AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI**

INSTYTUT INFORMATYKI


Praca dyplomowa

# *Hybrid algorithms for workflow scheduling problem in quantum devices based on gate model*

*Zastosowanie hybrydowych rozwiązań dla problemu szeregowania aplikacji typu workflow dla komputera kwantowego o modelu bramkowym*


| | |
|---|---|
| Autor: | Julia Plewa, Joanna Sieńko |
| Kierunek studiów: | Informatyka |
| Opiekun pracy: | dr inż. Katarzyna Rycerz |


Kraków, 2021

**Acknowledgements**

First and foremost, we wish to express our deepest and most sincere gratitude to our supervisor, Dr. Katarzyna Rycerz, whose continuous and invaluable support motivated us to continue our research even in the toughest moments. Secondly, we would like to thank the Qiskit team for the supportive community they provide, and specifically Dr. Hiroshi Horii for his willingness to help us with some of the technical challenges. We would like to also thank Mr. Adam Glos for his eagerness to share his work and discuss ideas. Lastly, we also extend our gratitude to CYFRONET, as this research would not have been possible without the PLGrid Infrastructure.

**Streszczenie**

Obliczenia kwantowe stale zyskują na popularności, pomimo iż era supremacji kwantowej jest jeszcze daleko przed nami. Dzisiejsze komputery Noisy Intermediate-Scale Quantum (NISQ) znane są ze swoich ograniczonych rozmiarów oraz szumu, jaki generują. Próbując więc uruchomić większy problem na komputerze NISQ, należy zazwyczaj zmienić sposób reprezentacji tego problemu. Prowadzi to do bardziej skomplikowanych obwodów, a w rezultacie do większego zaszumienia wyników. W niniejszej pracy rozważany jest ten kompromis w kontekście konkretnego problemu optymalizacyjnego – szeregowania aplikacji typu workflow. Porównano trzy kodowania o różnej gęstości: one-hot, binarne i domain wall oraz przetestowano ich wydajność w zastawieniu z dwoma popularnymi hybrydowymi algorytmami kwantowo-klasycznymi: QAOA i VQE. W celu uzyskania jak najlepszych wyników wykorzystane zostały różne wartości parametrów tychże algorytmów, a także sprawdzono inne najnowocześniejsze ulepszenia, takie jak dedykowane danemu kodowaniu miksery QAOA. Eksperymenty przeprowadzono na symulatorze kwantowym, a najbardziej udane z nich powtórzono z dodanym modelem szumu, co poskutkowało zauważalnie gorszymi wynikami. Ostatecznie udowodniono, że pomimo swojej popularności kodowanie one-hot nie zawsze jest najlepsze. Użycie gęstszego kodowania, takiego jak binarne lub domain wall, może pozwolić na zakodowanie większych problemów, a czasem nawet na uzyskanie lepszych wyników.

**Abstract**

Although the age of quantum supremacy is still far ahead of us, quantum computing has been steadily growing in popularity. The Noisy Intermediate-Scale Quantum (NISQ) computers of today are known for their limited size and the noise they generate. Therefore, when trying to run a larger problem on a NISQ computer, one typically has to alter the way this problem is represented, which tends to result in a more complicated circuit and, in turn, more noisy results. In this thesis, we consider this trade-off in the context of a specific optimization problem – workflow scheduling. We compare three encoding schemes of varying density: one-hot, binary, and domain wall, and test their performance against two well-known hybrid quantum-classical algorithms: QAOA and VQE. In an attempt to obtain the best results possible, we try manipulating various parameters of the algorithms and test out other state-of-the-art improvements, such as dedicated QAOA mixers. The experiments are run on a simulator and the most successful experiments are later repeated with an added noise model, resulting in noticeably worse results. Ultimately, we prove that, despite its popularity, one-hot encoding is not always the best, and that using a denser encoding scheme, such as binary or domain wall, can allow for encoding larger problems and sometimes even for better results.

# Contents

**1 Preface** **6**
1.1 Motivation . . . 6
1.2 State of the art . . . 7
1.2.1 Quantum and hybrid quantum-classical algorithms . . . 7
1.2.2 Optimization problems . . . 7
1.2.3 Quantum algorithm optimizations . . . 8
1.2.4 Workflow scheduling . . . 10
1.3 Goals of this work . . . 10
1.4 Author contributions . . . 11
1.5 Content of this work . . . 11

**2 Quantum computing concepts** **13**
2.1 Qubits . . . 13
2.2 Dirac notation . . . 14
2.3 Tensor product . . . 14
2.4 Quantum gates . . . 15
2.5 Measurement . . . 17
2.6 Superposition . . . 19
2.7 Entanglement . . . 21
2.8 Quantum computers . . . 21

**3 Optimization problems in quantum computing** **24**
3.1 Problem representation . . . 24

3.1.1 Hamiltonian . . . . . . . . . . 24
3.1.2 Ising model . . . . . . . . . . 25
3.1.3 QUBO . . . . . . . . . . 26
3.2 Optimization algorithms . . . . . . . . . . 27
3.2.1 VQE . . . . . . . . . . 27
3.2.2 QAOA . . . . . . . . . . 30
3.2.3 Comparison . . . . . . . . . . 35
3.3 Encoding schemes . . . . . . . . . . 35
3.3.1 One-hot encoding . . . . . . . . . . 36
3.3.2 Binary encoding . . . . . . . . . . 38
3.3.3 Domain wall encoding . . . . . . . . . . 41
3.3.4 QAOA mixers . . . . . . . . . . 45
3.3.5 Comparison . . . . . . . . . . 47

**4 Workflow scheduling** **51**
4.1 Basic intuition . . . . . . . . . . 51
4.1.1 Objective function . . . . . . . . . . 52
4.1.2 Constraints . . . . . . . . . . 53
4.1.3 Optimized function . . . . . . . . . . 55
4.2 Formal definition . . . . . . . . . . 55

**5 Solution implementation** **58**
5.1 Encoding-dependent solution representations . . . . . . . . . . 58
5.1.1 One-hot encoding . . . . . . . . . . 59
5.1.2 Binary encoding . . . . . . . . . . 60
5.1.3 Domain wall encoding . . . . . . . . . . 61
5.2 Mixers . . . . . . . . . . 63
5.2.1 One-hot encoding . . . . . . . . . . 63
5.2.2 Domain wall encoding . . . . . . . . . . 64

**6 Experiment design** **66**
6.1 Optimization algorithms . . . . . . . . . . 66

6.1.1 Classical optimizers . . . . . . . . . . 67
6.1.2 Initial point selection . . . . . . . . . . 68
6.1.3 Initial state . . . . . . . . . . 69
6.1.4 QUBO parameter selection . . . . . . . . . . 69
6.1.5 Experiment randomization . . . . . . . . . . 72
6.1.6 Result evaluation metrics . . . . . . . . . . 73
6.2 Considered workflows . . . . . . . . . . 73
6.2.1 Small problem . . . . . . . . . . 74
6.2.2 Large problems . . . . . . . . . . 75

**7 Evaluation of the results** **78**
7.1 Smaller problem . . . . . . . . . . 79
7.1.1 One-hot encoding . . . . . . . . . . 79
7.1.2 Binary encoding . . . . . . . . . . 90
7.1.3 Domain wall encoding . . . . . . . . . . 97
7.1.4 Objective function weight selection . . . . . . . . . . 105
7.1.5 Encoding comparison . . . . . . . . . . 108
7.1.6 Noisy results . . . . . . . . . . 112
7.2 Larger problem . . . . . . . . . . 117

**8 Conclusions and future works** **123**
8.1 Achieved goals and observations . . . . . . . . . . 123
8.1.1 General findings . . . . . . . . . . 123
8.1.2 Problem-specific findings . . . . . . . . . . 124
8.2 Future work . . . . . . . . . . 125

**List of Figures** **127**

**List of Tables** **129**

**Bibliography** **132**

# Chapter 1

# Preface

## 1.1 Motivation

Quantum computing is a relatively young area of science, but since its inception, researchers have believed it to be the future of computing [61]. For a long time, this remained mostly theoretical, but in the last few years there's been a noticeable spike of interest in the subject from both researchers and the industry. This growing interest can be most easily demonstrated by the steadily growing number of research papers that are published annually[1] and the very promising findings many of them have been reporting [1, 74]. But quantum computing is no longer just a theoretical research subject – in 2018, Gartner placed it on its annual chart of emerging technologies, estimating it would be adapted by the mainstream within five to ten years [51]. As reported by McKinsey & Company, the number of quantum-related companies has been growing rapidly ever since [52]. We've also seen many of the leading technological companies either join this race or announce their plans to do so in the near future [5, 56, 14].

With the growing quality of quantum computers and algorithms, we are at an age when real-life applications of quantum computing are finally becoming feasible. In this thesis, we will focus on workflow scheduling, a problem derived from the equally fast-growing world of cloud computing.

[1] For example, an ArXiv (`https://arxiv.org`) search for the term *quantum computing* returns 852 papers published in 2010, 1369 papers published in 2015, and 3283 papers published in 2020.

## 1.2 State of the art

Quantum optimization and workflow scheduling are both rapidly growing research areas, but not much research has been done with regard to combining the two.

### 1.2.1 Quantum and hybrid quantum-classical algorithms

Although quantum physics dates back to the 19th century, the first theoretical groundwork for quantum computing wasn't laid until the 1980s [4]. Many early quantum algorithms were created with only theoretical applications in mind. An example of this could be the Deutsch-Jozsa algorithm from 1992 [15] or the more general Bernstein-Vazirani algorithm that was published a year later [6]. Other algorithms, such as Grover's database search algorithm from 1996 [28] or Shor's fast factorization algorithm from 1994 [62], do have potential practical applications, but their effectiveness is impaired by the limitations of today's quantum machines – both due to their size and the noise they generate.

To a certain extent, these problems can be bypassed by hybrid quantum-classical algorithms [7]. Such algorithms combine the power of quantum computing with the abundant resources provided by classical computing. Typically, they follow a similar pattern: a classical computer prepares and provides input to a quantum computer, the quantum computer performs some computations using this input, and afterwards it feeds the output of those computations back to the classical computer, which in turn processes this data further and feeds it back to the quantum computer for another iteration. A very common application of this approach can be found in solving optimization problems.

### 1.2.2 Optimization problems

The role of quantum computing in solving NP-hard problems can be understood quite intuitively, even without considering any of the specifics. Certain characteristics of quantum computing, such as superposition, allow us to consider multiple states from the configuration space at the same time. This characteristic is very valuable when it comes to problems that would normally require exponential time, as is the case for NP-hard optimization problems [11]. Recent research in the field of quantum optimization has been quite promising

with achievements in fields such as biology [41, 50], resource distribution [23], machine learning [27], or finance [49].

A very well-known and versatile hybrid quantum-classical algorithm is the Quantum Approximate Optimization Algorithm (QAOA) introduced in 2014 [19]. This heuristic algorithm can be used for various combinatorial optimization problems. Notably, for a period of time, when applied to the MAX-E3LIN2 combinatorial problem of bounded occurrence, it was proven to have an approximation ratio better than that of any classical algorithm [20]. An implementation of this algorithm can be found in frameworks such as IBM's Qiskit[2] or Xanadu's PennyLane[3].

Another fundamental algorithm used for optimization is the Variational Quantum Eigensolver (VQE) introduced by Peruzzo et al. in 2014 [55]. In some ways, this algorithm can be seen as a generalization of QAOA. Just as QAOA, the VQE algorithm is a hybrid quantum-classical algorithm, and it utilizes a quantum subroutine run inside a classical optimization loop. Implementations of VQE can also be found in numerous libraries, such as Qiskit[4] or Pennylane[5].

An approach alternative to quantum optimization is utilized in quantum annealing. This algorithm is based on classical simulated annealing, and in the current formulation was first described in 1998 [34]. Quantum annealing is supported by the hardware created commercially by D-Wave Systems [33]. Such machines present themselves as an alternative to classical general-use quantum machines, but their abilities are a bit more limited.

While the algorithms mentioned above represent the most popular and well-researched approaches to quantum optimization, numerous other methods and advancements have been proposed in recent years [37, 29, 25].

### 1.2.3 Quantum algorithm optimizations

While considerable advancements are being made in regard to creating new and more powerful quantum software, it is important to note that the main limitations actually stem from

[2]https://qiskit.org/documentation/stubs/qiskit.aqua.algorithms.QAOA.html
[3]https://pennylane.readthedocs.io/en/stable/code/qml_qaoa.html
[4]https://qiskit.org/documentation/stubs/qiskit.aqua.algorithms.VQE.html
[5]https://pennylane.readthedocs.io/en/stable/_modules/pennylane/vqe/vqe.html

the quality of quantum hardware. We do not yet have access to quantum machines that are sufficiently noise-free and have enough qubits for us to be able to reach the age of *quantum supremacy* [61]. In order to make the best of the Noisy Intermediate-Scale Quantum (NISQ) machines of today [8], many researchers focus on finding different ways to fit larger problems on a limited number of qubits. This can be achieved either by finding more space-efficient methods of representing problems or by finding ways to split up larger problems into smaller portions.

One-hot encoding is the state-of-the-art encoding scheme in both machine learning and in quantum computing. However, as it is not the most space-efficient, other methods have been proposed and analyzed. Binary encoding is another commonly known encoding scheme – it is much more space-efficient than one-hot encoding, however in the context of quantum computing it is known to result in more complex circuits. This trade-off between space efficiency and circuit depth has been researched extensively [26, 22, 13]. Other approaches proposed in recent years include domain wall encoding [12, 13] and minimal encoding [66].

Another interesting development is the introduction of dedicated mixing operators (also known as *mixers*) [30, 69]. These mixers can be used in combination with QAOA to limit the configuration space to some feasible subspace in order to avoid configurations representing incorrect states altogether. This approach can also be used in combination with different encoding schemes [12]. Custom QAOA mixers have recently been introduced in Qiskit and can also be found in other frameworks, such as Pennylane[6].

Since the quantum resources available today are still quite limited, researchers have also considered ways of splitting various optimization problems into smaller portions before submitting them to a quantum computer. Kurowski et al. created a time-window-based heuristic for the Job Shop Scheduling problem [36, 44]. The D-Wave library has also recently been expanded with a component called Hybrid that allows for problem decomposition[7].

[6] https://pennylane.readthedocs.io/en/stable/code/api/pennylane.qaoa.mixers.xy_mixer.html

[7] https://docs.ocean.dwavesys.com/projects/hybrid/en/latest/

### 1.2.4 Workflow scheduling

Workflow scheduling is a combinatorial optimization problem whose goal is to assign a series of tasks to some available resources while meeting certain QoS requirements (such as a predefined time limit) and minimizing the overall cost. With the growing prevalence of cloud computing, resource allocation and synchronization problems have been researched extensively. Recent solutions of the workflow scheduling problem include adaptive algorithms [32], cooperative evolutionary algorithms [70], various heuristics and meta-heuristics [10, 47], as well as many other approaches [40, 53, 63].

Advancements are also being made with regard to solving optimization and scheduling problems using quantum algorithms [71, 17]. For instance, the Job Shop Scheduling problem has had promising results in combination with the aforementioned sliding-window heuristic [36, 44]. In terms of workflow scheduling, some research has been done utilizing VQE [64] and quantum annealing [67, 68] in combination with the one-hot encoding scheme.

## 1.3 Goals of this work

The main goal of this thesis was to investigate the capability of the previously mentioned hybrid algorithms, QAOA and VQE, to optimize the workflow scheduling problem. In particular, this includes:

- to analyze the known implementations of workflow scheduling problem and introduce improvements,
- to explore various encoding schemes and utilize their advantages in order to solve larger problems,
- to test and compare the performance of QAOA and VQE,
- to compare the available classical optimizers used as classical subroutines of these hybrid algorithms,
- to research other possible improvements such as QAOA mixers,
- to test out the performance of the proposed solution on a simulator using noise models.

## 1.4 Author contributions

The conducted experiments rely on various parameters, such as the encoding, the algorithm, the optimizer, and the QAOA mixer. The authors chose to divide their work based on the implemented encoding schemes in the following way:

- Julia Plewa: one-hot and binary encoding,
- Joanna Sieńko: domain wall encoding and result aggregation.

For each encoding, the authors tested the performance of QAOA and VQE, compared their efficiency for different classical optimizers, and – when possible – compared the performance of QAOA paired with either the default or an encoding-specific mixer. After running the experiments on a quantum simulator, the samples with the best results were then run again using real hardware noise models and compared to the previous results.

## 1.5 Content of this work

This thesis focuses on applying two hybrid quantum-classical algorithms – QAOA and VQE – to an optimization problem known as workflow scheduling. In this chapter, we discussed the current state of research in this area, as well as the goals of this thesis. Chapter 2 introduces some important quantum computing concepts, both high- and low-level. In Chapter 3, we discuss how an optimization problem can be represented on a quantum computer and how the different representations influence the quality and the cost of the results. Chapter 4 defines the workflow scheduling problem and introduces its formal definition. In Chapter 5, we delve into the specifics of the proposed solution and describe how workflow scheduling in particular can be represented on a quantum computer. Chapter 6 explains the reasoning behind certain design decisions and introduces the specific instances of workflow scheduling that are used in this thesis. Finally, Chapters 7 and 8 present and evaluate the obtained results, as well as provide ideas for future work.

d

# Chapter 2

# Quantum computing concepts

In this section, we declare some basic quantum computing concepts that are instrumental in understanding the ideas discussed in this thesis.

## 2.1 Qubits

A classical computer operates on bits, which are either 0's or 1's. In order to store a piece of information, the computer creates a string of zeros and ones, such as 10011010. Similarly, a quantum computer operates on qubits [46]. The state of a qubit resembles a kind of box, called a *ket* and denoted as $|\ \rangle$, into which a value representing the state is placed. Therefore, there are two distinguishable states that a qubit can store: $|0\rangle$ and $|1\rangle$. Typically, we consider a superposition of those states, which will be discussed in more detail in Section 2.6.

There are many ways of expressing the state of several qubits, all of which can be used interchangeably. For example, two qubits can exist in any of the four possible states:

$$|0\rangle|0\rangle, \ \ |0\rangle|1\rangle, \ \ |1\rangle|0\rangle, \ \ |1\rangle|1\rangle. \tag{2.1}$$

For states with multiple qubits, the following shortened form is usually easier to use:

$$|00\rangle, \ |01\rangle, \ |10\rangle, \ |11\rangle. \tag{2.2}$$

If we consider the given collection of zeros and ones as a binary representation of an integer, we arrive at an even shorter form:

$$|0\rangle\,,\ |1\rangle\,,\ |2\rangle\,,\ |3\rangle\,. \tag{2.3}$$

## 2.2 Dirac notation

The ket notation was introduced by Paul Dirac, hence it is known as the Dirac notation. It was introduced in the early days of quantum theory as a way to mathematically denote vectors, and it is commonly used to represent the physical state of a qubit or a collection of qubits. It is also possible to represent the state as a row vector, which is called a *bra*. Together, these two operators create the so-called *bra-ket*, denoted as $\langle\psi|\varphi\rangle$, which yields a single number [31].

Using Dirac's ket and bra notation, the two orthogonal states of a single qubit can be represented as column or row vectors:

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, \qquad |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}, \qquad \langle 0| = \begin{pmatrix}1 & 0\end{pmatrix}, \qquad \langle 1| = \begin{pmatrix}0 & 1\end{pmatrix}. \tag{2.4}$$

## 2.3 Tensor product

An alternative notation of the multi-qubit state is the tensor product, which is designed as a form of vector multiplication. Using the tensor product, Eq. 2.2 can be rewritten as:

$$|0\rangle = |0\rangle \otimes |0\rangle\,, \quad |1\rangle = |0\rangle \otimes |1\rangle\,, \quad |2\rangle = |1\rangle \otimes |0\rangle\,, \quad |3\rangle = |1\rangle \otimes |1\rangle\,. \tag{2.5}$$

The tensor product is considered to be the most natural way to represent multi-qubit states, as it leads to a generalization of the column-vector form presented in Eq. 2.4. Let us illustrate

this with an example of the following three-qubit state:

$$|6\rangle = |110\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}. \tag{2.6}$$

If we label each component of the resulting vector with an index $i \in \{0,...,7\}$, then the column representation of a qubit in state $|i\rangle$ contains a single 1 at index $i$, and is otherwise filled with 0's, just like in Eq. 2.4. In the example shown in Eq. 2.6, we are looking at a specific case in which $i = 6$.

## 2.4 Quantum gates

Quantum computers transform the initial state of qubits into some final form via reversible actions. These actions, called *gates*, take the form of unitary matrices, which means they fulfill the condition

$$A^{\dagger}A = AA^{\dagger} = I, \tag{2.7}$$

where $A^{\dagger}$ is a hermitian transpose of a matrix $A$ and $I$ is an identity matrix.

The four basic quantum gates are:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{2.8}$$

$I$ is called the identity operator, as it does not change the quantum state. $X$ is called the NOT operator (or sometimes the flipping operator), as it interchanges the states of $|0\rangle$ and $|1\rangle$. In order to reverse the NOT operator, one needs to apply the $X$ gate once again, which

brings the state to its original form. Operators $X$, $Y$, and $Z$ are often grouped into the vector $\sigma$, consisting of matrices $\sigma_x$, $\sigma_y$, and $\sigma_z$. Together, they are known as the Pauli matrices and have many useful purposes in quantum computing.

CNOT is another widely used quantum gate. Using the notation $C_{ij}$ to represent this gate, the $i$th qubit is called the *controlled* qubit and the $j$th qubit is called the *target* qubit. The CNOT gate leaves the state of the target qubit unchanged if the control qubit is equal to $|0\rangle$, and applies the NOT operator to the target qubit if the control qubit is equal to $|1\rangle$. In both cases, the state of the control qubit remains unchanged. As CNOT is designed to be a two-qubit gate, there are two possible CNOT gates: $C_{01}$ and $C_{10}$. The $C_{01}$ gate interchanges the states $|10\rangle = |2\rangle$ and $|11\rangle = |3\rangle$, and leaves $|00\rangle = |0\rangle$ and $|10\rangle = |1\rangle$ unchanged, whereas the $C_{10}$ gate interchanges the states $|01\rangle$ and $|11\rangle$, and leaves $|00\rangle$ and $|10\rangle$ unchanged. The matrix representations of these gates are:

$$C_{01} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad C_{10} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}. \tag{2.9}$$

An extension of the CNOT gate to the three-qubit state is called the *Toffoli gate* (or the controlled-controlled-not). $T_{xyz}$ flips the $z$th (target) qubit if the $x$th and the $y$th qubit are both equal to $|1\rangle$. An example of the Toffoli gate might be $T_{021}$, in which only the states $|101\rangle = |5\rangle$ and $|111\rangle = |7\rangle$ interchange (because they are the only two states with the control qubits both equal to $|1\rangle$).

The Hadamard (or Walsh-Hadamard) operator is another crucial quantum gate. This gate brings the initial quantum state into a superposition of all possible quantum states (described in more detail in Section 2.6). The two-dimensional matrix takes the form

$$H = \frac{1}{\sqrt{2}}(X + Z) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \tag{2.10}$$

which, applied to the *basis state* $|0\rangle$, maps it to $|+\rangle = \dfrac{|0\rangle + |1\rangle}{\sqrt{2}}$, and, applied to the basis state $|1\rangle$, maps it to $|-\rangle = \dfrac{|0\rangle - |1\rangle}{\sqrt{2}}$.

As stated at the begging of this section, a quantum computer performs reversible actions. However, there is a single irreversible action that a quantum computer can perform, called a *measurement*, which is the only way to obtain useful information from qubits.

## 2.5 Measurement

As described in Section 2.4, there are three Pauli matrices – $X$, $Y$, and $Z$ – which are especially important when considering measurements. They all share the same two eigenvalues, $+1$ and $-1$, and the corresponding eigenvectors are:

$$
\begin{aligned}
X\text{-eigenvectors:} \quad & |+\rangle = \tfrac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad |-\rangle = \tfrac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \\
Y\text{-eigenvectors:} \quad & |+i\rangle = \tfrac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ i \end{pmatrix}, \quad |-i\rangle = \tfrac{-1}{\sqrt{2}} \begin{pmatrix} 1 \\ -i \end{pmatrix}, \\
Z\text{-eigenvectors:} \quad & |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.
\end{aligned} \tag{2.11}
$$

In quantum mechanics, a measurement is the testing of a physical system in order to get a numerical result. The measurement is strongly coupled with another fundamental concept of quantum physics, the *expectation value*, the formal definition of which is

$$\langle A_\psi \rangle = \langle \psi | A | \psi \rangle . \tag{2.12}$$

After replacing $A$ with the equivalent sum of its eigenvectors $\psi_i$, weighted by its eigenvalues

$\lambda_i$, Equation 2.12 can be rewritten as

$$\langle A_\psi \rangle = \langle \psi | A | \psi \rangle = \langle \psi | \left( \sum_i \lambda_i | \psi_i \rangle \langle \psi_i | \right) | \psi \rangle = \sum_i \lambda_i \langle \psi | \psi_i \rangle \langle \psi_i | \psi \rangle = \sum_i \lambda_i | \langle \psi_i \psi \rangle |^2. \tag{2.13}$$

The expectation value can thus be defined as the average of the possible outcomes of a measurement $\lambda_i$, weighted by the probabilities of those measurements $|\langle \psi_i \psi \rangle|^2$ (since, as described in Section 2.2, the bra-ket operation always yields a number).

Equation 2.13 leads to the most important concept of quantum mechanics: the only viable real outcomes of an operator are its eigenvalues. Therefore, when measured, a state $|\psi\rangle$ collapses into one of the eigenstates of the measuring operator [31]. For example, based on Equation 2.11, the expectation value formula for the Pauli $Z$ operator takes the form:

$$\langle Z_\psi \rangle = 1 | \langle 0 | \psi \rangle |^2 + (-1) | \langle 1 | \psi \rangle |^2. \tag{2.14}$$

The difference between a measurement and an expectation value is that the measurement is a single numerical outcome of a quantum circuit, while the expectation value is an average of the possible outcomes. As shown in Eq. 2.14, the expectation value of $Z$ is related to the measurement of its eigenvalues, 1 and $-1$, while the measured state $|\psi\rangle$ collapses into either $|0\rangle$ or $|1\rangle$, which is the so-called *measurement in the computational basis*. We can just as well measure the eigenvalues of the $X$ operator in the similarly popular *$X$-basis* or the eigenvalues of the $Y$ operator in the not so widely used *$Y$-basis*.

An important fact about the expectation value is that it is not synonymous with the most probable outcome, as the name might wrongly suggest. Let us demonstrate this with a simple example. Assuming that an operator $A$ has two eigenvalues, $+1$ and $-1$, and it assigns to them an equal probability, then the expectation value of $A$ is equal to 0,

$$\langle A \rangle = -1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = 0, \tag{2.15}$$

which cannot be the most probable outcome, as it is not even one of the possible outcomes.

## 2.6 Superposition

Superposition and entanglement (the latter will be described in the following section) are the source of a quantum computer's real power [72]. While a classical bit can exist in either one of two states at any given moment, a qubit can exist in the state of both $|0\rangle$ and $|1\rangle$ simultaneously. A quantum register made of $N$ qubits can therefore exist in $2^N$ states at once. Due to this, a quantum computer processes multiple entries at the same time instead of looking at a single specific entry. This results in a tremendous speedup. While a classical computer needs $n$ operations to retrieve an element from an array, a quantum computer needs only $\sqrt{n}$ operations using Grover's algorithm [28]. As mentioned in the previous section, the state measurement operation, performed as the last operation in a quantum circuit, means that the superposition of states is collapsed into a single state – in other words, a qubit becomes a classical bit.

The mathematical definition of superposition is as follows. As presented in Eq. 2.4, any two-qubit state $|\psi\rangle$ can be represented as a two-dimensional vector,

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \end{pmatrix}, \tag{2.16}$$

where $\alpha_0$ and $\alpha_1$ are complex numbers denoting the *amplitudes*. A requirement put on these numbers is the normalization condition:

$$\|\alpha_0\|^2 + \|\alpha_1\|^2 = 1. \tag{2.17}$$

For example, a qubit in the state of $|1\rangle$ is equivalent to a superposition of $|0\rangle$ and $|1\rangle$ with amplitudes of 0 and 1 respectively. The state $|\psi\rangle$ from Eq. 2.16 is considered to be in a superposition of $|0\rangle$ and $|1\rangle$, with corresponding amplitudes of $\alpha_0$ and $\alpha_1$.

This representation of a qubit might not always be enough, because qubits represented in this way may not be distinguishable. For example, when states $|+\rangle$ and $|-\rangle$ are measured, the probability of 0 and 1 will be equal, and it will be impossible to distinguish between them. Thus, in order to have a more detailed description of a qubit's state, an alternative form is

often used,

$$|\psi\rangle = \alpha_0 |0\rangle + e^{i\varphi}\alpha_1 |1\rangle, \tag{2.18}$$

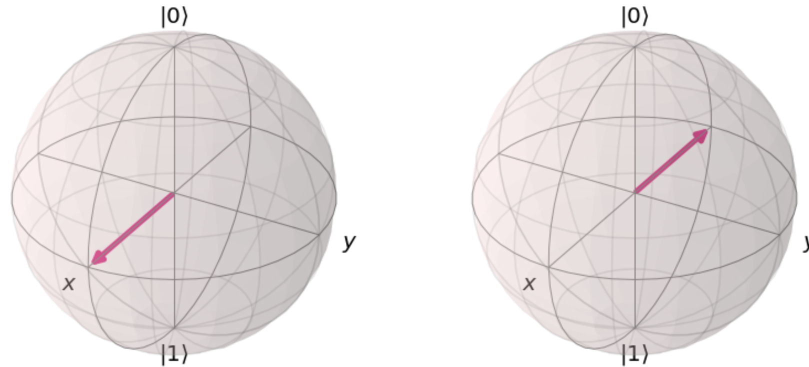
where $\alpha_0$ and $\alpha_0$ are real numbers and $\varphi$ is the *phase* between the basis states. Since the qubit is normalized (as per Eq. 2.17), the trigonometric identity can be used, and the amplitudes can be expressed in terms of a single variable $\theta$:

$$\alpha_0 = \cos\frac{\theta}{2}, \quad \alpha_1 = \sin\frac{\theta}{2}. \tag{2.19}$$

Therefore, the state of any qubit can be represented in terms of two real numbers, $\varphi$ and $\theta$, as

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle. \tag{2.20}$$

Using the representation from Eq. 2.20, we can visualize the state of $|\psi\rangle$ on the Bloch sphere, as shown in Fig. 2.1. $\theta$ is the radius between the positive $z$-axis and the vector, and $\varphi$ is the radius between the positive $x$-axis and the vector. Even though measuring the previously mentioned $|+\rangle$ and $|-\rangle$ states will return the same probabilities, using this representation they are distinguishable.




**(a)** $|+\rangle$ state with $\theta = \frac{\pi}{2}$ and $\varphi = 0$ **(b)** $|-\rangle$ state with $\theta = \frac{-\pi}{2}$ and $\varphi = 0$

**Figure 2.1:** The visual representation of two states, generated with Qiskit

Having introduced the concept of the Bloch sphere, we shall now mention the rotation gates. These gates allow us to rotate the qubit's state along one of the three axes: $x$, $y$, or $z$.

Their matrix representations are as follows:

$$R_x(\theta) = e^{-i\frac{\theta}{2}X} = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}, \tag{2.21}$$

$$R_y(\theta) = e^{-i\frac{\theta}{2}Y} = \begin{pmatrix} \cos\frac{\theta}{2} & \sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}, \tag{2.22}$$

$$R_z(\theta) = e^{-i\frac{\theta}{2}Z} = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}. \tag{2.23}$$

## 2.7 Entanglement

Entanglement is considered to be the most mysterious aspect of quantum mechanics. It means that some state vectors cannot be rewritten as a tensor product of other states (see Section 2.3). Such individual states are intimately related to each other, or *entangled*. The physical interpretation is as follows: if two entangled particles were light years away, the outcome of a measurement of the first particle would always determine the other measurement's outcome. The most popular example of entanglement is the following two-qubit state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \tag{2.24}$$

which can be achieved by applying the Hadamard gate to one qubit and then using this qubit as the control qubit of a CNOT gate. As mentioned before, it is impossible to rewrite the above equation as a tensor product of two states, as they are entangled.

## 2.8 Quantum computers

Currently, quantum computers are built in two ways: based on quantum gates or based on quantum annealing. The first group, often referred to as universal quantum computers, is more popular and more extensively developed, as it can be used to solve a wide range of

problems. Algorithms that only can be run on this kind of machines have been developed by researchers for years, such as Shor's algorithm for factorizing large numbers [62] or Grover's algorithm for quickly searching through massive data sets [28]. The most popular gate-based quantum computers are developed by IBM, which presently offers processors with as many as 65 qubits[1].

The second group of machines is quantum annealers. They are used exclusively for solving optimization problems and, instead of using quantum circuits, they use the classical Ising model (which will be described in Section 3.1.2). Therefore, every problem to be solved on a quantum annealer must be converted into the form of the Ising model. The D-Wave system solves such problems by slowly evolving to the lowest energy of the input model. Although the most popular company creating quantum annealers, D-Wave, has been on the market for only a decade, it has been growing rapidly. Their systems currently have over 5000 qubits [45].

Quantum computers can outperform classical computers in terms of speed, but they have their drawbacks. One of those drawbacks is decoherence, which is the "loss of purity of the state of a quantum system as the result of entanglement with the environment" [72]. The consequences of this effect can be mitigated via a technique called *Quantum Error Correction* (QEC). Although real quantum devices do not work perfectly, there is a way to check whether an implemented quantum algorithm really works. One can use a quantum simulator, in which the modelled qubits do not decohere or produce any error and persist in their ideal state [9].

Real quantum computers and simulators with an open-source API are provided by IBM Quantum Experience[2], Riggeti[3], Google[4], and D-Wave[5], among others.

## Summary

In this chapter, we introduced some important notation used in quantum computing. We also delved into some more complicated concepts, such as superposition or entanglement. We

[1]`https://quantum-computing.ibm.com/services?services=systems`
[2]`https://quantum-computing.ibm.com/`
[3]`https://www.rigetti.com/`
[4]`https://quantumai.google/`
[5]`https://www.dwavesys.com/`

concluded by discussing the real-life quantum computers of today.

# Chapter 3

# Optimization problems in quantum computing

In this chapter, we describe the representation of optimization problems in classical and quantum computers. We then present two popular quantum algorithms commonly used in the context of optimization problems. Finally, we introduce and compare different methods of encoding such problems using classical and quantum hardware.

## 3.1 Problem representation

In this section, we describe several models used to represent optimization problems from the perspective of both physics and computer science.

### 3.1.1 Hamiltonian

In quantum mechanics, a Hamiltonian is an operator describing the possible energies of a system, including both the kinetic and the potential energy. Any Hamiltonian may be written as

$$\hat{H} = \sum_{i\alpha} h^i_\alpha \sigma^i_\alpha + \sum_{ij\alpha\beta} h^{ij}_\alpha \sigma^i_\alpha \sigma^j_\beta + ..., \tag{3.1}$$

where $h$ is a real number, the Latin letters identify the subsystem on which the operator acts, and the Greek letters identify the Pauli operator ($\sigma_x$, $\sigma_y$, and $\sigma_z$). A wide range of physical systems is covered by this definition, including the quantum Ising model.

As mentioned in Section 2.5, a quantum system can exist in various states (*eigenstates*) and each state has a corresponding energy (*eigenvalue*), which we can represent with an equation knows as the characteristic equation or the *eigenequation* of $H$,

$$\hat{H}\,|\psi_\lambda\rangle = \lambda\,|\psi_\lambda\rangle\,. \tag{3.2}$$

In any system, the state with the lowest eigenvalue is called the *ground energy state*, to which each physical system tends to head toward.

### 3.1.2 Ising model

The classical Ising model can be written as a quadratic function of $n$ spins[1], $s_i = \pm 1$,

$$H(s_1,...,s_n) = \sum_{i<j} J_{ij} s_i s_j + \sum_{i}^{n} h_i s_i. \tag{3.3}$$

The quantum version of this is defined in the form of a Hamiltonian as

$$H_{\text{Ising}} = \sum_{i<j} J_{ij} Z_i Z_j + \sum_{i}^{n} h_i Z_i, \tag{3.4}$$

where $J_{ij}$ and $h_i$ are real numbers and $Z_i$ is a Pauli $Z$ matrix acting on the $i$th qubit. This can be rewritten as the tensor product of matrices,

$$Z_i = I_2^{\otimes i-1} \otimes Z \otimes I_2^{\otimes N-i}, \tag{3.5}$$

where $N$ is the total number of qubits, $I_2$ is an identity matrix of size $2\times 2$ and $I_2^{\otimes i-1}$ denotes consecutively applying $I_2$ gate for $i-1$ times. Since $I$ and $Z$ are both diagonal matrices, $H_{\text{Ising}}$

[1] The notation $\sum_i^N$ means a sum over the range $[0,N)$. Unless specified otherwise, it can be assumed that all sums in this thesis exclude the upper bound.

is also diagonal, resulting in the fact that for the $N$-qubit model, each state $|0\rangle, ..., |N-1\rangle$ is its eigenvector.

The Hamiltonian presented in Eq. 3.4 is a commuting operator, which is sought-after in classical systems more so than in quantum systems. In order to make it non-commuting, another type of interaction must be added to it – a Pauli $X$ operator [16]. The resulting model, called the *transverse-field Ising model*, is defined as

$$H_{\text{transverse}} = -A \sum_{i}^{n} X_i + B \cdot H_{\text{Ising}}, \tag{3.6}$$

where $A$ and $B$ are positive constants and $X_i = I_2^{\otimes i-1} \otimes X \otimes I_2^{\otimes N-i}$, similarly to Eq. 3.5 [43].

### 3.1.3 QUBO

While the Ising model is traditionally used in statistical mechanics, the QUBO (Quadratic Unconstrained Binary Optimization) model is used in computer science [39]. It is defined as an upper-diagonal matrix $Q$ of real weights and a vector of binary variables $x$. The objective function is defined as

$$f(x) = \sum_{i} Q_{ii} x_i + \sum_{i<j} Q_{ij} x_i x_j. \tag{3.7}$$

The problem to solve can then be expressed as

$$\min_{x \in \{0,1\}^n} x^T Q x. \tag{3.8}$$

When working on a D-Wave quantum computer (described in Section 2.8), each optimization problem must be passed in the QUBO form. Although many non-trivial optimization problems can be converted to this form, for some it is impossible, in which case the PUBO (Polynomial Unconstrained Binary Optimization) formulation must be used. PUBO is a generalization of QUBO allowing for polynomials of orders greater than quadratic in the objective function.

To solve a QUBO (or PUBO) problem on a quantum device, the domain of binary variables $\{0, 1\}$ needs to be converted into the Ising-model domain $\{1, -1\}$ (the eigenvalues of

the $Z$ operator). This can be achieved by replacing each $x_i$ in Eq. 3.7 with the operator

$$\frac{I - Z_i}{2}, \tag{3.9}$$

whose eigenvalues are 0 and 1. This way, a classical binary model can be solved on a quantum computer.

