Application of Quantum Computing to Use Cases from Travel & Transport Industries

Bachelorarbeit

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik

an der

Dualen Hochschule Baden-Württemberg Stuttgart

von

Lucine Madadi

Abgabedatum 06. September 2021

Bearbeitungszeitraum	12 Wochen
Matrikelnummer	3634158
Kurs	INF18A
Ausbildungsfirma	IBM Deutschland GmbH
	Ehningen
Betreuer der Ausbildungsfirma	Dr. Jan-Rainer Lahmann
Gutachterin der Hochschule	Prof. Dr. Carmen Winter

Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeitmit dem Thema: "Application of Quantum Computing to Use Cases from Travel & Transport Industries" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort	Datum	Unterschrift

Acknowledgements

I have received quite a lot of support in the process of writing this thesis. I would like to thank my supervisors – Dr. Jan-Rainer Lahmann, for his invaluable feedback, and Prof. Dr. Carmen Winter, for her input on structure and content. I would further like to thank Fabian Klos, who gave additional support on the subject of implementing the QAOA, and Nima Barraci and Oliver Haßa, who gave input on the formulation of the passenger routing use case. Additional thanks go to Stefan Wörner and Vaibhaw Kumar, who were kind enough to lend some insight into the implementation and testing process.

Abstract

Quantum computing is one of the most promising emerging technologies today. Quantum computers have the potential to solve problems in ways a classical computer cannot, opening up a realm of possibilities. However, the necessary hardware is still limited in size: currently, the largest universal quantum computers work with qubit numbers below 100. This necessitates the usage of efficient algorithms in order to solve relevant problems using limited resources. One of those algorithms is the Quantum Approximate Optimization Algorithm (QAOA).

It serves as an optimization algorithm that can be applied to a host of different problems. In this project, a proposed method of adapting the QAOA for routing problems [1] shall be examined, implemented – using IBM's quantum computing framework (Qiskit) – and evaluated. Tests are run on a simulator; the routing problems include single- and multi-vehicle routing as well as the travelling salesperson problem. In addition, a passenger routing use case in the context of aviation will be formulated and implemented.

In the result evaluation, the QAOA has been found to work well for problems with 10 qubits or below; for larger problems, the results often feature a large number of inadmissible solutions. The Variational Quantum Eigensolver (VQE), a similar algorithm, has been found to yield better results and was used for testing instead of the QAOA, though both algorithms perform worse with growing problem size, especially with qubit numbers above approx. 20.

For both algorithms, parameter tuning constitutes a central part in improving their performance. For problems with high qubit numbers, finding suitable parameters gets more challenging. Due to the large number of parameter combinations, this process is aided by having knowledge of the problem and algorithm at hand.

Methods for finding suitable parameters, as well as the effects of using a real quantum computer compared to a simulator, are left as further avenues for research.

Abstract

Quantum-Computing ist eine der zukunftsweisenden Technologien des aktuellen Zeitalters. Quantencomputer ermöglichen Ansätze für Problemlösungen, die durch klassische Computer nicht realisierbar sind. Jedoch ist die notwendige Hardware noch begrenzt: derzeit haben die größten universalen Quantencomputer noch Qubit-Zahlen unter 100. Um mit den – auch in absehbarer Zeit – begrenzten Ressourcen relevante Probleme lösen zu können, bedarf es daher effizienter Algorithmen. Einer dieser Algorithmen ist der Quantum Approximate Optimization Algorithm (QAOA), ein Optimierungsalgorithmus, der für eine Reihe von Problemen eingesetzt werden kann.

In dieser Arbeit sollen Formulierungen des QAOA für Routing-Probleme aus [1] umgesetzt und evaluiert werden. Darunter sind Single- und Multi-Vehicle-Routing sowie das Travelling Salesperson Problem. Zusätzlich wird ein Use-Case aus dem Luftfahrtbereich formuliert und implementiert. Verwendet werden soll für die Implementierung IBMs Quantum-Framework Qiskit; getestet wird mit einem Simulator.

Die Implementierung hat aufgezeigt, dass der QAOA für Probleme mit einer Größe von 10 Qubits oder weniger gute Ergebnisse liefert. Für größere Probleme sind die Ergebnisse häufig nicht zulässig. Der Variational Quantum Eigensolver (VQE), ein verwandter Algorithmus, erzielte in Tests bessere Ergebnisse, wenngleich beide Algorithmen für Probleme mit Qubit-Zahlen von etwa 20 und mehr schlechtere Ergebnisse erzielen.

Parameter-Tuning ist für beide Algorithmen ein integraler Bestandteil dessen, bessere Ergebnisse zu erhalten, und ist für Probleme mit hohen Qubit-Zahlen komplizierter. Der Prozess der Parameterfindung kann dadurch tiefere Analyse und besseres Verständnis von Problem und Algorithmus unterstützt werden.

Methoden zur Parameterfindung und das Testen mit einem echten Quantencomputer stellen weitergehende Forschungsmöglichkeiten dar.

Contents

List of Figures

List of Tables

Abbreviations

MV	Multi-Vehicle	3
nsg	numbered sliced graph	32
QAOA	Quantum Approximate Optimization Algorithm	1
QUBO	Quadratic Unconstrained Binary Optimization	13
sg	sliced graph	30
SV	Single-Vehicle	3
TSP	Travelling Salesperson Problem	3
VQE	Variational Quantum Eigensolver	15

Chapter 1

Introduction

Quantum computing is a relatively young field on the intersection of physics, mathematics, and computer science. In the last decades, it has evolved from a theoretical possibility to a way of computing that completely eschews boundaries previously thought rigid. As an emerging and evolving technology, quantum computing has the potential to fundamentally change how certain problems are approached — though performant hardware is necessarily a prerequisite.

Currently, the majority of universal quantum computers work with two-digit numbers of qubits, and so the challenge lies in demonstrating and testing the capabilities of quantum computing using limited hardware. Quantum computing as a rapidly growing field promises significant changes in computing, in a variety of areas of high relevance in today's age.

One such application area is optimization, a field where quantum computers could potentially be much more performant than classical ones. This is due to the fact that optimization problems can grow rapidly, combinatorially, with the size of the inputs given. For classical computing, the immense number of possible solutions makes them increasingly difficult and time-intensive to compute. Quantum algorithms like the Quantum Approximate Optimization Algorithm (QAOA) take an approach that can compute large problems, is scalable, and thus might prove to be an impactful improvement for solving optimization problems.

Routing optimization in particular is an area that is—especially in a world as globalized as ours—of great importance in a variety of different industries: any field where the transportation of goods or people is a central task, be it aviation, logistics, or postal delivery.

In practice, in real-world scenarios, routing problems often become very large in scale.

Due to the limitations mentioned above, it therefore becomes important to find small, contained use cases suitable for demonstrating the usage of a particular quantum algorithm. The objective of this work is to implement formulations of routing problems for the QAOA for small test cases. The respective formulations are those found in [1]. In addition to the four scenarios covered in the paper, a use case for routing in aviation, a simplified version of the passenger routing problem, will be formulated and implemented. The goal is to illustrate the possibilities and limitations of the QAOA for solving routing problems.

After this introduction, chapter 2 will give an overview over the various topics that are relevant in the context of this work. Chapter 3 will describe the approach taken in the implementations, which, along with their results, follow in chapter 4 for the use cases in [1] and in chapter 5 for the passenger routing use case. Chapter 6 will conclude this work with a summary and a brief outlook on the future of quantum computing.

Chapter 2

State of the Art – Routing & Quantum Computing

2.1 Routing Optimization

Routing problems are ubiquitous in the modern world. Routing has long been relevant for travel, commerce, and other areas, but in a world as globalized as this, the problems to be solved take on scales that grow in orders of magnitudes.

The central conceit of routing is to find the optimal path (or paths) from a number of possibilities, often while respecting a number of constraints. Routing is therefore one application in the broader field of optimization. What defines the optimal path can change from scenario to scenario; typical factors to minimize are distance, financial cost, or time.

There are a number of different routing problems, with a different focus each. They can be broadly categorized based on the type of scenario. For instance, one can distinguish between Single-Vehicle (SV) and Multi-Vehicle (MV) routing: In the former, only one path is needed; in the latter, the goal is to find several paths for different vehicles. While in some cases, a multi-vehicle problem can be seen as a combination of several single-vehicle problems, oftentimes the individual path choices have an effect on the whole and therefore need to be optimised holistically.

Problems can also be categorised based on whether the sought route needs to pass one destination or several. When computing the shortest route between two points, there is only one destination; in the case of the Travelling Salesperson Problem (TSP), there are several. Here, the goal is to find a path that connects a number of given nodes in a way that minimizes a chosen metric while visiting each node only once. This excepts the starting point, which is simultaneously the last destination. In the TSP, every node to be visited constitutes a destination node.

As mentioned, routing problems appear in a variety of areas. In an industry context, routing is of great importance especially to transport-focused areas such as aviation or freight transport, where choosing the shortest route or assigning paths to different vehicles are constant issues. In theory, the goal is to find the best path for a given scenario; in practice, the number of possibilities is oftentimes so large that finding the best one is, if not outright infeasible, oftentimes extremely resource-intensive and time-consuming. Therefore, a solution that is close to the optimum often suffices for the specific purposes, compared to an ideal solution that takes several times as long to compute. This is especially true in cases where the problem at hand is a time-sensitive one and a solution must not only be valid (i.e., adhering to all constraints), but must also be calculated quickly.

One category of algorithms using this approximate approach are heuristic algorithms. They make use of approximations and estimations based on available data in order to gauge the optimal solution to a problem. Heuristics need to be specific to the problem at hand in order to be effective. Since heuristics do not need to calculate the exact metric and often do not need to traverse all available data, they can reduce the complexity of solving the problem. This also means they are often less accurate and reliable, but faster.

Using heuristics is one out of many different approaches and ways to optimize the process of route-finding. Some well-known methods used for classical computers will be described briefly, after a short explanation of graphs.

Traditionally, path finding algorithms take a graph as input. A basic graph consists of a set of vertices (also called nodes) V; a set of edges E that connect them; and, optionally, additional properties of vertices and edges, for instance a weight w_{e_i} of an edge e_i.

Graphs can have different properties, some of which shall be mentioned briefly here. Graphs with edge weights are called weighted graphs; unweighted graphs can essentially be conceptualized as graphs with uniform weights. The weight of an edge represents a value for a specific metric, such as distance or duration. Graphs whose edges do not specify directions (i.e. every edge is bidirectional) are called undirected; directed if they do specify directions. Finally, if a graph's edges allow for a loop, the graph is called cyclical; it is called acyclical otherwise.

