, size::_I-1, 1, z_loop_inner>(H{}, termIndex);
    }
  };

template<int size=1>
constexpr auto genDiagonalHamiltonian(){

  //Random constant floats (100)
 float random_floats[] = {-0.5744, 0.8371, -0.3355, 0.5271, 0.4706, -0.8478, 0.6888, 0.3292, -0.9607,
     0.3222, -0.5105, -0.9897, -0.7715, 0.7951, 0.9434, -0.4727, 0.9197, 0.4264, -0.4883, 0.621, 0.4916,
     0.1467, 0.0373, -0.28, -0.7621, 0.2613, 0.5845, -0.1643, -0.3134, -0.7457, -0.0752, 0.3431, 0.3516,
     -0.2646, -0.2114, -0.4217, 0.3649, -0.3971, 0.9196, 0.2383, -0.6071, -0.939, 0.6689, -0.2127,
     -0.3769, 0.1525, -0.088, 0.2868, 0.8808, -0.0309, -0.4611, 0.1063, -0.6852, 0.4813, -0.2738, 0.2184,
     0.9295, 0.9794, -0.5543, 0.156, -0.9739, 0.6748, 0.5962, -0.2649, -0.3602, 0.4841, 0.3065, -0.2893,
     0.1669, -0.2922, 0.6322, 0.9242, -0.7872, -0.0055, 0.4468, -0.3218, 0.9994, -0.4223, 0.3314, 0.1887,
     -0.242, -0.3767, -0.9138, 0.8556, -0.3891, -0.004, -0.1904, 0.3853, 0.1728, 0.8095, 0.4985, -0.5314,
     -0.0891, 0.5509, -0.2526, 0.1154, 0.1072, -0.3161, 0.2092, 0.8787};

  //Static constexpressions
  constexpr auto static_size = VQE_Loops::static_int<size>(); //Size of the size by size Hamiltonian
 constexpr auto neg_index = VQE_Loops::static_int<-1>();     //Negative index for only I loop

  //Append pauli Z operators on each diagonal;
  using Ham_Z = decltype(utils::static_for<0, size-1, 1, z_loop_outer>(Hamiltonian<size, size>(),
     static_size));

  //Declare Hamiltonian H
 Ham_Z H;

  //Load coeffients to hamiltonian (op to 100)
 for(int i = 0; i < size; i++){
        H._coeffs[i] = random_floats[i];
    }

  return H;
};

//////////////////Ising Chain Hamiltonian////////////////////////////////

  //Loops over all qubits and assigns ZZ or I gate bases on qubit index and term index for neighbouring
     qubits
  template<index_t start, index_t end, index_t step, index_t index>
  struct zz_loop_inner
  {
    static constexpr auto pauli_mat(std::integral_constant<bool, false>){ return 'I';}
    static constexpr auto pauli_mat(std::integral_constant<bool, true>){  return 'X';}

    template<typename H, typename TermIndex>
    inline constexpr auto operator()(H&& h, TermIndex) noexcept
    {
      //Get pauli char based on index (neighbours)
      constexpr char pauli = pauli_mat(std::integral_constant<bool,index==TermIndex::_I or index==
     TermIndex::_I+1>{});

      //Return new Hamiltonian with added pauli gate
      return typename std::decay<H>::type::template add_pauli<pauli> {};
    }
  };

  //Loops over all Hamiltonian terms
  template<index_t start, index_t end, index_t step, index_t index>
  struct zz_loop_outer
  {
    template<typename H, typename size>
    inline constexpr auto operator()(H&& h, size) noexcept
    {
      constexpr auto termIndex = VQE_Loops::static_int<index>();
      return utils::static_for<0, size::_I-1, 1, zz_loop_inner>(H{}, termIndex);
    }
  };

  //Loops over all qubits and assigns X or I gate bases on qubit index and term index
  template<index_t start, index_t end, index_t step, index_t index>
  struct x_loop_inner
  {
    static constexpr auto pauli_mat(std::integral_constant<bool, false>){ return 'I';}
    static constexpr auto pauli_mat(std::integral_constant<bool, true>){  return 'Z';}

    template<typename H, typename TermIndex>
```

```
    inline constexpr auto operator()(H&& h, TermIndex) noexcept
    {
      //Get pauli char based on index
      constexpr char pauli = pauli_mat(std::integral_constant<bool,index==TermIndex::_I>{});

      //Return new Hamiltonian with added pauli gate
      return typename std::decay<H>::type::template add_pauli<pauli> {};
    }
  };

  //Loops over all Hamiltonian terms
  template<index_t start, index_t end, index_t step, index_t index>
  struct x_loop_outer
  {
    template<typename H, typename size>
    inline constexpr auto operator()(H&& h, size) noexcept
    {
    constexpr auto termIndex = VQE_Loops::static_int<index>();
    return utils::static_for<0, size::_I-1, 1, x_loop_inner>(H{}, termIndex);
    }
  };

  //Generate hamiltonian describing the energies of a 1-D Ising chain
  template<int size>
  constexpr auto genIsingHamiltonian(){

    //Constexpr terms based on chain size
    constexpr auto static_size = VQE_Loops::static_int<size>();
    constexpr int numTerms = 2*size-1;
    constexpr int numQubits = size;

    //Create hamiltonian with ZZ terms for neighbouring qubits
    using Ham_ZZ = decltype(utils::static_for<0, size-2, 1, zz_loop_outer>(Hamiltonian<numTerms, numQubits
     >(), static_size));

    //Append pauli X operators on each qubit
    using Ham_X = decltype(utils::static_for<0, size-1, 1, x_loop_outer>(Ham_ZZ(), static_size));

    //Declare Hamiltonian H
    Ham_X H;

    //Load coefficients in H
    float h = 1.0; //Tranverse field
    float J = 1.0; //Interaction strength
    for(int i = 0; i <= size-2; i++){ H._coeffs[i] = -J; }
    for(int i = 0; i <= size-1; i++){ H._coeffs[i+size-1] = -h; }

    return H;
  };

}; //namespace Hamiltonians

#endif //VQE_HAMILTONIANS_HPP
```

Listing E.10: QAOA_Hamiltonians.hpp code of the QPack benchmark in LibKet.

Python Code: Data processing

```python
import json
from os import listdir, remove, makedirs
import matplotlib.pyplot as plt
from math import log10, atan, pi
from matplotlib.ticker import MaxNLocator
from statistics import mean
from qpu_names import qpu_names
import numpy as np
from scipy.optimize import minimize
from matplotlib import gridspec

problem_names = {'MCP' : 'QAOA: MaxCut problem',
                 'DSP' : 'QAOA: Dominating set problem',
                 'MIS' : 'QAOA: Maximal independent set problem',
                 'TSP' : 'QAOA: Traveling salesperson problem',
                 'RH' : 'VQE: Random Hamiltonian',
                 'IC' : 'VQE: Ising chain'}

def loadBenchmarkData(data_dir):
    '''
    @brief Function to load in QPack benchmark data from selected data path