## 3.2 Optimization algorithms

In this section, we describe the hybrid quantum-classical algorithms VQE and QAOA, provide the theoretical ideas behind them, and review their basic flows. The section concludes with a comparison of the two algorithms.

### 3.2.1 VQE

The Variatonal Quantum Eigensolver (VQE) algorithm, proposed in 2013 by Alberto Peruzzo et al. [55], is a hybrid quantum-classical algorithm used to find the smallest eigenvalue of a given Hamiltonian and the corresponding eigenvector. Its main application is in solving large chemical problems, such as the problem of finding the ground state energy of molecules. VQE is an alternative to the QPE (Quantum Phase Estimation) algorithm [2], but with the advantage of smaller circuits depth, which is especially important for the current NISQ era of quantum computing.

As stated in Section 3.1.1, every Hamiltonian $H$ has eigenstates with their corresponding eigenvalues, however the eigenstate $\langle \psi_\lambda \rangle$ is typically not known a priori and must be estimated by calculating the expectation value of $H$ (see Eq. 2.13) at any given state $|\psi\rangle$:

$$\langle \hat{H} \rangle = \langle \psi | H | \psi \rangle = E(\psi). \tag{3.10}$$

The above equation can be rewritten for the eigenstate instead of any state:

$$\langle \hat{H} \rangle = \langle \psi_\lambda | H | \psi_\lambda \rangle = E(\psi_\lambda). \tag{3.11}$$

For the eigenstate $|\psi_0\rangle$ associated with the lowest eigenvalue, the energy would be equal to the ground state energy $E_0$. Since the ground state energy is the lowest possible energy, when an arbitrary state $|\psi\rangle$ is chosen, its energy $E_\psi$ must be greater or equal to $E_0$. This is known as the *variational principle* or *Rayleigh–Ritz method* [73]:

$$\langle\hat{H}\rangle = \langle\psi_\lambda|H|\psi_\lambda\rangle >= E_0. \tag{3.12}$$

The preparation of the state $|\psi\rangle$, to which the Hamiltonian will be applied, is the role of the *ansatz* operator, which will be defined later in this section.

Based on the above, Eq. 3.1 can be rewritten as

$$\langle\hat{H}\rangle = \sum_{i\alpha} h^i_\alpha \langle\sigma^i_\alpha\rangle + \sum_{ij\alpha\beta} h^{ij}_\alpha \langle\sigma^i_\alpha \sigma^j_\beta\rangle + ... \tag{3.13}$$

Thus, the evaluation of $\langle\hat{H}\rangle$ reduces it to the sum of the expectation values of all its terms for the quantum state $|\psi\rangle$, multiplied by some real constant. Each term is formally a tensor product of $n$ simple Pauli gates, e.g. the 4-qubit state $\sigma^1_x\sigma^2_y$ (which is synonymous with the notation $X_1Y_2$, defined in Eq. 3.5) is equal to $|I\rangle\otimes|\sigma_x\rangle\otimes|\sigma_y\rangle\otimes|I\rangle$, and its expectation value can be efficiently estimated by local measurements on each qubit. Taking the Hamiltonian

$$\hat{H} = 0.2\cdot ZI + 0.5\cdot IY + 0.4\cdot XZ, \tag{3.14}$$

then for a given state $|\psi\rangle$, the expectation value of the Hamiltonian can be calculated by adding the expectation values of its Pauli terms,

$$\langle\hat{H}\rangle = \langle\psi|\hat{H}|\psi\rangle = 0.2\cdot\langle\psi|ZI|\psi\rangle + 0.5\cdot\langle\psi|IY|\psi\rangle + 0.4\cdot\langle\psi|XZ|\psi\rangle. \tag{3.15}$$

Thus, for an $n$-qubit state, the expectation value of the resultant $2^n\times 2^n$ Hamiltonian would need to be evaluated, which would result in a tremendous memory complexity on a classical computer.

The VQE algorithm achieves this by creating a separate quantum circuit for each Pauli term. Since the measurement of the circuit is done in the $Z$-basis, the $X$ and $Y$ operators need to be aligned with the basis, therefore:

- each $X$ operator is mapped to the $R_y(-\frac{\pi}{2})$ gate, as the eigenvectors $|+\rangle$ and $|-\rangle$ (see Fig. 2.1) need to be rotated to $\sigma_z$'s eigenvectors $|0\rangle$ and $|1\rangle$,
- each $Y$ operator is mapped to the $R_y(\frac{\pi}{2})$ gate, as the eigenvectors $|i\rangle$ and $|-i\rangle$ need to be rotated to $\sigma_z$'s $|0\rangle$ and $|1\rangle$,
- $Z$ operators are not mapped to any gates in the circuit.

The algorithm calculates the expectation values of such circuits, sums them all, and obtains the expectation value of $\hat{H}$. The routine is then done over and over again for different states $|\psi\rangle$, generated by the already mentioned *ansatz*. One could try to simply generate all the possible values of $|\psi\rangle$, however that would be a very inefficient approach. Therefore, an ansatz is created as a parametrized circuit – manipulating the parameters of the ansatz results in different ansatz states. With a good ansatz and proper parameters, it is possible to have access to the subspace of the system that contains $|\psi_0\rangle$. With an improperly chosen ansatz, the quantum circuit won't be capable of generating the desired $|\psi_0\rangle$, and therefore of finding the optimal (ground energy) solution.

As mentioned above, VQE is a hybrid quantum-classical algorithm. A classical computer controls the selection of the parameters of the ansatz. At each step of the algorithm, the classical computer will change the parameters for the ansatz using some classical optimizer (e.g. COBYLA or L-BFGS-B, which will be described in more detail in Section 6.1.1), so that the state $|\psi\rangle$ will have a lower expectation value than its predecessor. If the new expectation value is lower, the algorithm "knows" that it is going in the right direction, otherwise the opposite is true. The goal is to have the algorithm arrive at a place where changing the parameters of the ansatz does not decrease the expectation value anymore. The state $|\psi\rangle$ used in the last step would then become the eigenstate $|\psi_0\rangle$ corresponding to the lowest eigenvalue (the lowest energy $E_0$) [35]. This cooperation between the quantum and the classical computer in achieving the goal of the algorithm – finding the lowest energy of the system – is presented in Fig. 3.1.

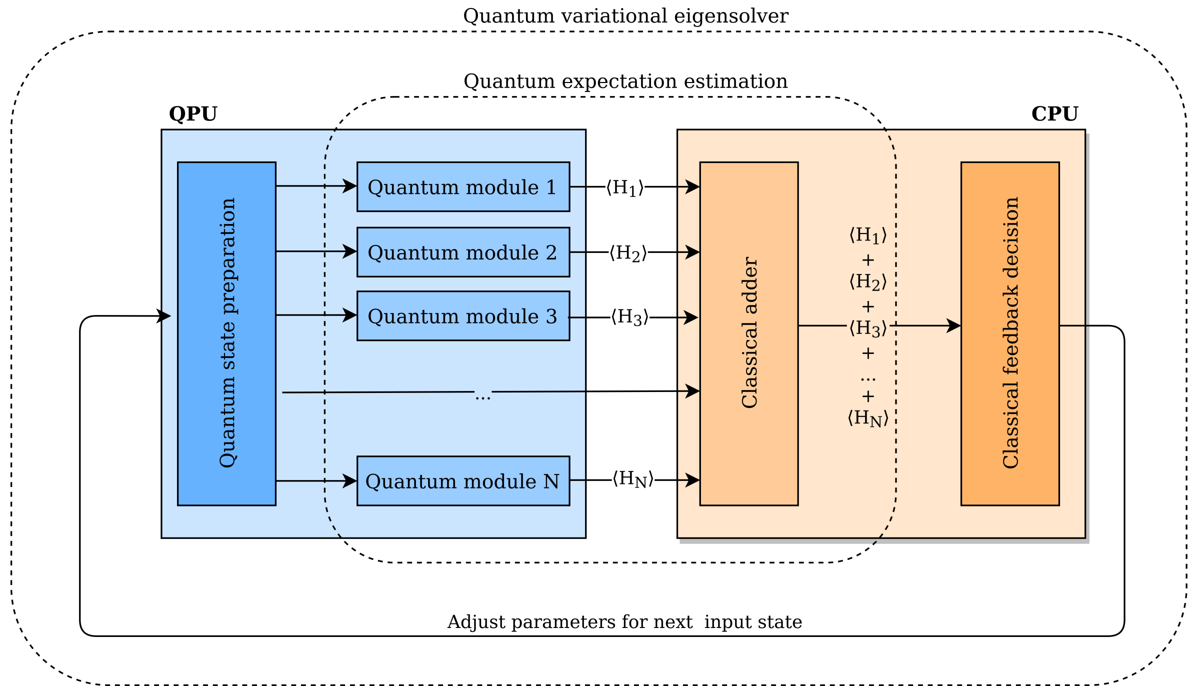



**Figure 3.1:** An overview of VQE

To sum up, the main parts of the VQE algorithm are:

1. preparing the $\langle\psi\rangle$ state with the parametrized circuit (ansatz),
2. calculating the expectation value $\langle H_i \rangle$ of each Pauli term using separate quantum circuits (*quantum modules*) which consist of $R_x$ and $R_y$ rotations, allowing for measurements in the $Z$-basis,
3. adding all the expectation values on a classical computer,
4. optimizing the parameters of the ansatz using a classical routine, based on the expectation value sum.

### 3.2.2 QAOA

The Quantum Approximate Optimization Algorithm (QAOA) was proposed by Farhi et al. in 2014 [19]. Its main application is to solve combinatorial optimization problems.

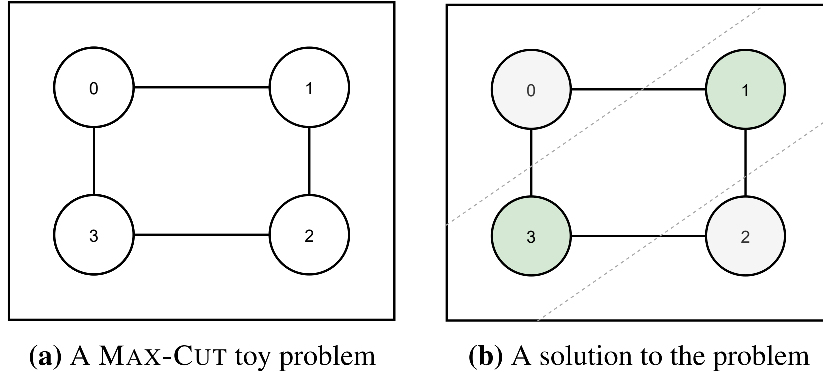
When describing QAOA, the MAX-CUT problem shall be used as a reference, as it is the problem used in the original paper [19] and the most common application of QAOA in literature. In the MAX-CUT problem, a graph with $m$ edges and $n$ vertices is considered. The desired solution of the problem is such a partition $z$ of the graph, which divides the vertices into two sets, maximizing function

$$C(z) = \sum_{\alpha=1}^{m} C_\alpha(z). \tag{3.16}$$

In the above equation, $C(z)$ is the counter of cut edges, while $C_\alpha(Z)$ is defined as

$$C_\alpha(Z) = \begin{cases} 1 & \text{if } z \text{ places one vertex from the } \alpha^{th} \text{ edge in A and the other in B,} \\ 0 & \text{otherwise.} \end{cases} \tag{3.17}$$

An example of the MAX-CUT problem and its solution can be seen in Fig. 3.2. In this case, the solution of the problem would be the bit string $z = 0101$, indicating that vertices 0 and 3 belong to set $A$ and vertices 1 and 2 to set $B$, while the value of the maximized function (the number of edges cut) is $C = 4$.




**(a)** A MAX-CUT toy problem **(b)** A solution to the problem

**Figure 3.2:** A MAX-CUT toy problem and its solution

While the aforementioned problem is rather simple and theoretical, QAOA is designed to solve problems with *exponential growth* or even *combinatorial explosion*. Typically, for big instances of such problems, there are no efficient classical methods to solve them.

There are various concepts and ideas behind QAOA that contribute to the fact that this algorithm actually works. The first of them is the Quantum Adiabatic Algorithm (QAA) [21]. Let us consider a quantum system described by the Hamiltonian $B$ with a ground state $|s\rangle$ and another Hamiltonian $C$ whose ground state is sought after. To find the ground state of $C$, QAA would start in $|s\rangle$ and then it would need to run for some long enough long time $T$. The formal definition of it is

$$H(t) = \left(1 - \frac{t}{T}\right) B + \left(\frac{t}{T}\right) C, \tag{3.18}$$

where $t$ is the current time and $T$ is the total time. If $\frac{t}{T}$ is changed slowly enough, the system always stays in the ground state of $H\left(\frac{t}{T}\right)$ and then, at the end of the algorithm, it finds itself in the ground state of $C$.

The second concept behind QAOA is the *time evolution of quantum states* (or the so-called *Schrödinger picture*). It asserts that in quantum mechanics it is the states that evolve in time, whereas the operators stay the same. It is derived from the Schrödinger equation, that is

$$ih\frac{\partial}{\partial t}|\psi(t)\rangle = H|\psi(t)\rangle. \tag{3.19}$$

The solution to the Schrödinger equation for the time-independent Hamiltonian $H$ is

$$|\psi(t)\rangle = e^{-i\hat{H}t}|\psi(0)\rangle, \tag{3.20}$$

where the operator

$$\hat{U}(t) = e^{-i\hat{H}t} \tag{3.21}$$

is called the *time-evolution* operator, which describes the quantum dynamics behind evolving the state $\psi(0)$ into the state $\psi(t)$ at some future time $t$ [42].

The third concept is called *trotterization*. QAOA is designed to work with combinatorial problems whose Hamiltonians are complicated, e.g. they are time-dependent and cannot be solved using Eq. 3.20. As the name should suggest, QAOA is about approximation, rather than about directly calculating the ground state, and since a Hamiltonian is often a sum (see Eq. 3.1 and 3.14) of large individual terms, e.g. $H = \sum_{j=1}^{M} h_j P_j$, a useful approximation

method might be the Suzuki-Trotter expansion [65],

$$e^{-iH(t)} \approx \prod_{j=1}^{M} e^{-ih_j P_j t} + \text{error}, \tag{3.22}$$

which is used to approximate the evaluation of a time-dependent Hamiltonian with a piecewise Hamiltonian that is constant in small enough time intervals:

$$U(t,t_0) = U(t,t_{n-1})U(t_{n-1},t_{n-2})...U(t_2,t_1)U(t_1,t_0). \tag{3.23}$$

The classical equivalent of trotterization is the approximation of a curve function, which can be done via a piecewise linear function. The more segments of linear functions there are, the better the curve function is approximated.

Having described the concepts behind QAOA, let us derive the two key operators defined for the algorithm, namely the cost Hamiltonian,

$$U(C,\gamma) = e^{-i\gamma C} = \prod_{\alpha=1}^{m} e^{-i\gamma C_\alpha}, \tag{3.24}$$

and the mixing Hamiltonian,

$$U(B,\beta) = e^{-i\beta B} = \prod_{j=1}^{n} e^{-i\beta \sigma_j^x}, \tag{3.25}$$

where $\gamma$ and $\beta$ are the angles, $m$ is the number of constraints in the problem, and $n$ is the number of qubits (as in the MAX-CUT problem presented at the beginning of this section in Fig. 3.2). The initial state $|s\rangle$ is the superposition over the computational basis states,

$$|s\rangle = \frac{1}{\sqrt{2^n}} \sum_z |z\rangle. \tag{3.26}$$

Based on the above equations, a quantum state dependent on angles $\gamma$ and $\beta$ is defined as

$$|\gamma,\beta\rangle = U(B,\beta_p)U(C,\gamma_p)...U(B,\beta_1)U(C,\gamma_1)\,|s\rangle\,, \tag{3.27}$$

where $p$ is a parameter describing the number of repetitions of the $U(B,\gamma_p)U(C,\beta_p)$ sequence, and thus the number of angles to be optimized. The correlation with QAA is that both algorithms perform an operator time evolution, but in QAOA this is done through an alternation of $U(B,\gamma)$ and $U(C,\beta)$, where the sum of the angles is the total running time. For a good approximation, angles $\gamma$ and $\beta$ should be small and the algorithm should have a long running time, therefore a large $p$ is expected. An overview of QAOA is shown in Fig. 3.3.

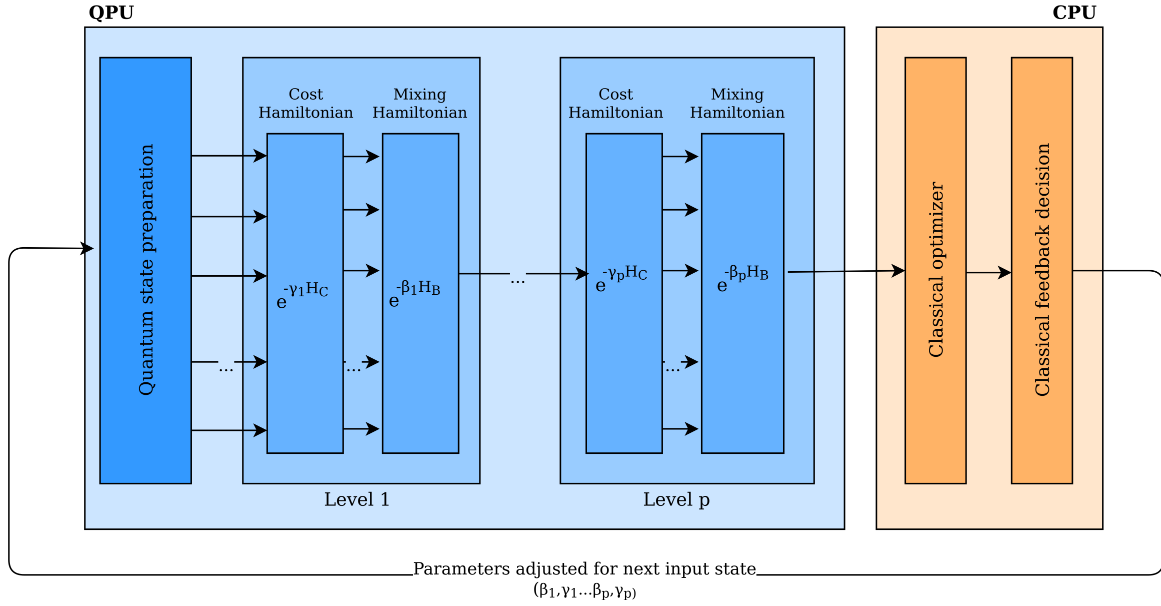



**Figure 3.3:** An overview of QAOA

To sum up, the main parts of the QAOA algorithm are:

1. preparing an initial state by applying the Hadamard gate on each qubit,
2. applying a sequence of $U(C,\gamma_i)U(B,\beta_i)$ gates $p$ times,
3. measuring the circuit on a quantum computer in the computational $Z$-basis,
4. using a classical optimization subroutine to find the new angles $(\beta_1,\gamma_1,...,\beta_p,\gamma_p)$,
5. stopping the algorithm once the optimization objective is met.

### 3.2.3 Comparison

Both VQE and QAOA are hybrid quantum-classical algorithms that find near-optimal solutions to combinatorial optimization problems. Among many similar algorithms, it is these two that are suitable for the NISQ era. While VQE (an alternative to QPE) was originally proposed to approximate the ground state of chemical systems, it can just as well be used to solve combinatorial problems, as can QAOA.

VQE and QAOA use a parameterized quantum circuit (ansatz) to generate the $|\psi(\theta)\rangle$ states used to minimize the total energy of the model, i.e., the expectation value of Hamiltonian $H$, $\langle\psi(\theta)|H|\psi(\theta)\rangle$. Since the ansatz in these two algorithms is different, the number of parameters to optimize differs as well. For VQE it depends on the particular ansatz form that is used. In this thesis, it will be assumed to depend on the number of qubits $n$ and the number of ansatz repetitions provided via the *reps* parameter[2], in the following way: $n \cdot (\text{reps} + 1)$. For QAOA, the number of parameters to optimize depends solely on the $p$ parameter, as the number of parameters to optimize is equal to $2 \cdot p$. This means that for a 10-qubit problem and *reps*/*p* set to 2, VQE will be optimizing 30 numbers, while QAOA only 4. Based on the above, QAOA is often described as an example of VQE, with a particular form of ansatz.

The difference in the type of problem that these algorithms can solve is that QAOA solves only Ising problems (with Hamiltonians as diagonal matrices), while VQE can solve any kind of Hamiltonian. Since the Ising model contains only $\sigma_z$ terms, no change in the measurement basis is required in QAOA [3].

## 3.3 Encoding schemes

Gate-model quantum machines are based on quantum bits (qubits), while many optimization problems involve discrete variables rather than binary. Such variables include integers, but can include other problem representations as well, e.g. continuous variables or multiple mutually exclusive options. An example of a discrete variable in the workflow scheduling problem might be the index of a machine on which a certain task should be performed, which will be discussed in more detail in Chapter 4.

[2] `https://qiskit.org/documentation/stubs/qiskit.circuit.library.RealAmplitudes.html`

In this chapter, we'll consider a simple toy example of categorical data, which can be seen in Table 3.1. Each row contains a description of a person, along with their age and favorite color. Let's assume the *Favorite color* column can have five possible values (*red*, *blue*, *green*, *yellow*, and *pink*) and it's those values that we will be trying to encode.

| Name | Age | Favorite color |
|---|---|---|
| Eve | 24 | red |
| Bob | 36 | yellow |
| Alice | 14 | blue |
| Jane | 41 | green |

**Table 3.1:** An example of categorical data

Three different methods of encoding these discrete variables into qubits will be presented in this section. For each of those methods, we will have to establish two functions:

- a binary function that takes an encoded bit string and validates if this string represents a specific category, e.g. "Does this bit string represent *blue*?", by returning a 1 if the answer is positive and a 0 if it's not,
- a function that ensures a bit string contains a valid encoding – it should reach its minimum when faced with a valid encoding and return higher values otherwise.

Both of those functions are necessary for implementing the objective function and the constraints in most optimization problems. We will discuss the specific objective and constraints needed for workflow scheduling in Chapter 4.

### 3.3.1 One-hot encoding

Looking at our example in Table 3.1, the first obvious issue is that the *Favorite color* column contains strings, which cannot be easily encoded on a quantum computer. The easiest solution would be to simply number the possible categories, however, as discussed at the beginning of this section, we have to find a way of actually translating those values into qubit states, as there is no out-of-the-box way of storing integers on a quantum computer.

One-hot encoding is commonly used in different areas of research, including statistics, machine learning, or even digital circuits. In quantum computing it is used primarily due to the fact that all values have to be manually mapped into binary strings, but in other areas of computing the reason is more nuanced. In machine learning, it is often not preferable to suggest to the algorithm a relationship between two categories – for instance, if we label *red* as 0 and *green* as 2, an algorithm such as a decision tree might unnecessarily assume that $red < green$.

One-hot encoding relies on the concept of *dummy variables*. A dummy variable can take the value of 0 or 1, indicating the absence or presence of some categorical quality. The example from Table 3.1 has been mapped to one-hot encoding in Table 3.2. Each row can contain only a single 1 in the five one-hot columns: *Prefers red*, *Prefers green*, *Prefers blue*, *Prefers yellow*, and *Prefers pink*. For consistency, let us collapse those columns back into one, as shown in Table 3.3. The placement of this single 1 indicates the category, i.e. the bit string 00001 can be translated to mean *pink*. Any string that does not follow this pattern, such as 00000 or 01011, does not correspond to any valid state.

| Name | Age | Prefers red | Prefers green | Prefers blue | Prefers yellow | Prefers pink |
|---|---|---|---|---|---|---|
| Eve | 24 | 1 | 0 | 0 | 0 | 0 |
| Bob | 36 | 0 | 0 | 0 | 1 | 0 |
| Alice | 14 | 0 | 0 | 1 | 0 | 0 |
| Jane | 41 | 0 | 1 | 0 | 0 | 0 |

**Table 3.2:** An example of one-hot-encoded categorical data

| Name | Age | Favorite color |
|---|---|---|
| Eve | 24 | 10000 |
| Bob | 36 | 00010 |
| Alice | 14 | 00100 |
| Jane | 41 | 01000 |

**Table 3.3:** An example of one-hot-encoded categorical data, simplified

### Bit string interpretation

As mentioned at the beginning of this section, for every encoding we need to have a way of telling if a given state represents a specific category. The interpretation of one-hot-encoded strings is very straightforward. In order to evaluate if string $s$ represents the $i$th state, we simply have to look at the $i$th bit in $s$, that is

$$f_{\text{one-hot}_i}(s) = s_i. \tag{3.28}$$

For instance, looking at the example from Table 3.3, for the category *pink* we would simply look at the fourth index in vector 00001, meaning the result of this function would be positive. If we were to validate this vector against *green*, the result would be negative.

### Bit string feasibility

The second function needed for each encoding is used for penalizing invalid vectors. The function should reach its minimum, typically equal to 0, when applied to valid states, and otherwise it should return higher values. For one-hot encoding, this function is defined as

$$g_{\text{one-hot}}(s) = \left(1 - \sum_{i}^{n} s_i\right)^2, \tag{3.29}$$

where $n$ is the number of states.

## 3.3.2 Binary encoding

The idea behind binary encoding is very simple: we map each categorical value into an integer and use those values to represent our data, like shown in Table 3.4. A mapping between those values and the original categories has to be maintained separately.

| Name | Age | Favorite color |
|---|---|---|
| Eve | 24 | $0_{10} = 000_2$ |
| Bob | 36 | $3_{10} = 011_2$ |
| Alice | 14 | $2_{10} = 010_2$ |
| Jane | 41 | $1_{10} = 001_2$ |

**Table 3.4:** A simple example of binary-encoded categorical data

### Bit string interpretation

Although binary encoding seems straightforward at first, its interpretation is much harder than that of one-hot encoding. The function answering the question "Does this vector match the $i$th color?" assumes a different form for each value of $i$. Its general form is defined as

$$f_{\text{binary}_i}(s) = \prod_{j}^{N} \left(1 - \left(s_j - b_j^{i,N}\right)^2\right), \tag{3.30}$$

where $N = \lceil \log_2 n \rceil$ is the number of bits in $s$ and $b^{i,N}$ is the $N$-bit binary-encoded equivalent of $i$ [26].

Considering our *yellow* example again, this function would assume the form

$$f_{\text{binary}_{3_{10}=011_2}}(s) = \left(1 - (s_0 - 0)^2\right)\left(1 - (s_1 - 1)^2\right)\left(1 - (s_2 - 1)^2\right), \tag{3.31}$$

which can be simplified to

$$\begin{aligned} f_{\text{binary}_{3_{10}=011_2}}(s) &= \left(1 - s_0^2\right)\left(1 - \left(s_1^2 + 2 - 2s_1\right)\right)\left(1 - \left(s_2^2 + 2 - 2s_2\right)\right) = \\ &= \left(1 - s_0^2\right)\left(2s_1 - s_1^2\right)\left(2s_2 - {s_2}^2\right). \end{aligned} \tag{3.32}$$

Since $s_0$, $s_1$, and $s_2$ can only assume the values of 0 and 1, we can safely apply the identity $s_i = s_i^2$ and remove the squares, resulting in

$$f_{\text{binary}_{3_{10}=011_2}}(s) = (1 - s_0)s_1 s_2. \tag{3.33}$$

It is easy to notice that each part of this product assumes the value of 0 if the specific bit in

vector $s$ doesn't match the expected value, and otherwise it assumes the value of 1. Therefore, the result of this function is 1 if every bit matches the specific state and 0 if there is at least one bit that differs.

The functions for the remaining colors from Table 3.4 are

$$
\begin{aligned}
f_{\text{binary}_{0_{10}=000_2}}(s) &= (1-s_0)(1-s_1)(1-s_2), \\
f_{\text{binary}_{1_{10}=001_2}}(s) &= (1-s_0)(1-s_1)s_2, \\
f_{\text{binary}_{2_{10}=010_2}}(s) &= (1-s_0)s_1(1-s_2).
\end{aligned} \tag{3.34}
$$

**Bit string feasibility**

A big advantage of binary encoding is that it is much *denser* than one-hot encoding. If the number of feasible states $n$ is equal to $2^N$, there are no states that are infeasible on the basis of encoding. Otherwise, if the number of states is not a power of two, we'd need to include a clause penalizing such invalid states.

In order to write this function, we need to compare the bit string in question to each of the infeasible states. In our example from Table 3.2, we established the five possible values for the *Favorite color* column. The number of bits needed for this encoding is $N = \lceil \log_2 n \rceil$, which in our specific example is equal to 3. Subsequently, we numbered the feasible states from 0 to 4, meaning that the allowed configurations are: $000_2$, $001_2$, $010_2$, $011_2$, $100_2$. This leaves out the other three combinations: 101, 110, and 111.

In order to penalize one of those states, we have to take all of the infeasible configurations specific to our problem, apply the formula established in Eq. 3.30, and sum the results, which formally is defined as

$$
g_{\text{binary}}(s) = \sum_{i \in P} f_{\text{binary}_i}(s), \tag{3.35}
$$

where $P$ is a set containing the infeasible states. This function will return a 1 for an infeasible state, and for any feasible state it will return a 0.

In our toy example, after applying the same simplifications as in Eq. 3.33, this function

would assume the form

$$\begin{aligned} g_{\text{binary}}(s) = f_{\text{binary}_5}(s) + f_{\text{binary}_6}(s) + f_{\text{binary}_7}(s) = \\ = s_0(1-s_1)s_2 + s_0 s_1(1-s_2) + s_0 s_1 s_2. \end{aligned} \tag{3.36}$$

### 3.3.3 Domain wall encoding

Unlike the previous two encodings, which are derived from classical computer science, domain wall encoding originates from physics. It was proposed by Chancellor as an alternative method of encoding discrete variables into qubits, and it is based on the physics of domain walls in one-dimensional Ising spin chains [12]. This new encoding of discrete variables will be possibly more useful in near-term applications, as it requires fewer qubits than the traditional one-hot method.

In a one-dimensional Ising model (see Eq. 3.4), a *domain wall* exists between two qubits $i$ and $i+1$, if their bit values are not equal, that is if

$$\langle Z_i Z_{i+1} \rangle = -1, \tag{3.37}$$

where $\langle\ \rangle$ denotes the *expectation value* described in Eq. 2.13 and $Z_i$ follows the definition from Eq. 3.5. Since the $Z$ gate affects the basis states $|0\rangle$ and $|1\rangle$ in the following way:

$$Z|0\rangle = 1|0\rangle, \quad Z|1\rangle = -1|1\rangle, \tag{3.38}$$

it can be seen that for adjacent qubits in the same basis states, the operation in Eq. 3.37 will return a 1, which indicates the lack of a domain wall,

$$\langle 00|Z_0\, Z_1|00\rangle = \langle 11|Z_0\, Z_1|11\rangle = 1, \tag{3.39}$$

while for adjacent qubits in different basis states it will return a $-1$, which denotes the existence of a domain wall,

$$\langle 01|Z_0\, Z_1|01\rangle = \langle 10|Z_0\, Z_1|10\rangle = -1. \tag{3.40}$$

For example, in the one-dimensional Ising model shown in Figure 3.4 (top), there are infinitely strong penalties holding the qubit at index $-1$ in the basis state of 1 and the qubit at index 4 in the basis state of 0. As the values of the first and the last qubit are fixed, they can be ignored, leaving only the middle segment of qubits indexed with $0...n-2$, as presented in Figure 3.4 (bottom), where $n$ is the number of values to encode (e.g. in the *favorite color* example mentioned in 3.3.1, $n$ is equal to five). Then, based on Eq. 3.37, the Hamiltonian of the model takes the simplified form

$$H_n = -\lambda(-Z_0 + \sum_{i}^{n-3} Z_i Z_{i+1} + Z_{n-2}), \tag{3.41}$$

where $\lambda$ is a sufficiently large positive constant enforcing that the system should be found in a logically correct subspace. The terms $Z_{-1}$ and $Z_{n-1}$ are not present in the formula, since, as already mentioned, their bit values are fixed and can be replaced with *-1* and *1*, respectively.

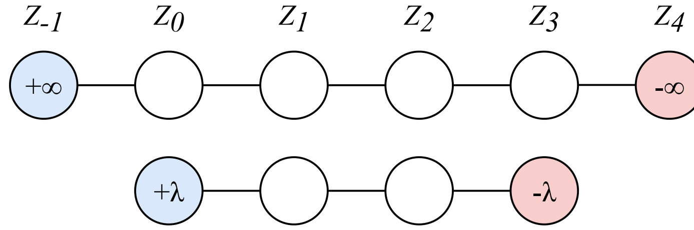



**Figure 3.4:** Top: The one-dimensional ferromagnetic Ising chain encoding. Bottom: An equivalent model without the fixed qubits, which encodes $\mathbb{Z}_5$

According to Eq. 3.37 and Eq. 3.41, the energy of a one-dimensional ferromagnetic chain of qubits is proportional to the number of domain walls. Therefore, the model has the lowest possible energy when there is only one domain wall between any of the $n$ pairs of consecutive qubits, and it has a higher energy when there are more walls. Since there are $N = n-1$ qubits, we can encode a discrete variable $x \in \mathbb{Z}_n$, where $\mathbb{Z}_n = \{0, 1, ..., n-1\}$.

In the Ising model from Fig. 3.4, there are five discrete variables that can be encoded, just as in the favorite color example shown in Table 3.1. A domain-wall-encoded mapping of the color categories from that example is presented in Table 3.5. In order to find the encoded value, it is necessary to find the position of the domain wall – that is, the place where the value

of a bit changes with respect to its predecessor. Any state that has more than one domain wall is an invalid state, and it does not correspond to any discrete variable, e.g. the state 1001 has three domain walls at positions with indices $i \in \{1,2,3\}$.

| Name | Age | Favorite color |
|---|---|---|
| Eve | 24 | 0000 |
| Bob | 36 | 1110 |
| Alice | 14 | 1100 |
| Jane | 41 | 1000 |

**Table 3.5:** An example of domain-wall-encoded categorical data

In order to assign a specific discrete variable to the domain wall position $i$, $\delta_i$ is defined as

$$\delta_i = \frac{1}{2}(Z_i - Z_{i+1}), \tag{3.42}$$

which returns the energy of 1 at the domain wall location $i$ and 0 otherwise (assuming there is only one domain wall). If there are multiple domain walls (e.g. 1001), the function returns the energies $\delta_i \in \{-1,0,1\}$. The energy of the operator is, as described in Section 3.1.1, the operator's eigenvalue.

#### Bit flips

As stated in the previous section, the energy of a one-dimensional ferromagnetic chain of qubits is proportional to the expectation value of the number of domain walls. Furthermore, flipping a single qubit with index $i$ can affect the domain wall configuration in three different ways:

- if $\langle Z_{i-1} \rangle = \langle Z_{i+1} \rangle = \langle Z_i \rangle$, the flip creates two domain walls, increasing the energy of the model by $4\lambda$,
- if $\langle Z_{i-1} \rangle = \langle Z_{i+1} \rangle \neq \langle Z_i \rangle$, the flip destroys two domain walls, decreasing the model's energy by $4\lambda$,
- if $\langle Z_{i-1} \rangle \neq \langle Z_{i+1} \rangle$, the flip moves the existing domain wall and the energy of the model does not change.

As a result of the above, if $\langle Z_{-1}\rangle \neq \langle Z_{n-1}\rangle$, there must be an odd number of domain walls between the qubits at indices $-1$ and $n-1$. In the four-qubit model from Fig. 3.4 it would be possible to have one, three, or five domain walls.

### Bit string interpretation

Even though domain wall encoding is based on qubits and quantum physics, it is possible to translate the formulas from above sections into classical bits. Assuming there are five categories (colors) to encode, as in Table 3.2, the domain wall vector would have the length of four bits. The possible encodings are presented in Table 3.5. We can then define a function for determining if a given state represents a specific category. For that, we need to calculate the difference of subsequent bits as defined in

$$f_{\text{domain-wall}_i}(s) = \begin{cases} 1 - s_0 & \text{if } i = 0, \\ s_i - s_{i+1} & \text{if } 0 < i < N-2, \\ s_{N-1} & \text{if } i = N-1, \end{cases} \tag{3.43}$$

where $N = n-1$ is the number of bits, as previously defined.

Looking at the example from Table 3.5, for the category *red*, we would look at the difference $1 - s_0$, whereas for *blue* we would look at the difference between $s_1$ and $s_2$. When this formula is used on a vector with only one domain wall, it returns either a 1, indicating that the represented color does indeed match the color in question, or a 0 if it doesn't. However, for a vector with more than one domain wall, it might also return the value of $-1$, which needs to be handled in the problem being modeled.

### Bit string feasibility

The second function that's important to define is used to minimize the energy of logically valid solutions and maximize the energy of incorrect ones. For domain wall encoding, this function assumes the form

$$g_{\text{domain-wall}}(s) = (1-s_0)^2 + \sum_{i}^{N-2} (s_i - s_{i+1})^2 + {s_{N-1}}^2. \tag{3.44}$$

For each encoding in Table 3.5, the above function would yield the value of 1 because there is only one domain wall, while for an infeasible solution, such as 1010, it would yield a value greater than 1, which in this specific case equals 3 as there are three domain walls.

### 3.3.4 QAOA mixers

As described in Section 3.2.2, a QAOA circuit uses two operators: the cos operator defined in Eq. 3.24, which depends on the optimization objective and the encoding, and the mixing operator defined in Eq. 3.25. The idea behind the latter is to preserve the feasible subspace and to provide transitions between configurations that belong to this subspace.

The default mixing operator used in QAOA is the $X$ mixer defined as

$$H_X = \sum_{i}^{N} X_i, \tag{3.45}$$

where $X_i$ is the Pauli $X$ operator acting on the $i$th qubit (similarly to $Z_i$ from Eq. 3.5). This operator acts as a simple bit flip, allowing for transitions between any state and its neighbors [30].

Using the default QAOA mixer has its drawbacks, as this mixing strategy may cause the system to appear in an incorrect state. For example, in one-hot encoding we could have the invalid state of 10100, and in domain wall encoding the state 0010. Certainly, the problem could be solved via post-processing, yet it is still problematic. Hence, instead of using transverse field mixers, it is often preferable to use a dedicated mixer that mixes only through the constraint subspace of the problem, preferably through the valid states.

#### One-hot encoding

For one-hot encoding, the feasible subspace is quite limited. The only feasible states are the ones that include exactly a single 1. Using the $X$ mixer would result in all those infeasible states being considered, which often forces one to include additional constraints penalizing the invalid states. Instead of the default mixer, one-hot encoding can be used in combination

with the dedicated $XY$ mixer, defined as

$$H_{XY} = \sum_{i}^{N-1} (X_i X_{i+1} + Y_i Y_{i+1}), \tag{3.46}$$

where $X_i$ and $Y_i$ are Pauli operators acting on the $i$th qubit, as in Eq. 3.5 [30, 69]. This operator works similarly to the SWAP gate – it turns the state $|01\rangle$ into $|10\rangle$ and vice versa, while the states $|00\rangle$ and $|11\rangle$ remain unchanged.