A graph can, computationally, be defined and represented in a number of different ways, depending on its properties and the data structures at one's disposal. One possibility is to save the graph data in a matrix of size $|V| \cdot |V|$. A given entry at m, n contains the weight of the edge that leads from a vertex with index m – that is, v_m – to the vertex v_n. For undirected graphs, this matrix would contain a large number of superfluous entries, since any entry at m, n is identical to the entry at n, m. Another option is to merely store the graph's edges in a list structure (or a key-value store if the edges are named) or alternatively by simply storing each vertex's neighbors, if the graph is unweighted. The approach should be chosen based on how the program will later need to work with the data and where the priorities lie: access and update speed, storage, etc.

For straightforward shortest-path routing problems, common algorithms that lend themselves to the matter are Dijkstra's algorithm, the A* algorithm, and breadth- or depth-first search. They take a graph as input and traverse it, following their specific approach. They then return the path with the lowest edge weight, for example in the form of a list of vertices to traverse.

The A* algorithm, for instance, works roughly as follows. It begins at the start vertex and considers all directly accessible vertices (neighbor vertices). For each of the vertices, it first computes the distance between the vertex and the start vertex, and then, using a heuristic, the estimated distance between the vertex and the destination vertex. This constitutes the vertex' cost. From the set of accessible vertices, the one with the lowest cost is chosen as the next vertex in the route.

From then, that vertex is expanded – its neighbor vertices are added to the set of accessible vertices and evaluated, or, if some are already in the set, their cost is updated. Vertices that have been expanded are placed in the set of 'closed' vertices. From the set of accessible vertices, the vertex with the lowest cost is chosen. Since this vertex does not necessarily have to be an extension of the previously chosen vertex, it can open a new path. Thus, different paths can open up by expanding different vertices, and the steps of expanding and comparing continue in a loop.

The algorithm terminates when the destination vertex is chosen as the next vertex in a path. If the estimation is chosen well (for one, it needs to underestimate the actual cost), then A* returns the shortest path to the destination from the start vertex.

For more complex optimization in routing and pathfinding, there is a large variety of approaches, including the Branch and Cut method, column generation, and tabu search (a heuristic) [2] [3], or gradient descent, an optimization method used for finding function minima. These approaches do not directly use a graph as input; instead, they aim to find the optimum of a function based on the given graph. This allows for the formulation of more complicated problems – they take into account factors other than mere path weight, such as specific constraints that must be fulfilled (time-related constraints, for instance). For real-life purposes, this is more suitable as it allows for closer modelling of use cases.

2.2 Quantum Computing

The following section will provide a short introduction to quantum computing and its mathematical and historical context, as well as discussing some of the hardware behind quantum computers.

2.2.1 Mathematical Framework

Quantum computers fundamentally differ from classical computers – the state of their smallest units of data, called qubits (quantum bits), is not limited to the values of 1 and 0, as is the case with bits in classical computers. Instead, they can take on values that are superpositions (linear combinations) of the basis vectors $|0\rangle$ and $|1\rangle$ (written in the so-called Dirac or Bra-ket notation). This state of a qubit allows for operations that go beyond those used in classical computing. The state of a qubit, also referred to as its wave function, can be expressed as follows:

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

where $|\alpha|^2 + |\beta|^2 = 1$. $|0\rangle$ and $|1\rangle$ are not related to the values 0 and 1; these are simply the designations for the two common basis vectors of a one-qubit system. Any other computational basis can be defined if the vectors are linearly independent and of length 1.

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, |0\rangle\,, |1\rangle \in \mathbb{C}^2$$

Any qubit can therefore be represented by a vector, where the first component is α and the second β. In the ket notation, $|x\rangle$ represents a column vector while $\langle x|$ represents the corresponding transpose, a row vector.

The state of a two-qubit system can similarly be represented by a vector. This vector is constructed by calculating the tensor product of the two individual vectors – that is, by multiplying each component of the first vector with the second vector. The following example shows a vector representing a three-qubit system.

$$|1\rangle\,|0\rangle\,|1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{pmatrix}$$

A qubit system of any size can be represented as the tensor product of its qubits, as long as it does not contain entangled qubits, though entangled states can still be represented by vectors. Since the dimension of the tensor product of two vectors is equal to the product of the vectors' lengths, the tensor product of n two-dimensional vectors will have dimension 2^n and thus be in $\mathbb{C}^{2^n}$. This exponential growth in size is why large quantum computers are difficult to simulate using classical computers.

Using a geometric model, a single qubit can be described using the so-called Bloch (or Poincaré) sphere, where the two states $|0\rangle$ and $|1\rangle$ are located at the two poles, respectively. Any point on the surface of the sphere other than the poles represents a state of superposition of those states.

With the Bloch sphere as a basis, the state $|\psi\rangle$ (psi) of a single qubit can also be defined by two angles φ (phi) and θ (theta), where φ acts essentially like the longitudinal angle ('east-west' alignment) and θ like the latitudinal one ('north-south' alignment). The relationship between angles and linear coefficients is summarised in this formula [4]:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle \;+\; e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle \tag{2.2}$$

where $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi)$.

These models allow for a description of a single qubit in a state of superposition. Any such qubit will, once measured, invariably 'collapse' into one of the values of the computational basis it is measured in. If the computational basis used is $|0\rangle$ and $|1\rangle$, then it will collapse into one of these states. The probability of collapsing into one value is equal to the square of the absolute value of its linear coefficient – here, $|\alpha|^2$ for $|0\rangle$ and $|\beta|^2$ for $|1\rangle$. This is the reason why the sum of $|\alpha|^2$ and $|\beta|^2$ needs to equal 1.

The manipulation of the so-called probability amplitudes α, β that influence which value a qubit collapses to is therefore of great importance in the formulation of quantum

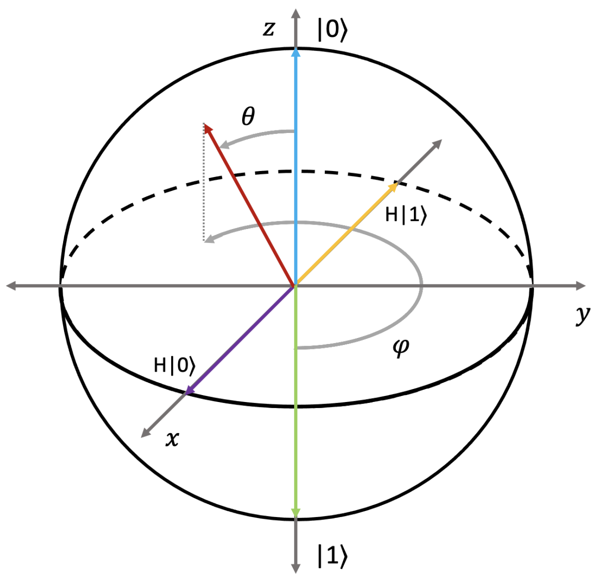


Figure 2.1: Bloch sphere representation of qubit states. $|0\rangle$ is marked in blue, $|1\rangle$ in green. The purple and yellow arrows show these states after a Hadamard gate (explained on p. 9) has been applied to them, respectively. The red arrow illustrates the angle representation of a qubit.

algorithms. Being able to implement entanglement – 'bindings' between qubits such that manipulation of one will affect the other – is another important touchstone. The third important concept in quantum computing is quantum interference, an effect of the wave-particle duality observable in quantum particles. Quantum interference can influence an outcome's probability[1].

Both superposition and entanglement, as well as other operations on qubits, can be realized by using specific gates on qubits. They are the quantum computing equivalent to logical gates in classical computing; a quantum circuit is essentially made of a series of quantum gates combined to achieve a specific result. These gates, or operators, can be represented by matrices that can be applied to the vector representation of a qubit using matrix multiplication. They can, for single qubits, also be conceptualized as rotations on the Bloch sphere, just as matrices can represent rotations for vectors in general.

[1]The double-slit experiment is one prominent example that showcases quantum interference.

As an example, one can consider the Pauli X, Y, and Z matrices. They perform rotations of 180° around the X, Y and Z axis, respectively.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, X\left|0\right\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, X\left|1\right\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{2.3}$$

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, Y\left|0\right\rangle = \begin{pmatrix} 0 \\ i \end{pmatrix}, Y\left|1\right\rangle = \begin{pmatrix} -i \\ 0 \end{pmatrix} \tag{2.4}$$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, Z\left|0\right\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, Z\left|1\right\rangle = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \tag{2.5}$$

Another example is the Hadamard gate. It can put a qubit from a single basis state into superposition. Its matrix is the following:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.6}$$

Applying it to $\left|0\right\rangle$ and $\left|1\right\rangle$ results in the following linear combinations:

$$H(\left|0\right\rangle) = \frac{1}{\sqrt{2}}\left|0\right\rangle + \frac{1}{\sqrt{2}}\left|1\right\rangle = \left|+\right\rangle \tag{2.7}$$

$$H(\left|1\right\rangle) = \frac{1}{\sqrt{2}}\left|0\right\rangle - \frac{1}{\sqrt{2}}\left|1\right\rangle = \left|-\right\rangle \tag{2.8}$$

This corresponds to a 180°-rotation around the angle bisector between the X and the Z axes. It can also be conceptualized as a 180° rotation around the Z axis and a subsequent 90° rotation around the Y axis. As seen in fig. 2.1, in the Bloch sphere representation the state $\left|0\right\rangle$ is projected onto the intersection of the X axis and the equator in the positive section of the X axis, while $\left|1\right\rangle$ is projected onto its opposing point, in the negative section of the X axis. The resulting states, $\left|+\right\rangle$ and $\left|-\right\rangle$, are commonly used as alternative basis states.

A gate acting on one qubit is of dimension 2×2; a gate acting on n qubits has dimension $2^n \times 2^n$, since the number of columns needs to be equal to the dimension of the corresponding qubit vector for the multiplication to work.