    @params data_dir : Path to data directory of QPack benchmark files

    @return Dictionary of relevant benchmark parameters used to compute
            benchmark scores
    '''

    benchmark_data = {}
    for qpu in listdir(f'{data_dir}'):
        benchmark_data[qpu] = {}
        for problem in listdir(f'{data_dir}/{qpu}'):
            benchmark_data[qpu][problem] = {}
            for size in listdir(f'{data_dir}/{qpu}/{problem}'):
                benchmark_data[qpu][problem][size] = {}

                benchmark_data[qpu][problem][size]['qjob average'] = 0
                benchmark_data[qpu][problem][size]['exec average'] = 0
                benchmark_data[qpu][problem][size]['cjob average'] = 0
                benchmark_data[qpu][problem][size]['exp val'] = 100
                benchmark_data[qpu][problem][size]['opt iterations'] = 0
                benchmark_data[qpu][problem][size]['total alg time'] = 0
                benchmark_data[qpu][problem][size]['total q time'] = 0
                benchmark_data[qpu][problem][size]['total c time'] = 0

                numReps = 0
                for rep in listdir(f'{data_dir}/{qpu}/{problem}/{size}'):
                    file = open(f'{data_dir}/{qpu}/{problem}/{size}/{rep}')
                    data = json.load(file)

                    #Add queue zeroes if missing
                    if 'Queue durations [ms]' not in data:
                        data['Queue durations [ms]'] = [0.0]

                    #Filter missing queue data
                    missing_idx = []
                    for index, queue_time in  enumerate(data['Queue durations [ms]']):
                        if queue_time < 0:
                            missing_idx.append(index)

                    for idx in reversed(missing_idx):
                        data['Queue durations [ms]'].pop(idx)
                        data['QJob durations [ms]'].pop(idx)
```

```
64
65                      #Load data into dict
66                      benchmark_data[qpu][problem][size]['qjob average'] += mean(data['QJob durations [ms]'
    ]) - mean(data['Queue durations [ms]'])
67                      benchmark_data[qpu][problem][size]['exec average'] += mean(data['Circuit execution
    durations [ms]'])
68                      benchmark_data[qpu][problem][size]['cjob average'] += mean(data['Optimizer durations [
    ms]'])
69                      benchmark_data[qpu][problem][size]['qubits'] = data['Qubits']
70                      benchmark_data[qpu][problem][size]['depth'] = data['Depth']
71                      benchmark_data[qpu][problem][size]['shots'] = data['Shots']
72                      benchmark_data[qpu][problem][size]['num params'] = len(data['Optimizer params'])
73                      if data['Expectation Value'] is None:
74                          print(f'{data_dir}/{qpu}/{problem}/{size}/{rep}')
75                      if data['Expectation Value'] < benchmark_data[qpu][problem][size]['exp val']:
76                          benchmark_data[qpu][problem][size]['exp val'] = data['Expectation Value']
77                      benchmark_data[qpu][problem][size]['opt exp val'] = data['Optimal Expectation Value']
78                      benchmark_data[qpu][problem][size]['opt iterations'] += data['Optimizer iterations']
79                      benchmark_data[qpu][problem][size]['total alg time'] += data['Total Algorithm duration
     [s]']
80                      benchmark_data[qpu][problem][size]['total q time'] += data['Total Quantum duration [s]
    ']
81                      benchmark_data[qpu][problem][size]['total c time'] += data['Total Classic duration [s]
    ']
82                      numReps += 1
83
84                  if numReps > 0:
85                      benchmark_data[qpu][problem][size]['qjob average'] /= numReps
86                      benchmark_data[qpu][problem][size]['exec average'] /= numReps
87                      benchmark_data[qpu][problem][size]['cjob average'] /= numReps
88                      benchmark_data[qpu][problem][size]['opt iterations'] /= numReps
89                      benchmark_data[qpu][problem][size]['total alg time'] /= numReps
90                      benchmark_data[qpu][problem][size]['total q time'] /= numReps
91                      benchmark_data[qpu][problem][size]['total q time'] /= numReps
92
93      return benchmark_data
94
95  def plotData(QPack_data, qpus = [], reduce=False, subdir=None):
96      '''
97      @brief Plots raw data metrics from QPack benchmark for each problem
98      '''
99
100     #Get all available QPU data if no subset is provided
101     if not qpus:
102         qpus = list(QPack_data.keys())
103
104     #Set problems to process
105     problems = []
106     for qpu in qpus:
107         for problem in list(QPack_data[qpu].keys()):
108             if problem not in problems:
109                 problems.append(problem)
110
111     #Get QueST data
112     file = open("QuEST_Scores.json")
113     QuEST_data = json.load(file)
114
115     #Loop over all problems
116     for problem in problems:
117
118         #Create figures
119         if reduce:
120             fig, ((ax1, ax2, ax3), (ax4, ax5, ax6)) = plt.subplots(2, 3)
121             fig.set_size_inches(8, 5)
122         else:
123             fig, ((ax1, ax2, ax3), (ax4, ax5, ax6), (ax7, ax8, ax9)) = plt.subplots(3, 3)
124             fig.set_size_inches(8,8)
125
126
127         legend = []
128
129         #Set maximum and minimum problem size variables
130         minSize = float('inf')
131         maxSize = 0
132
133         #Loop over all selected qpus
134         for qpu in qpus:
135
136             #If a qpu has not evaluated a problem, skip this qpu
137             if problem not in list(QPack_data[qpu].keys()):
138                 continue
139
140             #Add current QPU to legend
141             legend.append(str(qpu_names[int(qpu, base=16)]))
```

```
142
143             #Get problem sizes for this qpu
144             sizes = list(QPack_data[qpu][problem].keys())
145
146             #Retrieve data from JSON object
147             qjob_average = [QPack_data[qpu][problem][size]['qjob average'] for size in sizes]
148             exec_average = [QPack_data[qpu][problem][size]['exec average'] for size in sizes]
149             cjob_average = [QPack_data[qpu][problem][size]['cjob average'] for size in sizes]
150             numQubits = [QPack_data[qpu][problem][size]['qubits'] for size in sizes]
151             part_speed =  [QPack_data[qpu][problem][size]['depth']*QPack_data[qpu][problem][size]['shots'
    ]/(QPack_data[qpu][problem][size]['qjob average']/1000) for size in sizes]
152             part_acc = [(QuEST_data[problem]['exp val'][QuEST_data[problem]['sizes'].index(int(size[1:]))
    ]-QPack_data[qpu][problem][size]['exp val'])/QuEST_data[problem]['exp val'][QuEST_data[problem]['
    sizes'].index(int(size[1:]))] for size in sizes]
153             exp_val_average = [QPack_data[qpu][problem][size]['exp val'] for size in sizes]
154             opt_iterations = [QPack_data[qpu][problem][size]['opt iterations'] for size in sizes]
155             ttime = [QPack_data[qpu][problem][size]['total alg time'] for size in sizes]
156             qtime = [QPack_data[qpu][problem][size]['total q time'] for size in sizes]
157             ctime = [QPack_data[qpu][problem][size]['total c time'] for size in sizes]
158
159             int_sizes = [int(size[1:]) for size in sizes]
160
161             #Sort data and plot on axes
162             zipped = sorted(zip(int_sizes, qjob_average, exec_average, cjob_average, numQubits, part_speed
    , part_acc, exp_val_average, opt_iterations, ttime, qtime, ctime))
163
164             ax1.plot(list(item[0] for item in zipped), list(item[1] for item in zipped), marker = 'o',
    markersize=3)
165             ax2.plot(list(item[0] for item in zipped), list(item[2] for item in zipped), marker = 'o',
    markersize=3)
166             ax3.plot(list(item[0] for item in zipped), list(item[3] for item in zipped), marker = 'o',
    markersize=3)
167             ax4.plot(list(item[0] for item in zipped), list(item[7] for item in zipped), marker = 'o',
    markersize=3)
168             ax5.plot(list(item[0] for item in zipped), list(item[6] for item in zipped), marker = 'o',
    markersize=3)
169             ax6.plot(list(item[0] for item in zipped), list(item[8] for item in zipped), marker = 'o',
    markersize=3)
170
171             if not reduce:
172                 ax7.plot(list(item[0] for item in zipped), list(item[9] for item in zipped), marker = 'o',
     markersize=3)
173                 ax8.plot(list(item[0] for item in zipped), list(item[10] for item in zipped), marker = 'o'
    , markersize=3)
174                 ax9.plot(list(item[0] for item in zipped), list(item[11] for item in zipped), marker = 'o'
    , markersize=3)
175
176             #Find smallest and largest problem size
177             if zipped[0][0] < minSize:
178                 minSize = zipped[0][0]
179
180             if zipped[-1][0] > maxSize:
181                 maxSize = zipped[-1][0]
182
183         #Plot QuEST baseline
184         quest_zip = sorted(zip(QuEST_data[problem]['sizes'], QuEST_data[problem]['exp val']))
185         quest_sizes = list(item[0] for item in quest_zip)
186         quest_exps = list(item[1] for item in quest_zip)
187         start = quest_sizes.index(minSize)
188         end = quest_sizes.index(maxSize)
189         ax4.plot(quest_sizes[start:end+1], quest_exps[start:end+1] , color='red', marker = '', label="
    baseline")
190
191         # Force x-axis integers
192         ax1.xaxis.set_major_locator(MaxNLocator(integer=True))
193         ax2.xaxis.set_major_locator(MaxNLocator(integer=True))
194         ax3.xaxis.set_major_locator(MaxNLocator(integer=True))
195         ax4.xaxis.set_major_locator(MaxNLocator(integer=True))
196         ax5.xaxis.set_major_locator(MaxNLocator(integer=True))
197         ax6.xaxis.set_major_locator(MaxNLocator(integer=True))
198         if not reduce:
199             ax7.xaxis.set_major_locator(MaxNLocator(integer=True))
200             ax8.xaxis.set_major_locator(MaxNLocator(integer=True))
201             ax9.xaxis.set_major_locator(MaxNLocator(integer=True))
202
203         #y-axis scale
204         ax1.set_yscale("log")
205         ax2.set_yscale("log")
206         ax3.set_yscale("log")
207         if not reduce:
208             ax7.set_yscale("log")
209             ax8.set_yscale("log")
210             ax9.set_yscale("log")
211
```