### Domain wall encoding

As stated in the Section 3.3.3, flipping a qubit adjacent to a single domain wall does not change the overall number of domain walls in the model. Therefore, a domain wall mixer should only cause a bit to flip if it is adjacent to a domain wall and. This property is satisfied by the following Hamiltonian:

$$H_{mix} = \sum_{i}^{N-1} (Z_{i-1}\, X_i - X_i\, Z_{i+1}). \tag{3.47}$$

This mixer flips the $i$th qubit only when it is in direct neighborhood of a domain wall, i.e. if the values of the qubits at indices $i-1$ and $i+1$ are different, otherwise it does not introduce any change to the qubit. Take as an example the state 1100, representing the value of 2 (or the *blue* color category from Table 3.5). If this mixer was applied to the qubits with indices $\{0,3\}$, then $Z_{i-1} = Z_{i+1}$ and the terms in the above equation would reduce. On the other hand, if the mixer was applied to the qubits with indices $\{1,2\}$, then $Z_{i-1} \neq Z_{i+1}$, so the terms in the above equation would sum up to $-X_i - X_i = -2X_i$, which flips the $i$th qubit. The constant preceding the operator can be neglected.

This approach allows for transitions exclusively between consecutive states, instead of transitions between any two states. Yet, it is still an open question whether in real life problems a custom mixer specific to a particular encoding will perform better than the default $X$ mixer.

### 3.3.5 Comparison

A comparison of the encoding methods presented in this section can be found in Table 3.6.

| | one-hot | binary | domain wall |
|---|---|---|---|
| $N$ (no. of qubits) | $n$ | $\lceil log_2(n) \rceil$ | $n-1$ |
| no. of couplers for encoding | $n(n-1)$ | 0 if $n = 2^N$,<br>complicated otherwise | $n-2$ |
| intra-variable connectivity | complete | N/A or complicated | linear |
| maximum order<br>needed to evaluate single values | 1 | $\lceil log_2(n) \rceil$ | 1 |
| maximum order<br>needed for two-variable interactions | 2 | $2\lceil log_2(n) \rceil$ | 2 |
| custom mixer available | yes | no | yes |
| density | $\frac{n}{2^n}$ | $\frac{n}{2^{\lceil log_2(n) \rceil}}$ | $\frac{n}{2^{n-1}}$ |

**Table 3.6:** A comparison of binary, one-hot, and domain wall encoding

#### Number of qubits

The number of qubits scales linearly in both domain wall encoding and one-hot encoding, although the latter does save one qubit. In binary encoding, the number scales logarithmically.

#### Number of two-body couplers for encoding

In order to determine the number of couplers for a specific encoding, we have to look at its bit string feasibility function. For our purposes, the number of two-body couplers could be understood as the number of second-degree terms in the obtained polynomial, with the exclusion of terms that only contain a single variable, e.g. in the polynomial $x_0x_1 + x_0^2 + 2x_1$ the number of couplers would be equal to 1.

For one-hot encoding, the feasibility function was defined in Eq. 3.29. After expanding this function, we end up with the polynomial

$$
\begin{aligned}
g_{\text{one-hot}}(s) &= (1-\sum_{i}^{n} s_i)^2 = \\
&= (1-(s_0+s_1+s_2+...+s_{n-1}))^2 = \\
&= 1^2 - 2(s_0+s_1+s_2+...+s_{n-1}) + (s_0+s_1+s_2+...+s_{n-1})^2.
\end{aligned}
\tag{3.48}
$$

Following the rules described above, we can disregard the first two terms, which leaves us with

$$
\hat{g}_{\text{one-hot}}(s) = (s_0+s_1+s_2+...+s_{n-1})^2. \tag{3.49}
$$

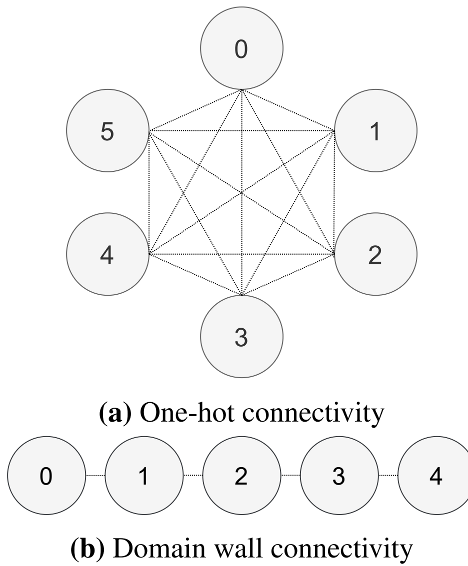
After expanding this polynomial, the number of terms is equal to $n^2$, however, we can ignore all terms of the form $s_i^2$, which leaves us with the final number of couplers being equal to $n(n-1)$.

The feasibility function for binary encoding was defined in Eq. 3.35. The number of two-body couplers depends on the value of $n$. If $n$ is a power of two, there are no infeasible states and the number of couplers is equal to 0. Otherwise, the function assumes the form of a more complicated polynomial, like the one seen in Eq. 3.36. This polynomial will typically require $N$-body interactions, which means that the implementation is much more complicated.

The feasibility function for domain wall encoding was defined in Eq. 3.44. It can be seen that in the formula there are $n-2$ two-body terms, since the basis states of qubits with indices *-1* and $N$ are fixed.

**Intra-variable connectivity**

The intra-variable connectivity property is directly related to the number of couplers described in the previous section. In one-hot encoding, each bit is coupled with every other bit, so the connectivity is complete (see Fig. 3.5a). In binary encoding, the connectivity depends on the value of $n$ again – if it's a power of two, there's no connectivity required, and otherwise the relationship is complicated. In domain wall encoding, the connectivity is linear, since each qubit is coupled only with its neighbors (see Fig. 3.5b).




**(a)** One-hot connectivity

**(b)** Domain wall connectivity

**Figure 3.5:** One-hot and domain wall qubit connectivity

#### Maximum order needed to evaluate single values

The maximum order needed to evaluate or penalize single values is directly related to the bit string interpretation function we defined for each encoding discussed in this section. It can be understood simply as the order of the $f(s)$ polynomial.

The bit string interpretation function for one-hot encoding was defined in Eq. 3.28. The function has the order of 1. For binary encoding, this function was introduced in Eq. 3.30. Its order is equal to $N = \lceil \log_2 n \rceil$. The version of this function used for domain wall encoding was defined in Eq. 3.44 – it has the order of 1.

#### Maximum order needed for two-variable interactions

The order needed for describing two-body interactions is directly related to the order needed for the evaluation of a single variable. The mechanism is the same, however this time two variables have to be considered instead of one. Therefore, the order for each encoding is doubled.

### Custom mixer availability

The $XY$ mixer suitable for one-hot encoding and the dedicated mixer for domain wall encoding were both described in 3.3.4. Binary encoding does not have a dedicated mixer operator, however due to the density of this encoding, there is often no need for a mixer other than the default $X$ mixer.

### Encoding density

Binary encoding is the densest of the three. The ration of feasible states to all states is equal to $\frac{n}{2^{\lceil log_2(n) \rceil}}$. If $n = 2^N$, none of the states are infeasible. The number of qubits used for one-hot and domain wall encoding scales linearly, so the density is much worse – $\frac{n}{2^n}$ for one-hot and $\frac{n}{2^{n-1}}$ for domain wall. Encoding density is important to consider, as it describes how much of our solution is space is occupied by infeasible states.

## Summary

In this section, we discussed how optimization problems can be represented and solved on a quantum computer. We began by defining the different representations, such as the Hamiltonian. Afterwards, we introduced the two quantum algorithms that will be used throughout this thesis, VQE and QAOA. Finally, we dove into two commonly used encoding schemes – one-hot and binary encoding – and compared them with a newly introduced alternative encoding called domain wall encoding.

# Chapter 4

# Workflow scheduling

This chapter explains the version of workflow scheduling that will be used throughout this thesis and introduces the formal definition.

## 4.1 Basic intuition

A workflow scheduling problem consists of $N$ tasks and $M$ types of machines. The tasks have to be completed in a specific order that is defined in the form of a directed acyclic graph (DAG). The execution of all tasks has to be completed within a deadline $d$.

It is important to note that while there are $M$ types of machines, the number of machines of each specific type is unlimited, meaning two tasks can be simultaneously running on two machines of the same type without any interference. For simplicity, instead of saying "machine of type 3", we will use the phrase "machine 3". This does not mean that all the tasks are executed on a single machine simultaneously, but rather that all of them are executed on different machines of this type.

Let's consider the following toy problem: we have two machines (A and B) and three tasks (1, 2, and 3). The cost of running each task on a specific machine is defined in Table 4.1. The number of time units required to run each task on a specific machine can be seen in Table 4.2. Figure 4.1 shows the DAG that represents the dependencies between the tasks.

| | Machine A | Machine B |
|---|---|---|
| Task 1 | 1 | 4 |
| Task 2 | 2 | 6 |
| Task 3 | 5 | 4 |

**Table 4.1:** Toy problem: cost matrix

| | Machine A | Machine B |
|---|---|---|
| Task 1 | 5 | 4 |
| Task 2 | 2 | 3 |
| Task 3 | 5 | 3 |

**Table 4.2:** Toy problem: time matrix

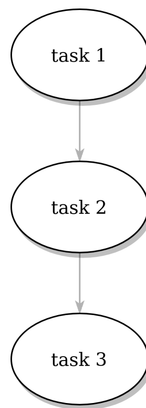



**Figure 4.1:** Toy problem: task order DAG

### 4.1.1 Objective function

Our objective is to minimize the cost of executing the series of tasks. Looking at our toy problem, if we execute the first task on machine A and the remaining two tasks on machine B, the cost will be equal to $1+6+4=11$.

The basic definition of this function is

$$\text{objective} = A \cdot \sum_{i}^{N} \text{cost}_{i\text{th task}} \tag{4.1}$$

where $A$ is a constant.

### 4.1.2 Constraints

While our goal is to minimize the objective function, there are also certain constraints that have to be met.

**Time constraint**

The time of execution does not have to minimized like the cost, but it does need to be kept below a deadline $d$. Looking at our toy problem and assuming the deadline is equal to $d = 12$, we could break this constraint by executing tasks 1 and 3 on machine A and task 2 on machine B ($5+3+5 = 13 > 12$). A solution satisfying this constraint could, for instance, take the form of having all the tasks be run on machine B ($4+3+3 = 10 \leq 12$).

The basic formulation of this constraint is

$$\sum_{i}^{N} \text{time}_{i\text{th task}} \leq d. \tag{4.2}$$

This inequality causes some complications, since we don't want to penalize solutions that require more units of time, as long as they stay within the time limit. In our example, we shouldn't favor using machine B for task 1 and machine A for tasks 2 and 3 ($4+2+5 = 11 \leq 12$) over using machine B for tasks 1 and 2 and machine A for task 3 ($4+3+5 = 12 \leq 12$). From the algorithm's perspective, these solutions should be equivalent, and we should favor the one with a lower cost.

The mechanism used to convert the inequality into an equality is based on *slack variables*. The slacks are additional non-negative variables in the solution vector that declare an offset added to the obtained value. The equality takes the following form:

$$\sum_{i}^{N} \text{time}_{i\text{th task}} + \sum \text{slacks} = d. \tag{4.3}$$

Looking at our toy problem, if we take a solution that includes running all the tasks on machine B ($4+3+3 = 10 \leq 12$), we would need the sum of our slack variables to assume the value of 2 to satisfy the equality $10+2 = 12$.

To figure out the maximum value of the slack sum (and therefore the number of bits

required to encode it), we need to look at the shortest possible path. In our toy example, it's $4+2+3=9 \leq 12$ (tasks 1 and 3 on machine B and task 2 on machine A). Therefore, to encode our solutions in the form of an equality, we will need the slacks to assume a value from the range $[0,4]$.

The final issue with the time constraint is that we do not have a mechanism to enforce equality conditions when using optimization algorithms such as the ones discussed in this thesis. We can only minimize a single function, which means that we have to somehow add this constraint to the already established objective function. The constraint converted to a function that can be minimized assumes the following form:

$$\text{constraint}_{\text{time}} = B \cdot \left( d - \left( \sum_{i}^{N} \text{time}_{i\text{th task}} + \sum \text{slacks} \right) \right)^{2}. \tag{4.4}$$

This function meets its minimum when $\sum_i^N$ time for task $i + \sum$ slacks is equal to the deadline and grows quadratically as we get away from the minimum (regardless of whether we're going over or below the deadline). The factor $B$ is included to ensure we have a proper weight distribution between the three summands.

While our toy problem includes only a single path, it is important to note that when solving problems with more complex DAGs (like the one in Figure 4.1), each path that goes through the graph needs a separate inequality condition (since we want each path to fit in the deadline) and therefore each path has its separate slacks. For a problem with a set of paths $R$, we arrive at the following formula:

$$\text{constraint}_{\text{time}} = B \cdot \sum_{r \in R} \left( d - \left( \sum_{i \in r} \text{time}_{i\text{th task}} + \sum_{j \in r} \text{slack}_{j} \right) \right)^{2}. \tag{4.5}$$

**Feasibility constraint**

As we noted, the number of machines of each type is unlimited, so it is possible to execute all tasks on machines of the same type. Therefore, we do not need to concern ourselves with spreading tasks across machines in any specific way. However, we have to ensure that each

task will be assigned to exactly one valid machine. This constraint is implemented via the bit string feasibility function described in Chapter 3.3. In some cases, the constraint can be skipped, for instance when using binary encoding to represent $2^N$ states. Additional scenarios for when the constraint can be skipped will be discussed in Chapter 5. Nevertheless, the basic idea is that we want to minimize the function

$$\text{constraint}_{\text{tasks}} = C \cdot \text{number of tasks without a machine.} \tag{4.6}$$

This penalty typically grows linearly with the number of incorrectly assigned tasks.

### 4.1.3 Optimized function

The full function takes the form:

$$\begin{gathered} A \cdot \sum \text{cost}_{i\text{th task}} + B \cdot \sum_{r \in R} \left( d - \left( \sum_{i \in r} \text{time}_{i\text{th task}} + \sum \text{slacks} \right) \right)^2 \\ + C \cdot \text{no. of tasks with no machine.} \end{gathered} \tag{4.7}$$

For vectors that meet the deadline and correctly assign machines to all tasks, the minimized function is equivalent to the objective (cost) function. For configurations that break one or two of the required conditions, penalties are assigned according to the two functions described above. The specifics of this function are encoding-dependent and will be discussed in the following chapters. We will also discuss the intuition behind choosing good values for $A$, $B$, and $C$.

## 4.2 Formal definition

In order to map the problem defined in the previous section to a QUBO problem, the following constants have to be defined:

- $N$ – the number of tasks,

- $M$ – the number of machines,
- $R$ – the number of paths,
- $d$ – the maximum number of time units allocated to the completion of all the tasks.

We also have to consider the following matrices:

- $C(i,j)$ – the cost of executing task $i$ on machine $j$,
- $T(i,j)$ – the number of time units required to execute task $i$ on machine $j$,
- $S(k)$ – the number of slack variables for path $k$,
- $P(i,k)$ – a binary function of the following form:

$$P(i,k) = \begin{cases} 1 & \text{if task } i \text{ is lies on path } k, \\ 0 & \text{otherwise.} \end{cases} \tag{4.8}$$

While the definitions described above are written in such a way that they are not dependent on the chosen encoding, it is important to point out that $V$, the solution vector returned by a quantum computer, does depend on the encoding. This vector contains a one-dimensional representation of the solution, including both the task-machine pairings and the slack variables. The interpretation of this vector will depend on the encoding, as it will be based on the bit string interpretation function defined for each encoding in Chapter 3.3. We will discuss the specific implementations in Chapter 5.

In order to establish a generic definition of our objective function, let us define two additional matrices. Their role is to map the state described by $V$ into an encoding-independent representation of this state. The vector will be converted into two matrices:

- the solution matrix $X$, which has the form:

$$X(i,j) = \begin{cases} 1 & \text{if task } i \text{ is executed on machine } j, \\ 0 & \text{otherwise,} \end{cases} \tag{4.9}$$

- the slack matrix $Y$, where $Y(k,l)$ denotes the value of the $l$th slack on path $k$.

For simplicity, the elements of the above-mentioned matrices $C(i,j)$, $T(i,j)$, $S(k)$, $P(i,j)$, $X(i,j)$, and $Y(k,l)$ will be denoted as $c_{i,j}$, $t_{i,j}$, $s_k$, $p_{i,j}$, $x_{i,j}$, and $y_{k,l}$, respectively.

Using the notation established in the paragraphs above, we arrive at the following generic formulation of the objective function:

$$\begin{aligned} O(X,Y) = A \cdot \sum_{i}^{N} \sum_{j}^{M} c_{i,j} x_{i,j} + B \cdot \sum_{k}^{R} \left( d - \left( \sum_{i}^{N} \sum_{j}^{M} p_{i,k} t_{i,j} x_{i,j} + \sum_{l}^{s_k} 2^l y_{k,l} \right) \right)^2 \\ + C \cdot \sum_{i}^{N} \left( 1 - \sum_{j}^{M} x_{i,j} \right)^2 . \end{aligned} \tag{4.10}$$

This definition includes the typical implementation of slack variables that is based on a straightforward binary representation of the slack value.

# Summary

This chapter derived the constraints and the objective function that will be used to solve the workflow scheduling problem. Using a simple toy example, we demonstrated the importance of the three components of the objective function. Finally, we arrived at a generic definition of the problem, which can be adapted to different encodings in a simple way. The specific encoding-dependent implementations will be discussed in Chapter 5.

# Chapter 5

# Solution implementation

In Chapter 4, we introduced a generic definition of workflow scheduling. This definition, summarized by Eq. 4.10, relies on two matrices: the task-machine matrix $X(i,j)$ and the slack matrix $Y(k,l)$. These matrices are formed from vector $V$, which is a one-dimensional encoding-dependent representation of the system's state. In this chapter, we discuss the specific implementations of $X$ and $Y$ for each of the discussed encoding schemes. After that, we return to the subject of QAOA mixers, previously discussed in Chapter 3.3.4, and we define the mixers to be used in this problem.

## 5.1 Encoding-dependent solution representations

In Chapter 3.3, we discussed three different encoding schemes: one-hot, binary, and domain wall. For each encoding, we defined two functions: the bit string interpretation function and the bit string feasibility function. The first function will be used to implement matrix $X$. The second function, in some cases, will be used to simplify the feasibility constraint.

In order to implement matrix $Y$, the same formula will be used for every encoding:

$$Y(k,l) = V(N\hat{M} + \sum_{r}^{k} s_r + l), \tag{5.1}$$

where $\hat{M}$ is the number of bits needed to encode $M$ machines and $s_r$ is the number of slack

variables for path $r$.

### 5.1.1 One-hot encoding

The one-hot encoding scheme is quite straightforward. To encode each machine we need $M$ bits and the meaning of those bits can be resolved by applying the function defined in Eq. 3.28. To encode $M$ machines and $N$ tasks, we need $N \cdot M$ bits. We store those values in a single flattened vector, where the first $M$ bits encode the first task, the second $M$ bits encode the second task, and so on. The remaining bits in vector $V$ correspond to the slack values. An example of $V$ for three tasks and three machines can be seen in Figure 5.1. In this case, the first task will be executed on the second machine (machine with index 1), the second task will be executed on the first machine (index 0), and the third task will be executed on the second machine as well. The slacks will be resolved to $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11$.

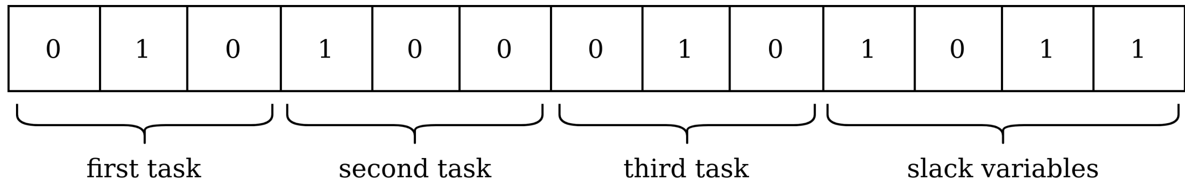



**Figure 5.1:** An example of a one-hot encoded workflow scheduling vector $V$

In order to translate the solution vector into the $X$ matrix, we apply the formula

$$X(i, j) = V(i \cdot M + j). \tag{5.2}$$

The universal feasibility function for one-hot encoding was defined in Eq. 3.29. This function closely resembles the feasibility constraint for workflow scheduling, as it was defined in Eq. 4.7. Therefore, there is no room for any additional optimization of this constraint.

For a three-bit task vector, like the one in Figure 5.1, we have three feasible states: 001, 010, and 100. The majority of the remaining configurations correspond to situations in which a single task would have to be executed on multiple machines (for instance, 111 would mean that the task is executed on all three machines). These configurations are penalized not only by the feasibility constraint, but also by the sheer cost of having these additional elements in

the cost and time functions – executing a single task on multiple machines will always cost more and take longer than executing it on a single machine. A notable exception is vector 000 – it denotes a situation in which a task is not executed on any machine. Unlike the other infeasible states, this configuration would seem preferable to the optimizer over other legitimate configurations, and it is the reason why the feasibility constraint is required when one-hot encoding is used (unless it is used in conjunction with a QAOA mixer).

### 5.1.2 Binary encoding

Using binary encoding, to encode each of the $M$ machines, we need $\hat{M} = \lceil \log_2 M \rceil$ bits. Following the thought process from the previous section, in this case we will also flatten the $N$ machine encodings into a single vector $V$, in which the first $\hat{M}$ bits will describe the first task, the next $\hat{M}$ bits the second task, and so on. The remaining bits in vector $V$ will be used for slack variables. This encoding is illustrated in Figure 5.2, which described the same configuration as Figure 5.1.

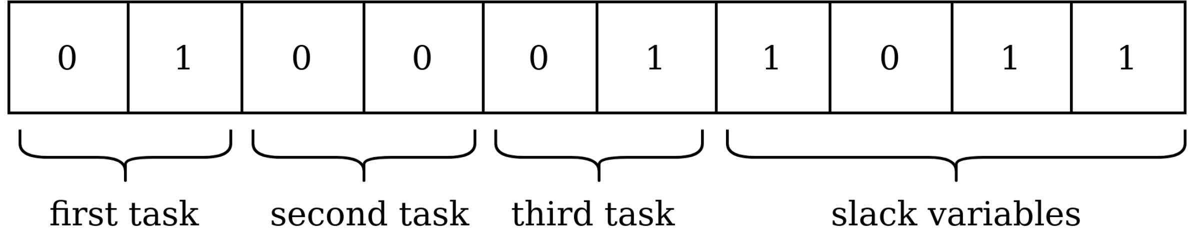



**Figure 5.2:** An example of a binary encoded workflow scheduling vector $V$

In order to decode each machine from vector $V$, we will use the function defined in Eq. 3.30. The full mapping to $X$ takes the following form:

$$X(i,j) = \prod_{m}^{\hat{M}} \left(1 - \left(V(i \cdot \hat{M} + m) - b_i^{j,\hat{M}}\right)^2\right), \tag{5.3}$$

where $b_i^{j,\hat{M}}$ is the $i$th bit of the binary-encoded number $j$, encoded over $\hat{M}$ bits.

The formulation of the feasibility constraint, as defined in Eq. 4.10, can be simplified for binary encoding. In one-hot encoding, we had to ensure that each task would be run on

exactly one machine (instead of zero machines or ten machines). In binary encoding, there is no state corresponding to executing a task on multiple machines or no machines – a task is simply run on the machine with the encoded index. When $M = 2^{\hat{M}}$, every combination of bits corresponds to some valid machine, so no feasibility constraint is needed. Otherwise, we have to ensure that the encoded machine index does not exceed $M$. To do that, we use the same method as in the feasibility function for binary encoding, which was defined in Eq. 3.35. After applying this function, we arrive at

$$
\begin{aligned}
O_{\text{binary}}(X,Y,\hat{V}) =& A\sum_i^N\sum_j^M c_{i,j}x_{i,j} + B\cdot\sum_k^R\left(d - \left(\sum_i^N\sum_j^M p_{i,k}t_{i,j}x_{i,j} + \sum_l^{s_k} 2^{s_k-l-1}y_{k,l}\right)\right)^2 \\
&+ C\cdot\sum_i^N\sum_{j=M}^{2^{\hat{M}}} g_{\text{binary}\,j}(\hat{v}_i),
\end{aligned}
\tag{5.4}
$$

where $\hat{V}(i)$ is the binary string representing the machine on which task $i$ is executed. For each machine, we check if its index happens to fall between $M$ and $2^{\hat{M}}$. In this formulation, the feasibility constraint will be equal to the number of tasks executed on infeasible machines.

### 5.1.3 Domain wall encoding

In domain wall encoding, we need $\hat{M} = M - 1$ bits to encode $M$ machines. Just like previously, we flatten the vector to describe the $N$ tasks, using $N \cdot (M-1)$ bits in total. The remainder of vector $V$ is used for slack variables. An example of a domain wall encoded vector $V$, analogous to the one in Figures 5.1 and 5.2 can be seen in Figure 5.3.

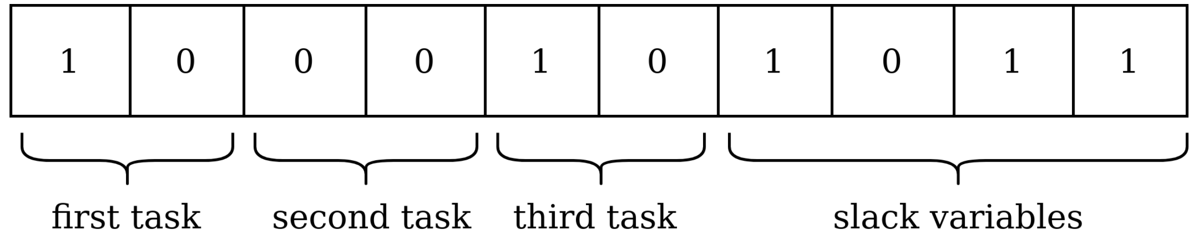



**Figure 5.3:** An example of a domain wall encoded workflow scheduling vector $V$

In order to define a translation between $V$ and $X$, let us first define a matrix $W$ of size $N \times M$ that takes the form

$$W(i,j) = \begin{cases} 1 - V(i \cdot (M-1)) & \text{if } j = 0, \\ V(i \cdot (M-1) + j) - V(i \cdot (M-1) + j + 1) & \text{if } 0 < j < M-1, \\ V((i+1) \cdot (M-1) - 1) & \text{if } j = M-1. \end{cases} \tag{5.5}$$

This matrix corresponds directly to the definition from Eq. 3.43, the only difference is that it looks at the appropriate indices of $V$. This matrix can contain the values $\{-1, 0, 1\}$ and its meaning can be decoded in the following way:

$$W(i,j) = \begin{cases} 1 & \text{if task } i \text{ is executed on machine } j, \\ 0 & \text{if task } i \text{ is not executed on machine } j, \\ -1 & \text{if this is an infeasible configuration.} \end{cases} \tag{5.6}$$

While this is not yet synonymous with one-hot encoding, it is relatively similar. Instead of vectors 001, 010, and 100, we can denote the same states with vectors 11, 10, and 00. In such a two-bit system, we only have one infeasible state: 01, which, after applying the formula defined above, would be decoded into the vector $[1, -1, 1]$. As described in Section 3.3.3, the appearance of a $-1$ is a characteristic of invalid vectors. This issue can be bypassed in different ways, but in our general implementation, we simply square the value of $W(i, j)$. After this operation, the vector $[1, -1, 1]$ would be converted to $[1, 1, 1]$, which is similar to an infeasible one-hot state. This way, we arrive at the final formula for the conversion between $V$ and $X$, which is defined as

$$X(i,j) = W(i \cdot M, j)^2. \tag{5.7}$$

As already stated, by squaring these values, we arrive at a pattern similar to one-hot encoding, with one notable exception – there is no state synonymous with vector 000. All the infeasible configs correspond to a situation in which multiple machines are selected for a specific task. Since these vectors are already penalized by both the cost function and the time function, it is sometimes possible to drop the feasibility constraint, as the optimizer should

never prioritize such a config over a feasible one.

As was the case for binary encoding in Eq. 5.4, when considering domain wall encoding, we also have a way of simplifying the feasibility constraint by employing the feasibility function defined in Eq. 3.44. A correct domain wall vector has exactly one wall, while incorrect vectors have more walls. Therefore, we do not have to create a function enforcing that the number of walls is equal to 1 and instead, we can just minimize the number of walls in the following way:

$$
\begin{aligned}
O_{\text{domain-wall}}(X,Y,V) = & A\sum_{i}^{N}\sum_{j}^{M} c_{i,j}x_{i,j} + B\cdot\sum_{k}^{R}\left(d-\left(\sum_{i}^{N}\sum_{j}^{M} p_{i,k}t_{i,j}x_{i,j}+\sum_{l}^{s_k}2^{s_k-l-1}y_{k,l}\right)\right)^2 \\
& + C\cdot\sum_{i}^{N}\left((1-v_{i\cdot\hat{M}})^2+\sum_{j}^{M-2}(v_{i\cdot\hat{M}+j}-v_{i\cdot\hat{M}+j+1})^2+{v_{(i+1)\cdot\hat{M}-1}}^2\right).
\end{aligned}
\tag{5.8}
$$

## 5.2 Mixers

Two of the used encoding schemes can be used in combination with custom mixers. The $XY$ mixer for one-hot and the custom mixer for domain wall were both described in Section 3.3.4.

It is important to note is that in our representation of workflow scheduling, these mixers can be used only on the bits describing the specific tasks. The slack bits need to have the default $X$ mixer applied to them.

### 5.2.1 One-hot encoding

In order to apply the $XY$ mixer defined in Chapter 3.25 to our one-hot-encoded workflow scheduling problem, we have to apply this mixer to the encoding of each task separately. We

also have to apply the default $X$ mixer to the slack variables. The full formula is defined as

$$\begin{aligned} H_{M_{\text{one-hot}}} = &\sum_{i}^{N}\sum_{j}^{M-1} I^{\otimes i\cdot M+j} \otimes X \otimes X \otimes I^{\otimes M(N-i-1)+M-2-j} \otimes I^{\otimes \hat{S}} \\ &+ \sum_{i}^{N}\sum_{j}^{M-1} I^{\otimes i\cdot M+j} \otimes Y \otimes Y \otimes I^{\otimes M(N-i-1)+M-2-j} \otimes I^{\otimes \hat{S}} \\ &+ \sum_{i}^{\hat{S}} I^{\otimes N\cdot M+i} \otimes X \otimes I^{\otimes \hat{S}-i-1}, \end{aligned} \tag{5.9}$$

where $\hat{S}$ denotes the number of slack variables, and the $I^{\otimes n}$ notation is used to describe a tensor product of $n$ identity matrices, as defined in Eq. 3.5.

When used in combination with this custom mixer, the objective function for one-hot encoding from Eq. 4.10 can be simplified to:

$$O_{\text{one-hot}}(X,Y) = A\sum_{i}^{N}\sum_{j}^{M} c_{i,j}x_{i,j} + B\sum_{k}^{R}\left(d - \left(\sum_{i}^{N}\sum_{j}^{M} p_{i,k}t_{i,j}x_{i,j} + \sum_{l}^{s_k} 2^l y_{k,l}\right)\right)^2. \tag{5.10}$$

The main difference is the exclusion of the feasibility constraint, which is possible, because the mixer ensures that only feasible configurations are processed and therefore no extra penalty should be needed.

### 5.2.2 Domain wall encoding

Similarly to the $XY$ mixer, in order to apply the custom domain wall mixer defined in Section 3.25 to our domain-wall-encoded problem, we have to apply this mixer to the encoding of each task separately. We also have to apply the default $X$ mixer to the slack variables. The

full formula is defined as

$$
\begin{aligned}
H_{M\text{domain-wall}} = \sum_i^N [ & -I^{\otimes i\cdot\hat{M}} \otimes X \otimes I^{\otimes (N-i-1)\cdot\hat{M}+M-2+\hat{S}} \\
& + \sum_j^{M-2} [(-1) I^{\otimes i\cdot\hat{M}+j} \otimes X \otimes Z \otimes I^{\otimes (N-i-1)\cdot\hat{M}+M-2-j+\hat{S}} \\
& \quad + I^{\otimes i\cdot\hat{M}+j} \otimes Z \otimes X \otimes I^{\otimes (N-i-1)\cdot\hat{M}+M-2-j+\hat{S}}] \\
& - I^{\otimes (i+1)\cdot\hat{M}-1} \otimes X \otimes I^{\otimes (N-i-1)\hat{M}+\hat{S}}] \\
& + \sum_i^{\hat{S}} I^{\otimes N\cdot\hat{M}+i} \otimes X \otimes I^{\otimes \hat{S}-i-1},
\end{aligned}
\tag{5.11}
$$

where $\hat{S}$ denotes the number of slack variables, and the $I^{\otimes n}$ notation is used to describe a tensor product of $n$ identity matrices, as defined in Eq. 3.5.

When used in combination with a custom mixer, the formula for the domain-wall-encoded objective function from Eq. 5.8 can be simplified to

$$
O_{\text{domain-wall}}(X,Y) = A\sum_i^N \sum_j^M c_{i,j} w_{i\cdot M,j} + B\sum_k^R \left( d - \left( \sum_i^N \sum_j^M p_{i,k} t_{i,j} w_{i\cdot M,j} + \sum_l^{s_k} 2^{s_k-l-1} y_{k,l} \right) \right)^2 . \tag{5.12}
$$

The main difference is the exclusion of the feasibility constraint (since feasibility is guaranteed by the mixer). Additionally, the transformation from Eq. 5.7 can be skipped, since it's only necessary when dealing with invalid configurations (as per Eq. 5.7).

## Summary

In this chapter, we derived the specific implementations of the workflow scheduling problem from Chapter 4 for each of the encodings presented in Section 3.3. We also defined the formulas of two QAOA mixers. The proposed implementation will be used in the experiments described in the two following chapters.

# Chapter 6

# Experiment design

In this chapter, we discuss the design of our experiments. Firstly, we focus on various experiment parameters, including the chosen algorithms and classical optimizers, the method used for initial point selection, and the selected QUBO parameters. We conclude the chapter by introducing the specific instances of workflow scheduling that will be used in our experiments.

## 6.1 Optimization algorithms

In previous works focusing on quantum workflow scheduling, the main focus has been VQE [64]. We have chosen to also consider QAOA. The inclusion of QAOA opens up the door for including a custom mixer in our computations. This lead us to arrive at the following combinations:

- QAOA with different mixers:
  - the default $X$ mixer and encodings:
    - one-hot,
    - domain wall,
    - binary,
  - a custom encoding-dependent mixer and encodings:

* one-hot ($XY$ mixer),
    * domain wall (domain wall mixer),

- VQE with encodings:
  - one-hot,
  - domain wall,
  - binary.

### 6.1.1 Classical optimizers

Although Qiskit offers numerous classical optimizers[1], previous research on workflow scheduling has focused on a single optimizer, specifically SPSA [64]. We have chosen to compare a few different optimizers to see whether any of them are preferable. The following optimizers were taken into account:

- COBYLA (Constrained Optimization By Linear Approximation optimizer), gradient-free [60, 57, 59],
- POWELL, gradient-free [58],
- NELDER-MEAD, gradient-free [48],
- L-BFGS-B (Limited-memory Broyden-Fletcher-Goldfarb-Shanno Bound), gradient-based [75].

Our choice of optimizers was backed by findings from literature, as well as personal experience. COBYLA and L-BFGS-B have been found to be fast even in noisy environments [38]. Research also suggests that SPSA, POWELL, and L-BFGS-B are all effective even in noisy environments, while COBYLA and NELDER-MEAD were found to handle noise worse [54]. SPSA was excluded from these tests due to its run time being very long.

[1] https://qiskit.org/documentation/apidoc/qiskit.aqua.components.optimizers.html

### 6.1.2 Initial point selection

The importance of an initial point is an issue overlooked in literature and in previous research. We have conducted additional small experiments testing the influence of this point on the results of a simple toy problem. The equation we tried to minimize was $x_0 + 5x_1 + 10x_2$ – its minimum occurs at vector 000. The results of this experiment can be seen in Figure 6.1. While most initial points render results that assign a high probability to the correct answer, there's also many points that result in incorrect results.

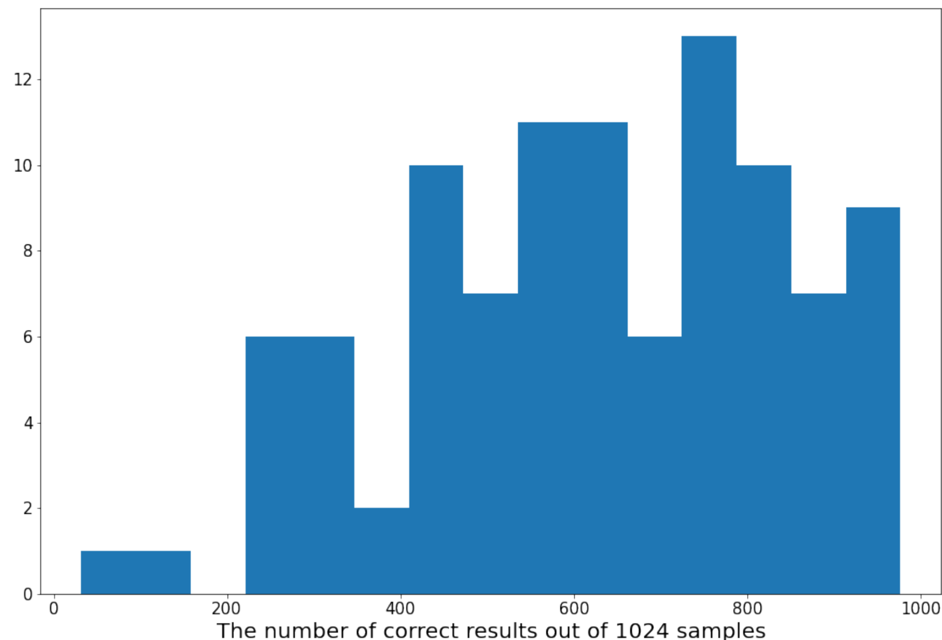



**Figure 6.1:** A histogram of the number of correct results in 1024 samples over 100 randomized initial points

This issue is not something that's often discussed in literature. We considered two approaches here: we could either look for a single point that renders the best result, or we could repeat the experiment enough times to be able to disregard the influence of randomization on our results. The first approach didn't seem to be a good choice for two reasons. Firstly, finding a good initial point is a tedious and partly manual process. Secondly, reporting on a single best result isn't particularly honest and realistic. This point can't be reused for other

problems, and it doesn't reflect on the quality of the solution, since it is the result of reverse engineering.

**Reusing initial points**

While there is no indication that these initial points can be reused as the size of the problem grows or that they're universally efficient regardless of the chosen classical optimizer, there is a way these values can be reused to improve one's results. The basic idea for both QAOA and VQE is that we can make more informed guesses for the initial point as the depth of the circuit grows. This approach is most often discussed for QAOA and its parameter $p$ [26], however it has also been suggested that it could be an efficient way to improve the results of VQE as the number of repetitions increases.