Applying single-qubit gates on systems of more than one qubit is done by calculating the tensor product of the gates and then applying it on the qubits. In a two-qubit system with state $\left|11\right\rangle$, where this follows the format $\left|q_1q_0\right\rangle$, applying the Pauli X gate to q_1 and the Hadamard gate to q_0 would have the following result:

$$\begin{aligned} X \otimes H\,|11\rangle &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} |11\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \\ 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \\ &= \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}} \cdot |00\rangle - \frac{1}{\sqrt{2}} \cdot |01\rangle + 0 \cdot |10\rangle + 0 \cdot |11\rangle \end{aligned} \tag{2.9}$$

As is apparent, q_1 has been flipped – the only basis vectors with a non-zero coefficient are ones where it is $|0\rangle$. In contrast, q_0 is now in superposition, with the coefficient of $|0\rangle$ being $\frac{1}{\sqrt{2}}$ and the coefficient of $|1\rangle$ $-\frac{1}{\sqrt{2}}$.

Were one to apply the X gate to q_1 without modifying q_0, the tensor product would be constructed from the X gate and the identity gate I, which leaves qubits unchanged.

Any such quantum gate, regardless of its dimension, needs to be reversible, a requirement that classical computing does not have. Quantum operators are unitary, meaning the inverse of an operator's matrix (e.g. the matrices in eq. 2.3 - 2.6) is equal to its conjugate transpose [5]. Gates like the Hadamard gate (which can put a qubit into superposition) are unique to quantum computing.

2.2.2 Quantum Hardware

The hardware of a quantum computer needs to be able to realize the theoretical concepts described. Over the last two decades, several different approaches have formed, using different implementations for the qubit. Possible approaches include using electronic circuits ('superconducting qubits'), or implementing qubits using electrons or photons, where electron spin and polarization, respectively, can be used to encode basis vectors.

All these approaches enable the implementation of qubits that allow for the realization of the aforementioned quantum computing concepts of superposition, entanglement, and interference.

There are a number of companies doing research around quantum computing and building quantum computers. Currently, the largest universal qubit computers have qubit numbers below 100 (such as the University of Science and Technology of China's

Jiuzhang with 76 qubits [6] or Google's *Bristlecone* with 72 qubits [7]), though simulators and quantum annealers can reach qubit numbers in the thousands, such as the D-Wave Advantage annealer [8]. Annealers differ from universal quantum computers in that they are adept at solving annealing (effectively, optimization) problems, though they are also limited to problems that can be formulated as such.

While quantum computing is in theory very powerful, in practice it still contends with problems such as non-trivial error rates (due to measurement errors, for example) and decoherence (meaning a quantum state cannot be maintained for long). Decoherence limits the number of gates that can be applied successively in a circuit. Additionally, the connectivity between qubits plays a role in how many gates are needed to solve a problem and as such is an important design factor; low connectivity decreases the efficiency of a quantum computer. Once these problems have been mitigated or solved, quantum computing will likely see a surge in usage in real-life application scenarios.

2.2.3 Historical Background

The fact that quantum computers are not already ubiquitous is not surprising – quantum computing is a relatively young field. Starting in the 1960s, research in quantum physics and computer science gradually advanced in a way that allowed theoretical frameworks for quantum computation to emerge. The concept of a quantum computer was first proposed in 1980 by Paul Benioff [9], who in his proposition put forth a way to simulate the capabilities of Turing machines using quantum mechanics. Richard Feynman, an early proponent, argued in 1981 that quantum computers might be able to model quantum systems that classical computers are unable to [10].

Building on these and other early theories, the first quantum computers were built in 1998 [11] [12] – then with two qubits, using nuclear magnetic resonance technology (NMR). This approach used the spin states of atomic nuclei to represent the qubit states. Since then, large strides have been made and a variety of approaches have been developed, as mentioned above.

For all of these approaches, proving their usefulness is an important goal – a concept referred to as 'quantum advantage'. The term was originally coined in 2012 as 'quantum supremacy' by its originator John Preskill [13]. 'Quantum advantage' as a term has emerged gradually a few years later[2].

At its core, quantum supremacy is about demonstrating that a quantum computer is

[2]This was to present alternate terminology – 'supremacy' implies an inherent improvement over classical computers, and has negative connotations stemming from the term 'white supremacy'.

capable of solving a problem that no classical computer can (or would take an exceedingly long time to), irrespective of the problem's usefulness. This has been achieved by some quantum computers – the aforementioned *Jiuzhang* [6] and Google's *Sycamore*[3] [15].

The term quantum advantage, on the other hand, has a number of different definitions – some equate it with quantum supremacy and treat the two terms as synonyms, others differentiate between them and consider them separate concepts. The core of the matter is that the problems used to prove quantum supremacy are generally ones specifically formulated to be easy to solve for quantum computers and difficult for classical computers. Essentially, they are often not relevant problems. This is why some definitions position quantum advantage as being about solving a relevant, real-life problem, generally in the realm of 'business or science' [16]. This has not been achieved yet and so remains an important area of research.

The relative nature of the definitions of both quantum supremacy and quantum advantage make them difficult to conclusively prove – for one, the development of classical computers and their algorithms is not stagnant; more performant hardware and faster algorithms that have not yet been developed might, in the future, lead to a significant speedup for a given problem on classical computers. For another, the formulation leaves open what is meant when speaking about comparatively long computing times for a classical computer, adding to the imprecise nature of the definitions.

Some developments, such as Shor's algorithm for prime factorization or Grover's algorithm for unstructured search, already strongly imply that quantum computers will play a significant role in computing in the coming decades. For optimization in particular, quantum computers signal a faster, more performant, and more scalable alternative to classical computers in the future.

As it is now, the current development of quantum computers echoes that of classical computers several decades ago – limited in scale and capability, owned and manufactured by a select few, and not yet very performant. Within a few decades, classical computers came to be not only affordable, but indispensable in modern life, with whole economies based on their continued functioning. How quantum computers will play into that dynamic long-term cannot be foreseen with certainty. However, the difference between classical and quantum computers is significant. It is therefore likely that the quantum computer will not replace the classical computer in the near future and that classical and quantum computers will exist in a complementary relationship rather than a competing one.

[3]Though Google's claim has been contested by IBM researchers, among others [14].

2.3 Quantum Approximate Optimization Algorithm

The Quantum Approximate Optimization Algorithm, or QAOA, is an algorithm used to solve optimization problems of second or lower degree using a quantum computer. Proposed by Edward Farhi, Jeffrey Goldstone and Sam Gutmann in 2014 [17], it suggests a new approach for quantum optimization. As the name implies, it is an approximate optimization algorithm, meaning its objective is not finding the optimal solution, but one close to the optimum.

The QAOA aims to find a solution z to a given cost function $C(z)$ such that it evaluates to a value close to the optimum of $C(z)$. $C(z)$ maps to the real numbers $\mathbb{R}$ [18], and z is comprised of n binary variables $z_0, \dots, z_{n-1}$. C(z) is a function made up of m clauses C_α, each incorporating z and of second degree at most. This is essentially a Quadratic Unconstrained Binary Optimization (QUBO) problem. One simple example would be the following:

$$C(z) = 2z_0 \cdot 5z_1 + 3z_1 \cdot 1z_2 \tag{2.10}$$

where $z = z_0z_1z_2$. Here, the number m of clauses (addends) is 2. The objective is to find a z that results in a value close to the optimum of $C(z)$.

The algorithm works as follows. First the qubits in the qubit string are initialized as $|0\rangle$ and then put into superposition by applying a Hadamard gate on each of them. Then the cost function C(Z) is constructed by substituting each binary variable z_i in $C(z)$ by the Pauli-Z matrix Z_i [19]. Instead of Z_i, sometimes the equivalent notation σ_i^z is used. This change means that C(Z) evaluates to a matrix, rather than a scalar, as C(z) does. If a z_i does not appear in a clause, it is represented by an identity gate. Applied to the example in 2.9, this would result in:

$$C(Z) = 2Z_0 \otimes 5Z_1 \otimes I + I \otimes 3Z_1 \otimes 1Z_2 \tag{2.11}$$

$$C(Z) = 2\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} \otimes 5\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} \otimes \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} + \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \otimes 3\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} \otimes 1\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} \tag{2.12}$$

Evaluating these tensor products results in matrices of dimension 2^n. C(Z) is also referred to as the cost hamiltonian or the problem hamiltonian. A hamiltonian, in the context of quantum computing, is an operator that describes the total energy within a quantum system. The optimal solution is essentially the ground state (state of lowest energy) of the cost hamiltonian.

Another relevant operator is B, also called the mixing operator or mixer hamiltonian. It is variable and can be adapted to the problem. The suggested definition for B in the original paper (which is commonly used) is:

$$B = \sum_{j=1}^{n} \sigma_j^x \tag{2.13}$$

Which is the sum of the individual σ_j^x – following the notation already used for σ_j^z, only using Pauli X gates instead of Z gates.

After this transformation, two unitary operators are applied to each qubit. This essentially rotates them around the Bloch sphere. The first operator is the unitary operator $U(C, \gamma)$ and the second operator is $U(B, \beta)$. The rotation angles are γ and β, respectively. The formulas, as defined in [17], are:

$$U(C, \gamma) = e^{-i\gamma C} = \prod_{\alpha=1}^{m} e^{-i\gamma C_\alpha} \tag{2.14}$$

$$U(B, \beta) = e^{-i\beta B} = \prod_{j=1}^{m} e^{-i\beta \sigma_j^x} \tag{2.15}$$

where $\gamma \in (0, 2\pi]$ and $\beta \in (0, \pi]$.

These operators can be applied in several iterations, where the parameter p determines the number of iterations. It therefore needs to take on values of at least 1, equivalent to applying $U(C, \gamma)$ and $U(B, \beta)$ once. The value of p can be tuned, and a higher value for p leads to monotonically better results, though it also takes longer to compute. Theoretically, a p value approaching infinity would always lead to the correct result [17].

This approach's advantage can be seen especially with problems that have a large number of possible solutions. In contrast to classical algorithms, the quantum algorithm can consider all qubits in superposition and, as such, all solutions simultaneously, which makes it more scalable.

When using the QAOA, the algorithm does not have to be built from the ground up. It can be found already implemented in quantum computing frameworks such as Qiskit, which will be used in this work. The algorithm takes the cost function for the optimization and then returns a result — the other parameters can be given optionally. The default value for p is generally 1. In the Qiskit implementation of the QAOA, if no values for γ and β are given, the algorithm will use random values [20]. During the optimization process, γ and β are then optimized using a classical optimizer, while the cost function's solution is evaluated using the quantum operations described above. This makes it a hybrid algorithm. The authors state that while one could simply evaluate the cost function for a large number of possible angle combinations, classical optimization is an efficient alternative, and they offer a calculation approach in their paper. Once the optimal values of γ and β have been determined, the result of the cost function evaluation is returned.