```
        # Add titles and lables
        fig.suptitle(problem_names[str(problem)])
        ax1.set_title('Average QJob \n duration')
        ax1.set_xlabel("Problem size")
        ax1.set_ylabel("Runtime [ms]")

        ax2.set_title('Average circuit \n execution duration')
        ax2.set_xlabel("Problem size")
        ax2.set_ylabel("Runtime [ms]")

        ax3.set_title('Average optimizer \n duration')
        ax3.set_xlabel("Problem size")
        ax3.set_ylabel("Runtime [ms]")

        ax4.set_title('Output state')
        ax4.set_xlabel("Problem size")
        ax4.set_ylabel("Expectation value")
        ax4.legend()

        ax5.set_title('Error')
        ax5.set_xlabel("Problem size")
        ax5.set_ylabel("Relative Error")

        ax6.set_title('Optimizer iterations')
        ax6.set_xlabel("Problem size")
        ax6.set_ylabel("Iterations")

        if not reduce:
            ax7.set_title('Total runtime')
            ax7.set_xlabel("Problem size")
            ax7.set_ylabel("Runtime [s]")

            ax8.set_title('Total Quantum runtime')
            ax8.set_xlabel("Problem size")
            ax8.set_ylabel("Runtime [s]")

            ax9.set_title('Total Classic runtime')
            ax9.set_xlabel("Problem size")
            ax9.set_ylabel("Runtime [s]")

        #Add legend
        fig.legend(legend, loc='upper center', bbox_to_anchor=(0.5, 0.01), fancybox=True, shadow=True,
    ncol=4)

        #Set tick marks
        ax1.get_xaxis().tick_bottom()
        ax1.get_yaxis().tick_left()

        #Save result in plots directory
        fig.tight_layout()
        if subdir:
            plt.savefig(f'plots/{subdir}/{str(problem)}.pdf', bbox_inches='tight')
        else:
            plt.savefig(f'plots/{str(problem)}.pdf', bbox_inches='tight')

        plt.close()

def computeScores(QPack_data, qpus=[]):
    '''
    @brief Computes sub-scores for selected QPUs

    @param Qpack_data : Dataset containing all relevant benchmark data
                        per QPU per problem

    @param qpus : Optional subset of qpus

    @return QPack Scores for each QPU and problem set

    '''

    #Get all available QPU data if no subset is provided
    if not qpus:
        qpus = list(QPack_data.keys())

    #Get QueST data
    file = open("QuEST_Scores.json")
    QuEST_data = json.load(file)

    #Create struct to store score resutls for all QPUs
    scores = {}

    for qpu in qpus:
        scores[qpu] = {}

```

```
        problems = list(QPack_data[qpu].keys())

        for problem in problems:

            scores[qpu][problem] = {}

            #Get problem sizes for this qpu
            sizes = list(QPack_data[qpu][problem].keys())

            #Runtime sub-score
            ave_troughput = mean([QPack_data[qpu][problem][size]['depth']*QPack_data[qpu][problem][size]['shots']/(QPack_data[qpu][problem][size]['qjob average']/1000) for size in sizes])
            scores[qpu][problem]['runtime'] = log10(ave_troughput)

            #Accuracy sub-score
            rel_errors = []
            for size in sizes:
                QuEST_score = QuEST_data[problem]['exp val'][QuEST_data[problem]['sizes'].index(int(size[1:]))]
                abs_error = QuEST_score-QPack_data[qpu][problem][size]['exp val']+10e-10
                rel_errors.append(abs_error/QuEST_score)
            ave_error = mean(rel_errors)

            #Accuracy score mapping
            a_acc = 10
            b_acc = 5
            scores[qpu][problem]['accuracy'] = a_acc*(pi/2 - atan(b_acc*(ave_error)))

            #Scalability sub-score
            qjob_averages = [QPack_data[qpu][problem][size]['qjob average'] for size in sizes]
            int_sizes = [int(size[1:]) for size in sizes]
            zipped = sorted(zip(int_sizes, qjob_averages, rel_errors))

            if(len(sizes) > 1):

                #Normalise qjobtimes
                qjob_array = np.array([item[1] for item in zipped])
                if(np.amax(qjob_array) != np.amin(qjob_array)):
                    qjob_array[-1] += 0.001
                norm_in = (qjob_array - np.amin(qjob_array))/(np.amax(qjob_array)-np.amin(qjob_array))

                def min_func(a):
                    x = np.array([item[0] for item in zipped])
                    y = x**a
                    norm_y = (y - np.amin(y))/(np.amax(y)-np.amin(y))
                    sq_diff = (norm_in-norm_y)**2
                    err = sum(sq_diff)
                    return err

                fit = minimize(min_func, 1, method='Nelder-Mead')

                a_scale = 10
                b_scale = 0.75
                scores[qpu][problem]['scalability'] = a_scale*(pi/2 - atan(b_scale*fit.x[0]))

            else:
                scores[qpu][problem]['scalability'] = 0

            #Capacity score
            max_size = 0
            threshold = 0.25
            for i in range(len(sizes)):
                if(zipped[i][2] < threshold and zipped[i][0] > max_size):
                    max_size = zipped[i][0]

            if(max_size):
                scores[qpu][problem]['capacity'] = QPack_data[qpu][problem][f'N{max_size:02d}']['qubits']
            else:
                scores[qpu][problem]['capacity'] = 0