### 6.1.3 Initial state

Initial state is another parameter passed to QAOA. By default, the initial state is a superposition of all possible configurations, however often it is preferable to replace that with a custom vector. Most notably, when QAOA mixers are used, it is necessary to pass the superposition of all *feasible* states. This way, the mixer can remain in the feasible subspace and only consider valid vectors. In our case, this superposition contained all the correctly encoded machine combinations with every possible slack variable value.

### 6.1.4 QUBO parameter selection

The definition of workflow scheduling from Eq. 4.7 includes three clauses. The first clause is the objective function, and the other two include penalties. Each clause has its respective weight, and by manipulating these weights we can alter the ordering of the energies of the possible configurations. This ordering has to follow two main principles:

- the optimal solution has to have the lowest energy,
- the energy of each correct solution has to be lower than that of an incorrect solution.

While these two principles generate a valid ordering, there are additional tweaks that can be implemented. Before we discuss them, let us define the following mutually exclusive types of solutions to the workflow scheduling problem:

- the optimal solution,
- a correct solution – a solution that represents a correct mapping and fits within the deadline (**note**: we do not include the optimal solution in the correct solution count),
- a semi-optimal solution – a solution identical to the optimal solution, but with an invalid slack configuration,
- a semi-correct solution – a solution identical to some correct solution, but with an invalid slack configuration,
- an incorrect solution – a solution that is not optimal, correct, semi-optimal, or semi-correct, meaning it either corresponds to an invalid configuration or it exceeds the deadline.

These metrics were specifically designed to add up to the number of total solutions (100%) returned from the experiment. The reason for highlighting these *semi-* solutions is that they cannot really be considered to be correct, since their energies includes a penalty for a seemingly exceeded deadline. The deadline is not actually exceeded, as it's just the slacks that give this impression, however the algorithm has no way of distinguishing between, for instance, a solution exceeding the deadline by 1 and a correct solution with a single extra slack. While the *semi-* solutions are not really desired, they are still preferable to incorrect solutions – a *semi-* solution corresponds to an actual correct configuration, so from a practical perspective it is quite sensible.

In a perfect setting, we would want the optimal and semi-optimal solutions to have the lowest energies, followed by correct and semi-correct solutions. The energies of incorrect solutions should be higher, preferably with feasible configurations having energies lower than infeasible solutions. This ordering is presented in Figure 6.2.

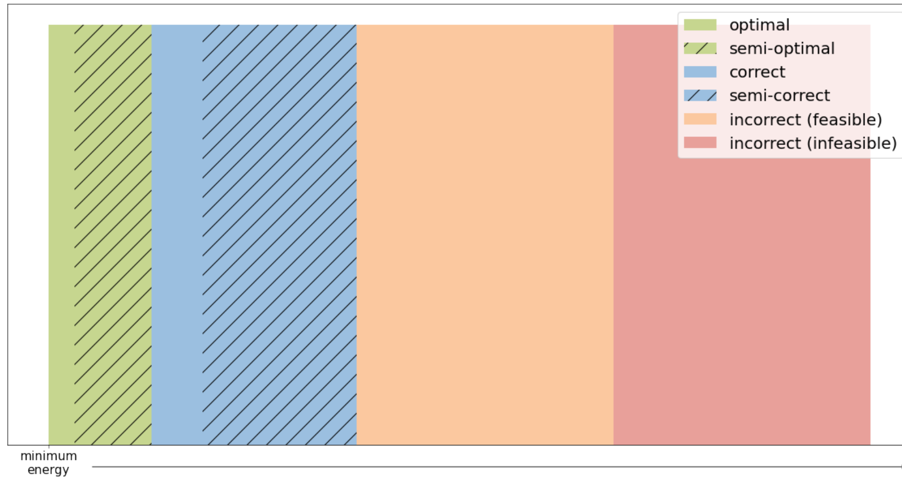



**Figure 6.2:** A one-dimensional visualization of the ideal ordering of solution energies

However, this ideal ordering is not possible due to penalties affecting the *semi-* solutions, which make it impossible to distinguish between solutions that exceed the deadline and solutions that are semi-correct or semi-optimal. Therefore, we initially settled on an approach that shall be referred to as the *naive ordering*, in which we do guarantee that the energies of optimal and correct solutions are the lowest, however all other solutions remain mixed with each other. This ordering is shown in Figure 6.3.

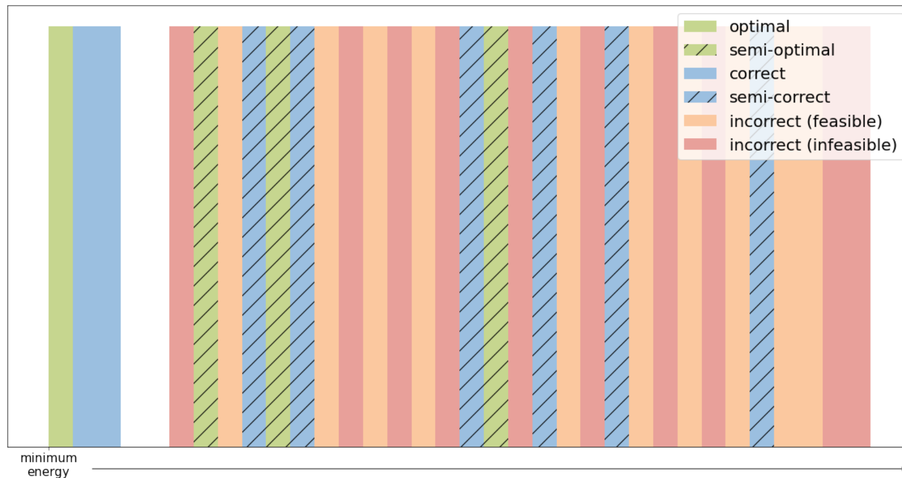



**Figure 6.3:** A one-dimensional visualization of the *naive ordering* of solution energies

Eventually, we tried manipulating the $A$, $B$, $C$ weights further to increase the gap between the correct solutions and the rest, but this was not particularly effective. Finally, we settled on an ordering, which will be referred to as the *feasibility-jump ordering*. This ordering guarantees the lowest energies for optimal and correct solutions and the highest energies for incorrect and infeasible solutions. In the middle, we have a mix of *semi-* solutions as well as incorrect feasible solutions. This ordering is illustrated in Figure 6.4.

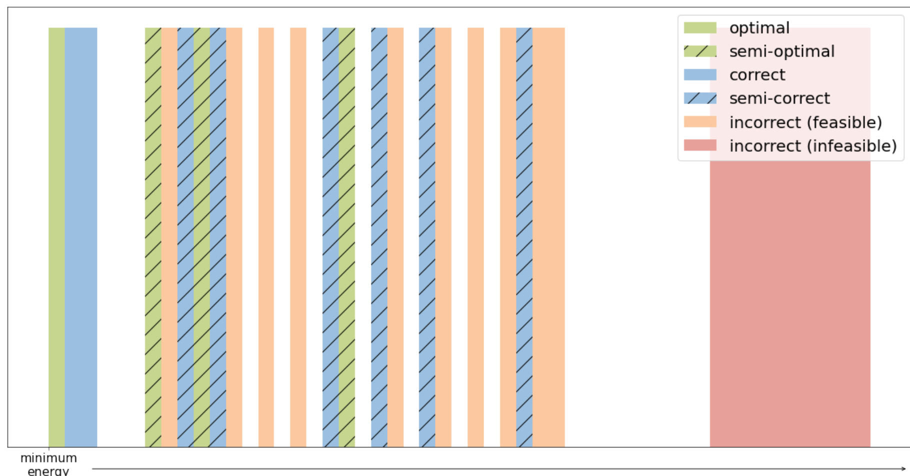



**Figure 6.4:** A one-dimensional visualization of the *feasibility jump ordering* of solution energies

This ordering was possible due to the feasibility constraint from Eq. 4.7 having a separate weight, $C$. By sending those infeasible configurations away from the rest, we found some improvement in the results. In Chapter 7 we will discuss the results for one-hot encoding and binary encoding for two sets of weights ensuring the naive ordering and the feasibility-jump constraint.

### 6.1.5 Experiment randomization

In previous research, experiments were conducted a single time and the presented results focused on a single most successful run [64]. We have chosen a different approach – each experiment was repeated a number of times and the results were averaged. This approach

seemed more reliable, since there is a lot of variation in the results depending on the selected initial point.

### 6.1.6 Result evaluation metrics

Different metrics can be used for evaluating the obtained results. A good metric is necessary not only for assessing the quality of a solution, but also when using the $p$-reusing trick described in 6.1.2. In this method, one needs to select the best solution for a specific $p$ and then the optimal point of this solution is used as the initial point for $p+1$.

The following metrics were taken into consideration:

- the number of correct solutions,
- the number of optimal solutions,
- the number of feasible solutions,
- the average energy between all solutions,
- the energy of the most frequent solution,
- the highest energy from among the found solutions,
- the lowest energy from among the found solutions.

Finally, we settled on the "average energy between all solution" approach, however, we admit that this metric is not perfect, as the standard deviation also appears to affect the results tremendously.

For result evaluation in Chapter 7 we shall be using the average energy, as well as the correct, optimal, incorrect, and feasible counts.

## 6.2 Considered workflows

Three different workflows were considered: a single small problem to be tested against each encoding, and two large problems designed to test the capabilities of the two denser encodings.

### 6.2.1 Small problem

The small problem was used for tests of all three encoding. It consists of three machines and three tasks. The costs and times for each machine and task can be seen in Tables 6.1 and 6.2. The DAG is shown in Figure 6.5. The number of required slack variables is 4. Therefore, the total number of qubits for the least dense encoding (one-hot) is

$$N \cdot \hat{M} + \hat{S} = N \cdot M + \hat{S} = 3 \cdot 3 + 4 = 13, \tag{6.1}$$

where $N$ is the number of tasks, $\hat{M}$ is the number of qubits needed to encode $M$ machines (which for one-hot encoding are equal) and $\hat{S}$ is the number of slack variables, as described in Section 5.1. Binary and domain wall encoding both required 10 qubits, since the total number of qubits for binary encoding is

$$N \cdot \hat{M} + \hat{S} = N \cdot \lceil \log_2 M \rceil + \hat{S} = 3 \cdot 2 + 4 = 10, \tag{6.2}$$

and for the domain wall encoding

$$N \cdot \hat{M} + \hat{S} = N \cdot (M - 1) + \hat{S} = 3 \cdot 2 + 4 = 10. \tag{6.3}$$

| | Machine A | Machine B | Machine C |
|---|---|---|---|
| Task 1 | 6 | 8 | 8 |
| Task 2 | 3 | 4 | 4 |
| Task 3 | 12 | 16 | 16 |

**Table 6.1:** Small problem: cost matrix

| | Machine A | Machine B | Machine C |
|---|---|---|---|
| Task 1 | 6 | 2 | 8 |
| Task 2 | 3 | 1 | 2 |
| Task 3 | 12 | 4 | 8 |

**Table 6.2:** Small problem: time matrix

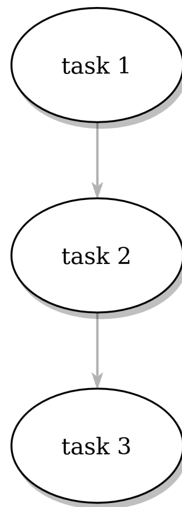



**Figure 6.5:** Small problem: task order DAG

### 6.2.2 Large problems

Two larger problems were used for tests of the more dense encoding – binary encoding and domain wall encoding. Both problems consisted of 4 tasks and both used the same DAG (shown in Figure 6.6), but each used a different number of machines and different cost and time matrices.

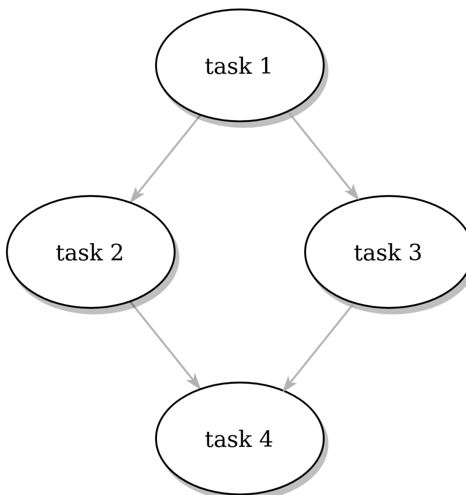



**Figure 6.6:** Large problem: task order DAG

### Binary encoding

The problem used to test binary encoding consisted of 4 tasks and 4 machines. The number of slack variables was 7 in total, 3 and 4 for each of the paths, respectively. The overall number of qubits is equal to $4 \cdot \lceil \log_2 4 \rceil + 3 + 4 = 4 \cdot 2 + 7 = 15$. The cost and time matrices can be seen in Tables 6.3 and 6.4.

| | Machine A | Machine B | Machine C | Machine D |
|---|---|---|---|---|
| Task 1 | 1 | 2 | 4 | 8 |
| Task 2 | 2 | 4 | 8 | 16 |
| Task 3 | 4 | 8 | 16 | 32 |
| Task 4 | 8 | 16 | 32 | 64 |

**Table 6.3:** Large problem (binary encoding): cost matrix

| | Machine A | Machine B | Machine C | Machine D |
|---|---|---|---|---|
| Task 1 | 4 | 2 | 2 | 1 |
| Task 2 | 8 | 4 | 4 | 2 |
| Task 3 | 16 | 8 | 8 | 4 |
| Task 4 | 32 | 16 | 16 | 8 |

**Table 6.4:** Large problem (binary encoding): time matrix

### Domain wall encoding

The problem used in domain encoding was slightly smaller due to the encoding being less dense – it contained 3 instead of 4 machines. The number of slack variables was 3 on each path, so the total number of qubits was equal to $4 \cdot (3-1) + 6 = 4 \cdot 2 + 6 = 14$. The cost and time matrices can be seen in Tables 6.5 and 6.6.

| | Machine A | Machine B | Machine C |
|---|---|---|---|
| Task 1 | 2 | 1 | 3 |
| Task 2 | 4 | 2 | 6 |
| Task 3 | 6 | 3 | 9 |
| Task 4 | 8 | 4 | 12 |

**Table 6.5:** Large problem (domain wall encoding): cost matrix

| | Machine A | Machine B | Machine C |
|---|---|---|---|
| Task 1 | 4 | 5 | 3 |
| Task 2 | 8 | 10 | 6 |
| Task 3 | 12 | 15 | 9 |
| Task 4 | 16 | 20 | 12 |

**Table 6.6:** Large problem (domain wall encoding): time matrix

## Summary

We began this chapter by explaining how the experiments were designed, including the choice of classical optimizers, the selection and randomization of parameters, and the choice of metrics used to evaluate the results. The chapter concludes with a description of three workflow scheduling instances, the results of which will be presented in the next chapter.

# Chapter 7

# Evaluation of the results

In this chapter, we describe the results obtained from the experiments described previosly in Chapter 6. The experiments were conducted on the Prometheus supercomputer at the AGH UST Academic Computer Centre CYFRONET with the usage of the PLGrid infrastructure.

As mentioned in Section 6.1, for each out of three encodings, the QAOA and VQE algorithms were both tested. In addition, for the encodings with a dedicated mixer available, QAOA was tested with the custom and the default mixer separately. In our case, as described in Section 3.3.4, only binary encoding did not have a custom mixer, which means that we tested eight combinations of encodings and algorithms. Then, for each such pair, we tested four different optimizers (see Section 6.1.1), and for each *encoding-algorithm-optimizer* triple, we repeated the experiment with the *p*/*reps* parameters ranging from 1 to 3 (as described in Sections 3.2.2 and 3.2.3).

Therefore, for the smaller problem instance we ran $8 \cdot 4 \cdot 3 = 96$ experiments, while for the larger problem, since one-hot encoding was omitted (which will be described in Section 7.2), we ran $5 \cdot 4 \cdot 3 = 60$ experiments. Each experiment was repeated 1000 times to get the most reliable averages. Additionally, it is worth mentioning that every repetition measures the quantum circuit 1024 times, which is defined via the *shots*[1] parameter.

The results shown in this section below are presented in the form of *box plots*[2].

[1] https://qiskit.org/documentation/apidoc/execute.html

[2] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html

## 7.1 Smaller problem

The first problem used in our experiments was the smaller problem defined in Section 6.2.1. The problem was chosen so that all its three encodings would fit on 15 or fewer qubits. The goal was to eventually run these experiments on the IBM Melbourne machine, the largest publicly available IBM quantum computer.

### 7.1.1 One-hot encoding

Using one-hot encoding, we tested QAOA with both the default $X$ mixer and the custom $XY$ mixer (described in Section 3.3.4), as well as VQE.

#### QAOA with a default mixer

The eigenvalues obtained in this experiment can be seen in Figure 7.1. The *eigenvalue* will henceforth mean the average eigenvalue from the given sample (repetition), since, as already mentioned, a single sample measures the quantum circuit 1024 times and thus returns 1024 eigenstates with their corresponding eigenvalues. This metric, along with others, is described in Section 6.1.6.

Overall, this algorithm performed quite poorly, as for all optimizers the result medians were in the 6000–14000 range, while the minimum eigenvalue, marked with a red line in Fig. 7.1, was around $-5000$. The worst results were obtained by the L-BFGS-B optimizer, and the best results were obtained using the POWELL optimizer with $p$ set to 2. It can be observed that **increasing the $p$ parameter to 2 improved the results, while increasing it to 3 had the opposite effect**. The strategy used for choosing initial points when increasing the $p/reps$ parameter was presented in Section 6.1.2.

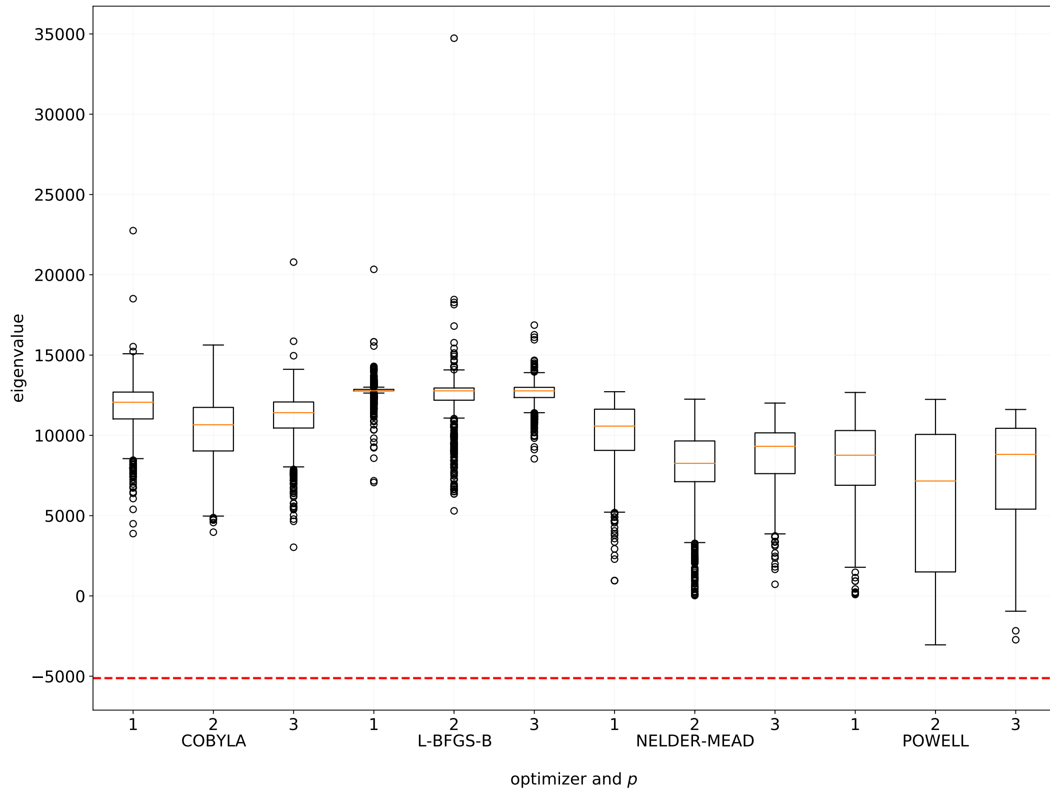
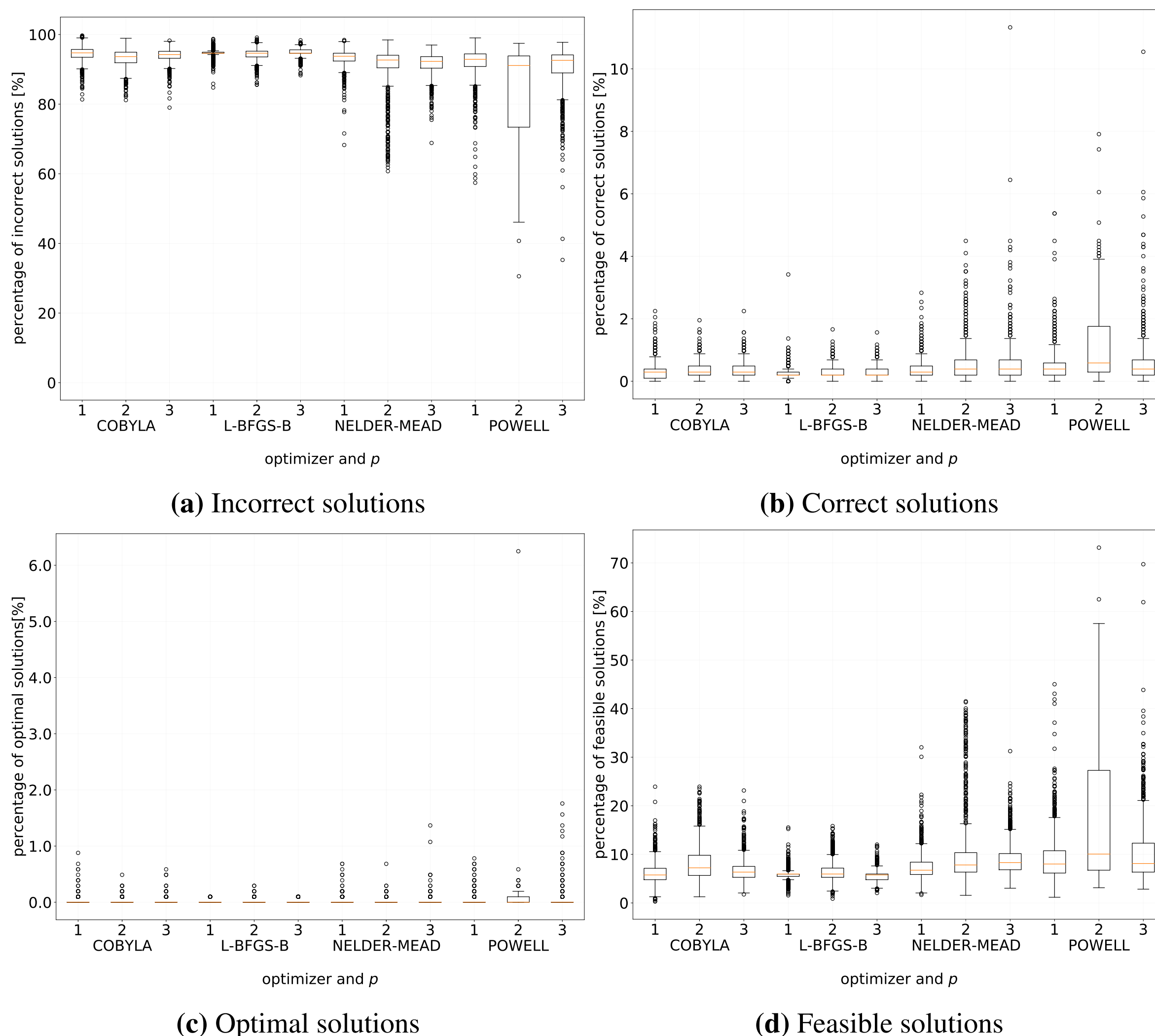



**Figure 7.1:** The eigenvalues for one-hot encoding and QAOA with a default mixer

An analysis of incorrect, correct, optimal, and feasible solution percentages, as obtained by QAOA with a default mixer, is shown in Fig. 7.2. The performance of this encoding is rather unsatisfactory, as all optimizers gave a median incorrect solution percentage higher than 90%. The POWELL optimizer with $p = 2$ performed the best, as can be seen in all four charts. This combination had the lowest percentage of incorrect solutions and the highest median percentage of correct, optimal, and feasible solutions.




**(a)** Incorrect solutions

**(b)** Correct solutions

**(c)** Optimal solutions

**(d)** Feasible solutions

**Figure 7.2:** A result breakdown for one-hot encoding and QAOA with a default mixer

### QAOA with a custom mixer

The eigenvalues obtained by QAOA with a custom mixer are presented in Fig. 7.3. Similarly to its performance in conjunctions with the default mixer, this method also did not give satisfactory results. Among the four optimizers, L-BFGS-B performed the worst, while POWELL was slightly better than the rest. Increasing the $p$ parameter improved the results only for the NELDER-MEAD optimizer, while for the others it did not bring any improvement.

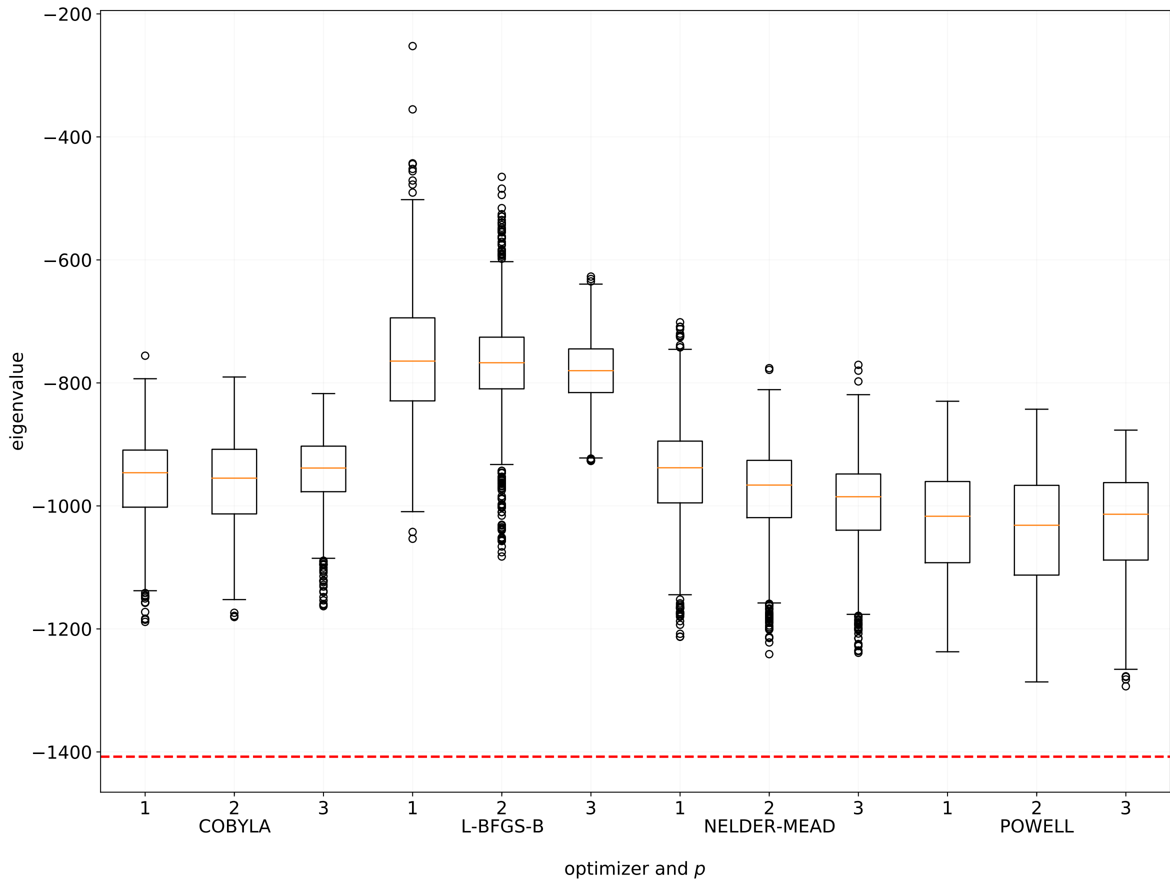
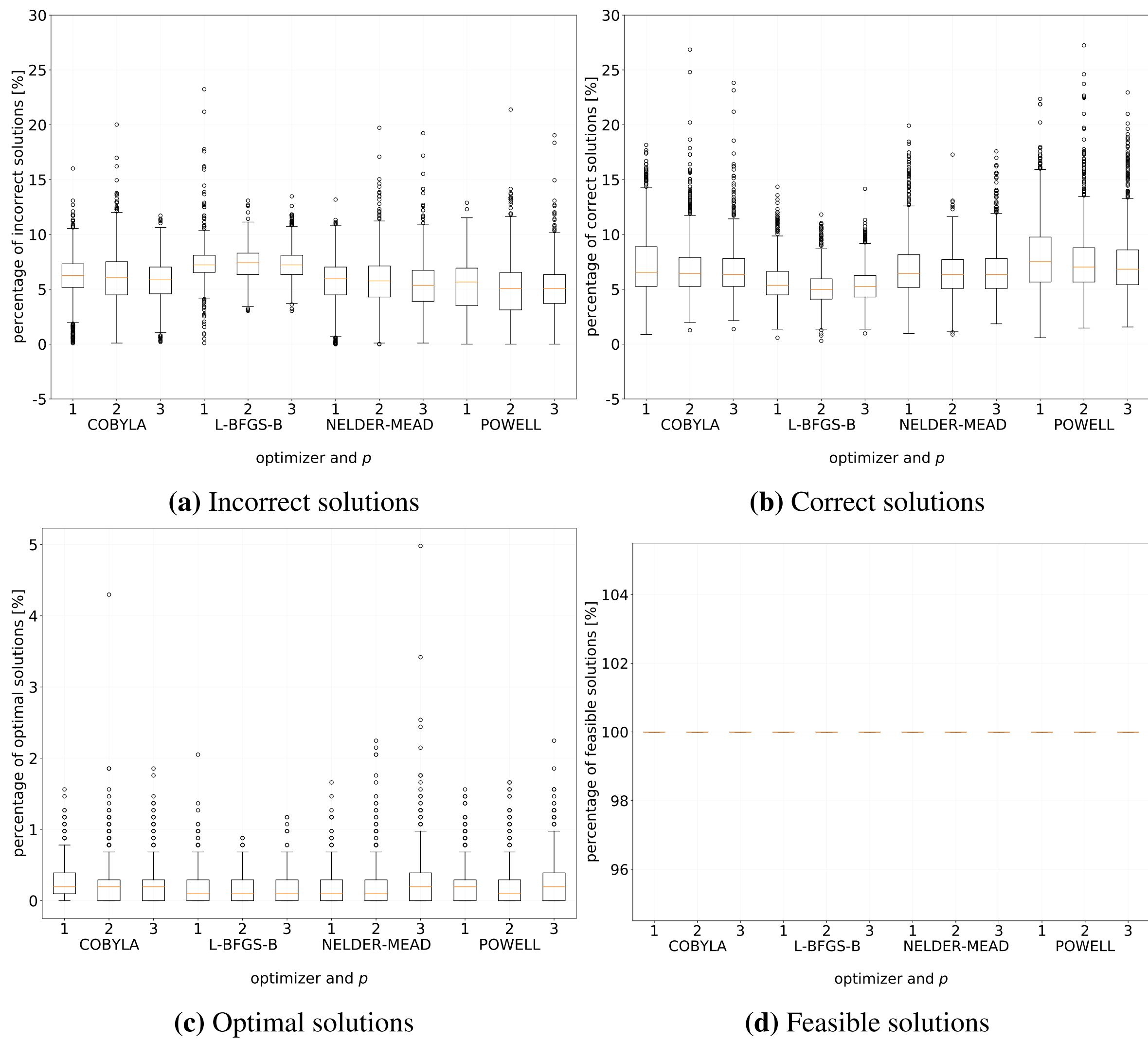



**Figure 7.3:** The eigenvalues for one-hot encoding and QAOA with a custom mixer

An analysis of incorrect, correct, optimal, and feasible solution percentages, as obtained by QAOA with a custom mixer, is shown in Fig. 7.4. From these plots, we can observe that all the optimizers performed similarly, with L-BFGS-B performing slightly worse than the others. The percentage medians of incorrect and correct solutions are in the range of 5%–10%, while the optimal solutions are in the range of 0%–0.3%, In each experiment, the percentage of feasible solutions is equal to 100%, because the custom mixer restricts the solution subspace to only the logically correct solutions.




**(a)** Incorrect solutions

**(b)** Correct solutions

**(c)** Optimal solutions

**(d)** Feasible solutions

**Figure 7.4:** A result breakdown for one-hot encoding and QAOA with a custom mixer

### VQE

The eigenvalues obtained by VQE, used in conjunction with one-hot encoding, are presented in Fig. 7.5. This algorithm gave us results better than the two versions of QAOA discussed above, but there is a significant variation in results depending on the chosen optimizers. L-BFGS-B and NELDER-MEAD performed the worst, while COBYLA and POWELL allowed

for much better results. The lowest eigenvalues were obtained by the POWELL algorithm with $reps = 2$. Increasing the *reps* parameter improved the results only for L-BFGS-B optimizer.

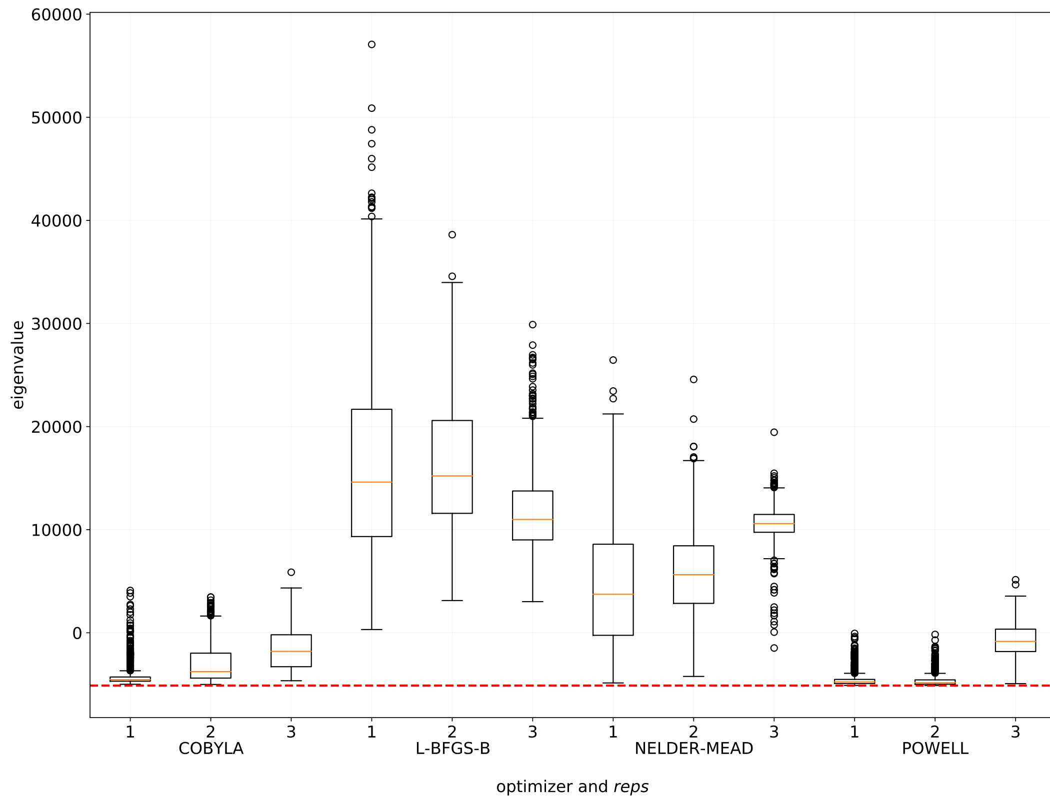
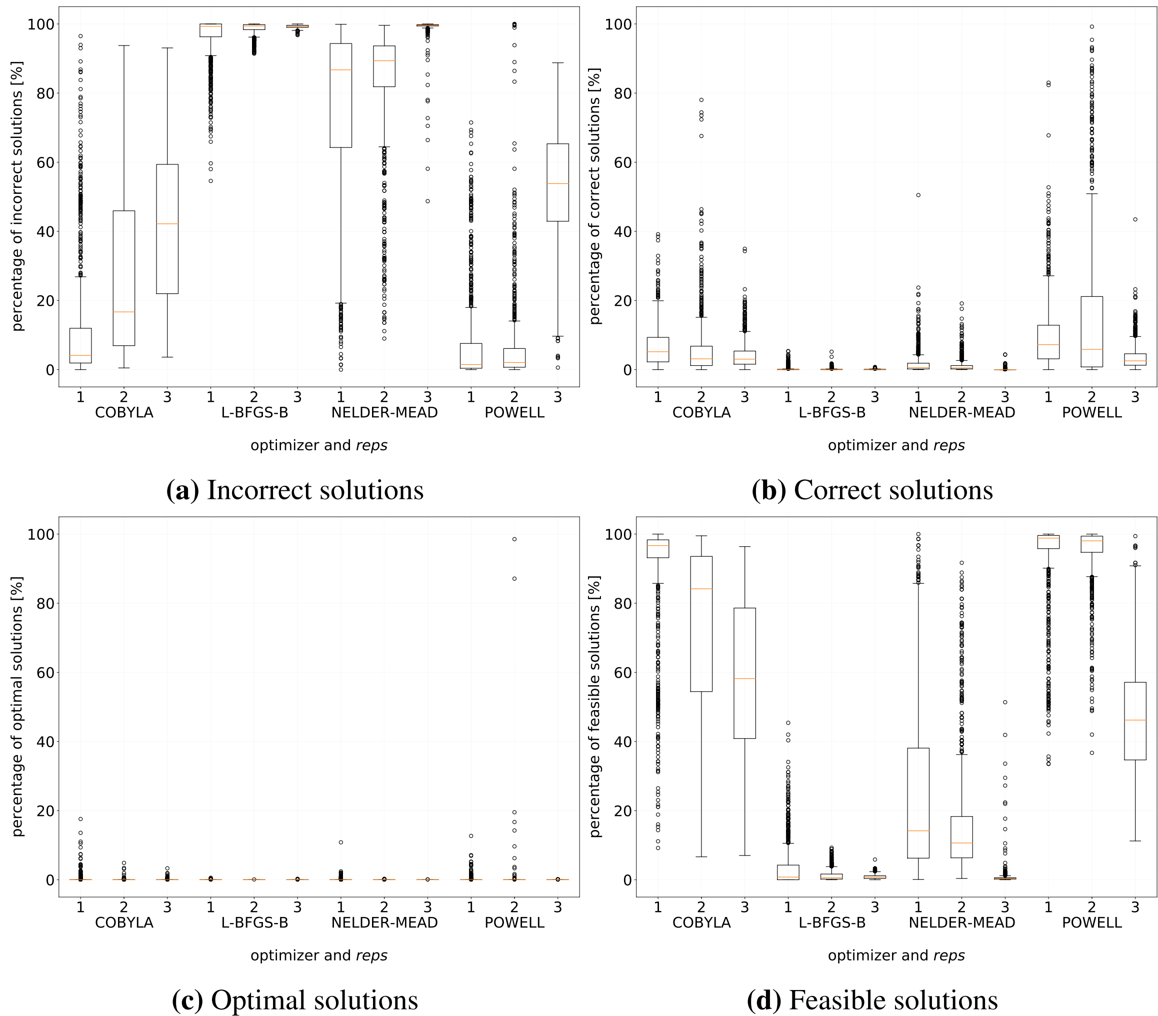



**Figure 7.5:** The eigenvalues for one-hot encoding with VQE

An analysis of incorrect, correct, optimal, and feasible solution percentages, as obtained by VQE, is shown in Fig. 7.6. The L-BFGS-B optimizer turned out to be the weakest, which can be noticed in the plot of incorrect solution percentages. The NELDER-MEAD optimizer performed slightly better, and the COBYLA optimizer had an even more significant decrease in the percentage of incorrect solutions.