Options for optimizers in Qiskit include COBYLA (Constrained Optimization by Linear Approximation), introduced in [21] and the SPSA (Simultaneous Perturbation Stochastic Approximation), introduced in [22], which uses gradient approximation, an iterative process. These will not be described in detail here, as the focus lies on the quantum computing aspect and the classical optimzier is variable. The default optimizer for the QAOA in Qiskit is SPSA with a limit of 1000 iterations [20] (this parameter is referred to as *maxiter*).

2.4 Variational Quantum Eigensolver

The Variational Quantum Eigensolver (VQE) was first introduced in 2014, by Alberto Peruzzo et al. [23]. Like the QAOA, the VQE can be used to approximate the ground state of a cost function for optimization purposes (for solving a QUBO problem, for instance), though can also be used to approximate the ground state of molecules. The QAOA is commonly seen as a specialized version of the VQE, which will be briefly described here. The VQE is, like the QAOA, a hybrid algorithm used for optimization.

Like in the QAOA (see equation 2.12), the cost function of the given problem is turned into a hamiltonian. The VQE then uses an ansatz, ansatz being German for approach. First, an initial guess at a solution to the problem is chosen; a common approach is using a state called Hartree-Fock ground state. This initial guess is called $|\Phi\rangle_{ref}$ in the original paper. The ansatz is essentially the function that applies the so-called variational form $U(\theta)$ to $|\Phi\rangle_{ref}$ to get a trial wave function $|\psi\rangle$ (also called ansatz state [24]):

$$|\psi\rangle = U(\theta)\,|\Phi\rangle_{ref} \tag{2.16}$$

The ansatz is implemented via a circuit, called ansatz circuit, and then applied to the initial guess. This process can be repeated, optimizing the variational parameter θ using the classical optimizer, and with that the trial wave function. The choice of ansatz influences which states are explored in the optimization.

In the QAOA, the ansatz circuit is composed of the two unitary operators $U(C, \gamma)$ and $U(B, \beta)$, alternated and repeated p times. It is therefore influenced by the problem's cost function itself. In the VQE, this is not generally the case; the ansatz can be chosen more flexibly. For instance, the ansatz circuit might only include a rotation around the Y axis, in which case only the states on the corresponding circle on the Bloch sphere are considered (if the initial state has all qubits in $|0\rangle$). In the VQE, like in the QAOA, the number of repetitions of the ansatz can be specified (in the QAOA, this is the parameter p).

The VQE therefore allows for more detailed specification of the ansatz, compared to the QAOA. The choice of ansatz strongly influences the result and thus can be adapted to better suit a given problem, though choosing the most suitable ansatz is not a straightforward process.

2.5 Standard Use Cases

This work is based on a paper titled 'Ising formulations of routing optimization problems' [1] which presents formulations for different routing scenarios for the QAOA. These formulations are constructed such that they keep the number of qubits needed for the computations relatively low (the first formulation, as per the paper, undercutting the quantum version of the A* algorithm). Due to the hardware limitations outlined above and due to error rates, reduction of the number of required qubits is an important aspect when working with quantum algorithms. The paper additionally puts the formulations in the context of aviation and describes possible applications for them.

Among the four scenarios are: single-vehicle routing; time-dependent single-vehicle routing; the travelling salesperson problem; and collision-free multi-vehicle routing. The formulations of the paper will be implemented in this work, and will be introduced briefly in the following.

For easier understanding, the first section will include the mathematical formulation of the problem. The other formulations will be supplied in their respective implementation sections in chapter 4.

The original paper's example graph, used to visualize some of the concepts introduced, is shown in fig. 2.2 for reference.

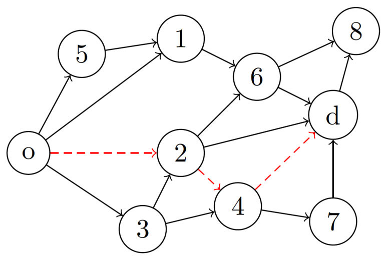


Figure 2.2: A directed graph G with 10 vertices, taken from [1]. One valid route has been marked in red.

2.5.1 Single-Vehicle Routing

The first scenario covers basic single-vehicle routing. The objective is to find the shortest path for one vehicle between two given vertices, meaning this (like the other scenarios) is a minimization problem. In the example graph in fig. 2.2, o represents the start vertex (the origin) and d the destination vertex. The given graph is weighted, though the edge weights are not shown in the figure.

The cost function is constructed by first including the cost of the path itself:

$$\sum_{(s,t)\in A} w(s,t)X_{(s,t)} \tag{2.17}$$

Here, the tuple (s,t) is made up of vertices s and t and represents an edge from s to t in the set A of edges. $w(s,t)$ returns the weight of the given edge. Finally, $X_{(s,t)}$ is a binary variable taking on a value of either 0 or 1, indicating whether edge (s,t) is part of the solution route (1) or not (0). X here is the string z of binary variables introduced in section 2.3, and will be optimized using the QAOA. In the optimization result, each qubit therefore represents one edge – if its value is 1, the edge is part of the solution route and its weight will be counted; if not, then the edge will not be passed and the corresponding term in the sum will be 0.

Additional penalty clauses ensure validity by increasing the cost in case a constraint is violated. The paper used the following notation for indicating whether a condition is fulfilled or not, essentially counting the edges in the set of edges A for which a given condition is true:

$$S(cond) := \sum_{\substack{(s,t)\in A\\ \text{fulfilling condition}\\ \text{cond}}} X_{(s,t)} \tag{2.18}$$

These constraints are listed in the following. The term in 2.19 ensures that there is exactly one outgoing edge from the start vertex. Any other value would not evaluate to zero and thus incur a penalty. The term in 2.20 guarantees that there will be no incoming edges at the start vertex. Both terms can not evaluate to a negative term, which is important – otherwise, penalties incurred from violating other constraints could theoretically be cancelled out.

$$P \cdot (S(s = o) - 1)^2 \tag{2.19}$$

$$P \cdot S(t = o) \tag{2.20}$$

For the destination node, the terms are similar; they merely need to be modified such that there is exactly one incoming edge and no outgoing edges.

The following constraint ensures that there is at most one outgoing edge from each vertex except for the start and destination vertices:

$$P \sum_{v \in V \setminus \{o,d\}} (S(s = v) - 1) \cdot S(s = v) \tag{2.21}$$

Substituting s with t results in the same constraint applied to incoming edges. Finally, for each vertex except the start and destination vertices, its number of incoming edges needs to equal the number of its outgoing edges[4]:

$$P \sum_{v \in V \setminus \{o,d\}} (S(t = v) - S(s = v))^2 \tag{2.22}$$

Since the optimized cost includes the penalty terms, the actual cost of the route is not returned, though it can easily be calculated using the qubit result.

2.5.2 Time-Dependent Single-Vehicle Routing

The second scenario, time-dependent single-vehicle routing, takes a different approach. Given are a graph G; a start vertex; and a destination vertex. Here, in order to incorporate the chronological aspect, some pre-processing is needed in order to create a modified graph G' using the original graph G as a basis. In G', the concept of time slices is introduced.

[4]In the paper, the difference is not squared; in the implementation of one of the authors, it is. Since squaring it ensures that the term does not evaluate to a negative value, this is adopted here.

The first slice, c_0, contains only the start vertex, and any slice c_n contains any vertex reachable in n steps, relative to the start vertex. It follows that G' is equal to or larger than G in vertex count, since a given vertex can conceivably be reached in different amounts of steps (see vertex 2 in the following figures).

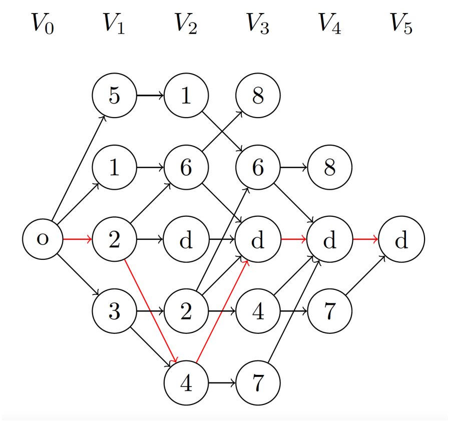


Figure 2.3: Graph G' built from G in fig. 2.2, with its time slices. The same path as in fig. 2.2 has been marked in red. V_i represents the set of vertices in slice c_i. [1]

A qubit in this scenario represents a vertex, not an edge. This means any edge needs to be identifiable by its start and destination vertices if the result is to be unambiguous. This, in turn, necessitates that in G' there only be at most one connection between two vertices for each direction. In case there is more than one edge with identical start and destination vertex in G, either one only of the vertices is chosen for G' (one might choose to only keep the cheapest edge), or another vertex needs to be added into G' as a stand-in for the original destination of the edge.

That way, providing a start and destination vertex allows definite identification of an edge. Its cost can then be considered in the cost function. If the solution path passes two vertices in succession that are not connected by an edge, a penalty is incurred.

The validity of the solution can then be ensured by again adding penalty clauses. These disincentivize any solution that does not pass exactly one vertex in any slice. To ensure the validity of solutions that reach the destination sooner than in the last slice, each slice after the first possible destination slice also contains the destination node. A supplementary edge is added to ensure the cost does not increase: it leads from the destination to itself, with a cost of 0. That way, in the routes that arrive earlier, the last vertices are always destination vertices. This can be seen illustrated with the marked path in figure 2.3.

Since the number of qubits equals the number of vertices in G', effective pre-processing can greatly decrease the size of the problem and with it the computing time.

2.5.3 Travelling Salesperson Problem

The third scenario is considers the travelling salesperson problem. Again, the original graph G is used to create G'. The time slices are constructed from G, but can be further reduced due to the fact that any slice other than the first and last can not contain the start vertex, as per the definition of the TSP.

In addition to computing the route cost and ensuring the route passes exactly one vertex in each slice, another penalty clause is added to enforce that each vertex (except for the start vertex) is passed exactly once.