        #Overall sub-scores
        scores[qpu]['Overall'] = {}
        scores[qpu]['Overall']['runtime'] = mean([scores[qpu][problem]['runtime'] for problem in problems])
        scores[qpu]['Overall']['accuracy'] = mean([scores[qpu][problem]['accuracy'] for problem in problems])
        scores[qpu]['Overall']['scalability'] = mean([scores[qpu][problem]['scalability'] for problem in problems])
        scores[qpu]['Overall']['capacity'] = mean([scores[qpu][problem]['capacity'] for problem in problems])

    return scores

def plotQPUScores(qpu_scores, qpus=[], subdir=None, xlimit=None):
```

```
    '''
    @brief Creates plot for all QPU benchmark sub-scores

    @param qpu_scores : Dictionary containing QPack scores for each QPU

    @param qpus : Optional subset of qpus
    '''

    #Get all available QPU data if no subset is provided
    if not qpus:
        qpus = list(qpu_scores.keys())

    #Loop over all selected qpus
    for qpu in qpus:

        #Get problems for this QPU
        problems = list(qpu_scores[qpu].keys())

        #Set bar axis
        X = np.arange(len(problems))
        fig = plt.figure()
        ax = fig.add_axes([0,0,1,1])
        ax.set_yticks(X, problems)
        ax.set_title(str(qpu_names[int(qpu, base=16)]))

        #Create plotting data lists
        plot_speed = []
        plot_accuracy = []
        plot_scale = []
        plot_cap = []
        for problem in problems:
            plot_speed.append(qpu_scores[qpu][problem]['runtime'])
            plot_accuracy.append(qpu_scores[qpu][problem]['accuracy'])
            plot_scale.append(qpu_scores[qpu][problem]['scalability'])
            plot_cap.append(qpu_scores[qpu][problem]['capacity'])

        #Plot all scores with some offset
        speed_bar = ax.barh(X + 0.20, plot_speed, 0.20)
        acc_bar = ax.barh(X + 0.00, plot_accuracy, 0.20)
        scale_bar = ax.barh(X - 0.20, plot_scale, 0.20)
        cap_bar = ax.barh(X - 0.40, plot_cap, 0.20)
        ax.legend(['Runtime', 'Accuracy', 'Scalability', 'Capacity'])

        #Add labels and score values
        ax.bar_label(speed_bar, fmt='%.2f',size= 10)
        ax.bar_label(acc_bar, fmt='%.2f', size= 10)
        ax.bar_label(scale_bar, fmt='%.2f', size= 10)
        ax.bar_label(cap_bar, fmt='%.2f', size= 10)
        ax.set_xlabel('Benchmark Score')
        if xlimit:
            ax.set_xlim(0, xlimit)

        #Save figure
        if subdir:
            plt.savefig(f'plots/{subdir}/benchmark_{str(qpu_names[int(qpu, base=16)])}.pdf', bbox_inches='
    tight')
        else:
            plt.savefig(f'plots/benchmark_{str(qpu_names[int(qpu, base=16)])}.pdf', bbox_inches='tight')

        plt.close()

def plotRadar(qpu_scores, qpus=[], reduce=False, subdir=None):
    '''
    @brief Creates plots for final benchmark scores on radar charts

    @param qpu_scores : Dictionary containing QPack scores for each QPU

    @param qpus : Optional subset of qpus
    '''

    #Get all available QPU data if no subset is provided
    if not qpus:
        qpus = list(qpu_scores.keys())

    #Setup radar plot
    categories = ['Runtime', 'Accuracy', 'Scalability', 'Capacity']
    categories = [*categories, categories[0]]
    label_loc = np.linspace(start=0, stop=2 * np.pi, num=len(categories))

    if not reduce:
        fig = plt.figure(figsize=(8, 4.5))
        gs  = gridspec.GridSpec(ncols=19, nrows=10, figure=fig)
        ax1 = fig.add_subplot(gs[0:9, 0:9], projection = 'polar')
        ax2 = fig.add_subplot(gs[2:6, 13:18])
```

```
456
457     else:
458         fig = plt.figure(figsize=(4.5, 4.5))
459         gs  = gridspec.GridSpec(ncols=10, nrows=10, figure=fig)
460         ax1 = fig.add_subplot(gs[0:9, 0:9], projection = 'polar')
461
462
463     #Compute total QPU score
464     max_score = 0
465     max_subscore = 0
466     for i, qpu in enumerate(qpus):
467         total_score = 0.5*(qpu_scores[qpu]['Overall']['runtime']+qpu_scores[qpu]['Overall']['scalability'
        ])*(qpu_scores[qpu]['Overall']['accuracy']+qpu_scores[qpu]['Overall']['capacity'])
468         ax1.plot(label_loc, [qpu_scores[qpu]['Overall']['runtime'], qpu_scores[qpu]['Overall']['accuracy'
        ], qpu_scores[qpu]['Overall']['scalability'], qpu_scores[qpu]['Overall']['capacity'], qpu_scores[qpu
        ]['Overall']['runtime']], label=str(qpu_names[int(qpu, base=16)]))
469         ax1.fill(label_loc, [qpu_scores[qpu]['Overall']['runtime'], qpu_scores[qpu]['Overall']['accuracy'
        ], qpu_scores[qpu]['Overall']['scalability'], qpu_scores[qpu]['Overall']['capacity'], qpu_scores[qpu
        ]['Overall']['runtime']], alpha=0.25)
470         if not reduce:
471             ax2.bar_label(ax2.bar(i, total_score, 0.35), fmt='%.1f',size= 8)
472         if total_score > max_score:
473             max_score = total_score
474         subs = [qpu_scores[qpu]['Overall']['runtime'], qpu_scores[qpu]['Overall']['scalability'],
        qpu_scores[qpu]['Overall']['accuracy'], qpu_scores[qpu]['Overall']['capacity']]
475
476         if max(subs) > max_subscore:
477             max_subscore = max(subs)
478     #Set figure titles and legends
479     fig.suptitle('QPack benchmark results', size=13)
480     ax1.set_thetagrids(np.degrees(label_loc), labels=categories)
481     ax1.tick_params(axis='y', labelsize=8)
482     #ax1.legend(prop={'size': 10}, loc='upper right',  bbox_to_anchor=(1.3, 1.07))
483     ax1.legend(prop={'size': 9}, loc='lower left',  bbox_to_anchor=(0.6, 0.8))
484     ax1.set_ylim(0, 1.1*max_subscore)
485
486
487     if not reduce:
488         ax2.xaxis.set_major_locator(MaxNLocator(integer=True))
489         ax2.set_xticklabels([' ', *[qpu_names[int(qpu, base=16)] for qpu in qpus]], rotation=45, ha="right
        ", rotation_mode="anchor")
490         ax2.set_ylabel('Benchmark Score')
491         ax2.set_xlabel('Quantum device')
492         ax2.set_ylim([0, 1.25*max_score])
493
494     #Save figure
495     if subdir:
496         plt.savefig(f'plots/{subdir}/benchmark_result.pdf', bbox_inches='tight')
497     else:
498         plt.savefig('plots/benchmark_result.pdf', bbox_inches='tight')
499
500     plt.close()
501
502 def main():
503     '''
504     @brief Main function of QPack data processing
505     '''
506     #Set path benchmark data
507     data_dir = "/home/huub/qpack/qpack_output/benchmark_new"
508
509     #Select subset of qpus
510     #qpus = []
511     qpus = [
512             ['3010008', '7000000', '6010013', '9000001'],                          #local ideal simulators
513             ['300001B', '3000010', '3000013', '3000017', '3000020', '300002E'],  #local noisy simulators
514             ['10000001', '4010001', '6010030'],                                    #remote ideal simulators
515             ['402001B', '402002E', '4020020', '4020013', '4020010', '4020017']   #remote hardware
516             ]
517
518     #Process benchmark data
519     sub_dirs = ['loc_sims', 'loc_sims_noisy', 'rem_sims', 'rem_qpu']
520     limits = [26, 17, 26, 17] #Max scores
521     QPack_data = loadBenchmarkData(data_dir)
522
523     for i, sub_dir in enumerate(sub_dirs):
524         makedirs(f"plots/{sub_dir}",  exist_ok = True)
525         plotData(QPack_data, qpus=qpus[i], reduce=True, subdir=sub_dir)
526         scores = computeScores(QPack_data, qpus=qpus[i])
527         plotQPUScores(scores, subdir=sub_dir, xlimit=limits[i])
528         plotRadar(scores, qpus=qpus[i], reduce=False, subdir=sub_dir)
529
530     return 0
531
532
```