Finally, the POWELL optimizer had the lowest percentage of incorrect solutions, as well as the highest percentage of correct solutions (with a few samples greater than 90%), the highest percentage of optimal solutions (with two outstanding samples greater than 80%), and the highest percentage of feasible solution (the median for $reps = 1$ was close to 100%). By

analyzing the four metrics, one can conclude that **increasing the value of the *reps* parameter did not result in any improvement**.




**(a)** Incorrect solutions

**(b)** Correct solutions

**(c)** Optimal solutions

**(d)** Feasible solutions

**Figure 7.6:** A result breakdown for one-hot encoding with VQE

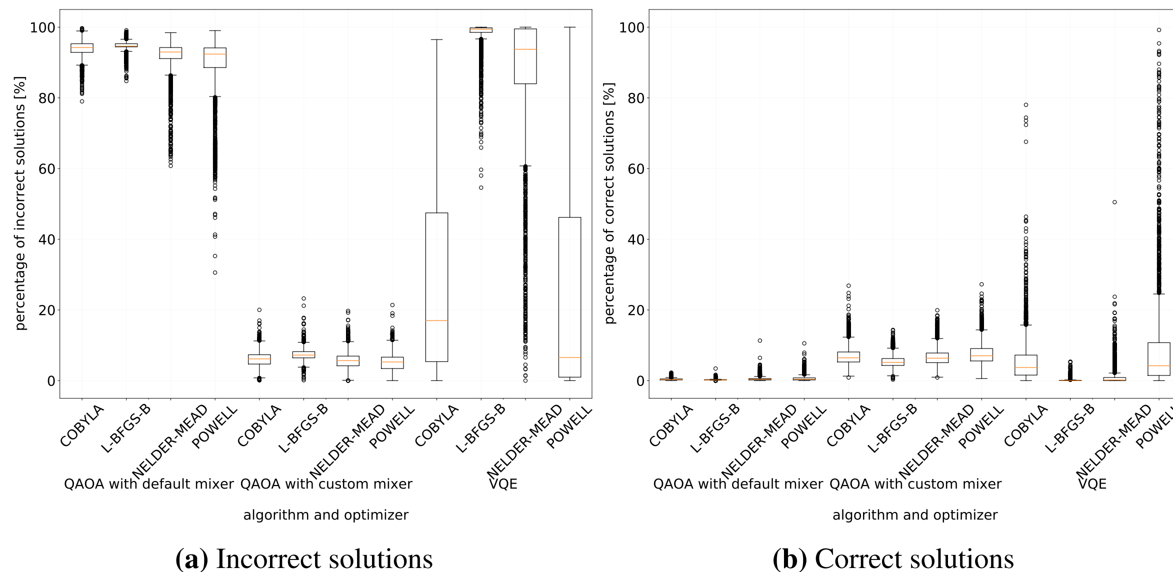
### One-hot encoding comparison

A comparison of correct and incorrect solution percentages for different algorithms used in conjunction with one-hot encoding is presented in Fig. 7.7.

QAOA with a custom mixer proved to be the best algorithm when considering the per-

centage of incorrect solutions (Fig. 7.7a), as for all four optimizers the percentage of incorrect solutions was lower than 20%. QAOA with a default mixer was mostly in the 80%–100% range and the VQE algorithm had a significantly higher variance, however the median for POWELL was similar to the median of incorrect solutions for QAOA with a custom mixer.

Comparing the percentages of correct solutions (Fig. 7.7b), one can see that QAOA with a default mixer had the worst performance, with typically less than 10% of correct solutions. QAOA with a custom mixer performed better, with typically up to 30% of correct solutions. The best results were obtained using VQE with COBYLA and POWELL – although the median is low, there are many outliers reaching as high as 100%. It can be noted that **VQE has the largest variance of correct solutions. The performance of QAOA is rather similar among the different optimizers**.




**(a)** Incorrect solutions

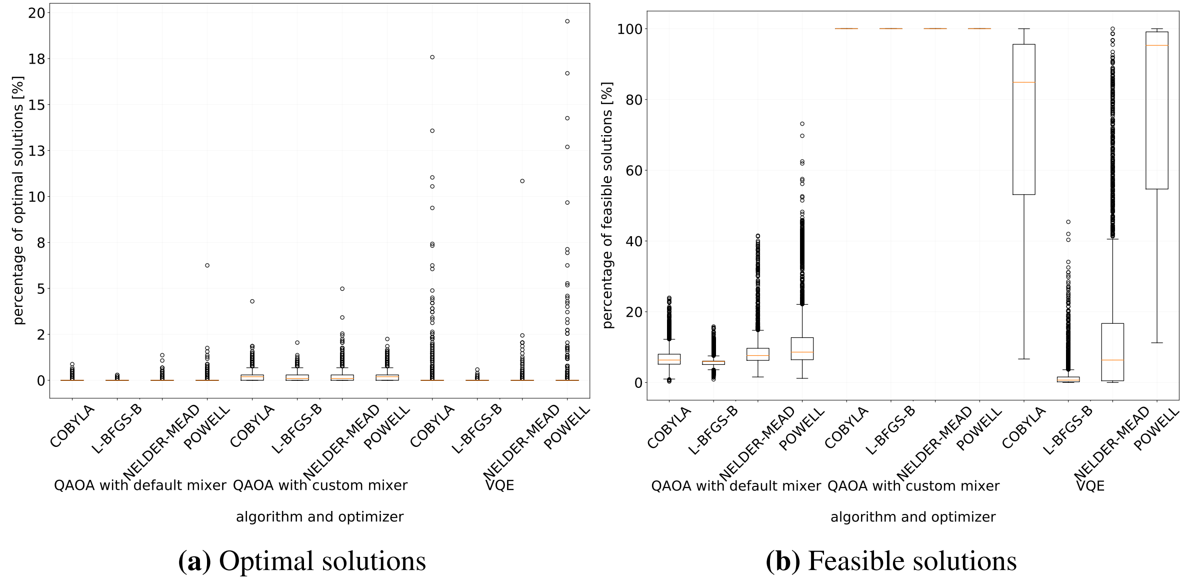
**(b)** Correct solutions

**Figure 7.7:** A result breakdown for all experiments with one-hot encoding: incorrect and correct percentages

A comparison of the percentages of optimal and feasible solutions for different algorithms and one-hot encoding is shown in Fig. 7.8.

QAOA with a custom mixer had the highest median of optimal solutions (although still below 1%). The results obtained for VQE with COBYLA and POWELL have some interesting

outliers, with the percentage of optimal solutions going up to almost 20%.




**(a)** Optimal solutions **(b)** Feasible solutions

**Figure 7.8:** A result breakdown for all experiments with one-hot encoding: optimal and feasible percentages. Note: two outliers for VQE with POWELL with more than 85% of optimal solutions were removed from the plot to increase readability

**As for the percentage of feasible solutions, since QAOA with a custom mixer is limited to a valid subspace, the percentage of feasible solutions is always equal to 100%**. For QAOA with a default mixer, the percentage of feasible solutions is quite similar for all optimizers, and the median oscillates around 10%–15%. For VQE, the percentage of feasible solutions has a large variance, but the median of the two best optimizers, COBYLA and POWELL, is quite high, greater than 80%.

### Best sample results

In this section, we will look at the energy histograms of the most successful QAOA and VQE experiments.

The experiments were selected manually – we first tried to find iterations with the highest optimal result percentage, then, if the first metric proved to be unsatisfactory, the highest correct result percentage, and as a last resort we considered the feasibility percentage. In the

end, both of the selected samples were from experiments based on the POWELL optimizer, with the number of *p/reps* being equal to 2.

Out of 1000 repetitions, the most successful experiment using QAOA with custom mixer produced 17 optimal and 279 correct results (out of 1024 *shots*, as stated at the beginning of this chapter). A histogram of the eigenstates returned by the simulator is shown in Fig. 7.9.

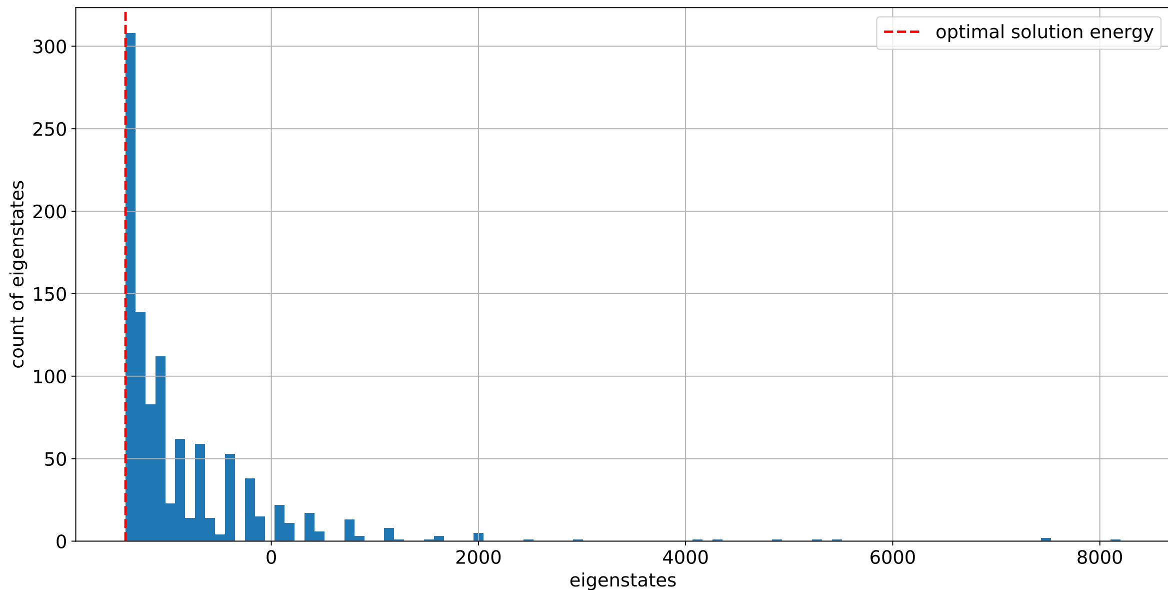



**Figure 7.9:** An eigenstate histogram for the best one-hot QAOA experiment

The most satisfactory experiment, based on VQE, produced 1009 optimal solutions. A histogram of the eigenstates returned from this experiment is shown in Fig. 7.10. As can be clearly seen when comparing it with the histogram of the best QAOA sample (Fig. 7.9), the eigenvalues are much closely concentrated towards the minimum eigenvalue.

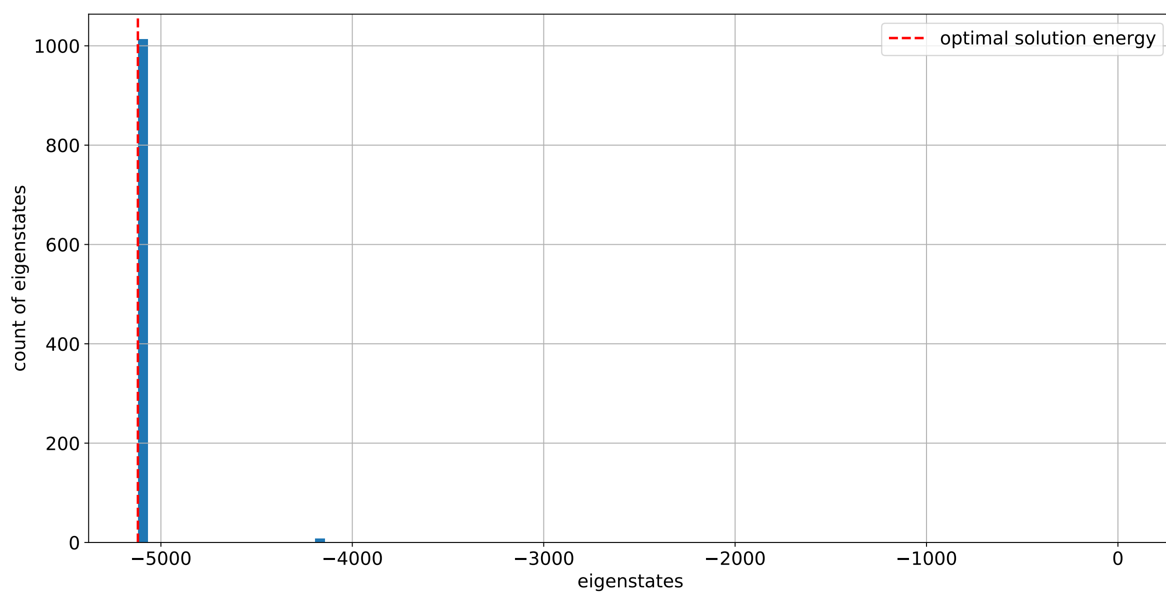



**Figure 7.10:** An eigenstate histogram for the best one-hot VQE experiment

### 7.1.2 Binary encoding

For binary encoding, we only considered a single version of QAOA, as no custom mixers are available for this encoding.

#### QAOA with a default mixer

The eigenvalues obtained by the QAOA algorithm used with binary encoding are presented in Fig. 7.11. The L-BFGS-B optimizer performed the worst. Interestingly, the COBYLA optimizer performed worse than NELDER-MEAD and the POWELL optimizer with the $p$ parameter set to 2 performed the best. It can be seen that increasing the $p$ parameter from 1 to 2 improved the results, especially for NELDER-MEAD and POWELL.

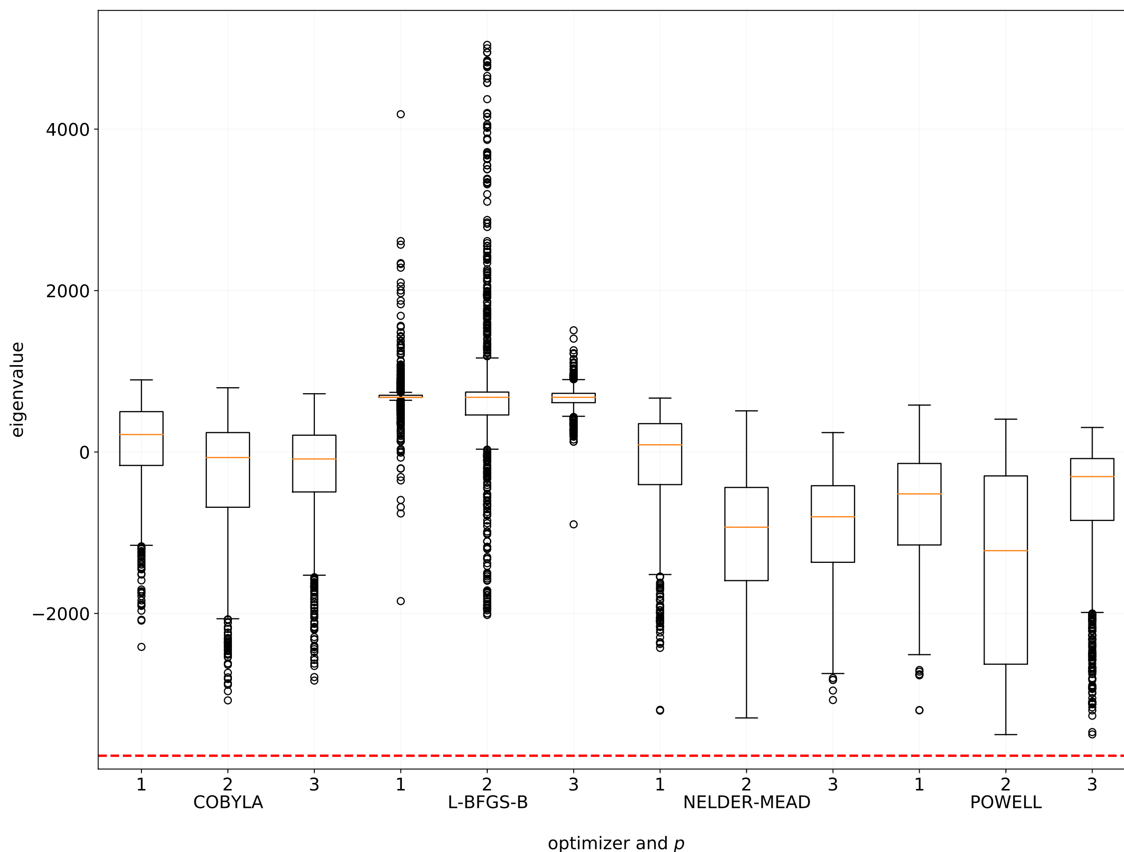
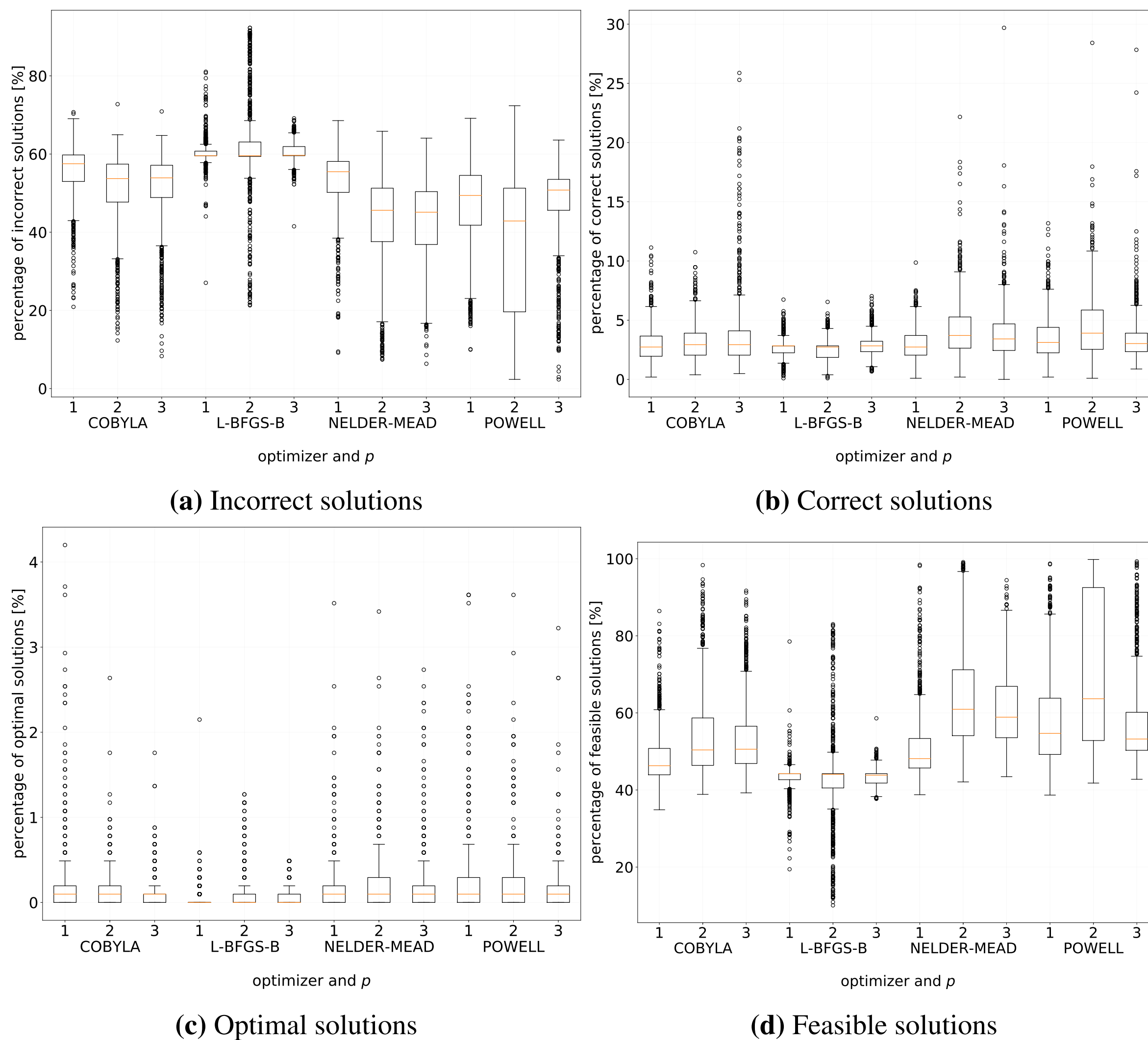



**Figure 7.11:** The eigenvalues for binary encoding with QAOA

An analysis of incorrect, correct, optimal, and feasible solution percentages, as obtained by VQE, is shown in Fig. 7.12. The incorrect solution percentage median was in the range of 40%–65% (POWELL with $p = 2$ obtained the lowest value). The correct solution percentages had a lower variance, as all medians were in the range of 2%–5%. The optimal solution percentages were very low ($< 0.2\%$), with the COBYLA optimizer having the best outlier sample. The feasible solution percentage median was always higher than 40%.




**(a)** Incorrect solutions

**(b)** Correct solutions

**(c)** Optimal solutions

**(d)** Feasible solutions

**Figure 7.12:** A result breakdown for binary encoding with QAOA

### VQE

The eigenvalues obtained by the VQE algorithm are presented in Fig. 7.11. As was the case with QAOA, the L-BFGS-B optimizer performed the worst. However, this time NELDER-MEAD performed poorly and COBYLA performed very well, even comparably to POWELL. Increasing the *reps* parameter did not improve the results, which can be clearly seen for the NELDER-MEAD optimizer.

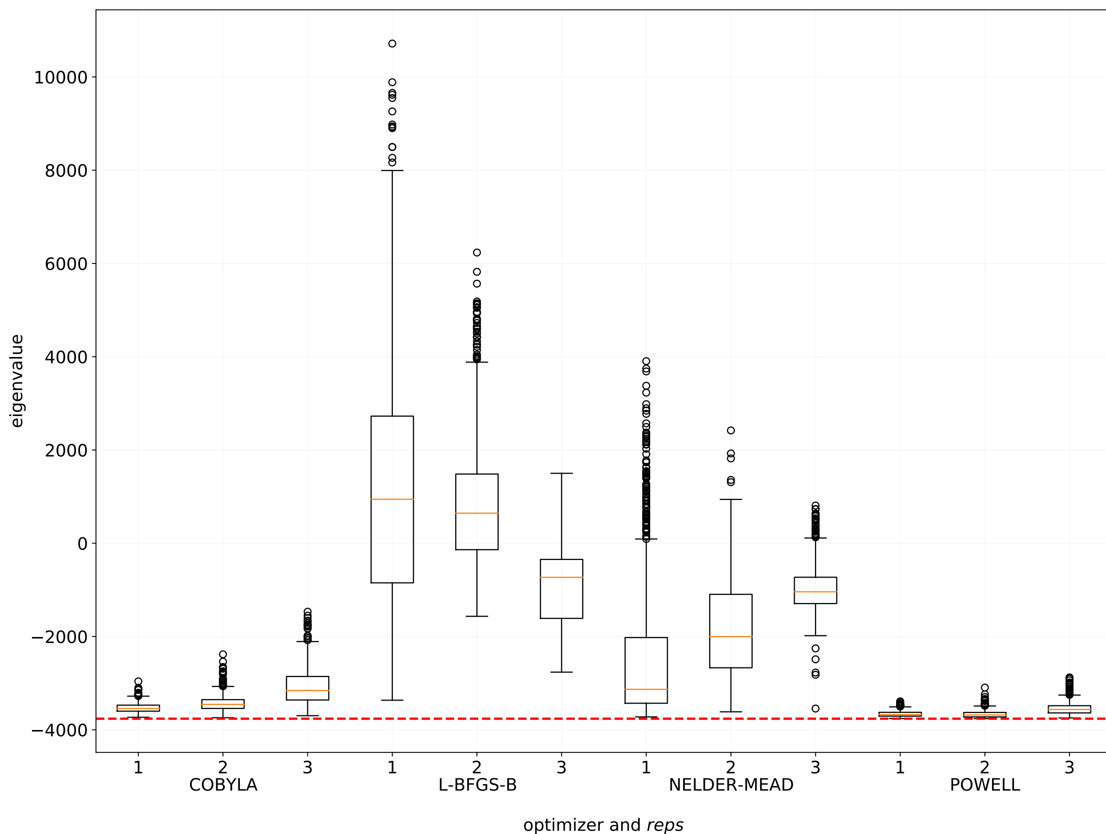
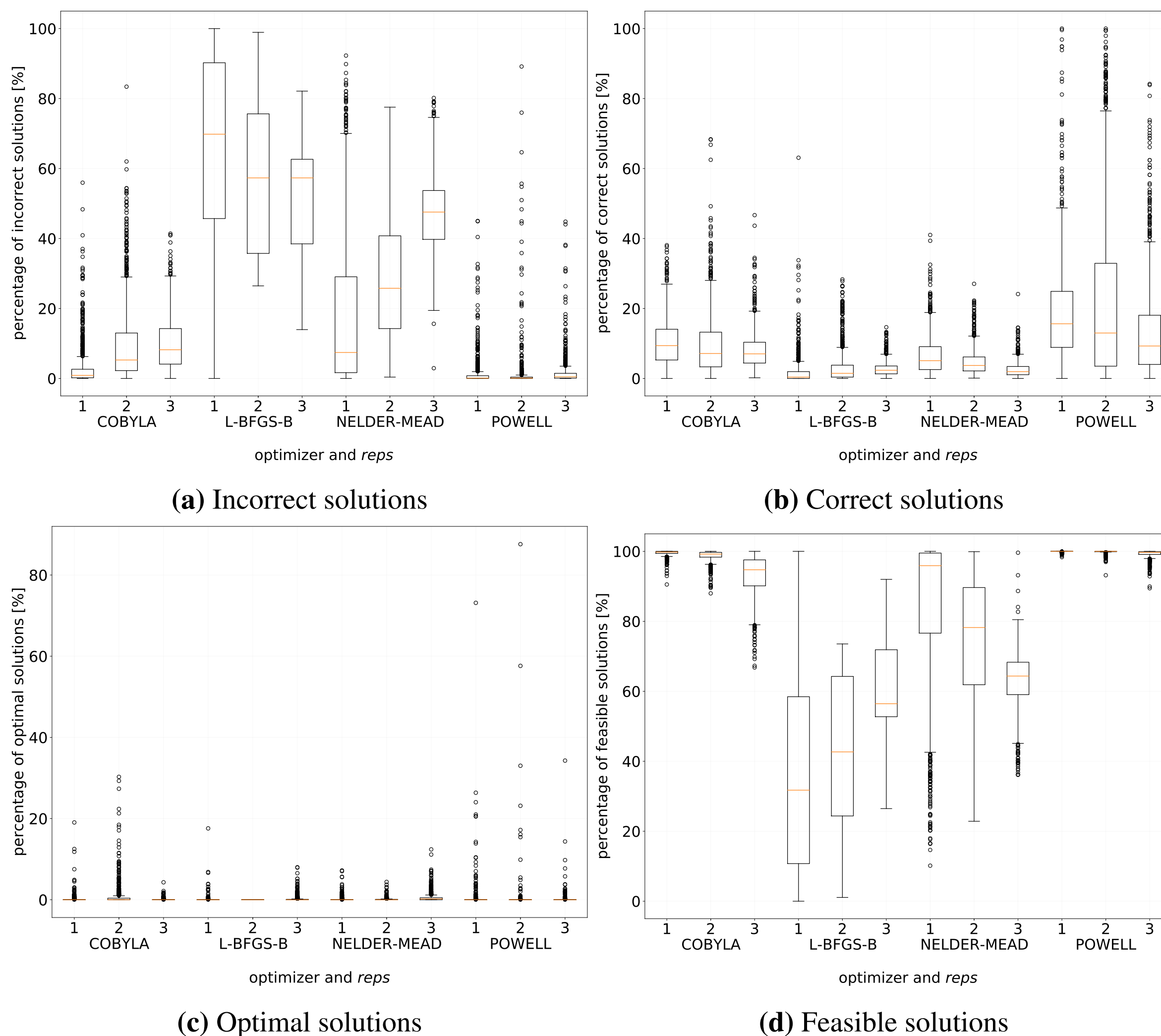



**Figure 7.13:** The eigenvalues for binary encoding with VQE

An analysis of incorrect, correct, optimal, and feasible solution percentages, as obtained by VQE, is shown in Fig. 7.14. It can be seen that the percentages of incorrect and feasible solutions are characterized by a large variance. As can be seen in Fig. 7.14 (a), L-BFGS-B had the highest number of incorrect solutions. NELDER-MEAD performed reasonably well,

although increasing *reps* actually significantly worsened the results. COBYLA had an incorrect result median that was lower than 10%, while POWELL achieved the best results with the median of incorrect solution percentages lower than 2%. An identical conclusion can be drawn based on the percentages of feasible solutions. In terms of the number of correct solutions, all of the optimizers obtained results with medians lower than 20%. POWELL with *reps* set to 1 and 2 obtained samples with optimal solution percentages of about 75% and 85%, respectively.




**(a)** Incorrect solutions

**(b)** Correct solutions

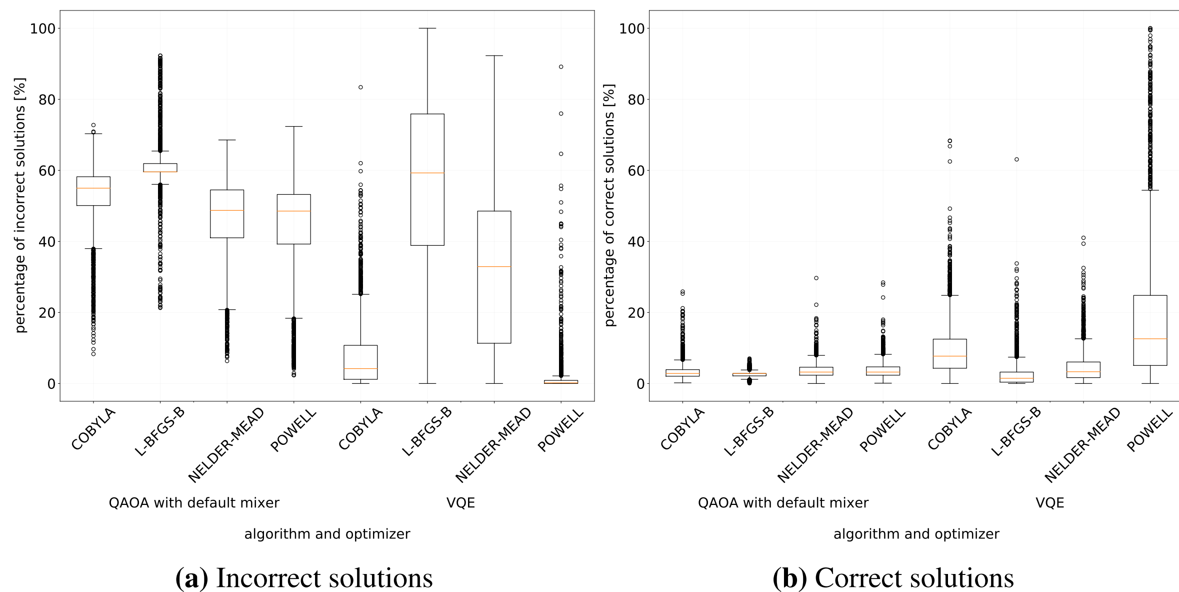
**(c)** Optimal solutions

**(d)** Feasible solutions

**Figure 7.14:** A result breakdown for binary encoding with VQE

### Binary encoding comparison

A comparison of QAOA and VQE is presented in Fig. 7.15 and Fig. 7.16. What can be clearly seen in 7.15 (a) and Fig. 7.16 (b) is that VQE is highly dependent on the chosen optimizer, while QAOA obtained more similar results among optimizers, e.g. for the COBYLA optimizer, the incorrect solution percentage median decreased from around 55% for QAOA to 5% for VQE. Similarly, a significant improvement occurred for the POWELL optimizer. The improvement was not as significant for NELDER-MEAD and no improvement was seen for L-BFGS-B.



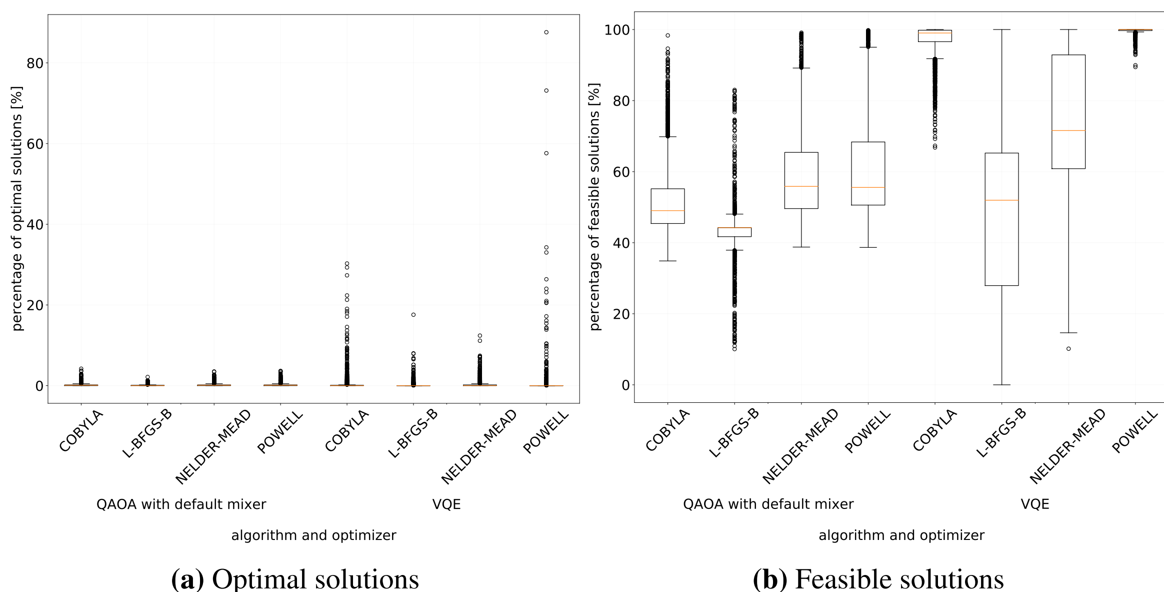

**(a)** Incorrect solutions

**(b)** Correct solutions

**Figure 7.15:** A result breakdown for all experiments with binary encoding: incorrect and correct percentages

The best combination of algorithm and optimizer is definitely VQE with POWELL, which had the lowest number of incorrect solution and the highest number of correct, feasible, and optimal solutions.

To sum up, for binary encoding, it was the VQE algorithm that gave the best results.




**(a)** Optimal solutions **(b)** Feasible solutions

**Figure 7.16:** A result breakdown for all experiments with binary encoding: optimal and feasible percentages

### Best sample results

For both QAOA and VQE, both of the selected best samples were based on the POWELL optimizer with *p*/*reps* parameters set to 2.

The best sample from QAOA had 1022 feasible solutions. A histogram of the eigenstates returned by the simulator is shown in Fig. 7.17.

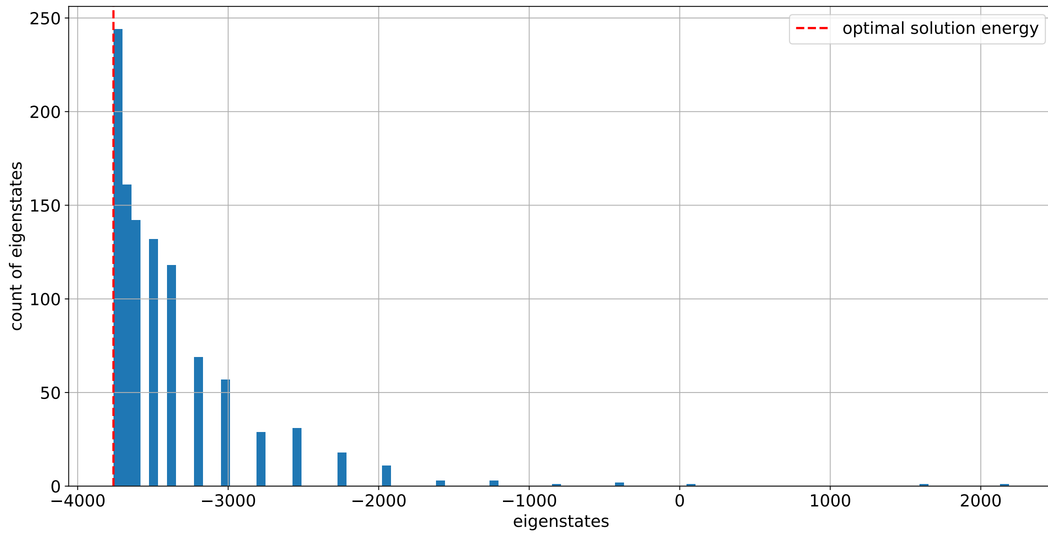



**Figure 7.17:** An eigenstate histogram for the best binary QAOA experiment

The best sample from the VQE algorithm had 897 optimal solutions. A histogram of the eigenstates returned from that sample is shown in Fig. 7.18. By analyzing Fig. 7.17 and Fig. 7.18, we can clearly conclude that the best sample for VQE obtained better results than the best one for QAOA.

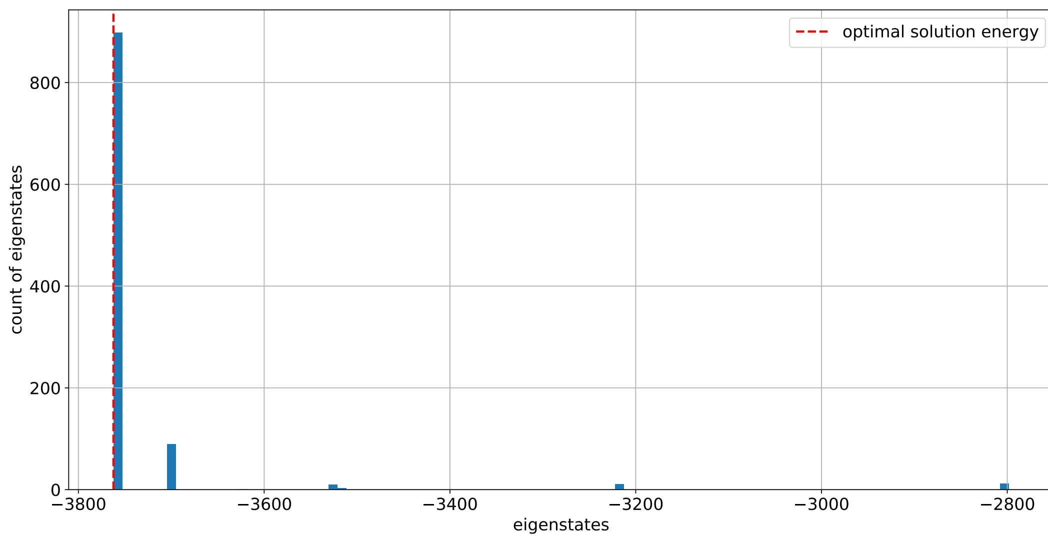



**Figure 7.18:** An eigenstate histogram for the best binary VQE experiment

### 7.1.3 Domain wall encoding

For domain wall encoding, three algorithms were tested: QAOA with a default mixer, QAOA with a custom mixer, and VQE.