2.5.4 Collision-Free Multi-Vehicle Routing

The fourth and last scenario covers collision-free multi-vehicle routing. It is similar to the second scenario, time-dependent single-vehicle routing. In this scenario, the goal is to find routes for each vehicle such that no two vehicles pass the same vertex at the same time (that is, in the same time slice). The vehicles have the same start point, but different destinations. Occupying the start vertex at the same time does not constitute a collision. This being a multi-vehicle problem where the vehicles have different destinations, each vehicle needs a custom graph G'. The cost function is then – like that of scenario 2 – comprised of the routes' costs, as well as penalty terms to ensure that each vehicle's route passes all time slices exactly once, and an additional clause penalizes any occurrence of vehicles being at the same vertex at the same time.

Chapter 3

Methodology

In this section, the approach for this paper will be outlined. The goal is to implement the scenarios in [1], with the exception of the first scenario, which has already been implemented by one of the authors. This first scenario will, however, be included in the tests. Based on the results of these scenarios, a more specific use case for passenger routing, particular to the aviation industry, will be formulated and implemented. The cases in the paper are roughly ordered according to ascending complexity and will be implemented in order; then the passenger routing use case will follow. Since it is built on the basis of the scenarios in the paper, these will be introduced first, in chapter 4. Based on their implementation and results, the passenger routing use case will then follow in chapter 5.

3.1 Implementation Details

The implementations will be realized in Python (version 3.7.3), in the form of Jupyter Notebooks. These allow for segmenting code and including written explanations and headings outside of code comments. The quantum computing libraries used are IBM's *qiskit* (v0.29.0) [25] and the Qiskit optimization library *qiskit_optimization* (v0.1.0). This framework allows access to IBM's quantum computers and simulators as well as providing tools for quantum optimization.

Additional libraries used include:

- *sympy* (v1.6.2) [26], a library for symbolic mathematics and the evaluation thereof,
- *pandas* (v1.1.3) [27], a library for data handling,

- *matplotlib* (v3.3.2) [28], a library for data visualization, and
- *networkx* (v.2.5) [29], a library for network creation, manipulation, and visualization.

For the purposes of this paper, no real quantum computers will be used for testing. This has several reasons: for one, error rates are still prevalent, with a high enough percentage that they have a significant effect on the results. Error minimization is its own field and not the focus of this paper, and so using a simulator is more suitable here. For another, usage of IBM's real quantum systems works via a queue system where jobs are submitted to the queue. For the QAOA, many separate circuits need to be submitted, which takes quite a long time overall.

The simulator used will be Qiskit's *qasm-simulator*, a general-purpose simulator. The *qasm-simulator* allows for several simulation methods, which per default are chosen automatically (based on the circuit that is executed) [30]. The simulator's upper limit for qubit numbers is roughly 32 (depending on the circuit), though the choice of simulator can also influence this limit. The simulator will be executed locally, on a notebook computer. The machine used for testing has 8GB of RAM and a 2,3 GHz Dual-Core Intel Core i5 processor.

Each of the use cases will be encoded as a QUBO, which will then be solved by the QAOA. Quadratic means that the problems can be at the most of second degree in respect to the optimized variables; unconstrained means that only the function itself is given to the QAOA. Any constraints, such as $X_3 = 0$, must be encoded within the function as a term and can not be added separately.

The validity of the quantum implementation will be ensured by cross-checking the results with those of a classical optimizer, IBM's CplexOptimizer, which can be accessed via the *qiskit_optimization* library. The problems covered are small enough that a classical optimization is easily realized.

3.2 Standard Use Cases and Passenger Routing Use Case

The standard use cases have been described in section 2.4; their mathematical formulations will follow in their implementation chapter. What remains is to outline the passenger routing use case.

This scenario models a passenger routing problem: an airline's flight is cancelled at short notice; the passengers then need to be rerouted by the airline to their planned destination. The input for this problem is a list of passengers with information on their cancelled flight, as well as a number of available flights that can be taken. For each passenger, the airline then needs to find an alternative route such that no flight is overbooked, each passenger can get their connecting flights, and the overall cost is minimized.

There are three different optimization metrics to choose from. One is the monetary cost, that is, the cost of all flights taken in the solution scenario. The second metric supplements this by including a simplified model of compensation payments (based on the 2004 European Air Passenger Rights Regulation) being paid by the airline, depending on factors like the difference between planned and actual arrival time. Instead of minimizing the flight cost, this metric optimizes flight and compensation cost together. The third metric aims to minimize the overall flight duration across all passengers, where 'overall flight duration' is defined as the length of time between the originally scheduled departure and the actual arrival.

The passenger data, then, needs to include information on the passengers' original flights (destination, departure time, distance, etc.). Flight data contains information like start and destination, flight cost, capacity, etc. The exact extent of the needed data will be outlined in chapter 5. Using that data, the goal then is to allocate flights to the passengers in such a way that

- the overall cost of the flights is minimized,
- no flight is overbooked, and
- every passenger can get their connecting flights; that is, the time between connecting flights must be above a certain threshold.

Testing of the implementations – both the passenger routing use case and the standard use cases – will mainly be done with specifically constructed test cases. This helps to identify edge case behavior and makes error detection easier. One test graph will be based on the example in [1] which has been introduced in section 2.4 of this work. In order to turn this graph into a weighted one, edge weights will be defined and shown in chapter 4. Other test cases will be constructed based on the different scenarios, such as the travelling salesperson problem or the passenger routing use case, and will be introduced alongside their relevant scenarios.

Chapter 4

Implementation of Standard Use Cases

This chapter will describe the implementation of the routing scenarios in [1]. For ease of understanding, first the symbols that will be used throughout the implementations will be defined, though some have already been touched upon in section 2.4.

- K is the set of vehicles/passengers. Multi-vehicle routing (section 4.4) covers vehicles; the passenger routing use case (chapter 5) covers passengers. In these scenarios, vehicles and passengers are equivalent – both are entities that are assigned routes. A passenger routing scenario can thus be adapted from a multi-vehicle routing scenario. K is the set of vehicles/passengers k_i with index i where $i \in \{0, 1, \ldots, |K| - 1\}$.

- C is the set of time slices of a given graph as delineated in the paper (and in section 2.4). c_i represents a time slice c_i where $i \in \{0, 1, \ldots, |C| - 1\}$. C_k is the set of slices of a given k.

- V is the set of vertices in a given graph, containing vertices v_i; $i \in \{0, 1, \ldots, |V| - 1\}$. V_c is the set of vertices within a given slice c.

- E is the set of edges in a given graph, with edges e_i where $i \in \{0, 1, \ldots, |E| - 1\}$. E_c is the set of edges within a given slice c. For multi-vehicle or multi-passenger formulations, $E_{c,k}$ is the set of edges within a slice c belonging to a passenger/vehicle k.

- P represents the penalty value used in the function polynomial. This term is used to raise the negative impact of any constraint violations. As per the paper, P should generally be higher than the sum of the cost of all edges in the graph.

- X represents the binary variables that constitute the result; that is, in the calculation, every X_i is linked to one qubit and takes on its result state. As such, any X_i can only take on values of either 0 or 1. The highest index, i_{max}, is dependent on the problem size. $X_{c,v}$ is the binary variable of a vertex v in a slice c. Similarly, the same applies to $X_{k,c,v}$ for a given passenger k, and for $X_{k,c,e}$ for a given edge e.

For any set, the highest index will also be referred to by max – the highest index in the set of time slices C is $c_{max} = |C| - 1$, for example.

As mentioned above, the graph from [1] will be used for testing. Since the implementations cover scenarios where edge weights are relevant, these have been added (see fig. 4.1). The weighted graph will be referred to as G1 and will be used throughout this chapter to illustrate preprocessing and other implementation aspects.

The start vertex has been renamed from o to 0, and the destination vertex from d to 9, since this graph will not only be used for single-vehicle routing, but for multi-vehicle routing as well. This makes the designation of a single destination vertex d misleading.

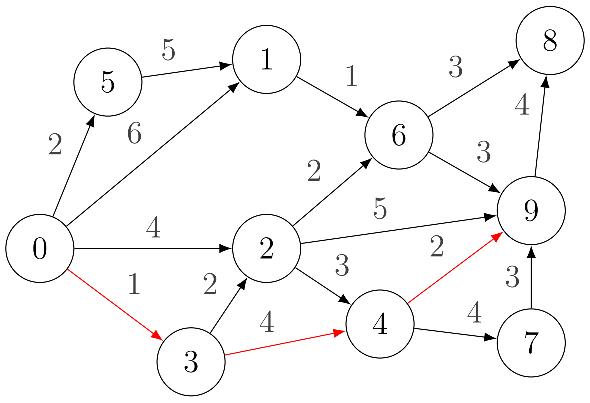


Figure 4.1: Weighted example graph G1; edge weights are marked. The optimal solution from vertex 0 to vertex 9 (0–3–4–9) with cost 7 is marked in red.

In the results of the test runs, the penalty value P, the QAOA parameter p, and the classical optimizer will be specified, with the limit of iterations for the optimizer in parentheses after it.

4.1 Single-Vehicle Routing

This scenario covers basic single-vehicle routing. Given a start vertex and a destination vertex, the objective is to find the shortest path that connects them. Since this use case has already been implemented [31] and, for the given example, consistently returned the optimal solution, it was not a point of focus in this work; it was only modified to update the code, since the Qiskit library has been restructured (Qiskit Aqua is deprecated as of April 2021 [32]), and to test a second graph. The relevant cost function has been explained in section 2.4.1.

The implementation has been tested with the graph of the original paper, G1 (see fig. 4.1), as well as with the smaller graph used in the implementation (see fig. 4.2), which will be referred to as G2. This smaller graph has four vertices and five edges. It therefore requires five qubits, where each represents one edge. The larger graph has 10 vertices and 17 edges and therefore requires 17 qubits.

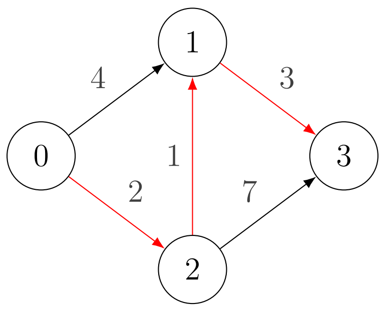


Figure 4.2: Graph G2 used for testing the single-vehicle routing implementation. The optimal solution from vertex 0 to vertex 3 (0–2–1–3) with cost 6 is marked in red.

An edge that leads from vertex A to vertex B can be expressed as a tuple (A,B). The predetermined indexing of edges in G2 (as per the implementation) can then be described as:

- 0: (0,2); 1: (0,1); 2: (2,1); 3: (2,3); 4: (1,3).