```
if __name__ == "__main__":
    main()
```

Listing F.1: Process QPack data & compute benchmark scores

```
# -*- coding: utf-8 -*-
"""
Created on Wed Feb  9 21:57:05 2022

@author: huub
"""

#Name lib:
qpu_names = {     'Constant1'                                : 0xAAA,
                  'Constant2'                                : 0xAAB,
                  'Constant3'                                : 0xAAC,
                  'Linear1'                                  : 0xBBA,
                  'Linear2'                                  : 0xBBB,
                  'Linear3'                                  : 0xBBC,
                  'Quadratic1'                               : 0xCCA,
                  'Quadratic2'                               : 0xCCB,
                  'Quadratic3'                               : 0xCCC,
                  'Exponential1'                             : 0xDDA,
                  'Exponential2'                             : 0xDDB,
                  'Exponential3'                             : 0xDDC,
                  'generic'                                  : 0x00000000,
                  'pyquil_visualizer'                        : 0x00000001,
                  'qasm2tex_visualizer'                      : 0x00000002,
                  'qiskit_visualizer'                        : 0x00000003,
                  'atos_qlm_feynman_simulator'               : 0x01000001,
                  'atos_qlm_linalg_simulator'                : 0x01000002,
                  'atos_qlm_stabs_simulator'                 : 0x01000003,
                  'atos_qlm_mps_simulator'                   : 0x01000004,
                  'qi_26_simulator'                          : 0x02000001,
                  'qi_34_simulator'                          : 0x02000002,
                  'qi_spin2'                                 : 0x02010001,
                  'qi_starmon5'                              : 0x02010002,
                  'qiskit_almaden_simulator'                 : 0x03000001,
                  'qiskit_armonk_simulator'                  : 0x03000002,
                  'qiskit_athens_simulator'                  : 0x03000003,
                  'qiskit_belem_simulator'                   : 0x03000004,
                  'qiskit_boeblingen_simulator'              : 0x03000005,
                  'qiskit_bogota_simulator'                  : 0x03000006,
                  'qiskit_brooklyn_simulator'                : 0x03000007,
                  'qiskit_burlington_simulator'              : 0x03000008,
                  'qiskit_cairo_simulator'                   : 0x03000009,
                  'qiskit_cambridge_simulator'               : 0x0300000A,
                  'qiskit_casablanca_simulator'              : 0x0300000B,
                  'qiskit_dublin_simulator'                  : 0x0300000C,
                  'qiskit_essex_simulator'                   : 0x0300000D,
                  'qiskit_guadalupe_simulator'               : 0x0300000E,
                  'qiskit_hanoi_simulator'                   : 0x0300000F,
                  'qiskit_jakarta_simulator'                 : 0x03000010,
                  'qiskit_johannesburg_simulator'            : 0x03000011,
                  'qiskit_kolkata_simulator'                 : 0x03000012,
                  'qiskit_lagos_simulator'                   : 0x03000013,
                  'qiskit_lima_simulator'                    : 0x03000014,
                  'qiskit_london_simulator'                  : 0x03000015,
                  'qiskit_manhattan_simulator'               : 0x03000016,
                  'qiskit_manila_simulator'                  : 0x03000017,
                  'qiskit_melbourne_simulator'               : 0x03000018,
                  'qiskit_montreal_simulator'                : 0x03000019,
                  'qiskit_mumbai_simulator'                  : 0x0300001A,
                  'qiskit_nairobi_simulator'                 : 0x0300001B,
                  'qiskit_ourense_simulator'                 : 0x0300001C,
                  'qiskit_paris_simulator'                   : 0x0300001D,
                  'qiskit_peekskill_simulator'               : 0x0300001E,
                  'qiskit_poughkeepsie_simulator'            : 0x0300001F,
                  'qiskit_quito_simulator'                   : 0x03000020,
                  'qiskit_rochester_simulator'               : 0x03000021,
                  'qiskit_rome_simulator'                    : 0x03000022,
                  'qiskit_rueschlikon_simulator'             : 0x03000023,
                  'qiskit_santiago_simulator'                : 0x03000024,
                  'qiskit_singapore_simulator'               : 0x03000025,
                  'qiskit_sydney_simulator'                  : 0x03000026,
                  'qiskit_tenerife_simulator'                : 0x03000027,
                  'qiskit_tokyo_simulator'                   : 0x03000028,
                  'qiskit_toronto_simulator'                 : 0x03000029,
                  'qiskit_valencia_simulator'                : 0x0300002A,
                  'qiskit_vigo_simulator'                    : 0x0300002B,
                  'qiskit_yorktown_simulator'                : 0x0300002C,
                  'qiskit_washington_simulator'              : 0x0300002D,
                  'qiskit_perth_simulator'                   : 0x0300002E,
```