#### QAOA with a default mixer

The eigenvalues obtained by QAOA with a default mixer are presented in Fig. 7.19. Among the optimizers used, it was POWELL that obtained energies closest to the minimum eigenvalue. NELDER-MEAD performed quite well also, while the COBYLA and L-BFGS-B routines performed similarly and obtained worse results. Increasing the $p$ parameter to 2 decreased the median, however increasing it to 3 didn't improve the results, but rather worsen them.

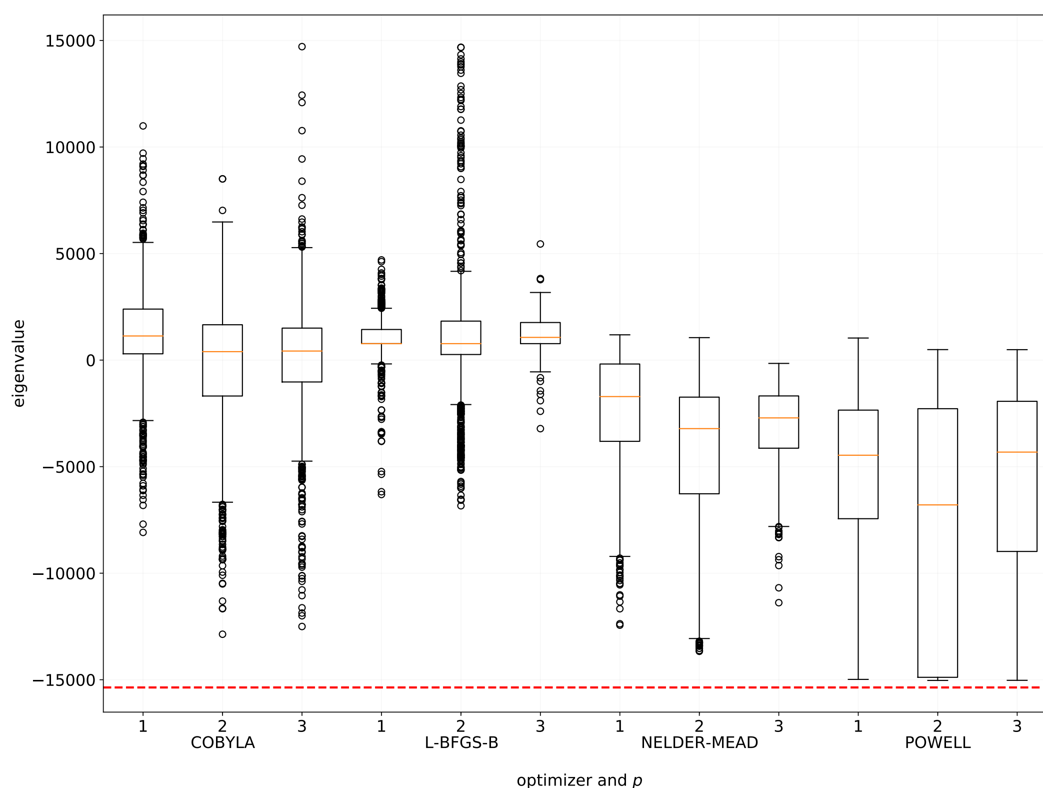
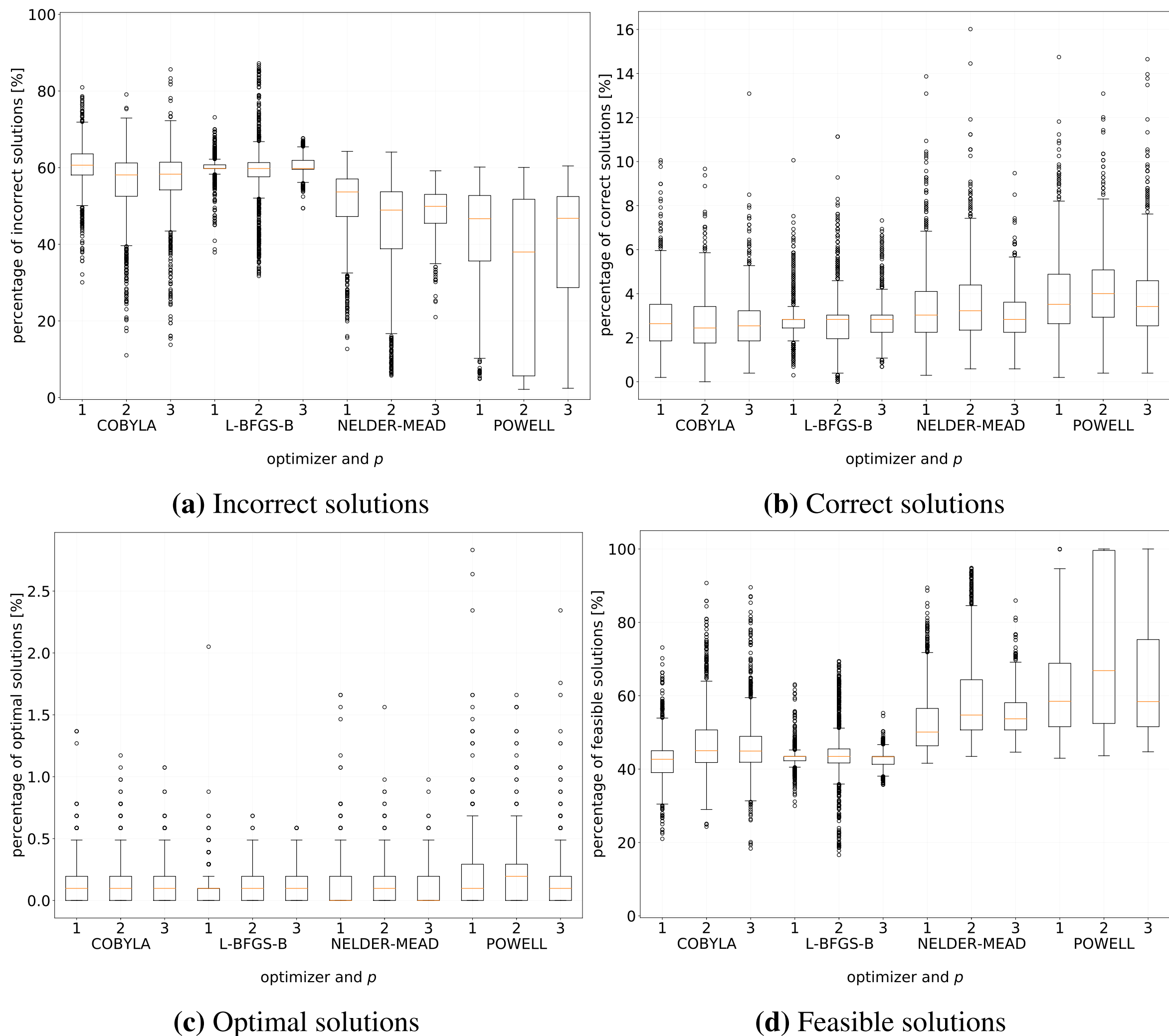



**Figure 7.19:** The eigenvalues for domain wall encoding and QAOA with a default mixer

An analysis of the specific metrics is shown in Fig. 7.20. As expected, the POWELL optimizer obtained the lowest incorrect solution percentage median, at around 35%, for $p = 2$, and the highest correct solution percentage median, at around 4%. NELDER-MEAD returned a sample with the highest percentage of incorrect solutions – 16%. Yet, such results are not satisfying. The optimal solution percentage is similar for all optimizers – around 0.1%. When comparing the feasible solutions, it can again be seen that the POWELL optimizer found more than 50% of them in most cases.




**(a)** Incorrect solutions

**(b)** Correct solutions

**(c)** Optimal solutions

**(d)** Feasible solutions

**Figure 7.20:** A result breakdown for domain wall encoding and QAOA with a default mixer

### QAOA with a custom mixer

The eigenvalues obtained by QAOA with a custom mixer are presented in Fig. 7.21. Compared to the default mixer, it can be seen that in this case the COBYLA optimizer obtained better results, similarly to the NELDER-MEAD routine. Yet it is still the POWELL algorithm that obtained energies closest to the minimum eigenvalue. In contrast to running QAOA with a default mixer, increasing the $p$ parameter to 2 or 3 did not change the output remarkably.

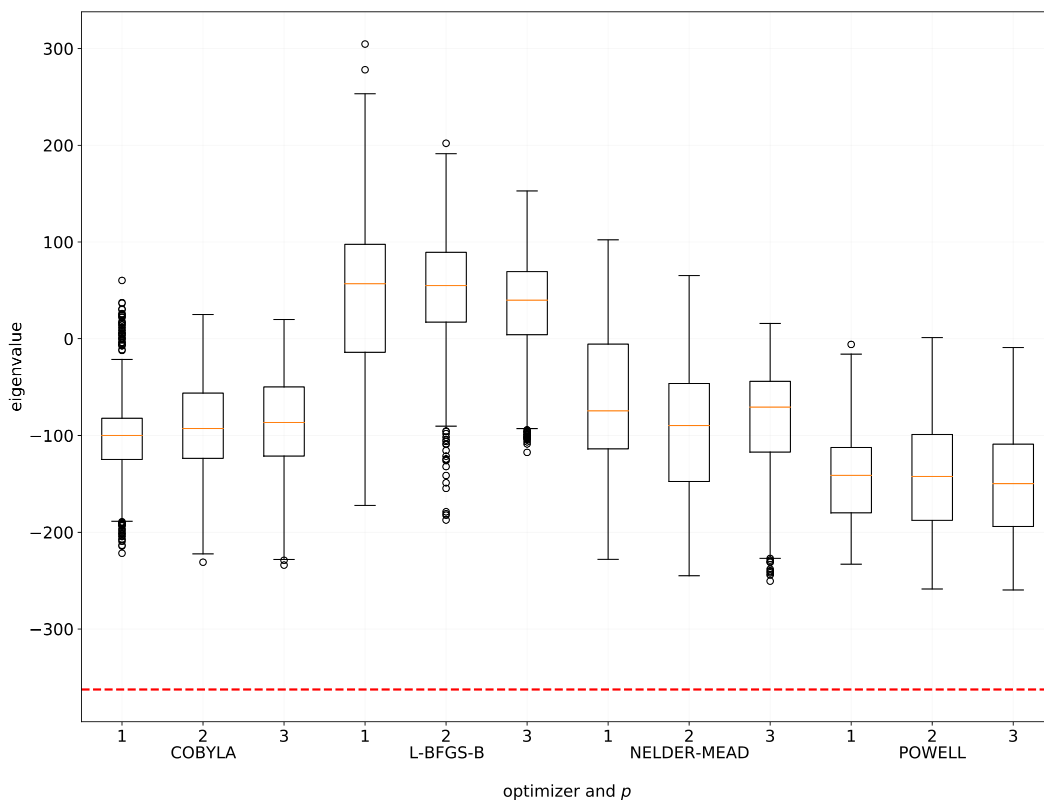
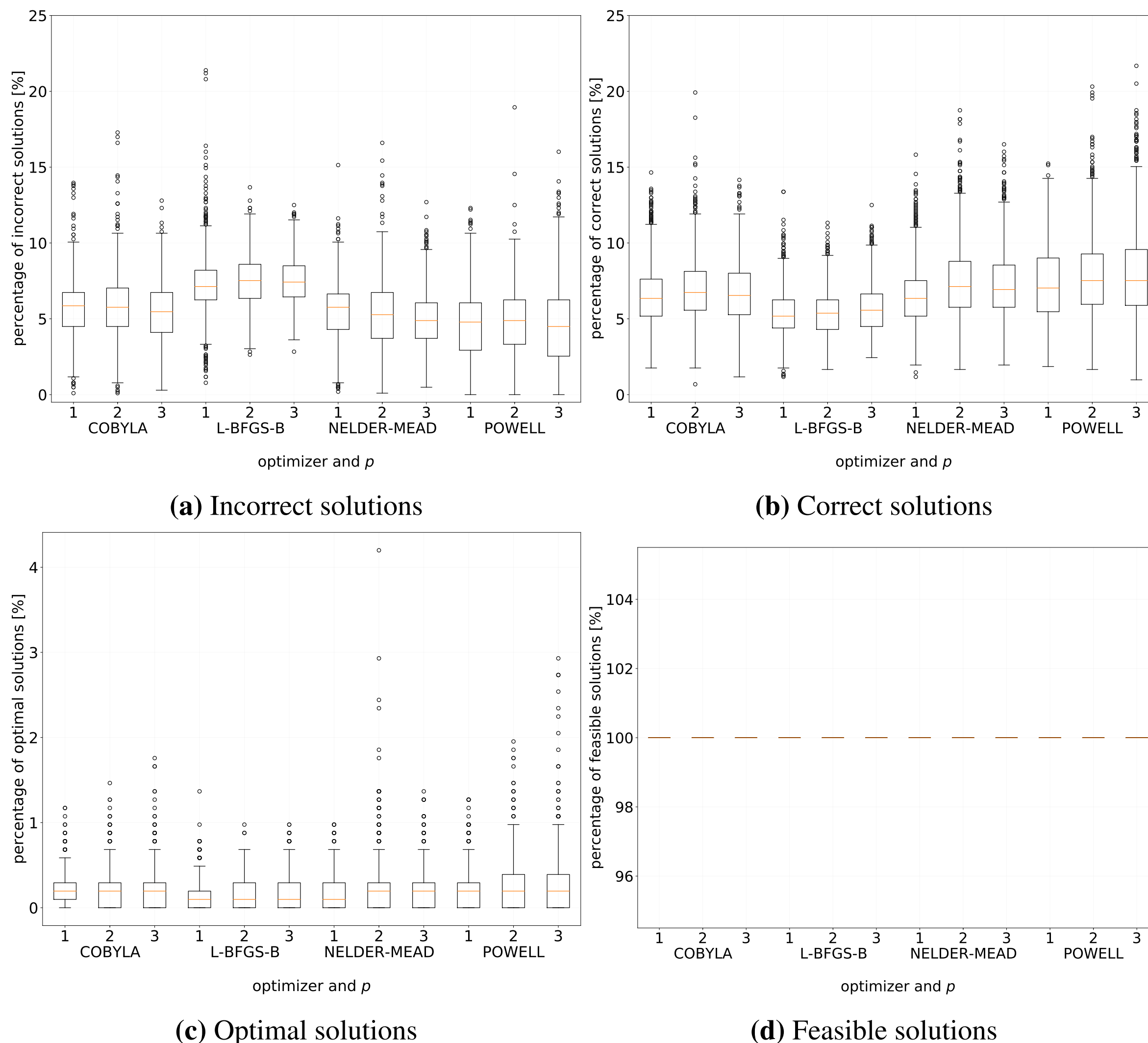



**Figure 7.21:** The eigenvalues for domain wall encoding and QAOA with a custom mixer

An analysis of incorrect, correct, optimal, and feasible solution percentages, as obtained by QAOA with a custom mixer, is shown in Fig. 7.22. In contrast to the previous algorithm, the percentages of incorrect, correct, and optimal solutions are not as dispersed. All subroutines performed rather well, with the incorrect solution percentage median being lower than 10%. The correct solution median increased from around 3% to 6%. It can be observed that

adding a dedicated mixer resulted in a decrease in incorrect solutions, but it did not help with finding optimal solutions. Since a custom mixer was used, the feasible solution percentage was equal to 100%.




**(a)** Incorrect solutions

**(b)** Correct solutions

**(c)** Optimal solutions

**(d)** Feasible solutions

**Figure 7.22:** A result breakdown for domain wall encoding and QAOA with a custom mixer

### VQE

The eigenvalues obtained by VQE are presented in Fig. 7.23. The VQE algorithm obtained the best and yet the most dispersed results simultaneously, as the COBYLA and POWELL optimizers performed much butter in comparison to their usage in QAOA, however L-BFGS-B has worse results. Based on the plot, it can be deducted, that the COBYLA and POWELL subroutines fit perfectly as VQE's classical subroutine.

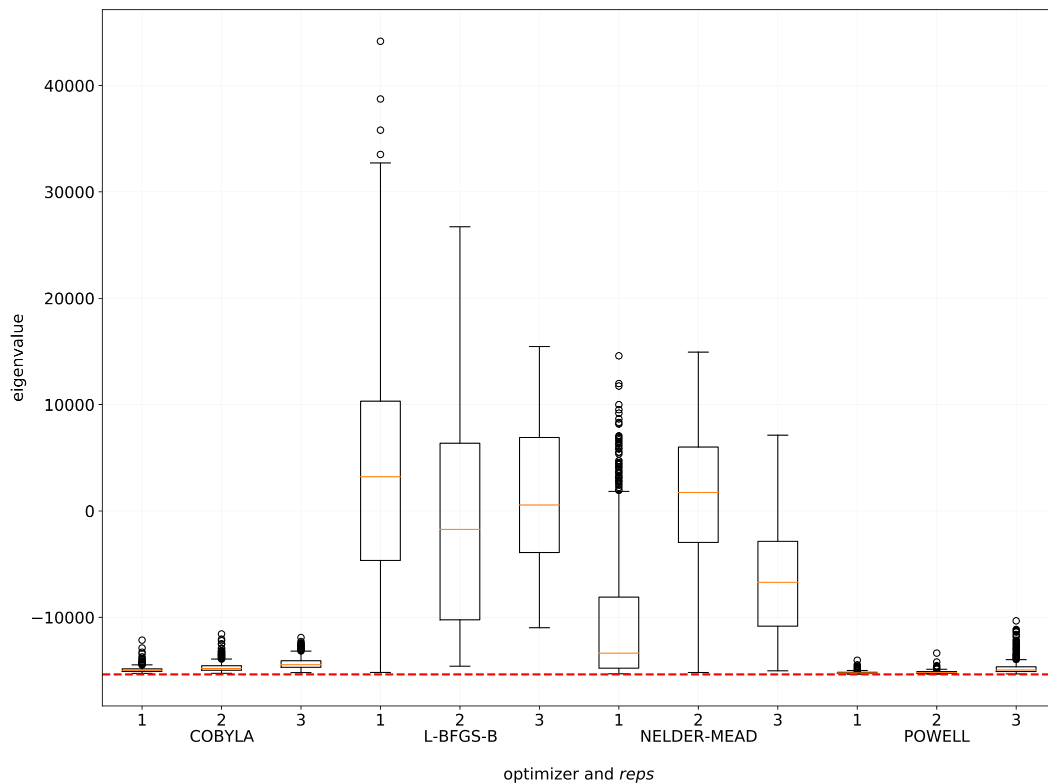
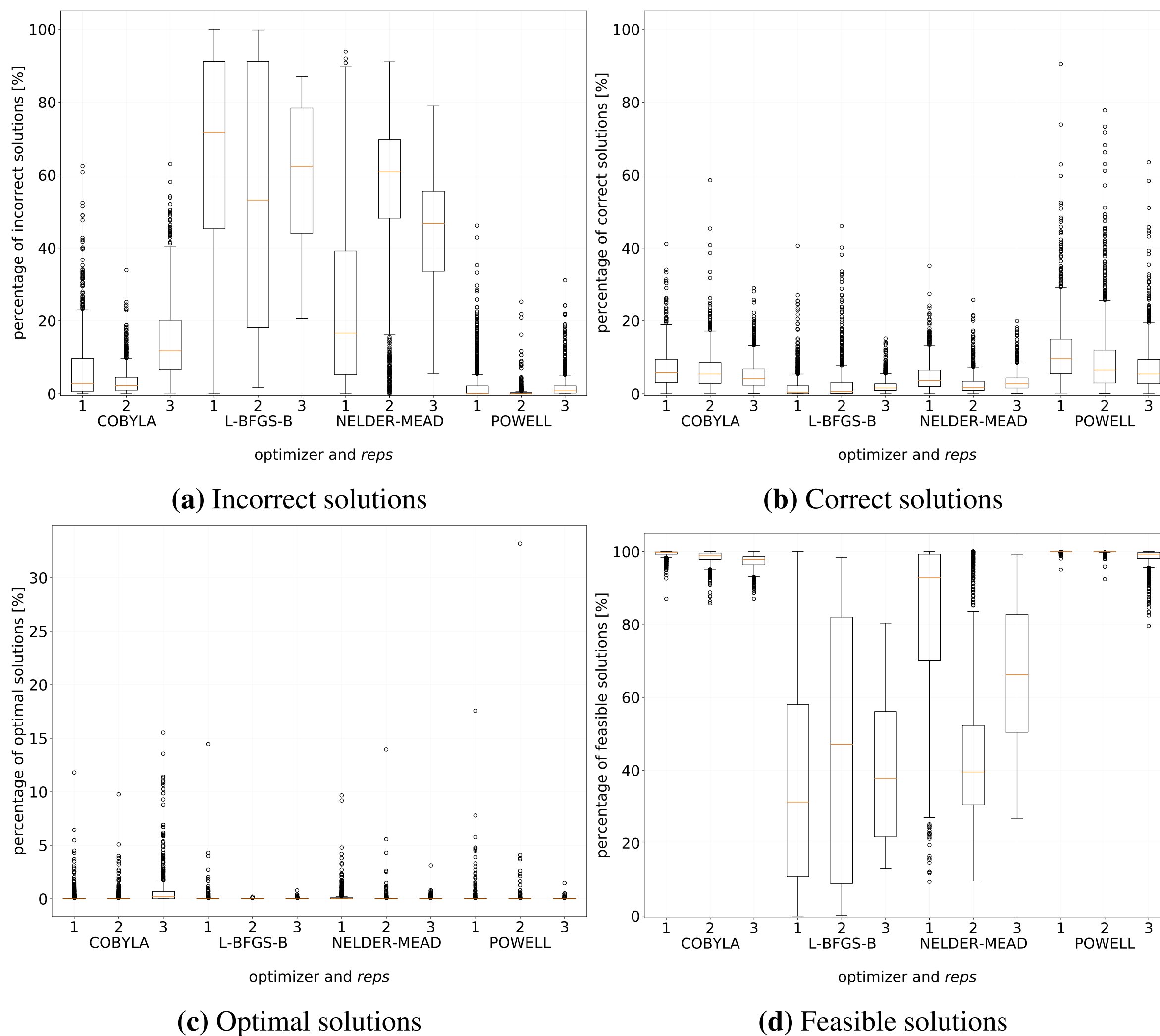



**Figure 7.23:** The eigenvalues for domain wall encoding with VQE

An analysis of incorrect, correct, optimal, and feasible solution percentages, as obtained by VQE, is shown in Fig. 7.24. COBYLA and POWELL had the lowest incorrect solution percentages (at around 0% for POWELL) and the highest feasible solution percentages among all optimizers. For the correct and the optimal solutions, there are no large differences to be seen between the optimizers, however the POWELL optimizer obtained an outlier sample with

90% of correct solutions and a sample with around 33% of optimal solutions, which are the best outputs for all of the domain wall experiments.




**(a)** Incorrect solutions

**(b)** Correct solutions

**(c)** Optimal solutions

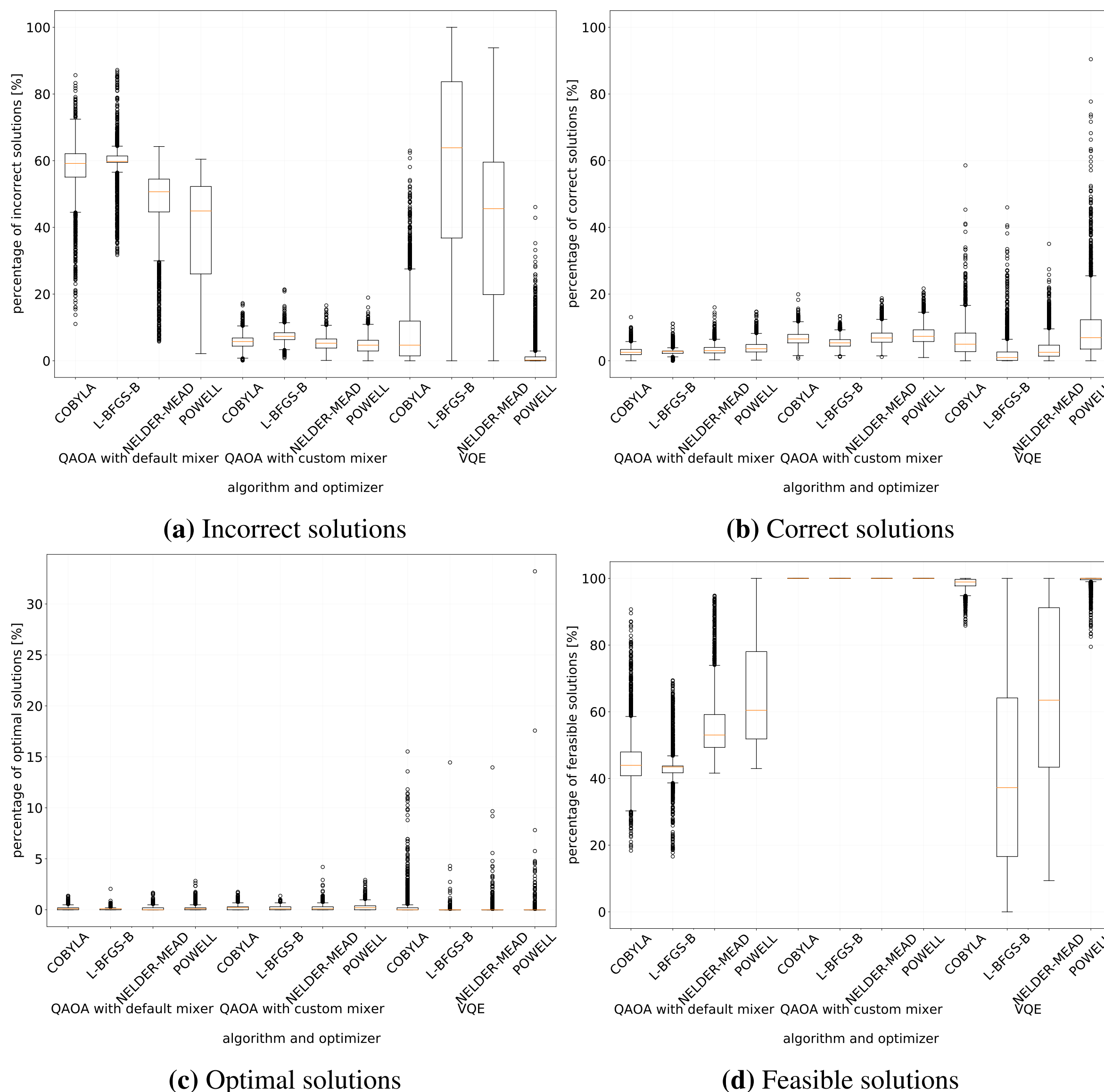
**(d)** Feasible solutions

**Figure 7.24:** A result breakdown for domain wall encoding with VQE

### Domain wall encoding comparison

A comparison of all three algorithms used with domain wall encoding is presented in Fig. 7.25. Based on the incorrect solution percentages, it can be concluded that **applying a custom**

**mixer to the QAOA algorithm improved the results** and that (similarly to the previous encodings) **the VQE algorithm was highly dependent on the optimizer used**. It is again the POWELL optimizer that obtained the best results, with some samples having around 35% optimal solutions,




**(a)** Incorrect solutions

**(b)** Correct solutions

**(c)** Optimal solutions

**(d)** Feasible solutions

**Figure 7.25:** A result breakdown for all experiments with domain wall encoding

### Best sample results

The best sample from the QAOA algorithm had 43 optimal solutions and 83 correct (but not optimal) solutions, and was achieved using a custom mixer, the NELDER-MEAD optimizer, with the $p$ parameter set to 2. A histogram of the eigenstates returned by the simulator is shown in Fig. 7.26.

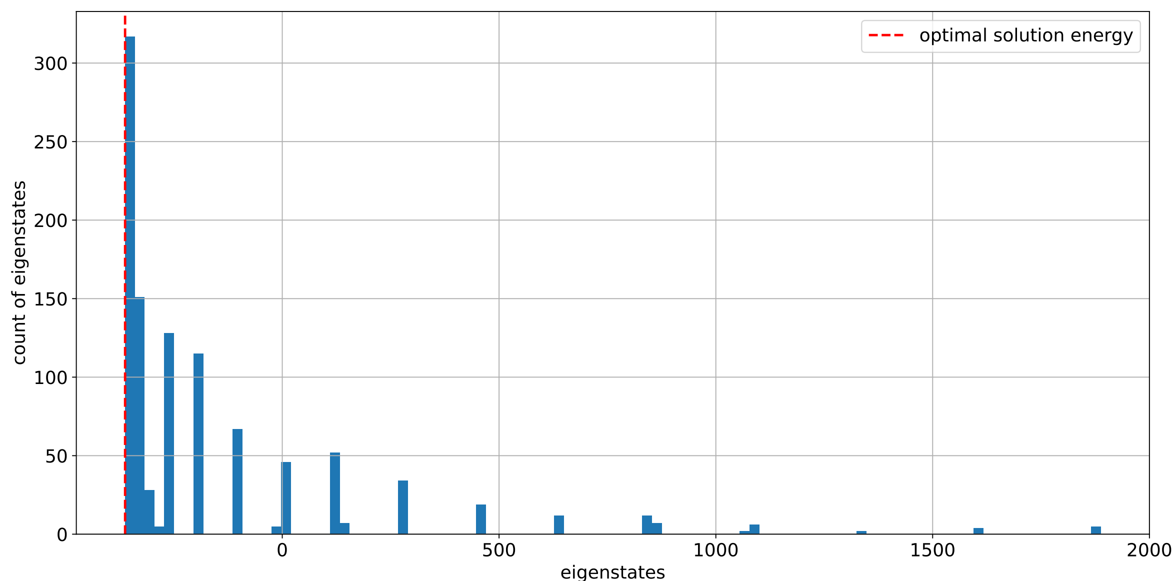



**Figure 7.26:** An eigenstate histogram for the best domain wall QAOA experiment

The best sample from the VQE algorithm had 340 optimal solutions and 86 correct (but not optimal) solutions. It was obtained using the POWELL optimizer and the *reps* parameter set to 2. A histogram of the eigenstates returned from that sample is shown in Fig. 7.27.

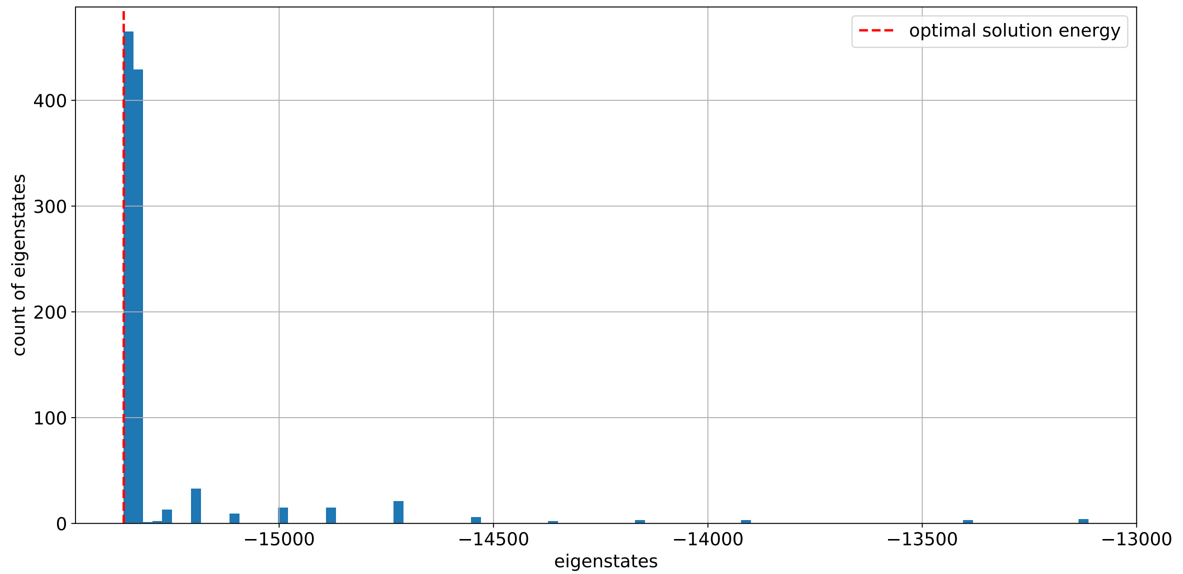



**Figure 7.27:** An eigenstate histogram for the best domain wall VQE experiment

### 7.1.4 Objective function weight selection

As mentioned in Section 6.1.4, we considered two variants of energy ordering: the *naive ordering* and the *feasibility jump ordering*. In the previous sections of this chapter, we presented the results for the latter, as we found that this ordering typically produced better results. This difference was most prominent when considering VQE.

A comparison of the incorrect solution percentages between the two orderings can be seen in Figures 7.28 and 7.29, for one-hot and binary encoding, respectively. **It can be observed that the *feasibility jump ordering* resulted in a significant decrease of incorrect results, especially when considering COBYLA and POWELL.**

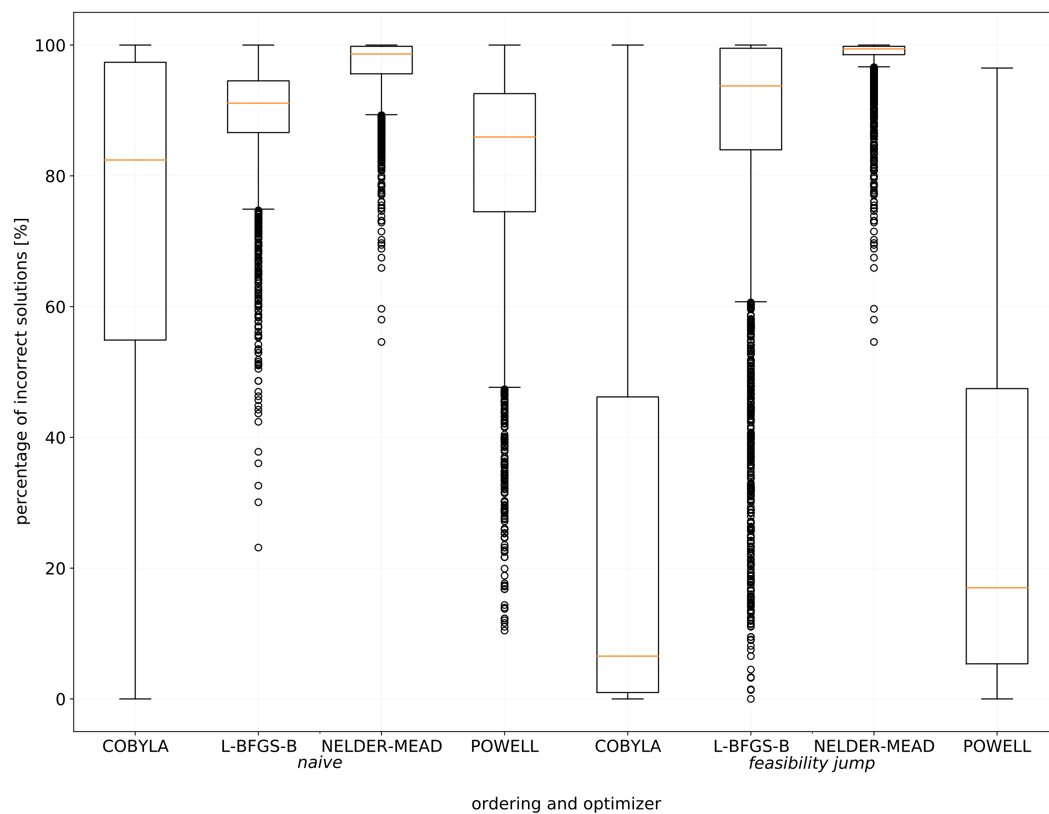



**Figure 7.28:** The difference in incorrect solution percentages for different orderings with one-hot encoding and VQE algorithm

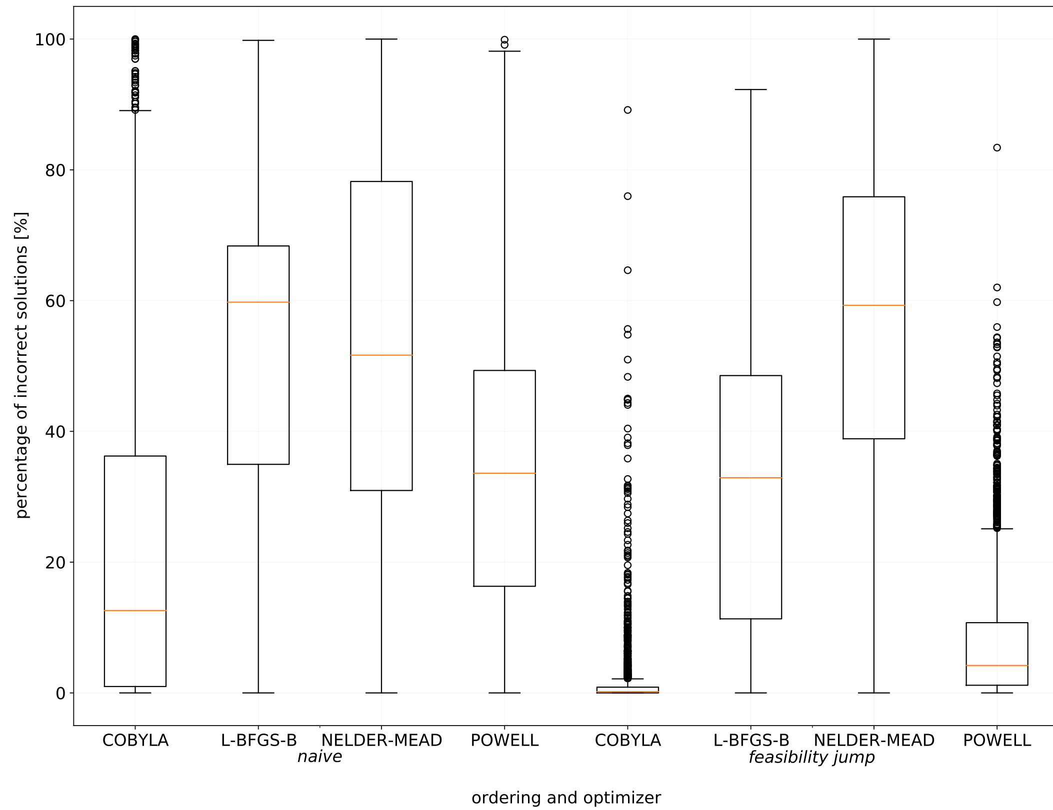



**Figure 7.29:** The difference in incorrect solution percentages for different orderings with binary encoding and VQE algorithm

### 7.1.5 Encoding comparison

In order to make the encoding comparison plots easier to read, the QAOA algorithm is compared in separate plots (Fig. 7.30 and 7.31), regardless of the mixer used, and the VQE algorithm is compared in separate plots (Fig. 7.32 and Fig. 7.33). The results obtained using the same optimizer are labelled with one color. For each optimizer used with a given encoding, there is only a single box plot drawn, as results for all *p*/*reps* parameters are merged together (no distinction is made between the results for different *p*/*reps* parameter values). Sample outliers were also removed to make the plots more readable.