The indexing determines how the result is to be interpreted. In the result string, which is made up of zeroes and ones, the digit with index i indicates whether edge e_i is traversed or not – 1 if yes, 0 if no. In this case, the correct qubit result is 10101 – the edges with index 0, 2, and 4 are part of the shortest solution route.

For the second graph, edges have been numbered according to ascending order of the first vertex (start) and then the second vertex (end) of the tuple:

- 0: (0,1); 1: (0,2); 2: (0,3); 3: (0,5); 4: (1,6); 5: (2,4), etc.

The optimal solution for graph G1 (marked in fig. 4.1), would be 001 000 000 101 000 00 (added spaces are for legibility).

Testing with the QAOA returned different results for the two graphs. For the smaller graph G2, testing with $p = 1$ and $p = 5$, a penalty value of $P = 50$, and the SPSA optimizer across 100 iterations consistently returned the optimal result.

	valid results	optimal results	mean optimization result
$P = 50$; $p = 1$; SPSA (500)	100%	96%	6.13
$P = 50$; $p = 5$; SPSA (500)	100%	96%	6.03

Table 4.1: Results for graph G2 across 100 iterations, with $p = 1$ and $p = 5$, respectively.

For the larger graph G1, different parameter configurations were tested, with p in {1,5,15}, P in {10, 20, 50, 250}, and with the classical optimizers COBYLA and SPSA with different values for their parameter *maxiter*, which determines the maximum number of iterations they perform. Due to computing times in the range of 30-40 minutes per iteration, the test cases run across 10 iterations each. A selection of the results can be seen in table 4.2.

	valid results	optimal results	mean optimization result
$P = 250$; $p = 3$; COBYLA (500)	40%	20%	307.3
$P = 20$; $p = 3$; COBYLA (500)	20%	10%	40.6
$P = 10$; $p = 3$; SPSA (500)	20%	0%	28.7
$P = 25$; $p = 5$; COBYLA (1000)	20%	0%	59.2
$P = 50$; $p = 15$; SPSA (200)	20%	10%	88.9

Table 4.2: Results for graph G1 for different test parameters across 10 iterations each. The numbers in parentheses specify the limit of iterations for the classical optimizer.

Throughout the testing process, different configurations were tested. In addition to the QAOA, the VQE algorithm was tested alongside it, and though 10-iteration runs are not necessarily representative, in repeated tests the VQE produced better and more consistent results. Therefore, the approach was changed – instead of using the QAOA, the VQE is used to run the tests for the use cases. In the following, test cases will (due to time constraints – depending on the parameters, some test runs can take up to several hours) consist of 10 runs, in order to validate the results.

Running a test for G1 with SPSA as the classical optimizer, $P = 10$, and using an ansatz with 2 repetitions lead to the following results:

	optimization cost	optimization result	chosen edges
1	7	00100000010100000	(0,3) (3,4) (4,9)
2	8	00100001100000000	(0,3) (2,9) (3,2)
3	8	00100010100000100	(0,3) (2,6) (3,2) (6,9)
4	9	01000010000000100	(0,2) (2,6) (6,9)
5	9	01000010000000100	(0,2) (2,6) (6,9)
6	11	00011000000010100	(0,5) (1,6) (5,1) (6,9)
7	11	00011000000010100	(0,5) (1,6) (5,1) (6,9)
8	12	00100000011000010	(0,3) (3,4) (4,7) (7,9)
9	21	00100000000000000	(0,3)
10	22	00100000000000000	(0,3)

Table 4.3: Results for graph G1 across 10 iterations using the VQE. The classical optimizer used is SPSA (200), $P = 10$, using an ansatz with 2 repetitions. Each 1 in the optimization result represents one edge in the "chosen edges" column, in order (left to right). The optimal result has been returned once.

The results are better than the ones seen in table 4.2, even compared to the last case, where $p = 15$ (here, the percentage of admissible results is 50, there it was 20). In contrast, the results of QAOA and VQE for the smaller graph G2 show little to no difference: Running a test for G2 with SPSA (200) as the classical optimizer, $P = 50$, and using an ansatz with 5 repetitions lead to the following result:

	optimization cost	optimization result	path
1	6	10101	(0,2) (2,1) (1,3)
2	6	10101	(0,2) (2,1) (1,3)
3	6	10101	(0,2) (2,1) (1,3)
4	6	10101	(0,2) (2,1) (1,3)
5	6	10101	(0,2) (2,1) (1,3)
6	6	10101	(0,2) (2,1) (1,3)
7	6	10101	(0,2) (2,1) (1,3)
8	6	10101	(0,2) (2,1) (1,3)
9	6	10101	(0,2) (2,1) (1,3)
10	6	10101	(0,2) (2,1) (1,3)

Table 4.4: Results for graph G2 across 10 iterations using the VQE. The optimizer used is SPSA, $P = 50$, using an ansatz with 5 repetitions. All results show the optimal solution.

As can be seen, the VQE results for G2 are consistently valid, and each result returned the optimal value. The QAOA results (see table 4.1) show similar results. The difference between the performance of the algorithms with regard to the two graphs G1 and G2 might be due to the different problem sizes of 17 and 6 qubits, respectively.

4.2 Time-Dependent Phenomena

The second scenario constitutes a different case than the first one. While this scenario, too, covers single-vehicle routing, this is done in a manner that allows for including time-related aspects. This formulation will be the basis for the TSP and collision-free multi-vehicle routing, where this model allows for checking if two vehicles occupy the same vertex at the same time. It is constructed differently than the previous one in that the binary variables (and therefore qubits) represent vertices, not edges. The result will give information not only on which vertices are passed, but also when – context that is not available in the formulation of the previous problem. In this scenario, other aspects can be factored in, such as varying edge costs depending on when the edge is passed, for instance.

However, the optimal solution for G1 for the previous use case and for the current one show the same path. The goal is the same, and so are the inputs: given a start vertex, a destination vertex, and a graph, the objective is to find the shortest path from start to destination.

4.2.1 Implementation

In this scenario, the inputs are start and destination vertex, the penalty value P, and a graph. The first issue, then, is the representation of the given graph. In the implementation of scenario 1, edges were grouped according to their connected vertices. This was suited to the scenario and fit the cost function, but is a little complicated to modify in this form.

Since the possibilities for defining a graph are varied, a simple array of edges was chosen for this purpose. Represented as tuples, they contain their respective start and destination vertices as well as the edge cost. As an example, the edge list for G1 would start as follows:

- (0,1,6), (0,2,4), (0,3,1), (0,5,2), (1,6,1), etc.

V can then be assembled by extracting all unique vertices. Describing a graph solely on the basis of its edges means that isolated vertices (vertices that, by definition, are not connected to another vertex) will not be included in V. This is intentional; these vertices have no relevance to a solvable routing problem unless they constitute both start and destination, in which case the solution is trivial. That case is an edge case, though it is nevertheless covered in this implementation. For any other case, these vertices cannot be traversed. As such, this approach ensures that (in non-trivial solutions) they will appear neither in the constructed graph G' nor, therefore, in the solution.

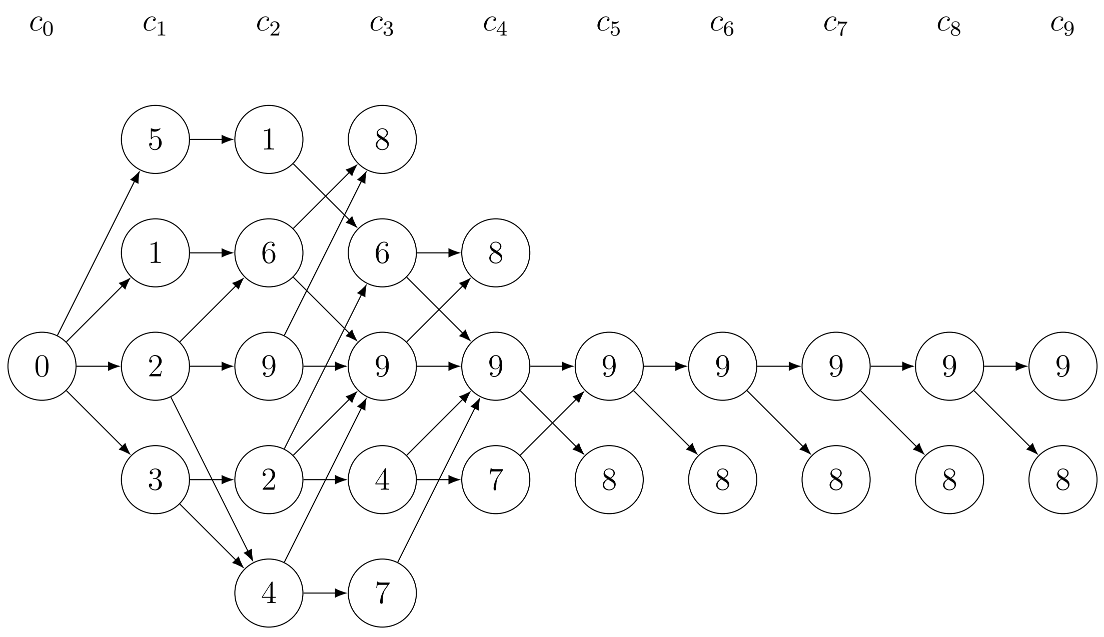
For this scenario, some preprocessing is necessary. First, as outlined in the paper, an additional, supplementary edge is added. This edge has a weight of 0 and connects the destination to itself. The reason for this has been brought up in 2.4 – it keeps solutions with less than the maximal number of possible steps (edges) from being penalized by 'filling the route up' with these supplementary edges. This will not add to the overall cost and still ensures that one vertex is passed in each time slice. An example can be seen in fig. 2.2, where the last two edges marked in red are supplementary edges.

Then G' itself is constructed. It is conceptualized as an array of arrays (a two-dimensional array). Within the main array, each sub-array represents a time slice. This two-dimensional array is referred to as sliced graph (*sg*). It is important to note that G' and *sg* are not equivalent. The structure *sg* only contains vertex data and no edge connections between vertices; to reconstruct G' from *sg*, edge information is required. To see an illustration of the difference, refer to figure 4.3 where both the graph G1' and its corresponding *sg* can be seen.

sg is initialized as an empty array within an empty array. The first element of this inner array (the first subarray) is set to be the start vertex. No additional vertex is added to c_0. Any subsequent slice c_i contains any vertex reachable from the start vertex in i steps, or, equivalently, any vertex directly reachable from c_{i-1}. This allows for isolated vertices to be the start vertex; since they have no outgoing edges, *sg* will simply consist of one time slice containing a single vertex. For G1, this would look something like the graph shown in fig 4.3, where the subarray with index 0 contains the elements of slice c_0, the subarray with index 1 represents c_1, etc.