```
                'qiskit_pulse_simulator'                    : 0x03010001,
                'Qiskit QASM Simulator'                     : 0x03010002,
                'qiskit_statevector_simulator'              : 0x03010003,
                'qiskit_unitary_simulator'                  : 0x03010004,
                'qiskit_aer_density_matrix_simulator'       : 0x03010005,
                'qiskit_aer_extended_stabilizer_simulator'  : 0x03010006,
                'qiskit_aer_matrix_product_state_simulator' : 0x03010007,
                'Qiskit Aer Simulator'                      : 0x03010008,
                'qiskit_aer_stabilizer_simulator'           : 0x03010009,
                'qiskit_aer_statevector_simulator'          : 0x0301000A,
                'qiskit_aer_superop_simulator'              : 0x0301000B,
                'qiskit_aer_unitary_simulator'              : 0x0301000C,
                'ibmq_almaden_simulator'                    : 0x04000001,
                'ibmq_armonk_simulator'                     : 0x04000002,
                'ibmq_athens_simulator'                     : 0x04000003,
                'ibmq_belem_simulator'                      : 0x04000004,
                'ibmq_boeblingen_simulator'                 : 0x04000005,
                'ibmq_bogota_simulator'                     : 0x04000006,
                'ibmq_brooklyn_simulator'                   : 0x04000007,
                'ibmq_burlington_simulator'                 : 0x04000008,
                'ibmq_cairo_simulator'                      : 0x04000009,
                'ibmq_cambridge_simulator'                  : 0x0400000A,
                'ibmq_casablanca_simulator'                 : 0x0400000B,
                'ibmq_dublin_simulator'                     : 0x0400000C,
                'ibmq_essex_simulator'                      : 0x0400000D,
                'ibmq_guadalupe_simulator'                  : 0x0400000E,
                'ibmq_hanoi_simulator'                      : 0x0400000F,
                'ibmq_jakarta_simulator'                    : 0x04000010,
                'ibmq_johannesburg_simulator'               : 0x04000011,
                'ibmq_kolkata_simulator'                    : 0x04000012,
                'ibmq_lagos_simulator'                      : 0x04000013,
                'ibmq_lima_simulator'                       : 0x04000014,
                'ibmq_london_simulator'                     : 0x04000015,
                'ibmq_manhattan_simulator'                  : 0x04000016,
                'ibmq_manila_simulator'                     : 0x04000017,
                'ibmq_melbourne_simulator'                  : 0x04000018,
                'ibmq_montreal_simulator'                   : 0x04000019,
                'ibmq_mumbai_simulator'                     : 0x0400001A,
                'ibmq_nairobi_simulator'                    : 0x0400001B,
                'ibmq_ourense_simulator'                    : 0x0400001C,
                'ibmq_paris_simulator'                      : 0x0400001D,
                'ibmq_peekskill_simulator'                  : 0x0400001E,
                'ibmq_poughkeepsie_simulator'               : 0x0400001F,
                'ibmq_quito_simulator'                      : 0x04000020,
                'ibmq_rochester_simulator'                  : 0x04000021,
                'ibmq_rome_simulator'                       : 0x04000022,
                'ibmq_rueschlikon_simulator'                : 0x04000023,
                'ibmq_santiago_simulator'                   : 0x04000024,
                'ibmq_singapore_simulator'                  : 0x04000025,
                'ibmq_sydney_simulator'                     : 0x04000026,
                'ibmq_tenerife_simulator'                   : 0x04000027,
                'ibmq_tokyo_simulator'                      : 0x04000028,
                'ibmq_toronto_simulator'                    : 0x04000029,
                'ibmq_valencia_simulator'                   : 0x0400002A,
                'ibmq_vigo_simulator'                       : 0x0400002B,
                'ibmq_yorktown_simulator'                   : 0x0400002C,
                'ibmq_washington_simulator'                 : 0x0400002D,
                'ibmq_perth_simulator'                      : 0x0400002E,
                'IBMQ QASM Simulator'                       : 0x04010001,
                'ibmq_almaden'                              : 0x04020001,
                'ibmq_armonk'                               : 0x04020002,
                'ibmq_athens'                               : 0x04020003,
                'ibmq_belem'                                : 0x04020004,
                'ibmq_boeblingen'                           : 0x04020005,
                'ibmq_bogota'                               : 0x04020006,
                'ibmq_brooklyn'                             : 0x04020007,
                'ibmq_cairo'                                : 0x04020008,
                'ibmq_burlington'                           : 0x04020009,
                'ibmq_cambridge'                            : 0x0402000A,
                'ibmq_casablanca'                           : 0x0402000B,
                'ibmq_dublin'                               : 0x0402000C,
                'ibmq_essex'                                : 0x0402000D,
                'ibmq_guadalupe'                            : 0x0402000E,
                'ibmq_hanoi'                                : 0x0402000F,
                'IBMQ Jakarta'                              : 0x04020010,
                'ibmq_johannesburg'                         : 0x04020011,
                'ibmq_kolkata'                              : 0x04020012,
                'ibmq_lagos'                                : 0x04020013,
                'ibmq_lima'                                 : 0x04020014,
                'ibmq_london'                               : 0x04020015,
                'ibmq_manhattan'                            : 0x04020016,
                'IBMQ Manila'                               : 0x04020017,
                'ibmq_melbourne'                            : 0x04020018,
                'ibmq_montreal'                             : 0x04020019,
```

```
                  'ibmq_mumbai'                              : 0x0402001A,
                  'IBMQ Nairobi'                             : 0x0402001B,
                  'ibmq_ourense'                             : 0x0402001C,
                  'ibmq_paris'                               : 0x0402001D,
                  'ibmq_peekskill'                           : 0x0402001E,
                  'ibmq_poughkeepsie'                        : 0x0402001F,
                  'IBMQ Quito'                               : 0x04020020,
                  'ibmq_rochester'                           : 0x04020021,
                  'ibmq_rome'                                : 0x04020022,
                  'ibmq_rueschlikon'                         : 0x04020023,
                  'ibmq_santiago'                            : 0x04020024,
                  'ibmq_singapore'                           : 0x04020025,
                  'ibmq_sydney'                              : 0x04020026,
                  'ibmq_tenerife'                            : 0x04020027,
                  'ibmq_tokyo'                               : 0x04020028,
                  'ibmq_toronto'                             : 0x04020029,
                  'ibmq_valencia'                            : 0x0402002A,
                  'ibmq_vigo'                                : 0x0402002B,
                  'ibmq_yorktown'                            : 0x0402002C,
                  'ibmq_washington'                          : 0x0402002D,
                  'IBMQ Perth'                               : 0x0402002E,
                  'openql_cc_light_compiler'                 : 0x05000001,
                  'openql_cc_light17_compiler'               : 0x05000002,
                  'openql_qx_compiler'                       : 0x05000003,
                  'rigetti_aspen_8_simulator'                : 0x06000001,
                  'rigetti_aspen_9_simulator'                : 0x06000002,
                  'rigetti_aspen_10_simulator'               : 0x06000003,
                  'rigetti_9q_simulator'                     : 0x06010001,
                  'rigetti_9q_square_simulator'              : 0x06010002,
                  'Rigetti 16Q L QVM'                        : 0x06010013, # (local opt)
                  'Rigetti 16Q R QVM'                        : 0x06010030, # (remote)
                  'rigetti_16q_square_simulator'             : 0x06010004,
                  'rigetti_aspen_8'                          : 0x06020001,
                  'rigetti_aspen_9'                          : 0x06020002,
                  'rigetti_aspen_10'                         : 0x06020003,
                  'QuEST Simulator'                          : 0x07000000,
                  'QX'                                       : 0x08000000,
                  'Cirq Simulator'                           : 0x09000001,
                  'cirq_bristlecone_simulator'               : 0x09000002,
                  'cirq_foxtail_simulator'                   : 0x09000003,
                  'cirq_sycamore_simulator'                  : 0x09000004,
                  'cirq_sycamore23_simulator'                : 0x09000005,
                  'Ionq Simulator'                           : 0x10000001,
                  'ionq_qpu'                                 : 0x10000002}
qpu_names = {value: key for key, value in qpu_names.items()}
```

Listing F.2: Dict with QPU IDs and names

```
import json
import matplotlib.pyplot as plt
from statistics import mean
from os import listdir, remove

#Custom JSON encoder, lists have no line breaks
class MyJSONEncoder(json.JSONEncoder):

  def iterencode(self, o, _one_shot=False):
    list_lvl = 0
    for s in super(MyJSONEncoder, self).iterencode(o, _one_shot=_one_shot):
      if s.startswith('['):
        list_lvl += 1
        s = s.replace('\n', '').rstrip().replace(" ", "")
      elif 0 < list_lvl:
        s = s.replace('\n', '').rstrip().replace(" ", "")
        if len(s) > 1:
            s = s[0] + " " + s[1:]
        if s and s[-1] == ',':
          s = s[:-1] + self.item_separator
        elif s and s[-1] == ':':
          s = s[:-1] + self.key_separator
      if s.endswith(']'):
        list_lvl -= 1
      yield s

#Find available qpu's and problems
data_dir = "/home/huub/qpack/qpack_output/baseline_new"