The performance of the QAOA algorithm from various encodings is shown in two figures. Fig. 7.30 presents the incorrect solution percentages and Fig. 7.31 shows the correct solutions percentages. The optimal and feasible percentages were not used here.

When comparing both the incorrect and correct solution percentages for QAOA with the default mixer, it can be concluded that **one-hot encoding – which is the most popular encoding used in literature for solving optimization problems – performed the worst out of the three possible encodings**. The newly proposed domain wall encoding and the well-known, but less popular, binary encoding performed significantly better, e.g. for the COBYLA optimizer, the percentage of correct solutions increased around 10 times when compared to one-hot encoding. The difference between the performance of domain wall and binary encoding is not that crucial.

**Applying a custom mixer to the QAOA algorithm resulted in significantly decreasing the percentage of incorrect solutions and increasing the percentage of correct solutions**. In this variant, the performance of one-hot encoding and domain wall encoding is comparable, so it can be concluded that one-hot encoding should be always used with a custom mixer, as it uses the largest number of qubits out of the three compared encodings, and therefore it has many infeasible states. What is worth noting, is that, **when used with custom mixers, both one-hot and domain wall encoding outperformed binary encoding**.

The incorrect solution percentages and correct solution percentages, as obtained by the VQE algorithm, using different encodings are presented in Fig. 7.32 and Fig. 7.33, respectively. When considering the performance of VQE, it can be again seen that one-hot encoding performed the worst, while domain wall and binary encoding both performed bet-

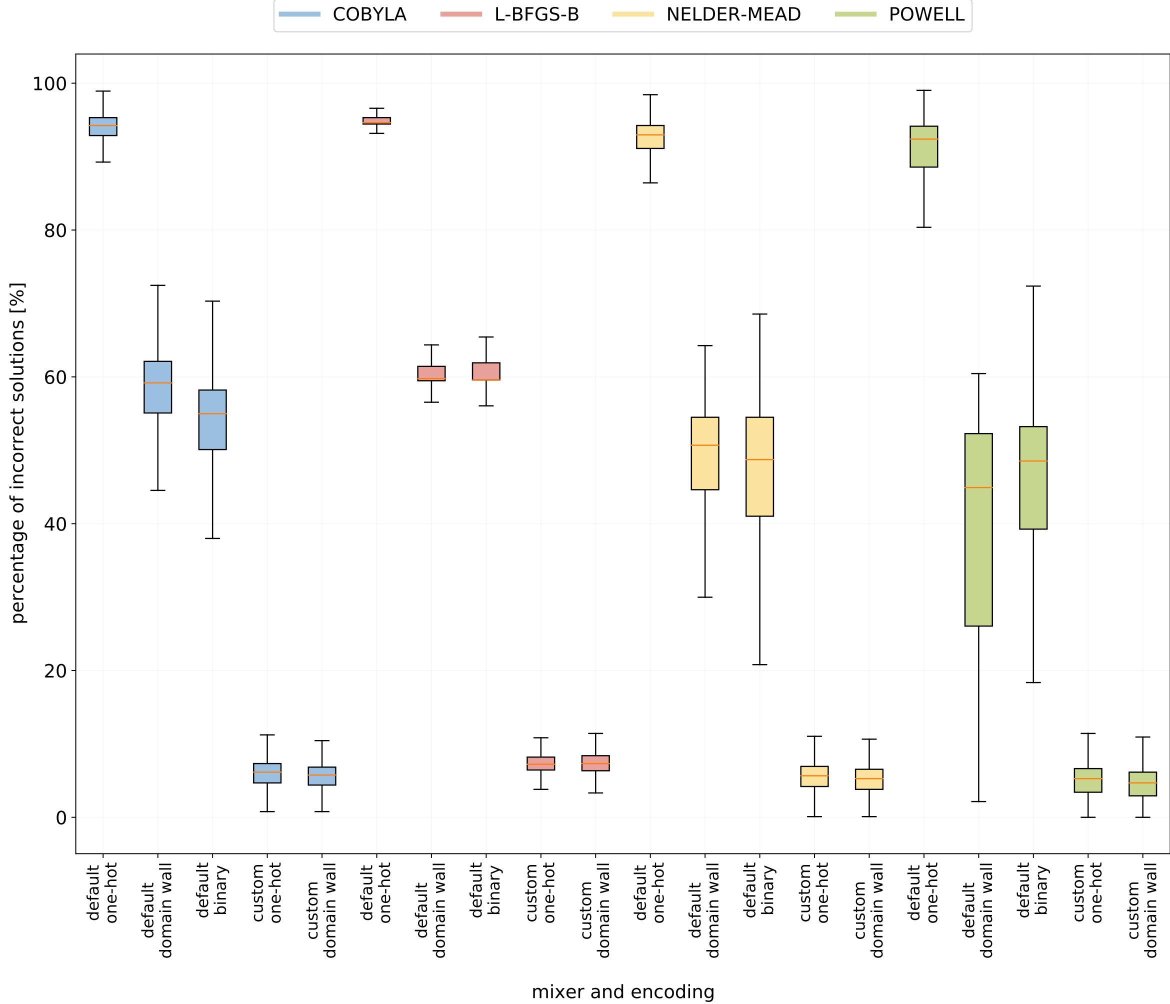



**Figure 7.30:** An incorrect solution percentage comparison between three encodings with QAOA

ter. The biggest difference between the two can be seen for the NELDER-MEAD optimizer, for which the median of incorrect solution percentages decreased from around 95% to around 50% for domain wall encoding and even around 30% for binary encoding.

An incorrect solution percentage analysis can lead to a conclusion that domain wall and binary encoding performed similarly, however when comparing the correct solution percentages it turns out that binary encoding performed better. For each of four optimizers, the

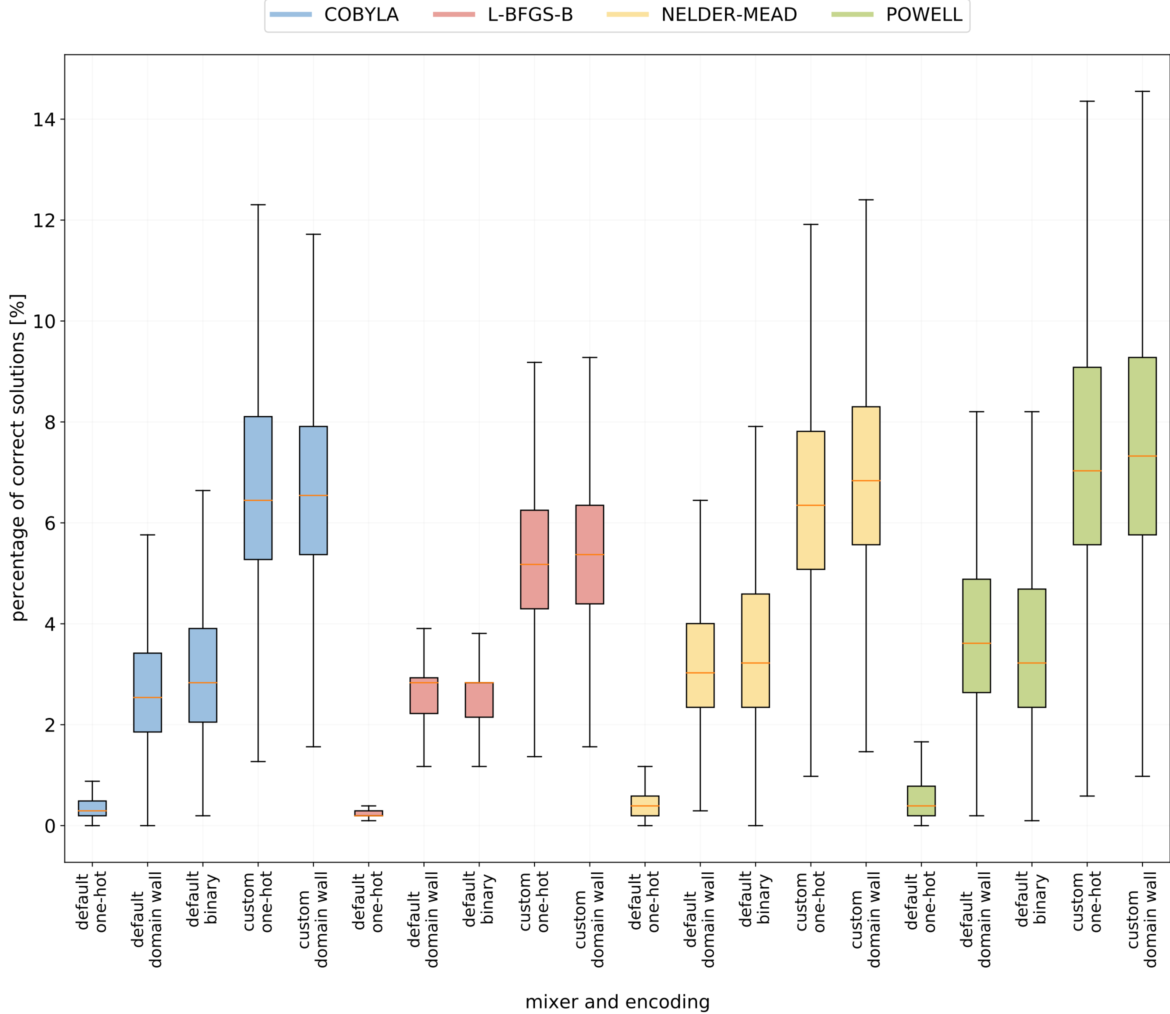



**Figure 7.31:** A correct solution comparison between the three encodings with QAOA

median correct solution percentage for binary encoding was higher than for domain wall encoding. The biggest difference can be seen for the POWELL optimizer, for which the percentage of correct solutions raised from around 7% to 12%.

To sum up the results for the smaller problem instance, for QAOA it is domain wall encoding with a custom mixer and the POWELL optimizer that performed the best – with a median of incorrect solutions at around 5% and a median of correct solutions at around 7%. Notably, one-hot encoding with a custom mixer performed only slightly worse. For VQE,

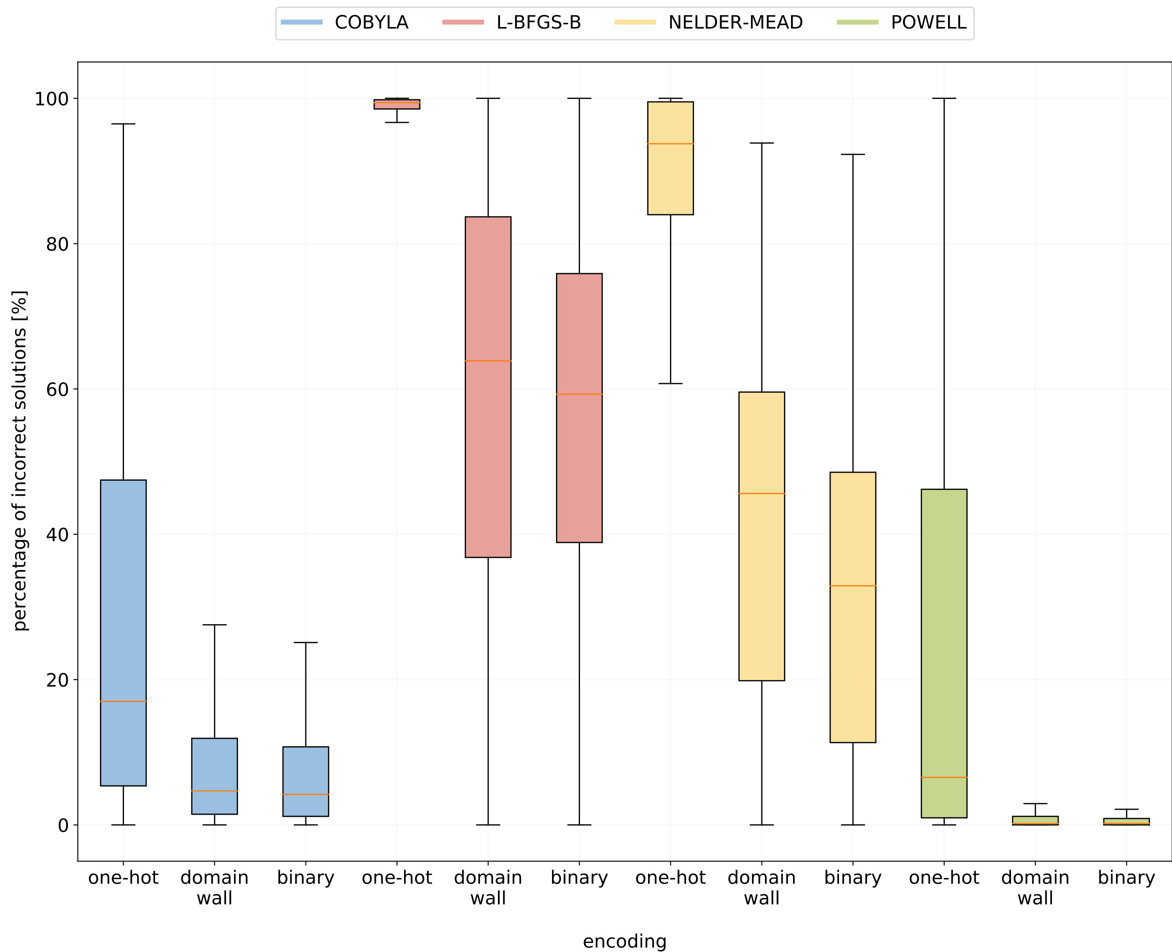



**Figure 7.32:** An incorrect solution comparison between three encodings with the VQE algorithm

it was definitely binary encoding with the POWELL optimizer that performed the best – with the median of incorrect solutions at around 0% and the median of correct solutions at around 12%. Thus, it can be observed that the **VQE algorithm performed better than QAOA** and that **applying a custom mixer for a specific encoding in QAOA yields better results**.

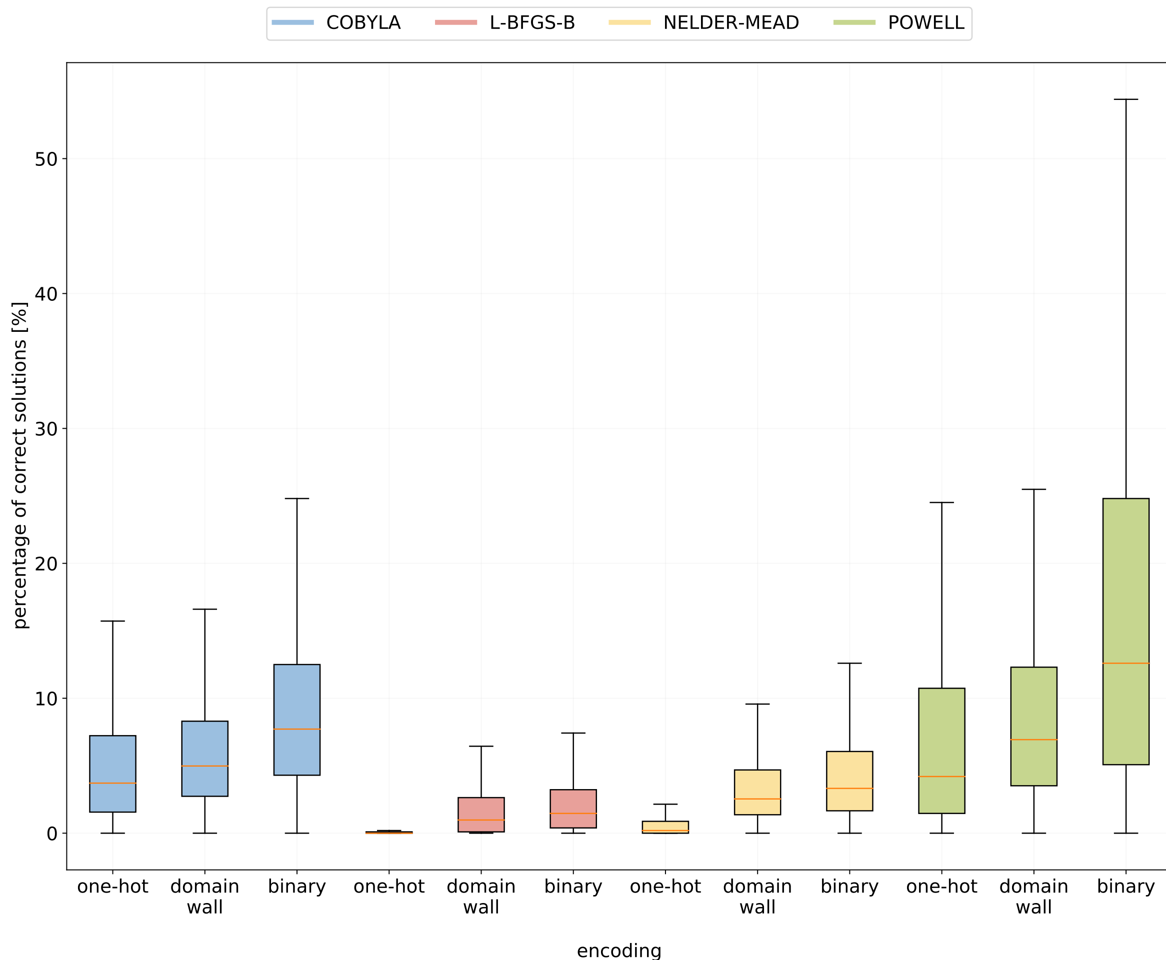



**Figure 7.33:** A correct solution comparison between three encodings with the VQE algorithm

### 7.1.6 Noisy results

Originally, one of the goals of this project was to run some of the most successful experiments on a real quantum computer in order to observe the influence of decoherence on the quality of the results. The problem described in this chapter was even designed with a specific machine in mind – the publically available 15-qubit IBMQ_16_MELBOURNE machine. Unfortunately, this machine was retired by IBM on July 7th 2021[3]. The largest remaining

[3] https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/retired-systems

machine has 5 qubits[4], which is not sufficient for our purposes. Eventually, we decided to run these experiments with simulated noise[5], using noise models created from three real quantum machines: Melbourne[6], Guadalupe[7], and Mumbai[8]. A specification of the machines that were used as mock noise sources is presented in Table 7.1.

| machine | number of qubits | quantum volume | processor type |
|---|---|---|---|
| Melbourne | 15 | 8 | Canary r1.1 |
| Guadalupe | 16 | 32 | Falcon r4P |
| Mumbai | 27 | 128 | Falcon r5.1 |

**Table 7.1:** A comparison of real quantum machine specifications

### One-hot encoding

The experiment chosen to be repeated for one-hot encoding and VQE originally returned close to 99% optimal results (see Section 7.1.1). Regardless of the chosen model, the noisy results are much worse – none of the experiments returned any optimal results and the probability of obtaining a correct result did not exceed 10%. A breakdown of those results can be seen in Table 7.2.

| noise model | optimal | semi-optimal | correct | semi-correct | incorrect | feasible |
|---|---|---|---|---|---|---|
| none | 98.5% | 1.3% | 0.0% | 0.1% | 0.1% | 99.9% |
| Melbourne | 0.0% | 0.0% | 1.8% | 27.7% | 70.5% | 29.5% |
| Guadalupe | 0.0% | 0.2% | 6.8% | 21.5% | 71.5% | 28.5% |

**Table 7.2:** A result breakdown for one-hot-encoded VQE with different noise models

[4]https://quantum-computing.ibm.com/services?services=systems

[5]https://qiskit.org/documentation/tutorials/simulators/2_device_noise_simulation.html

[6]https://github.com/Qiskit/qiskit-terra/blob/main/qiskit/test/mock/backends/melbourne/fake_melbourne.py

[7]https://github.com/Qiskit/qiskit-terra/blob/main/qiskit/test/mock/backends/guadalupe/fake_guadalupe.py

[8]https://github.com/Qiskit/qiskit-terra/blob/main/qiskit/test/mock/backends/mumbai/fake_mumbai.py

The experiment chosen to be repeated for one-hot encoding and QAOA originally returned 27.2% optimal results (see Section 7.1.1). Due to the inclusion of a dedicated mixer, all the returned samples were feasible. After including noise in the simulation, the results worsened noticeably – over 90% of the samples are now incorrect and the feasibility percentage lowered down to around 9%. A breakdown of those results can be seen in Table 7.3.

| **noise model** | **optimal** | **semi-optimal** | **correct** | **semi-correct** | **incorrect** | **feasible** |
|---|---|---|---|---|---|---|
| none | 1.7% | 1.7% | 27.2% | 63.9% | 5.6% | 100.0% |
| Melbourne | 0.0% | 0.4% | 0.7% | 7.4% | 91.5% | 9.9% |
| Guadalupe | 0.2% | 0.3% | 0.3% | 7.6% | 91.6% | 8.7% |

**Table 7.3:** A result breakdown for one-hot-encoded QAOA with different noise models

### Binary encoding

The experiment chosen to be repeated for binary encoding and VQE originally reached the optimal solution with the probability of almost 88% (see Section 7.1.2). The noisy simulation performed much worse, but not as badly as was the case for one-hot encoding. A summary of the results can be found in Table 7.4. In this scenario, we can see a bit of a difference between the three noise models. The Guadalupe model performed the best out of the three. It generated only 11% of optimal results, but the feasible result probability remained very high – it went from 100% to 83%, which is still acceptable.

| **noise model** | **optimal** | **semi-optimal** | **correct** | **semi-correct** | **incorrect** | **feasible** |
|---|---|---|---|---|---|---|
| none | 87.6% | 7.4% | 0.1% | 3.4% | 1.5% | 100.0% |
| Melbourne | 0.2% | 0.9% | 5.9% | 67.0% | 26.1% | 78.7% |
| Guadalupe | 11.0% | 5.5% | 4.6% | 40.0% | 38.9% | 82.7% |
| Mumbai | 2.9% | 4.2% | 3.9% | 28.7% | 60.3% | 81.1% |

**Table 7.4:** A result breakdown for binary-encoded VQE with different noise models

The original experiment chosen for QAOA and binary encoding was slightly less successful. It didn't generate a lot of optimal or correct solutions, but it generated very few incorrect solutions, and almost all the returned samples were feasible. After rerunning this experiment with noise, the results worsened and the probability of obtaining a feasible sample went down

to 56%. There is no noticeable difference between the three noise models. These results can be seen in Table 7.5.

| noise model | optimal | semi-optimal | correct | semi-correct | incorrect | feasible |
|---|---|---|---|---|---|---|
| none | 0.1% | 1.6% | 78.1% | 88.1% | 2.4% | 99.8% |
| Melbourne | 0.5% | 2.2% | 3.3% | 43.4% | 50.6% | 56.0% |
| Guadalupe | 0.3% | 1.9% | 3.4% | 41.1% | 53.3% | 54.8% |
| Mumbai | 0.1% | 2.5% | 2.8% | 42.9% | 51.7% | 53.8% |

**Table 7.5:** A result breakdown for binary-encoded QAOA with different noise models

### Domain wall encoding

The experiment chosen for domain wall and VQE originally returned an optimal answer with a 33% probability (see Section 7.1.3). The noise simulation performed much worse, as experiments with each model returned virtually 0% of optimal solutions. The greatest deterioration can be seen for the optimal and the semi-optimal metric. The noise models performed quite similarly, and their results are summarized in Table 7.6.

| noise model | optimal | semi-optimal | correct | semi-correct | incorrect | feasible |
|---|---|---|---|---|---|---|
| none | 33.2% | 39.0% | 8.4% | 17.9% | 1.5% | 100.0% |
| Melbourne | 0.0% | 1.1% | 5.9% | 67.0% | 26.1% | 10.8% |
| Guadalupe | 0.0% | 1.4% | 5.5% | 75.0% | 18.2% | 8.3% |
| Mumbai | 0.1% | 0.4% | 7.0% | 72.4% | 20.1% | 19.3% |

**Table 7.6:** A result breakdown for domain-wall-encoded VQE with different noise models

The experiment chosen for domain wall and QAOA originally returned an optimal answer with a 4% probability. The greatest deterioration after applying one of the noise models (see Table 7.7) can be seen for the incorrect solutions metric, as its percentage increased from 8% to 50%. Optimal and semi-optimal solutions percentage did decrease that much as for VQE simulation. Thus, the conclusion might be that QAOA algorithm – even using the noise model – returns eigenvalues from a wider range, as can be seen by analyzing the number of optimal and semi-optimal solutions, while VQE rather returns eigenvalues from a narrower range. The ordering of each metric on the eigenvalue axis was presented in Section 6.1.4

in Fig. 6.4. It is also important that the mixer did not work perfectly as it did in the simulator, which is due to the decoherence of a real quantum computer. Therefore, the solution space searched was not limited to only the feasible solutions.

| noise model | optimal | semi-optimal | correct | semi-correct | incorrect | feasible |
|---|---|---|---|---|---|---|
| none | 4.2% | 17.9% | 8.1% | 61.9% | 7.9% | 100.0% |
| Melbourne | 0.2% | 2.3% | 3.0% | 42.7% | 51.8% | 26.0% |
| Guadalupe | 0.2% | 1.7% | 2.5% | 46.8% | 48.8% | 28.1% |
| Mumbai | 0.2% | 1.8% | 2.6% | 45.1% | 50.3% | 27.1% |

**Table 7.7:** A result breakdown for domain-wall-encoded QAOA with different noise models

### Comparison

The noisy experiments presented in this section are not entirely comparable, as they were run with entirely different parameters. However, there is one clear observation that seems fair to make – out of the six experiments that were repeated with noise, all of them originally returned close to 100% feasible results. After adding noise, **the feasibility percentage lowered to 10-30% for both one-hot encoding and domain wall encoding, but it remained quite high for binary encoding: around 80% for QAOA and around 55% for VQE**, regardless of the chosen noise model. This could be explained with the fact that binary encoding has fewer infeasible states – to encode three states we use two qubits, so there is just one infeasible state (see Section 3.3.2), however this is exactly the same as for three-state domain wall encoding (see Section 3.3.3), therefore there has to be another reason. Although these results are very promising, it is important to note that **binary encoding grows in complexity, so for large problems its performance might be less satisfactory**.

Another interesting conclusion can be made in regard to the use of dedicated QAOA mixers (see Section 3.3.4). Such mixers were used for one-hot encoding and domain wall encoding, and the results are presented in Tables 7.3 and 7.7. The two noiseless simulations had a 100% feasibility probability, which is caused by the fact that the mixing operator didn't allow the algorithm to consider any infeasible states. **The noisy simulations did not perform good as well – the feasibility probability dropped considerably for both the encoding schemes**. The overall performance of the noisy QAOA simulations for one-hot and

domain wall encoding could also be affected by the fact that in both of those experiments, the objective Hamiltonian was simplified due to the assumption that only feasible configurations would be considered (see Section 5.2). Since the mixing operator did not act as expected, this resulted in the objective function potentially prioritizing infeasible configurations over feasible, or even correct, states.

## 7.2 Larger problem

In this section, we will provide the results for the two larger problem instances, as described in Section 6.2.2. The larger problem was designed to fit on the publicly available 15-qubit IBMQ_16_MELBOURNE machine, previously mentioned in Sec. 7.1. For one-hot encoding, it was impossible to design a larger problem instance, since the small problem already occupied 13 qubits.

For the small problem, we presented a deeper analysis of each encoding and algorithm (see Section 7.1), but for the larger problem instances, we will only describe a comparison of the incorrect and correct solution percentages between the two encodings. The performance of each specific algorithm within an encoding won't be evaluated.

A comparison of QAOA with both the default and the custom mixer for binary and domain wall encoding is presented in Fig. 7.34 (incorrect solution percentages) and Fig. 7.35 (correct solution percentages).

The first thing to notice from the percentage of incorrect solutions (see Fig. 7.34) is that for QAOA with a default mixer in every case it was binary encoding that performed better, e.g. for L-BFGS-B the median percentage of incorrect solutions dropped by about 10%. A similarly large improvement can be observed for POWELL, but not in terms of the median, but in terms of the lower quartile.

Next, it can be seen that **using a custom mixer for QAOA with domain wall encoding significantly improved the results**. For example, for the POWELL optimizer, the median of incorrect solutions decreased by about 35% when compared to binary encoding. The same relationship can be observed for all the optimizers used.

Another thing to note is that **again it was the POWELL optimizer that performed the best**. However, compared to the smaller problem (see Figure 7.30), the advantage of the

POWELL optimizer is larger – which may also be due to the fact that in the smaller problem, all optimizers had a low percentage of incorrect solutions when using QAOA with a custom mixer.

Compared to the smaller problem, the larger problem produced worse results, as the median of incorrect solution was never even close to 0%, regardless of the chosen optimizer. This is because the problem itself uses a large number of qubits and is therefore more difficult to solve.

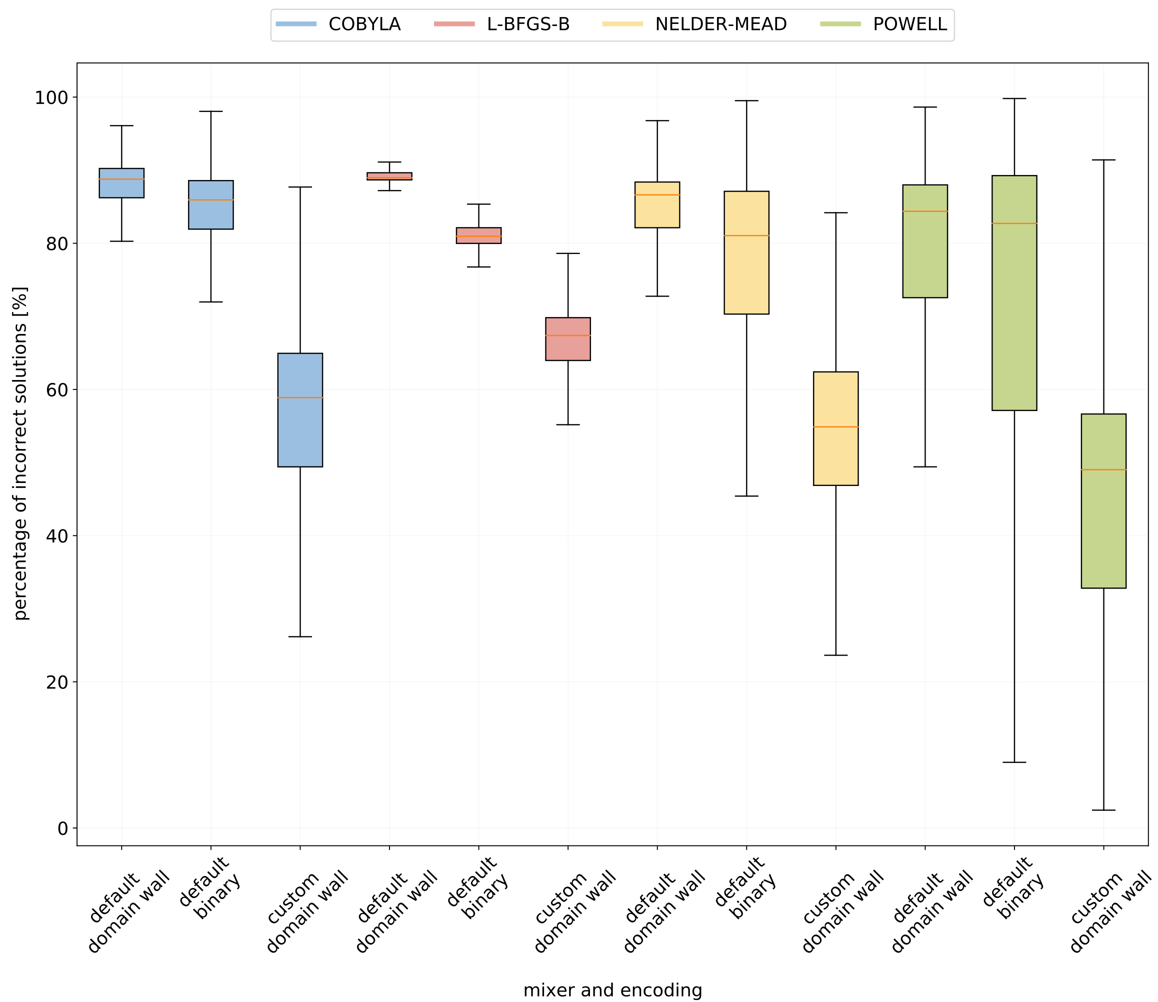



**Figure 7.34:** Larger problem instance: An incorrect solution comparison between two encodings with the QAOA algorithm

The number of correct solutions obtained in the larger problem instance using the QAOA algorithm is unsatisfactory for all encodings and possible mixers. The percentage of correct solutions (see Fig. 7.35) did not exceed 1.5% in all cases, which is much worse than for the smaller problem instance. However, a trend between the different optimizers can be observed, namely that the most correct solutions were obtained for domain wall encoding with a custom mixer.

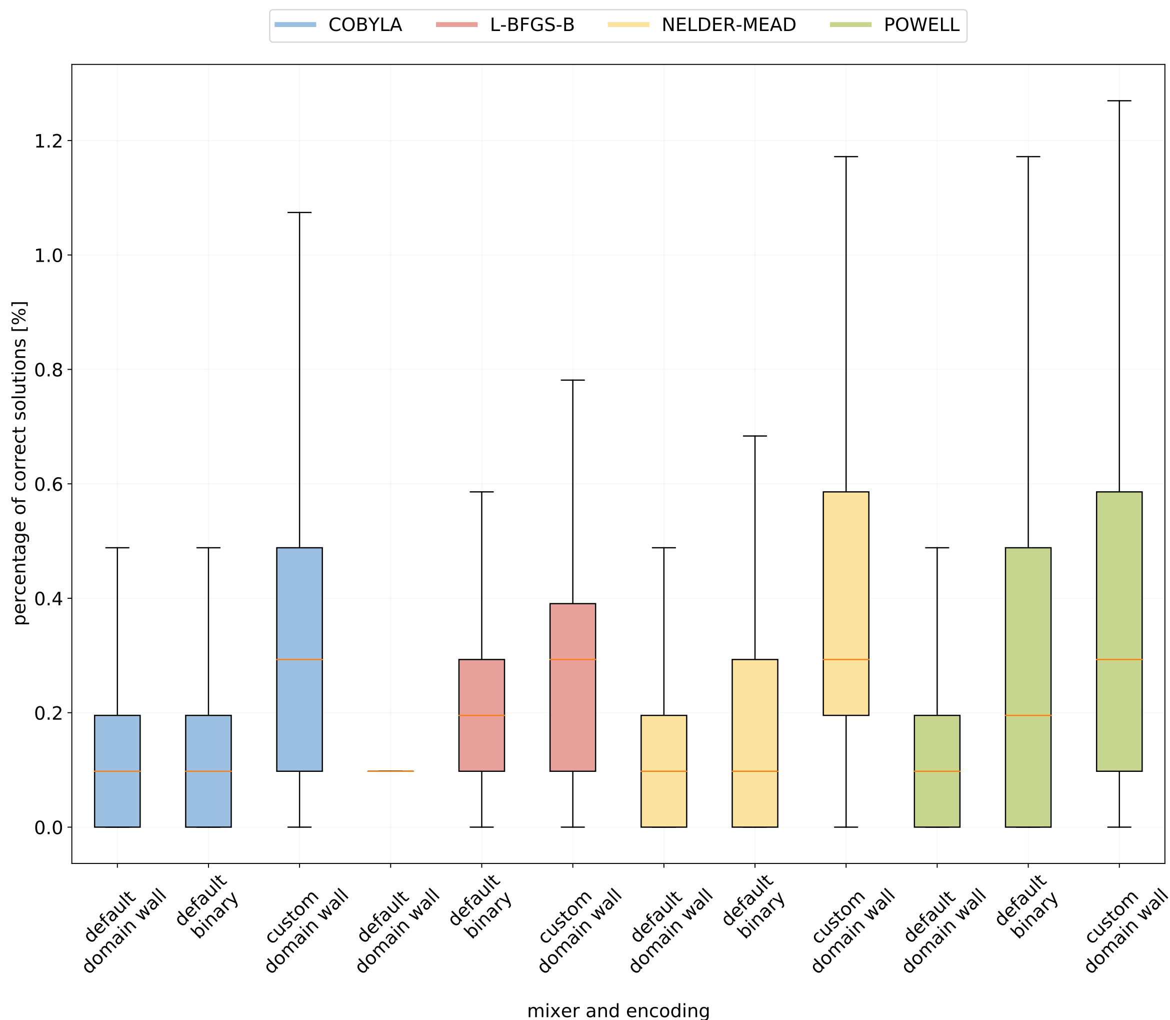



**Figure 7.35:** Larger problem instance: A correct solution comparison between two encodings with QAOA

The results obtained by VQE are presented in Fig. 7.36 and Fig. 7.37. Most importantly, VQE resulted in fewer incorrect solutions than QAOA (regardless of the mixer used) for the COBYLA and POWELL optimizers. This confirms a similar conclusion made in Section 7.1 that **VQE is strongly optimizer-dependent, as the L-BFGS-B and NELDER-MEAD optimizers performed worse than in QAOA, which clearly implies that they are not sufficient for VQE**.