It should be noted that in the diagram, the order of vertices in the graph does not correspond to the array order, it is changed in order to have a clear visualization. The array order is the one that the program returns; it is important for result interpretation, which is why the elements within subarrays are not ordered. To keep graphs like this one from going on infinitely, the upper limit to the number of time slices is $|V|$ (the number of vertices in V), as stated in the paper. This is because the longest G' for any given acyclical graph G can be at most of that length (i.e., there would be one vertex per slice). Cyclical graphs would in theory be of infinite length, though that is neither required nor practical for this scenario, which is why the same limit is imposed on them.

This example illustrates that G' can contain redundant information – the last five



c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
[[0],	[1,2,3,5],	[6,4,9,2,1],	[6,4,9,7,8],	[7,8,9],	[8,9],	[9,8],	[9,8],	[9,8],	[9,8]]

Figure 4.3: Initial state of G1' and the corresponding *sg*.

slices are identical. Since every vertex in G' will be represented by one qubit, reducing the number of vertices can, sometimes drastically, reduce the problem size. As such, trimming G' is an important part of preprocessing.

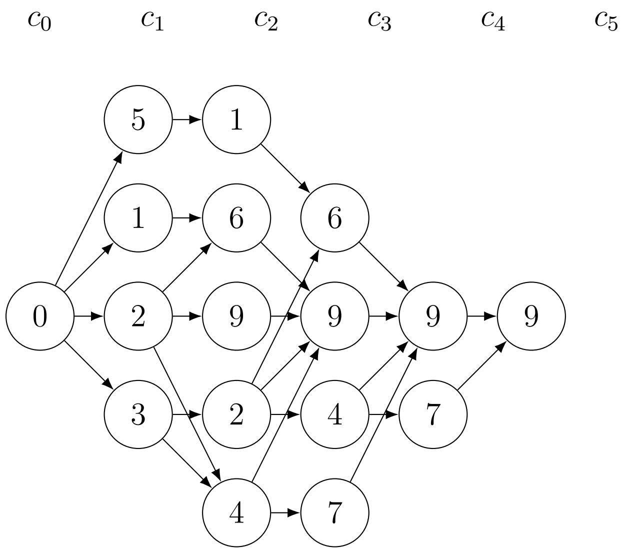
One trivial step is the removal of any vertex in the last slice except for the destination vertex, since any such vertex can not be part of a valid solution. Vertices in the previous slice that only lead to a now-removed vertex or to no vertex at all will likewise not lead to the destination and instead are designated 'dead ends.' Traversing G' backwards, any dead ends can thus be removed. Applying this to our example leads to the following state:

c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
[[0],	[1,2,3,5],	[6,4,9,2,1],	[6,4,9,7],	[7,9],	[9],	[9],	[9],	[9],	[9]]

Figure 4.4: Time slices of G1 after removal of dead ends, i.e. vertices that cannot be part of a valid solution.

A consequence of trimming is that G' might be left with a number of trailing slices only containing the destination vertex, as happened here, due to the supplementary edge.

All but one of them are removed, since they, too, are superfluous. In this case, these steps have reduced the initial number of vertices in G1' from 28 to 17.



c_0	c_1	c_2	c_3	c_4	c_5
[[0],	[1,2,3,5],	[6,4,9,2,1],	[6,4,9,7],	[7,9],	[9]]

Figure 4.5: G1' after removal of surplus slices

Then a new two-dimensional array is created. It is a copy of the sliced graph and identical in structure, but instead of the vertex labels, the subarrays contain new vertex indices. It is important for each vertex to be unambiguously identifiable in order to interpret the optimization result, which is why the labels do not suffice for this – vertices with the same label can appear several times within a G'. Therefore, all vertices are numbered separately. The start vertex gets index 0 by default; iterating over each time slice, this so-called numbered sliced graph (nsg) assigns an index to each vertex (see fig. 4.6). If a vertex is present in several time slices, each occurrence will be indexed separately. This makes referring to individual vertices in G' possible.

It is important to note that two sg for a given graph can differ in the ordering of vertices within the slices. As long as the qubit result is evaluated according to the vertex order in the corresponding sg, that order does not influence the result.

$$\begin{array}{cccccc} c_0 & c_1 & c_2 & c_3 & c_4 & c_5 \\ [[0], & [1,2,3,4], & [5,6,7,8,9], & [10,11,12,13], & [14,15], & [16]] \end{array}$$

Figure 4.6: The indexed pendant to *sg*, the *numbered sliced graph* (*nsg*). It contains the indices of the vertices in G1'. Referring to vertex 3 in slice c_2 in G1' returns its label; in the respective *nsg*, it returns its index.

Next, a graph matrix is created (see figure 4.7). It has dimension of $|V|$ * $|V|$. Any entry at m, n contains the edges with start m, destination n. Every entry in this matrix is an array and can therefore contain several edges. However, instead of using a three-dimensional array, a *pandas* data frame is used. It functions as a matrix, but allows for column and row labels. This means that it is not necessary for the vertex labels to be integers in the range 0,1,..., $|V| - 1$. Instead, the labels can take on any value (integers, characters, etc.), as long as all vertices in the original graph G have unique labels.

Of course, this matrix contains redundant information. If start and destination of an edge can be determined by looking at its position within the matrix, it is not necessary to store the whole edge, the cost alone would suffice. This is true in this case, where the edge cost is the only additional information within an edge, but in use cases where an edge holds additional information (and where two vertices can have identical start, destination, and cost), it is more convenient to simply store the edge. In cases where edges have a unique identifier key, this would be stored here instead of the edges themselves.

Here, an edge e_i with start m and destination n is sorted into column m and row n (see fig. 4.3). Any outgoing edges of a vertex can be found in its column, any incoming ones in its row.

As a next step, an edge cost matrix is built (see fig 4.8). It is similar in structure, but each entry contains a number instead of an array. At a given entry in column m, row n:

- if there is no connecting edge with start m and destination n, the entry is P,
- if there is one connecting edge, then the entry is the cost of that edge,
- if there are several connecting edges, the entry is the cost of the cheapest edge. This selection method can be changed, or one can create duplicate vertices with different labels as described in section 2.4 in order to include all edges.

	0	1	2	3	4	5	6	7	8	9
0	[]	[]	[]	[]	[]	[]	[]	[]	[]	[]
1	[(0,1,6)]	[]	[]	[]	[]	[(5,1,5)]	[]	[]	[]	[]
2	[(0,2,4)]	[]	[]	[(3,2,2)]	[]	[]	[]	[]	[]	[]
3	[(0,3,1)]	[]	[]	[]	[]	[]	[]	[]	[]	[]
4	[]	[]	[(2,4,3)]	[(3,4,4)]	[]	[]	[]	[]	[]	[]
5	[(0,5,2)]	[]	[]	[]	[]	[]	[]	[]	[]	[]
6	[]	[(1,6,1)]	[(2,6,2)]	[]	[]	[]	[]	[]	[]	[]
7	[]	[]	[]	[]	[(4,7,4)]	[]	[]	[]	[]	[]
8	[]	[]	[]	[]	[]	[]	[(6,8,3)]	[]	[]	[(9,8,4)]
9	[]	[]	[(2,9,5)]	[]	[(4,9,2)]	[]	[(6,9,3)]	[(7,9,3)]	[]	[(9,9,0)]

Figure 4.7: Edge matrix for graph G1. Each tuple represents an edge that goes from the first value in the tuple to the second. The third value is the edge cost, which in this case remains the same, independent of when an edge is passed.

	0	1	2	3	4	5	6	7	8	9
0	P	P	P	P	P	P	P	P	P	P
1	**6**	P	P	P	P	**5**	P	P	P	P
2	**4**	P	P	**2**	P	P	P	P	P	P
3	**1**	P	P	P	P	P	P	P	P	P
4	P	P	**3**	**4**	P	P	P	P	P	P
5	**2**	P	P	P	P	P	P	P	P	P
6	P	**1**	**2**	P	P	P	P	P	P	P
7	P	P	P	P	**4**	P	P	P	P	P
8	P	P	P	P	P	P	**3**	P	P	**4**
9	P	P	**5**	P	**2**	P	**3**	**3**	P	**0**

Figure 4.8: Edge cost matrix for graph G1. An entry in column m and row n shows the edge cost from m to n. If not edge exists, the edge cost is equal to P.

This edge cost matrix is not strictly necessary; one could just as much use the edge matrix. However, it is convenient and makes it easier to modify costs based on optional modifiers – the edges themselves would then remain unchanged. In cases where one would want to introduce changes in cost based on when a vertex is reached (i.e., make costs time-dependent), one would construct a matrix with dimensions equal to the number of total vertices in G'. That way, vertices that were the same in the original G would have different entries in the cost matrix and their cost could then be adapted depending on when they are passed and where they are reached from.

The next step is assembling the cost function. Since it has already been defined in

[1], it only needs to be implemented in sympy. The function will be split up and the individual addends will be explained separately for easier understanding.

$$P \sum_{c=0}^{c_{max}} \left(\sum_{v \in V_c} X_{c,v} - 1 \right)^2 \tag{4.1}$$

This above term iterates over the slices and then the vertices in the slices and ensures that in any given slice, exactly one vertex is traversed. The binary variable X for a given vertex is 1 if it is traversed, meaning the sum of binary variables of vertices v in a given slice nsg_{c_i} needs to be exactly 1. In order to ensure that the term cannot turn negative in value, it is squared. Then it is multiplied with P – if the number of traversed vertices in the slice is not equal to 1, the factor will evaluate to a value >0 and thus a penalty will be incurred.

$$\sum_{c=0}^{c_{max}-1} \sum_{\substack{v' \in V_{c+1} \\ v \in V_c}} X_{c+1,v'} \cdot X_{c,v} \cdot d_{v,v'} \tag{4.2}$$

This second term serves to calculate the total cost of the route. To determine it, one needs to iterate over the vertices in a slice c_i and the vertices in the next slice, c_{i+1}. Any combination of cross-slice vertex pairs could possibly be part of the route. If two given vertices are, then their corresponding binary variables are 1, meaning that the product of both of the binary variables and their edge cost d can only evaluate to a non-zero value if both vertices are traversed. The function d, given two vertices, returns P if no valid edge exists, and the edge weight otherwise. The value of d can be determined from the edge cost matrix (see fig. 4.8).