#Data file
QuEST_dict = {}
qpu = '7000000'

for problem in listdir(f'{data_dir}/{qpu}'):
    QuEST_dict[problem] = {}
```

```
    QuEST_dict[problem]['sizes'] = []
    QuEST_dict[problem]['exp val'] = []
    QuEST_dict[problem]['opt exp val'] = []

    for size in listdir(f'{data_dir}/{qpu}/{problem}'):
        QuEST_dict[problem]['sizes'].append(int(size[1:]))

        exp_val = float('inf')
        for rep in listdir(f'{data_dir}/{qpu}/{problem}/{size}'):
            file = open(f'{data_dir}/{qpu}/{problem}/{size}/{rep}')
            data = json.load(file)
            if data['Expectation Value'] is None:
                print(f'Null... Removing {data_dir}/{qpu}/{problem}/{size}/{rep}')
                remove(f'{data_dir}/{qpu}/{problem}/{size}/{rep}')
                continue

            if data['Expectation Value'] < exp_val:
               exp_val = data['Expectation Value']

        QuEST_dict[problem]['exp val'].append(exp_val)
        QuEST_dict[problem]['opt exp val'].append(data['Optimal Expectation Value'])

#Store new result file
json.dump(QuEST_dict, open("QuEST_Scores.json", "w"), indent=2, cls=MyJSONEncoder)

#Plot baseline result
fig, axs = plt.subplots(2, 3)
fig.set_size_inches(8,8)

problems = ['MCP', 'DSP', 'MIS', 'TSP', 'RH', 'IC']

for i, r_ax in enumerate(axs):
    for j, ax in enumerate(r_ax):
        problem = problems[i*3+j]
        ax.set_title(f'{problem} score')
        ax.set_xlabel("Nodes")
        ax.set_ylabel("Score")
        zipped = sorted(zip(QuEST_dict[problem]['sizes'], QuEST_dict[problem]['exp val'], QuEST_dict[
     problem]['opt exp val']))

        ax.plot(list(item[0] for item in zipped), list(item[1] for item in zipped), marker = 'o')
        ax.plot(list(item[0] for item in zipped), list(item[2] for item in zipped), marker = 'o')

#Add legend
fig.suptitle("QuEST Simulator score results")
fig.legend(["VQA", "Optimal"], loc='upper center', bbox_to_anchor=(0.5, 0.01), fancybox=True, shadow=True,
      ncol=5)

fig.tight_layout()

plt.savefig('plots/QuEST_scores.png', bbox_inches='tight')
```

Listing F.3: Parse QuEST baseline data

```
import json
from random import random
from statistics import mean
from os import makedirs

#Custom JSON encoder, lists have no line breaks
class MyJSONEncoder(json.JSONEncoder):

  def iterencode(self, o, _one_shot=False):
    list_lvl = 0
    for s in super(MyJSONEncoder, self).iterencode(o, _one_shot=_one_shot):
      if s.startswith('['):
        list_lvl += 1
        s = s.replace('\n', '').rstrip().replace(" ", "")
      elif 0 < list_lvl:
        s = s.replace('\n', '').rstrip().replace(" ", "")
        if len(s) > 1:
            s = s[0] + " " + s[1:]
        if s and s[-1] == ',':
          s = s[:-1] + self.item_separator
        elif s and s[-1] == ':':
          s = s[:-1] + self.key_separator
      if s.endswith(']'):
        list_lvl -= 1
      yield s

#Set mock data path
data_dir = "/home/huub/qpack/qpack_output/mock_data"

```

```
#Set some MCP parameters
MCP_scores = [0, 0, 1, 2, 4, 6, 8, 10, 12, 12, 14, 16, 18, 18, 20, 22, 24, 24, 26, 28, 30, 30, 32, 34, 36,
      36, 38, 40, 42, 42, 44, 46, 48, 48, 50, 52, 54, 54, 56, 58, 60, 60, 62, 64, 66, 66, 68, 70, 72, 72]
mcp_depth = [0, 0, 14, 32, 41] + [95+18*i for i in range(20)]

#Open QuEST baseline
file = open("QuEST_Scores.json")
QuEST_data = json.load(file)['MCP']

#Generate data parameters
noise = False
start = 2
end = 15

#Constant function 1
for size in range(start, end):
    data = {}

    iters = 50 + 5*size
    q_job_times = [(20 + noise*random()-0.5)*1 for i in range(iters)]
    c_job_times = [q_time + 100 for q_time in q_job_times]

    data['Average job duration [ms]'] = mean(q_job_times)
    data['Depth'] = mcp_depth[size]
    data['P'] = 3
    data['Shots'] = 4096
    data['QJob durations [ms]'] = q_job_times
    data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
    data['Optimal Expectation Value'] = -MCP_scores[size]
    data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    data['Optimizer durations [ms]'] = c_job_times
    data['Optimizer iterations'] = iters
    data['Qubits'] = size
    data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)] + noise*random()
    data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/AAA/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/AAA/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)

#Constant function 2
for size in range(start, end):
    data = {}

    iters = 50 + 10*size
    q_job_times = [(20 + noise*random()-0.5)*10 for i in range(iters)]
    c_job_times = [q_time + 100 for q_time in q_job_times]

    data['Average job duration [ms]'] = mean(q_job_times)
    data['Depth'] = mcp_depth[size]
    data['P'] = 3
    data['Shots'] = 4096
    data['QJob durations [ms]'] = q_job_times
    data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
    data['Optimal Expectation Value'] = -MCP_scores[size]
    data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    data['Optimizer durations [ms]'] = c_job_times
    data['Optimizer iterations'] = iters
    data['Qubits'] = size
    data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)] + noise*random() +
     1
    data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/AAB/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/AAB/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)


#Constantfunction 3
for size in range(start, end):
    data = {}

    iters = 50 + 15*size
    q_job_times = [(20 + noise*random()-0.5)*100 for i in range(iters)]
    c_job_times = [q_time + 100 for q_time in q_job_times]

```

```
    data['Average job duration [ms]'] = mean(q_job_times)
    data['Depth'] = mcp_depth[size]
    data['P'] = 3
    data['Shots'] = 4096
    data['QJob durations [ms]'] = q_job_times
    data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
    data['Optimal Expectation Value'] = -MCP_scores[size]
    data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    data['Optimizer durations [ms]'] = c_job_times
    data['Optimizer iterations'] = iters
    data['Qubits'] = size
    data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)]*(1-0.4*(size/end))
     + noise*random()
    data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/AAC/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/AAC/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)

#Linear function 1
for size in range(start, end):
    data = {}

    iters = 50 + 5*size
    q_job_times = [(10 + noise*random()-0.5)*size for i in range(iters)]
    c_job_times = [q_time + 100 for q_time in q_job_times]

    data['Average job duration [ms]'] = mean(q_job_times)
    data['Depth'] = mcp_depth[size]
    data['P'] = 3
    data['Shots'] = 4096
    data['QJob durations [ms]'] = q_job_times
    data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
    data['Optimal Expectation Value'] = -MCP_scores[size]
    data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    data['Optimizer durations [ms]'] = c_job_times
    data['Optimizer iterations'] = iters
    data['Qubits'] = size
    data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)] + noise*random()
    data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/BBA/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/BBA/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)

#linear function 2
for size in range(start, end):
    data = {}

    iters = 50 + 10*size
    q_job_times = [(10 + noise*random()-0.5)*10*size for i in range(iters)]
    c_job_times = [q_time + 100 for q_time in q_job_times]

    data['Average job duration [ms]'] = mean(q_job_times)
    data['Depth'] = mcp_depth[size]
    data['P'] = 3
    data['Shots'] = 4096
    data['QJob durations [ms]'] = q_job_times
    data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
    data['Optimal Expectation Value'] = -MCP_scores[size]
    data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    data['Optimizer durations [ms]'] = c_job_times
    data['Optimizer iterations'] = iters
    data['Qubits'] = size
    data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)] + noise*random() +
     1
    data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/BBB/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/BBB/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)

```