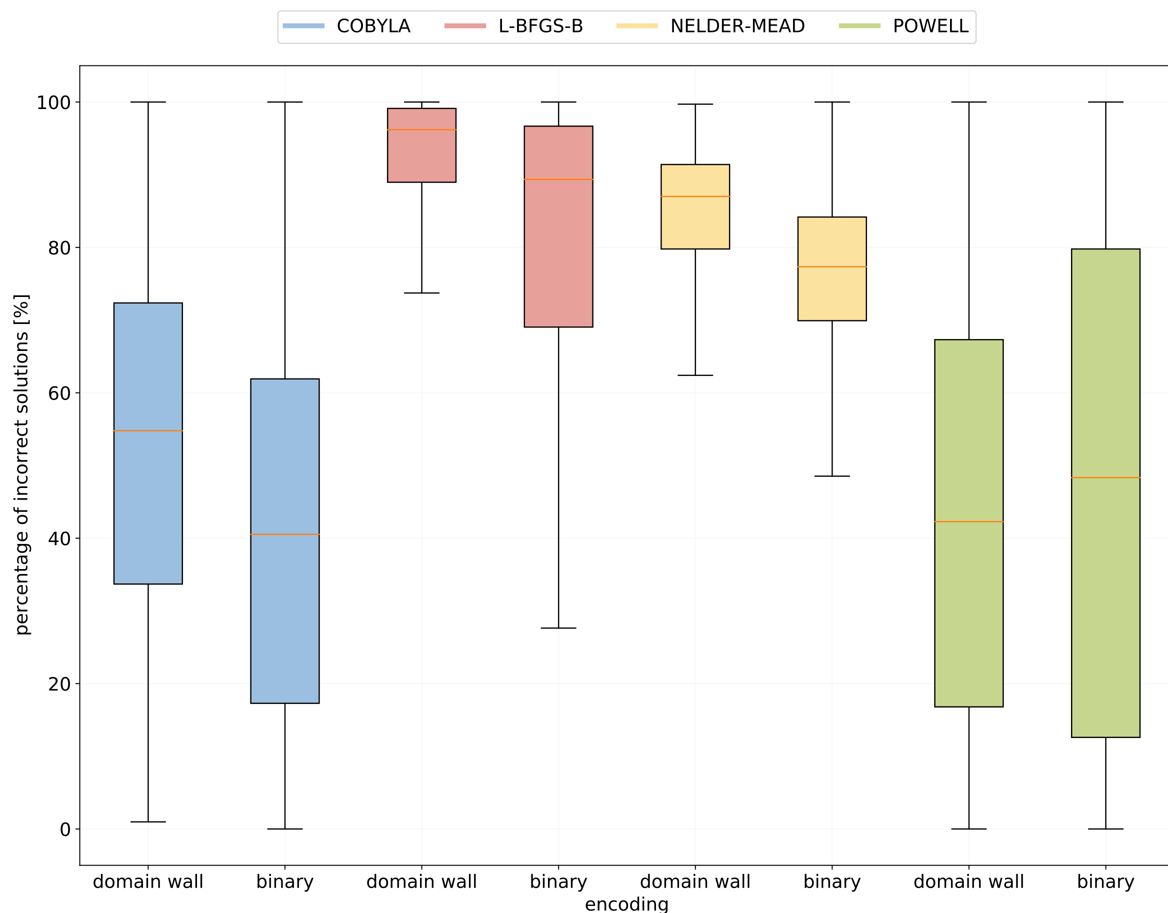



**Figure 7.36:** Larger problem instance: An incorrect solution comparison between two encodings with the VQE algorithm

Comparing the performance of binary and domain wall encoding with the VQE algorithm, we find that for the COBYLA, L-BFGS-B, and NELDER-MEAD optimizers, binary encoding

produced fewer incorrect solutions (see Fig. 7.36), while the opposite is true for the POWELL optimizer. It is noteworthy that for COBYLA and POWELL, the results span across the 0%–100% range.

While the analysis of incorrect solution numbers for VQE did not fully resolve which encoding performed better, it is the number of correct solutions (see Fig. 7.37) that might answer that question. As it turned out, for all optimizers, it is binary encoding that resulted in larger number of correct solutions. The results are obviously not satisfying (as they did not even exceed 1.6%), but the trend is clearly notable.

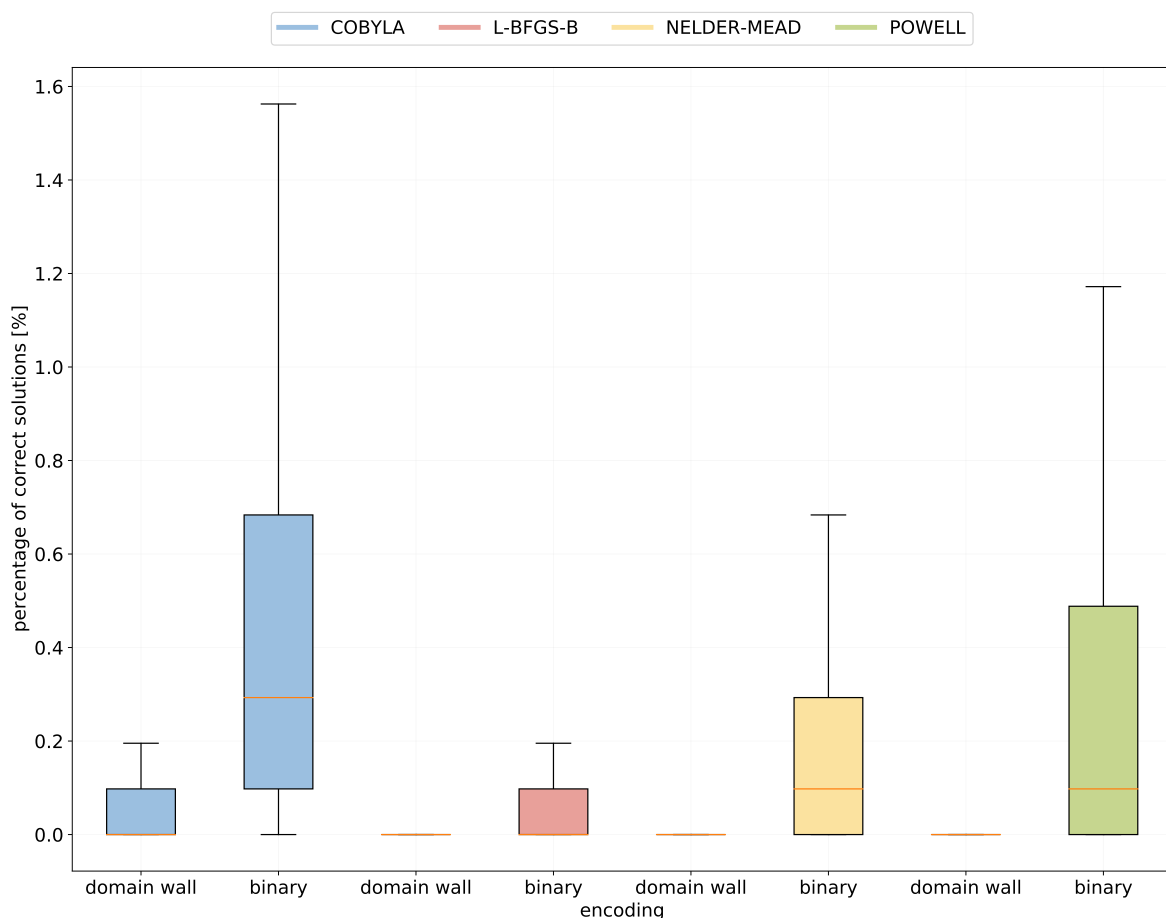



**Figure 7.37:** Larger problem instance: A correct solution comparison between two encodings with the VQE algorithm

What is worth noticing is that in this case it was the COBYLA optimizer that had the highest number of correct solutions, not the POWELL optimizer, which usually performed best. This agrees with the number of incorrect solutions, where COBYLA and POWELL had nearly the same median percentage of incorrect solutions (around 40%).

To conclude, for the larger problem instance, the VQE algorithm with binary encoding and COBYLA optimizer obtained the highest number of correct solutions (median of 0.3%) and at the same time the lowest number of incorrect ones (median of 40%). Comparing the percentage of correct solutions, it can be said that QAOA with custom mixer for domain wall encoding obtained a similar number of correct solutions, however VQE performed better in terms of the number of incorrect solutions. Nevertheless, this conclusion could indicate that **QAOA may have more potential in larger, more real-world problems**.

## Summary

This section describes the results obtained on the quantum simulator for smaller and larger problem instances. For the smaller problem instance, an in-depth analysis of each encoding and the algorithms available for it (QAOA with default or custom mixer and VQE) is presented. For the larger problem, only a comparison of encoding-specific algorithms is presented.

The most important conclusion is that the alternatives to the most commonly used one-hot encoding, perform quite well while using fewer qubits, which will have a huge impact on the results when solving problems on real quantum devices.

# Chapter 8

# Conclusions and future works

In today's quantum computing, we are limited to noisy near-term quantum devices with a relatively small number of qubits. Although the true potential of quantum computing has not yet been reached, many researchers focus on trying to improve the existing algorithms and methods, in order to prepare for the hardware of the future.

## 8.1 Achieved goals and observations

The goal of this thesis was to research various methods of improving the existing solutions of workflow scheduling. Additionally, we also wanted to compare different problem representations with the hope that they would allow us to solve larger problems.

### 8.1.1 General findings

While working on this thesis and designing our experiments, we made many general observations. They are as follows:

- Using encoding methods alternative to one-hot encoding allows for encoding larger problems.
- The inclusion of slack variables severely complicates the objective function and leads to the introduction of *semi-correct* and *semi-optimal* states. Those states are very hard

for the optimizer to handle, because they are penalized in the same way as states that break the time constraint. Additionally, the number of possible configurations increases majorly when slack variables are introduced, meaning for each valid state there are several versions of it that are penalized. With such numerous configurations, it becomes harder and harder to get to the optimal solution.

- Denser encodings, such as binary encoding, limit the number of infeasible configurations, which in some cases can lead to better results. In some specific situations, binary encoding can even limit the number of infeasible configurations to zero.

- Selecting good weights for the objective function and constraints is crucial. Introducing energy jumps between feasible and infeasible states leads to an improvement in the results.

- The inclusion of QAOA mixers allows the algorithm to disregard infeasible configurations, which in turn leads to an improvement of the simulation results. However, in our experiments, the noisy simulation results were not as satisfactory, which might suggest that the additional noise generated by the inclusion of a custom mixer might counteract any real benefits of the mixer.

### 8.1.2 Problem-specific findings

In Chapter 7, we presented our results in detail and discussed their importance. Those observations can be summarized as follows:

- For the problems solved in the thesis, it was the VQE algorithm that performed better, but for a bigger problem instance the difference between VQE and QAOA is not as significant. This may suggest that QAOA has more potential for larger problem instances.

- Increasing the $p$ parameter used by QAOA usually improved the results only when it was increased to 2. When increased to 3, the results deteriorated. For VQE, any increase in the *reps* parameter resulted in worse results. This trend was observed for both smaller and larger instances of the problem. Although it is not presented in this

thesis, we also made attempts to increase these parameters further, but we were not able to improve our results.

- The VQE algorithm is highly dependent on the chosen classical subroutine, and the best results were obtained when the COBYLA or POWELL optimizers were used. For QAOA there are no such discrepancies between optimizers.
- The use of custom mixers allowed us to reduce the complexity of the objective function (see Section 5.2). This performed well as a perfect simulation, however with the inclusion of noise, there were no real benefits from the mixer.

## 8.2 Future work

In this work, we focused on exploring several aspects of quantum optimization, but there are still many other approaches that could be tested.

Firstly, we believe that the inclusion of slack variables is worth reexamining, since those variables increase the complexity of the circuit and the number of qubits. Other methods of encoding inequality constrains should be considered, such as the Alternating Direction Method of Multipliers, based on augmented Lagrangians [24].

Another interesting direction for further research would be in finding better ways of determining the weights used for the objective and the constraints. As shown in Chapter 7, modifying those weights influences the results visibly, therefore some specific paradigm for choosing *good weights* should be established.

Thirdly, since we did not observe any noticeable improvement in the results of QAOA after increasing the $p$ parameter, we believe that this method should be explored further. In this thesis, we only included the results for low values of $p$, however we initially also conducted experiments for much larger values and those experiments did not return promising results. Other researchers report noticeable improvements for growing values of $p$ [26], which leads us to believe that our methodology might be worth reevaluating. A good starting point might be to reconsider the metric used for selecting the best angles from a number of runs. Some ideas for different metrics were discussed in Section 6.1.6. In this thesis, we chose to focus on the lowest average energy metric, however it is likely that other metrics would perform

better. Similarly, we experimented with modifying the *reps* parameter passed to VQE, and we did not observe any clear correlation between this value and the quality of the results.

While our research focused mainly on improving the results obtained on a simulator, in the future it will be necessary to guarantee reasonable results on a real quantum computer as well. The first step in this exploration is the inclusion of artificial noise models, which serve as an indication of how a specific implementation might perform under decoherence. Some of the techniques tackled in this thesis seemed to perform well as a perfect noiseless simulation, however the inclusion of noise severely affected the results. It would be particularly valuable to reexamine the performance of QAOA mixers to evaluate if their performance outweighs the potential implementation cost. Ultimately, in order to minimize the influence of decoherence on a real quantum computer, it would be a good idea to explore some error mitigation techniques [18].

# List of Figures

2.1 The visual representation of two states, generated with Qiskit . . . . . . . . . 20

3.1 An overview of VQE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.2 A MAX-CUT toy problem and its solution . . . . . . . . . . . . . . . . . . . 31
3.3 An overview of QAOA . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.4 Top: The one-dimensional ferromagnetic Ising chain encoding. Bottom: An equivalent model without the fixed qubits, which encodes $\mathbb{Z}_5$ . . . . . . . . . 42
3.5 One-hot and domain wall qubit connectivity . . . . . . . . . . . . . . . . . . 49

4.1 Toy problem: task order DAG . . . . . . . . . . . . . . . . . . . . . . . . 52

5.1 An example of a one-hot encoded workflow scheduling vector $V$ . . . . . . . 59
5.2 An example of a binary encoded workflow scheduling vector $V$ . . . . . . . . 60
5.3 An example of a domain wall encoded workflow scheduling vector $V$ . . . . 61

6.1 A histogram of the number of correct results in 1024 samples over 100 randomized initial points . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.2 A one-dimensional visualization of the ideal ordering of solution energies . . 71
6.3 A one-dimensional visualization of the *naive ordering* of solution energies . . 71
6.4 A one-dimensional visualization of the *feasibility jump ordering* of solution energies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
6.5 Small problem: task order DAG . . . . . . . . . . . . . . . . . . . . . . . . 75
6.6 Large problem: task order DAG . . . . . . . . . . . . . . . . . . . . . . . . 75

7.1 The eigenvalues for one-hot encoding and QAOA with a default mixer . . . . 80

7.2 A result breakdown for one-hot encoding and QAOA with a default mixer . . 81
7.3 The eigenvalues for one-hot encoding and QAOA with a custom mixer . . . . 82
7.4 A result breakdown for one-hot encoding and QAOA with a custom mixer . . 83
7.5 The eigenvalues for one-hot encoding with VQE . . . . . . . . . . . . . . . 84
7.6 A result breakdown for one-hot encoding with VQE . . . . . . . . . . . . . . 85
7.7 A result breakdown for all experiments with one-hot encoding: incorrect and correct percentages . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
7.8 A result breakdown for all experiments with one-hot encoding: optimal and feasible percentages. Note: two outliers for VQE with POWELL with more than 85% of optimal solutions were removed from the plot to increase readability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
7.9 An eigenstate histogram for the best one-hot QAOA experiment . . . . . . . 88
7.10 An eigenstate histogram for the best one-hot VQE experiment . . . . . . . . 89
7.11 The eigenvalues for binary encoding with QAOA . . . . . . . . . . . . . . . 90
7.12 A result breakdown for binary encoding with QAOA . . . . . . . . . . . . . 91
7.13 The eigenvalues for binary encoding with VQE . . . . . . . . . . . . . . . . 92
7.14 A result breakdown for binary encoding with VQE . . . . . . . . . . . . . . 93
7.15 A result breakdown for all experiments with binary encoding: incorrect and correct percentages . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
7.16 A result breakdown for all experiments with binary encoding: optimal and feasible percentages . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
7.17 An eigenstate histogram for the best binary QAOA experiment . . . . . . . . 96
7.18 An eigenstate histogram for the best binary VQE experiment . . . . . . . . . 96
7.19 The eigenvalues for domain wall encoding and QAOA with a default mixer . 97
7.20 A result breakdown for domain wall encoding and QAOA with a default mixer 98
7.21 The eigenvalues for domain wall encoding and QAOA with a custom mixer . 99
7.22 A result breakdown for domain wall encoding and QAOA with a custom mixer 100
7.23 The eigenvalues for domain wall encoding with VQE . . . . . . . . . . . . . 101
7.24 A result breakdown for domain wall encoding with VQE . . . . . . . . . . . 102
7.25 A result breakdown for all experiments with domain wall encoding . . . . . . 103
7.26 An eigenstate histogram for the best domain wall QAOA experiment . . . . . 104

7.27 An eigenstate histogram for the best domain wall VQE experiment . . . . . . 105
7.28 The difference in incorrect solution percentages for different orderings with one-hot encoding and VQE algorithm . . . . . . . . . . . . . . . . . . . . . . 106
7.29 The difference in incorrect solution percentages for different orderings with binary encoding and VQE algorithm . . . . . . . . . . . . . . . . . . . . . . . 107
7.30 An incorrect solution percentage comparison between three encodings with QAOA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
7.31 A correct solution comparison between the three encodings with QAOA . . . 110
7.32 An incorrect solution comparison between three encodings with the VQE algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
7.33 A correct solution comparison between three encodings with the VQE algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112
7.34 Larger problem instance: An incorrect solution comparison between two encodings with the QAOA algorithm . . . . . . . . . . . . . . . . . . . . . . . 118
7.35 Larger problem instance: A correct solution comparison between two encodings with QAOA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
7.36 Larger problem instance: An incorrect solution comparison between two encodings with the VQE algorithm . . . . . . . . . . . . . . . . . . . . . . . 120
7.37 Larger problem instance: A correct solution comparison between two encodings with the VQE algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 121

# List of Tables

3.1 An example of categorical data . . . . . 36
3.2 An example of one-hot-encoded categorical data . . . . . 37
3.3 An example of one-hot-encoded categorical data, simplified . . . . . 37
3.4 A simple example of binary-encoded categorical data . . . . . 39
3.5 An example of domain-wall-encoded categorical data . . . . . 43
3.6 A comparison of binary, one-hot, and domain wall encoding . . . . . 47

4.1 Toy problem: cost matrix . . . . . 52
4.2 Toy problem: time matrix . . . . . 52

6.1 Small problem: cost matrix . . . . . 74
6.2 Small problem: time matrix . . . . . 74
6.3 Large problem (binary encoding): cost matrix . . . . . 76
6.4 Large problem (binary encoding): time matrix . . . . . 76
6.5 Large problem (domain wall encoding): cost matrix . . . . . 77
6.6 Large problem (domain wall encoding): time matrix . . . . . 77

7.1 A comparison of real quantum machine specifications . . . . . 113
7.2 A result breakdown for one-hot-encoded VQE with different noise models . . 113
7.3 A result breakdown for one-hot-encoded QAOA with different noise models . 114
7.4 A result breakdown for binary-encoded VQE with different noise models . . 114
7.5 A result breakdown for binary-encoded QAOA with different noise models . 115
7.6 A result breakdown for domain-wall-encoded VQE with different noise models 115

7.7 A result breakdown for domain-wall-encoded QAOA with different noise models . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116

# Bibliography


[1] Arute F., Arya K., Babbush R., Bacon D., Bardin J., Barends R., Biswas R., Boixo S., Brandao F., Buell D., Burkett B., Chen Y., Chen Z., Chiaro B., Collins R., Courtney W., Dunsworth A., Farhi E., Foxen B., Martinis J.: Quantum supremacy using a programmable superconducting processor. In: *Nature*, vol. 574, pp. 505–510, 2019. URL `http://dx.doi.org/10.1038/s41586-019-1666-5`.

[2] Aspuru-Guzik A.: Simulated Quantum Computation of Molecular Energies. In: *Science*, vol. 309(5741), pp. 1704–1707, 2005. ISSN 1095-9203. URL `http://dx.doi.org/10.1126/science.1113479`.

[3] Barkoutsos P.K., Nannicini G., Robert A., Tavernelli I., Woerner S.: Improving Variational Quantum Optimization using CVaR. In: *Quantum*, vol. 4, p. 256, 2020. ISSN 2521-327X. URL `http://dx.doi.org/10.22331/q-2020-04-20-256`.

[4] Benioff P.: The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. In: *Journal of Statistical Physics*, vol. 22, pp. 563–591, 1980. URL `http://dx.doi.org/10.1007/BF01011339`.

[5] Benmeleh Y., Turner G., Day M.: Amazon Is Laying the Groundwork for Its Own Quantum Computer, 2020. URL `https://www.bloomberg.com/news/articles/2020-12-01/amazon-is-laying-the-groundwork-for-its-own-quantum-computer`.

[6] Bernstein E., Vazirani U.: Quantum Complexity Theory. In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pp. 11–20. Associ-

ation for Computing Machinery, New York, NY, USA, 1993. ISBN 0897915917. URL `http://dx.doi.org/10.1145/167088.167097`.

[7] Bertels K., Sarkar A., Hubregtsen. T., Serrao M., Mouedenne A.A., Yadav A., Krol A., Ashraf I.: Quantum Computer Architecture: Towards Full-Stack Quantum Accelerators. In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020. URL `http://dx.doi.org/10.23919/date48585.2020.9116502`.

[8] Bharti K., Cervera-Lierta A., Kyaw T.H., Haug T., Alperin-Lea S., Anand A., Degroote M., Heimonen H., Kottmann J.S., Menke T., Mok W.K., Sim S., Kwek L.C., Aspuru-Guzik A.: Noisy intermediate-scale quantum (NISQ) algorithms. arXiv:quant-ph/2101.08448, 2021.

[9] Bishnoi B.: Quantum-Computation and Applications. arXiv:quant-ph/2006.02799, 2020.

[10] Bouzidi M.R., Soltani A., Bouhank A., Daoudi M.: New Search Based Methods to Solve Workflow Scheduling Problem in Cloud Computing. In: *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*, pp. 647–652. 2018. URL `http://dx.doi.org/10.1109/CoDIT.2018.8394855`.

[11] Bovet D.P., Crescenzi P.: *Introduction to the Theory of Complexity*. Prentice Hall International (UK) Ltd., GBR, 1994. ISBN 0139153802.

[12] Chancellor N.: Domain wall encoding of discrete variables for quantum annealing and QAOA. In: *Quantum Science and Technology*, vol. 4(4), p. 045004, 2019. URL `http://dx.doi.org/10.1088/2058-9565/ab33c2`.

[13] Chen J., Stollenwerk T., Chancellor N.: Performance of Domain-Wall Encoding for Quantum Annealing. In: *IEEE Transactions on Quantum Engineering*, vol. 2, pp. 1–14, 2021. ISSN 2689-1808. URL `http://dx.doi.org/10.1109/tqe.2021.3094280`.

[14] Cho A.: IBM promises 1000-qubit quantum computer-a milestone-by 2023, 2020. URL `https://www.sciencemag.org/news/2020/09/ibm-promises-1000-qubit-quantum-computer-milestone-2023`.

[15] Deutsch D., Jozsa R.: Rapid solution of problems by quantum computation. In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, vol. 439, pp. 553–558, 1992. URL http://dx.doi.org/10.1098/rspa.1992.0167.

[16] Dutta A., Aeppli G., Chakrabarti B.K., Divakaran U., Rosenbaum T.F., Sen D.: *Quantum Phase Transitions in Transverse Field Spin Models: From Statistical Physics to Quantum Information*. Cambridge University Press, 2015. URL http://dx.doi.org/10.1017/CBO9781107706057.

[17] Elmougy S., Sarhan S., Joundy M.: A Novel Hybrid of Shortest Job First and Round Robin with Dynamic Variable Quantum Time Task Scheduling Technique. In: *J. Cloud Comput.*, vol. 6(1), 2017. ISSN 2192-113X. URL http://dx.doi.org/10.1186/s13677-017-0085-0.

[18] Endo S., Benjamin S.C., Li Y.: Practical Quantum Error Mitigation for Near-Future Applications. In: *Physical Review X*, vol. 8(3), 2018. ISSN 2160-3308. URL http://dx.doi.org/10.1103/physrevx.8.031027.

[19] Farhi E., Goldstone J., Gutmann S.: A Quantum Approximate Optimization Algorithm. arXiv:quant-ph/1411.4028, 2014.

[20] Farhi E., Goldstone J., Gutmann S.: A Quantum Approximate Optimization Algorithm Applied to a Bounded Occurrence Constraint Problem. arXiv:quant-ph/1412.6062, 2015.

[21] Farhi E., Goldstone J., Gutmann S., Sipser M.: Quantum Computation by Adiabatic Evolution. arXiv:quant-ph/0001106, 2000.

[22] Fuchs F., Øie Kolden H., Aase N.H., Sartor G.: Efficient Encoding of the Weighted MAX k-CUT on a Quantum Computer Using QAOA. In: *SN Computer Science*, vol. 2, p. 89, 2021. URL http://dx.doi.org/10.1007/s42979-020-00437-z.

[23] Gaily S.E., Imre S.: Derivation of Parameters of Quantum optimization in Resource Distribution Management. In: *2019 42nd International Conference on Telecommunica-*

*tions and Signal Processing (TSP)*, pp. 58–61. 2019. URL http://dx.doi.org/10.1109/TSP.2019.8769092.

[24] Gambella C., Simonetto A.: Multiblock ADMM Heuristics for Mixed-Binary Optimization on Classical and Quantum Computers. In: *IEEE Transactions on Quantum Engineering*, vol. 1, pp. 1–22, 2020. ISSN 2689-1808. URL http://dx.doi.org/10.1109/tqe.2020.3033139.

[25] Gilyén A., Arunachalam S., Wiebe N.: Optimizing Quantum Optimization Algorithms via Faster Quantum Gradient Computation. In: *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '19, pp. 1425–1444. Society for Industrial and Applied Mathematics, USA, 2019. URL http://dx.doi.org/10.1137/1.9781611975482.87.

[26] Glos A., Krawiec A., Zimborás Z.: Space-efficient binary optimization for variational computing. arXiv:quant-ph/2009.07309, 2020.

[27] Gomes J., McKiernan K.A., Eastman P., Pande V.S.: Classical Quantum Optimization with Neural Network Quantum States. arXiv:cond-mat.dis-nn/1910.10675, 2019.

[28] Grover L.K.: A Fast Quantum Mechanical Algorithm for Database Search. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pp. 212–219. Association for Computing Machinery, New York, NY, USA, 1996. ISBN 0897917855. URL http://dx.doi.org/10.1145/237814.237866.

[29] Gyongyosi L.: Quantum State Optimization and Computational Pathway Evaluation for Gate-Model Quantum Computers. In: *Scientific Reports*, vol. 10, p. 4543, 2020. URL http://dx.doi.org/10.1038/s41598-020-61316-4.

[30] Hadfield S., Wang Z., O'Gorman B., Rieffel E., Venturelli D., Biswas R.: From the Quantum Approximate Optimization Algorithm to a Quantum Alternating Operator Ansatz. In: *Algorithms*, vol. 12(2), p. 34, 2019. ISSN 1999-4893. URL http://dx.doi.org/10.3390/a12020034.

[31] J. A., Adedoyin A., Ambrosiano J., Anisimov P., Bärtschi A., Casper W., Chennupati G., Coffrin C., Djidjev H., Gunter D., Karra S., Lemons N., Lin S., Malyzhenkov A., Mascarenas D., Mniszewski S., Nadiga B., O'Malley D., Oyen D., Pakin S., Prasad L., Roberts R., Romero P., Santhi N., Sinitsyn N., Swart P.J., Wendelberger J.G., Yoon B., Zamora R., Zhu W., Eidenbenz S., Coles P.J., Vuffray M., Lokhov A.Y.: Quantum Algorithm Implementations for Beginners. arXiv:cs.ET/1804.03719, 2020.

[32] Jaybhaye S., Attar V.: Adaptive Workflow Scheduling Using Evolutionary Approach in Cloud Computing. In: *Vietnam Journal of Computer Science*, vol. 7, 2020. URL http://dx.doi.org/10.1142/S2196888820500104.

[33] Johnson M., Amin M., Gildert S., Lanting T., Hamze F., Dickson N., Harris R., Berkley A., Johansson J., Bunyk P., Chapple E., Enderud C., Hilton J.P., Karimi K., Ladizinsky E., Ladizinsky N., Oh T., Perminov I., Rich C., Thom M.C., Tolkacheva E., Truncik C., Uchaikin S., Wang J., Wilson B., Rose G.: Quantum annealing with manufactured spins. In: *Nature*, vol. 473, pp. 194–198, 2011. URL http://dx.doi.org/10.1038/nature10012.

[34] Kadowaki T., Nishimori H.: Quantum annealing in the transverse Ising model. In: *Physical Review E*, vol. 58(5), pp. 5355–5363, 1998. ISSN 1095-3787. URL http://dx.doi.org/10.1103/physreve.58.5355.

[35] Khachatryan D.: Tutorials for Quantum Algorithms with Qiskit implementations. URL https://github.com/DavitKhach/quantum-algorithms-tutorials/blob/master/variational_quantum_eigensolver.ipynb.

[36] Kurowski K., Węglarz J., Subocz M., Różycki R., Waligóra G.: Hybrid Quantum Annealing Heuristic Method for Solving Job Shop Scheduling Problem. In: *Computational Science – ICCS 2020*, pp. 502–515. Springer International Publishing, Cham, 2020. ISBN 978-3-030-50433-5. URL http://dx.doi.org/10.1007/978-3-030-50433-5_39.

[37] Lara P., Portugal R., Lavor C.: A New Hybrid Classical-Quantum Algorithm for Continuous Global Optimization Problems. In: *Journal of Global Optimization*, vol. 60, pp. 317–331, 2014. URL http://dx.doi.org/10.1007/s10898-013-0112-8.

[38] Lavrijsen W., Tudor A., Muller J., Iancu C., de Jong W.: Classical Optimizers for Noisy Intermediate-Scale Quantum Devices. In: *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2020. URL http://dx.doi.org/10.1109/qce49297.2020.00041.

[39] Lewis M., Glover F.: Quadratic Unconstrained Binary Optimization Problem Preprocessing: Theory and Empirical Analysis. arXiv:cs.AI/1705.09844, 2017.

[40] Li H., Liu H., Li J.: Workflow scheduling algorithm based on control structure reduction in cloud environment. In: *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 2587–2592. 2014. URL http://dx.doi.org/10.1109/SMC.2014.6974317.

[41] Li R., Di Felice R., Rohs R., Lidar D.: Quantum annealing versus classical machine learning applied to a simplified computational biology problem. In: *npj Quantum Information*, vol. 4, p. 14, 2018. URL http://dx.doi.org/10.1038/s41534-018-0060-8.

[42] Low G.H., Bauman N.P., Granade C.E., Peng B., Wiebe N., Bylaska E.J., Wecker D., Krishnamoorthy S., Roetteler M., Kowalski K., Troyer M., Baker N.A.: Q# and NWChem: Tools for Scalable Quantum Chemistry on Quantum Computers. arXiv:quant-ph/1904.01131, 2019.

[43] Lucas A.: Ising formulations of many NP problems. In: *Frontiers in Physics*, vol. 2, 2014. ISSN 2296-424X. URL http://dx.doi.org/10.3389/fphy.2014.00005.

[44] Marszałek K.: *Ocena środowiska do obliczeń kwantowych Rigetti Quantum Services*. Master's thesis, AGH University of Science and Technology, Kraków, 2020.

[45] McGeoch C., Farre P.: *The D-Wave Advantage System: An Overview*. D-Wave, 2020. URL https://www.dwavesys.com/media/s3qbjp3s/14-1049a-a_the_d-wave_advantage_system_an_overview.pdf.

[46] Mermin N.D.: *Quantum Computer Science: An Introduction*. Cambridge University Press, New York, 2007. ISBN 0521876583. URL http://dx.doi.org/10.1017/CBO9780511813870.

[47] Nandhakumar C., Ranjithprabhu K.: Heuristic and meta-heuristic workflow scheduling algorithms in multi-cloud environments – A survey. In: *2015 International Conference on Advanced Computing and Communication Systems*, pp. 1–5. 2015. URL http://dx.doi.org/10.1109/ICACCS.2015.7324053.

[48] Nelder J., Mead R.: A Simplex Method for Function Minimization. In: *Comput. J.*, vol. 7, pp. 308–313, 1965. URL http://dx.doi.org/10.1093/comjnl/7.4.308.

[49] Orús R., Mugel S., Lizaso E.: Quantum computing for finance: Overview and prospects. In: *Reviews in Physics*, vol. 4, p. 100028, 2019. ISSN 2405-4283. URL http://dx.doi.org/10.1016/j.revip.2019.100028.

[50] Outeiral C., Strahm M., Shi J., Morris G.M., Benjamin S.C., Deane C.M.: The prospects of quantum computing in computational molecular biology. In: *WIREs Computational Molecular Science*, vol. 11(1), p. e1481, 2021. URL http://dx.doi.org/10.1002/wcms.1481.

[51] Panetta K.: 5 Trends Emerge in the Gartner Hype Cycle for Emerging Technologies, 2018, 2018. URL https://www.gartner.com/smarterwithgartner/5-trends-emerge-in-gartner-hype-cycle-for-emerging-technologies-2018/.

[52] Pautasso L., Pflanzer A., Soller H.: The current state of quantum computing: Between hype and revolution, 2021. URL https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/tech-forward/the-current-state-of-quantum-computing-between-hype-and-revolution#.

[53] Pawlik M., Banach P., Malawski M.: Adaptation of Workflow Application Scheduling Algorithm to Serverless Infrastructure. In: U. Schwardmann, C. Boehme, D. B. Heras, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R.R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, S.L. Scott, eds., *Euro-Par 2019: Parallel Processing Workshops*, pp. 345–356. Springer International Publishing, Cham, 2020. ISBN 978-3-030-48340-1. URL http://dx.doi.org/10.1007/978-3-030-48340-1_27.

[54] Pellow-Jarman A., Sinayskiy I., Pillay A., Petruccione F.: A comparison of various classical optimizers for a variational quantum linear solver. In: *Quantum Information Processing*, vol. 20(6), 2021. ISSN 1573-1332. URL http://dx.doi.org/10.1007/s11128-021-03140-x.

[55] Peruzzo A., McClean J., Shadbolt P., Yung M.H., Zhou X.Q., Love P.J., Aspuru-Guzik A., O'Brien J.L.: A variational eigenvalue solver on a photonic quantum processor. In: *Nature Communications*, vol. 5(1), 2014. ISSN 2041-1723. URL http://dx.doi.org/10.1038/ncomms5213.

[56] Porter J.: Google wants to build a useful quantum computer by 2029, 2021. URL https://www.theverge.com/2021/5/19/22443453/google-quantum-computer-2029-decade-commercial-useful-qubits-quantum-transistor.

[57] Powell M.J.: A view of algorithms for optimization without derivatives. In: *Mathematics Today-Bulletin of the Institute of Mathematics and its Applications*, vol. 43(5), pp. 170–174, 2007. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.591.6481&rep=rep1&type=pdf.

[58] Powell M.J.D.: An Efficient Method for Finding the Minimum of a Function of Several Variables Without Calculating Derivatives. In: *The Computer Journal*, vol. 7(2), pp. 155–162, 1964. URL http://dx.doi.org/10.1093/comjnl/7.2.155.

[59] Powell M.J.D.: *A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation*, pp. 51–67. Springer Netherlands, Dor-

drecht, 1994. ISBN 978-94-015-8330-5. URL http://dx.doi.org/10.1007/978-94-015-8330-5_4.

[60] Powell M.J.D.: Direct search algorithms for optimization calculations. In: *Acta Numerica*, vol. 7, pp. 287–336, 1998. URL http://dx.doi.org/10.1017/S0962492900002841.

[61] Preskill J.: Quantum computing and the entanglement frontier. arXiv:quant-ph/1203.5813, 2012.

[62] Shor P.W.: Algorithms for quantum computation: discrete logarithms and factoring. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134. 1994. URL http://dx.doi.org/10.1109/SFCS.1994.365700.

[63] Smanchat S., Viriyapant K.: Taxonomies of workflow scheduling problem and techniques in the cloud. In: *Future Generation Computer Systems*, vol. 52, pp. 1–12, 2015. ISSN 0167-739X. URL http://dx.doi.org/10.1016/j.future.2015.04.019. Special Section: Cloud Computing: Security, Privacy and Practice.

[64] Stachoń M.: *Solving Optimization problems using Qiskit Aqua*. Master's thesis, AGH University of Science and Technology, Kraków, 2020. URL http://dice.cyfronet.pl/publications/source/MSc_theses/Malgorzata_Stachon_msc.pdf.

[65] Suzuki M.: Generalized Trotter's formula and systematic approximants of exponential operators and inner derivations with applications to many-body problems. In: *Communications in Mathematical Physics*, vol. 51, pp. 183–190, 1976. URL http://dx.doi.org/10.1007/BF01609348.

[66] Tan B., Lemonde M., Thanasilp S., Tangpanitanon J., Angelakis D.: Qubit-efficient encoding schemes for binary optimisation problems. In: *Quantum*, vol. 5, p. 454, 2021. URL http://dx.doi.org/10.22331/q-2021-05-04-454.

[67] Tomasiewicz D.: *Analysis of D'Wave 2000Q Applicability for Job Scheduling Problems*. Master's thesis, AGH University of Science and Technology, Kraków,

2020. URL http://dice.cyfronet.pl/publications/source/MSc_theses/Dawid_Tomasiewicz_msc.pdf.

[68] Tomasiewicz D., Pawlik M., Malawski M., Rycerz K.: Foundations for Workflow Application Scheduling on D-Wave System. In: V.V. Krzhizhanovskaya, G. Závodszky, M.H. Lees, J.J. Dongarra, P.M.A. Sloot, S. Brissos, J. Teixeira, eds., *Computational Science – ICCS 2020*, pp. 516–530. Springer International Publishing, Cham, 2020. URL http://dx.doi.org/10.1007/978-3-030-50433-5_40.

[69] Wang Z., Rubin N.C., Dominy J.M., Rieffel E.G.: XY mixers: Analytical and numerical results for the quantum alternating operator ansatz. In: *Physical Review A*, vol. 101(1), 2020. ISSN 2469-9934. URL http://dx.doi.org/10.1103/physreva.101.012320.

[70] Xiao Q., Zhong J., Feng L., Luo L., Lv J.: A Cooperative Coevolution Hyper-Heuristic Framework for Workflow Scheduling Problem. In: *IEEE Transactions on Services Computing*, 2019. ISSN 1939-1374. URL http://dx.doi.org/10.1109/TSC.2019.2923912.

[71] Yan Z., Shen H., Huang H., Deng Z.: Constrained Optimization via Quantum Genetic Algorithm for Task Scheduling Problem. In: H. Shen, Y. Sang, eds., *Parallel Architectures, Algorithms and Programming*, pp. 234–248. Springer Singapore, Singapore, 2020. ISBN 978-981-15-2767-8. URL http://dx.doi.org/10.1007/978-981-15-2767-8_22.

[72] Yanofsky N.S., Mannucci M.A.: *Quantum Computing for Computer Scientists*. Cambridge University Press, USA, 1 ed., 2008. ISBN 0521879965. URL http://dx.doi.org/10.1017/CBO9780511813887.

[73] Yuan X., Endo S., Zhao Q., Li Y., Benjamin S.C.: Theory of variational quantum simulation. In: *Quantum*, vol. 3, p. 191, 2019. ISSN 2521-327X. URL http://dx.doi.org/10.22331/q-2019-10-07-191.

[74] Zhong H.S., Wang H., Deng Y.H., Chen M.C., Peng L.C., Luo Y.H., Qin J., Wu D., Ding X., Hu Y., Hu P., Yang X.Y., Zhang W.J., Li H., Li Y., Jiang X., Gan L., Yang G.,

You L., Wang Z., Li L., Liu N.L., Lu C.Y., Pan J.W.: Quantum computational advantage using photons. In: *Science*, vol. 370(6523), pp. 1460–1463, 2020. ISSN 0036-8075. URL http://dx.doi.org/10.1126/science.abe8770.

[75] Zhu C., Byrd R.H., Lu P., Nocedal J.: Algorithm 778: L-BFGS-B: Fortran Subroutines for Large-Scale Bound-Constrained Optimization. In: *ACM Trans. Math. Softw.*, vol. 23(4), pp. 550–560, 1997. ISSN 0098-3500. URL http://dx.doi.org/10.1145/279232.279236.