```
194
195 #linear function 3
196 for size in range(start, end):
197     data = {}
198
199     iters = 50 + 15*size
200     q_job_times = [(10 + noise*random()-0.5)*100*size for i in range(iters)]
201     c_job_times = [q_time + 100 for q_time in q_job_times]
202
203     data['Average job duration [ms]'] = mean(q_job_times)
204     data['Depth'] = mcp_depth[size]
205     data['P'] = 3
206     data['Shots'] = 4096
207     data['QJob durations [ms]'] = q_job_times
208     data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
209     data['Optimal Expectation Value'] = -MCP_scores[size]
210     data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
211     data['Optimizer durations [ms]'] = c_job_times
212     data['Optimizer iterations'] = iters
213     data['Qubits'] = size
214     data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)]*(1-0.4*(size/end))
        + noise*random()
215     data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
216     data['Total Classic duration [s]'] = sum(c_job_times)
217     data['Total Quantum duration [s]'] = sum(q_job_times)
218
219     #create directories if non-existant
220     makedirs(f'{data_dir}/BBC/MCP/N{size:02d}', exist_ok = True)
221
222     with open(f'{data_dir}/BBC/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
223         json.dump(data, outfile, indent=2, cls=MyJSONEncoder)
224
225 #quadratic function 1
226 for size in range(start, end):
227     data = {}
228
229     iters = 50 + 5*size
230     q_job_times = [(10 + noise*random()-0.5)*(size**1.5) for i in range(iters)]
231     c_job_times = [q_time + 100 for q_time in q_job_times]
232
233     data['Average job duration [ms]'] = mean(q_job_times)
234     data['Depth'] = mcp_depth[size]
235     data['P'] = 3
236     data['Shots'] = 4096
237     data['QJob durations [ms]'] = q_job_times
238     data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
239     data['Optimal Expectation Value'] = -MCP_scores[size]
240     data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
241     data['Optimizer durations [ms]'] = c_job_times
242     data['Optimizer iterations'] = iters
243     data['Qubits'] = size
244     data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)] + noise*random()
245     data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
246     data['Total Classic duration [s]'] = sum(c_job_times)
247     data['Total Quantum duration [s]'] = sum(q_job_times)
248
249     #create directories if non-existant
250     makedirs(f'{data_dir}/CCA/MCP/N{size:02d}', exist_ok = True)
251
252     with open(f'{data_dir}/CCA/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
253         json.dump(data, outfile, indent=2, cls=MyJSONEncoder)
254
255 #quadratric function 2
256 for size in range(start, end):
257     data = {}
258
259     iters = 50 + 10*size
260     q_job_times = [(10 + noise*random()-0.5)*(size**2) for i in range(iters)]
261     c_job_times = [q_time + 100 for q_time in q_job_times]
262
263     data['Average job duration [ms]'] = mean(q_job_times)
264     data['Depth'] = mcp_depth[size]
265     data['P'] = 3
266     data['Shots'] = 4096
267     data['QJob durations [ms]'] = q_job_times
268     data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
269     data['Optimal Expectation Value'] = -MCP_scores[size]
270     data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
271     data['Optimizer durations [ms]'] = c_job_times
272     data['Optimizer iterations'] = iters
273     data['Qubits'] = size
274     data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)] + noise*random() +
        1
275     data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
```

```
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/CCB/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/CCB/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)

#quadratic function 3
for size in range(start, end):
    data = {}

    iters = 50 + 15*size
    q_job_times = [(10 + noise*random()-0.5)*(size**3) for i in range(iters)]
    c_job_times = [q_time + 100 for q_time in q_job_times]

    data['Average job duration [ms]'] = mean(q_job_times)
    data['Depth'] = mcp_depth[size]
    data['P'] = 3
    data['Shots'] = 4096
    data['QJob durations [ms]'] = q_job_times
    data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
    data['Optimal Expectation Value'] = -MCP_scores[size]
    data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    data['Optimizer durations [ms]'] = c_job_times
    data['Optimizer iterations'] = iters
    data['Qubits'] = size
    data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)]*(1-0.4*(size/end))
     + noise*random()
    data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/CCC/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/CCC/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)


#exponential function 1
for size in range(start, end):
    data = {}

    iters = 50 + 5*size
    q_job_times = [(10 + noise*random()-0.5)*(1.5**size) for i in range(iters)]
    c_job_times = [q_time + 100 for q_time in q_job_times]

    data['Average job duration [ms]'] = mean(q_job_times)
    data['Depth'] = mcp_depth[size]
    data['P'] = 3
    data['Shots'] = 4096
    data['QJob durations [ms]'] = q_job_times
    data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
    data['Optimal Expectation Value'] = -MCP_scores[size]
    data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    data['Optimizer durations [ms]'] = c_job_times
    data['Optimizer iterations'] = iters
    data['Qubits'] = size
    data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)] + noise*random()
    data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/DDA/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/DDA/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)

#exponential function 2
for size in range(start, end):
    data = {}

    iters = 50 + 10*size
    q_job_times = [(10 + noise*random()-0.5)*(2**size) for i in range(iters)]
    c_job_times = [q_time + 100 for q_time in q_job_times]

    data['Average job duration [ms]'] = mean(q_job_times)
    data['Depth'] = mcp_depth[size]
    data['P'] = 3
    data['Shots'] = 4096
    data['QJob durations [ms]'] = q_job_times
```

```
    data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
    data['Optimal Expectation Value'] = -MCP_scores[size]
    data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    data['Optimizer durations [ms]'] = c_job_times
    data['Optimizer iterations'] = iters
    data['Qubits'] = size
    data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)] + noise*random() +
     1
    data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/DDB/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/DDB/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)

#exponential function 3
for size in range(start, end):
    data = {}

    iters = 50 + 15*size
    q_job_times = [(10 + noise*random()-0.5)*(3**size) for i in range(iters)]
    c_job_times = [q_time + 100 for q_time in q_job_times]

    data['Average job duration [ms]'] = mean(q_job_times)
    data['Depth'] = mcp_depth[size]
    data['P'] = 3
    data['Shots'] = 4096
    data['QJob durations [ms]'] = q_job_times
    data['Circuit execution durations [ms]'] = [q_time*0.1 for q_time in q_job_times]
    data['Optimal Expectation Value'] = -MCP_scores[size]
    data['Optimizer params'] = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
    data['Optimizer durations [ms]'] = c_job_times
    data['Optimizer iterations'] = iters
    data['Qubits'] = size
    data['Expectation Value'] = QuEST_data['exp val'][QuEST_data['sizes'].index(size)]*(1-0.4*(size/end))
     + noise*random()
    data['Total Algorithm duration [s]'] = sum(c_job_times) + sum(q_job_times)
    data['Total Classic duration [s]'] = sum(c_job_times)
    data['Total Quantum duration [s]'] = sum(q_job_times)

    #create directories if non-existant
    makedirs(f'{data_dir}/DDC/MCP/N{size:02d}', exist_ok = True)

    with open(f'{data_dir}/DDC/MCP/N{size:02d}/dataset00.json', 'w') as outfile:
        json.dump(data, outfile, indent=2, cls=MyJSONEncoder)
```

Listing F.4: Generate synthetic data sets