The complete cost function can then be assembled:

$$P \sum_{c=0}^{c_{max}} \left(\sum_{v \in V_c} X_{c,v} - 1 \right)^2 + \sum_{c=0}^{c_{max}-1} \sum_{\substack{v' \in V_{c+1} \\ v \in V_c}} X_{c+1,v'} \cdot X_{c,v} \cdot d_{v,v'} \tag{4.3}$$

Before evaluating, one more step remains. The binary variables of vertices that are already known to be part of the route can be set to 1. In this case, the first and last binary variables are known to be the start and destination vertices and can be set to 1. If there were other vertices in the first or last time slice, they would be set to 0. However, these vertices have already been removed when constructing G', this is therefore not necessary.

The cost function is then transferred to the Quantum Approximate Optimization Algorithm to solve. The result is evaluated according to nsg and sg, and a solution text and the cost of the route can then be returned.

4.2.2 Results

The graphs used for testing were G1 (which has been used as an example in this section) and a different custom graph that will be called G3 (see fig. 4.10). G1 will be shown again for easier access.

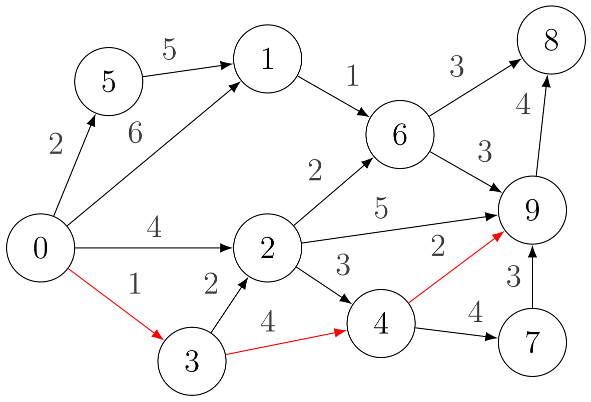


Figure 4.9: Weighted example graph G1. The optimal solution from vertex 0 to vertex 9 (0–3–4–9) with cost 7 is marked in red.

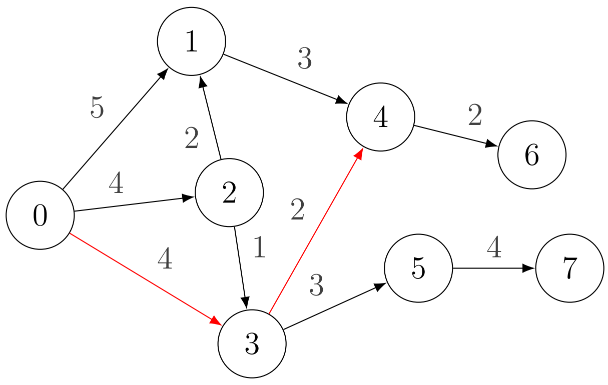
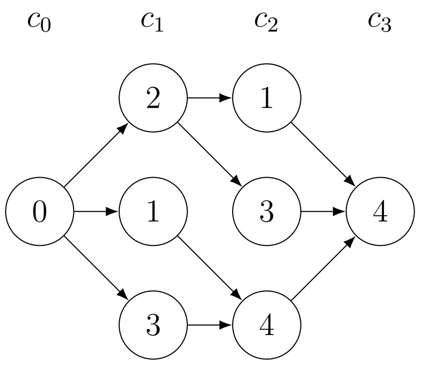


Figure 4.10: Graph G3, used for testing alongside G1. The optimal solution from vertex 0 to vertex 4 (0–3–4) with cost 6 is marked in red.

The sliced graph of G1 has a total of 17 vertices, as seen in fig. 4.5. The optimal solution is shown in fig. 4.9. The corresponding qubit sequence, according to G1's sg and nsg (see fig. 4.5 and 4.6), is 1-0010-01000-0010-01-1 (time slices separated by dashes for legibility).

For graph G3, *sg* and *nsg* are as follows:



[[0], [1, 2, 3], [4, 1, 3], [4]]

[[0], [1, 2, 3], [4, 5, 6], [7]]

Figure 4.11: G3' and its *sg* and *nsg*, trimmed.

Here, the optimal result would be 1-001-100-1.

For G3, using the VQE with SPSA, a penalty of 500, and an ansatz with 5 repetitions resulted in the optimal result being returned in all iterations (see table 4.6). The time slice sections in the optimization results have again been separated by a dash for legibility. Their interpretation can be seen in the column 'path'.

	optimization cost	optimization result	path
1	6	1-001-100-1	0-3-4-4
2	6	1-001-100-1	0-3-4-4
3	6	1-001-100-1	0-3-4-4
4	6	1-001-100-1	0-3-4-4
5	6	1-001-100-1	0-3-4-4
6	6	1-001-100-1	0-3-4-4
7	6	1-001-100-1	0-3-4-4
8	6	1-001-100-1	0-3-4-4
9	6	1-001-100-1	0-3-4-4
10	6	1-001-100-1	0-3-4-4

Table 4.5: Results for graph G3 across 10 iterations using the VQE. The optimizer used is SPSA (500), $P = 50$, using an ansatz with 5 repetitions. All solutions are optimal.

In order to ensure that the suboptimal performance of the QAOA is not limited to the first use case, this scenario was tested graph G1 using the QAOA – the cost of the optimal solution should again be 7. The parameter p was set to 25 (which generally should lead to better results than testing with low values for p), P was set to 50, and SPSA was used for the classical optimizer. The result of 10 iterations can be seen in table 4.3 – the result with the lowest optimization cost still incurred $1P$ – it did not return an admissible result.

	optimization cost	optimization result	path
1	51	1-0010-00100-0010-01-1	0-3-9-9-9-9
2	54	1-0010-00000-0001-01-1	0-3-none-7-9-9
3	54	1-0010-10000-0010-01-1	0-3-6-9-9-9
4	59	1-0100-10000-1000-01-1	0-2-6-6-9-9
5	102	1-0000-00001-0100-01-1	0-none-1-4-9-9
6	103	1-0000-00000-0001-01-1	0-none-none-7-9-9
7	103	1-0000-10000-1000-01-1	0-none-6-6-9-9
8	104	1-0010-00100-1000-01-1	0-3-9-6-9-9
9	111	1-0010-00010-0010-11-1	0-3-2-9-7;9-9
10	115	1-1000-10010-0010-01-1	0-1-6-9-9-9

Table 4.6: Results for graph G1 across 10 iterations using the QAOA. The optimizer used is SPSA ($maxiter = 250$), $P = 50$, $p = 25$. No admissible results have been returned.

Given that the best result in this set is still not admissible, the problem was again tested with the VQE in order to compare the results. Testing G1 using the VQE with SPSA (200) and a penalty of 15 using an ansatz with 8 repetitions resulted in the following:

	optimization cost	optimization result	path
1	7	1-0010-01000-0010-01-1	0-3-4-9-9-9
2	8	1-0010-00010-1000-01-1	0-3-2-6-9-9
3	11	1-0001-00001-1000-01-1	0-5-1-6-9-9
4	16	1-0010-00000-0010-01-1	0-3-none-9-9-9
5	20	1-0001-00000-1000-01-1	0-5-none-6-9-9
6	20	1-0001-10000-0010-01-1	0-5-6-9-9-9
7	24	1-0001-01000-0001-01-1	0-5-4-7-9-9
8	28	1-0010-00010-1010-01-1	0-3-2-6;9-9-9
9	32	1-0000-01000-0100-01-1	0-none-4-4-9-9
10	25	1-0110-00000-0010-01-1	0-2;3-none-9-9-9

Table 4.7: Results for graph G1 across 10 iterations using the VQE. The optimizer used is SPSA (200), $P = 15$, using an ansatz with 8 repetitions. The optimal result has been returned once.

Though only 3 of the results are valid, the optimal result was returned once. A number of results feature time slices that are not passed; two feature a time slice that was traversed twice. However, the overall results are better than what the QAOA returned (see table 4.6). All subsequent test cases will therefore make use of the VQE instead of the QAOA.

4.3 Travelling Salesperson Problem

The objective of the travelling salesperson problem is to essentially construct a round trip; for a given graph and a starting point, the goal is to traverse all other vertices exactly once and then return to the start. Once again, one aims to minimize the overall path cost.

4.3.1 Implementation

The implementation of the TSP starts like scenario 2 does. The preprocessing is largely similar; the time slices are set up, dead ends are trimmed, but there is no removal of slices. This is because in this case, a predetermined number of vertices and thus slices needs to be traversed; accordingly, the number of slices cannot be reduced. It equals $|V| + 1$, since every non-start vertex is traversed once, and the start vertex is traversed twice. An additional preprocessing step is made to ensure the start vertex is only present in the first and last slice, as well as being the only vertex in those slices. Then, the *nsg* is constructed again for indexing, as well as the corresponding graph matrix, edge cost matrix, and the cost function. Here, too, already known results for binary variables can be fixed in advance – in this case, the first and last variables are set to 1, since these represent the start vertex in the first and in the last slice.

The TSP cost function, like the previous use case's function, needs to ensure that only one vertex is passed in each slice – that term (equation 4.1) has already come up in the previous use case. Likewise, equation 4.2 is used to calculate the path cost. An additional term ensures that each vertex is only traversed once:

$$P \sum_{v \in V \setminus \{o\}} \left(\sum_{c=1}^{c_{max}-1} X_{c,v} - 1 \right)^2 \tag{4.4}$$

For each vertex, it counts how often it is traversed in different time slices. If the count is not equal to 1, the term within parentheses is not zero; a penalty is incurred. Squaring the inner term once again ensures that the term as a whole cannot turn negative. The complete cost function is then:

$$P \sum_{c=0}^{c_{max}} \left(\sum_{v \in V_c} X_{c,v} - 1 \right)^2 + \sum_{c=0}^{c_{max}-1} \sum_{\substack{v' \in V_{c+1} \\ v \in V_c}} X_{c+1,v'} \cdot X_{c,v} \cdot d_{v,v'} + P \sum_{v \in V \setminus \{o\}} \left(\sum_{c=1}^{c_{max}-1} X_{c,v} - 1 \right)^2 \tag{4.5}$$

4.3.2 Testing and Results

Since G1 is not suited for use in the TSP, two custom graphs G4 and G5 were constructed.

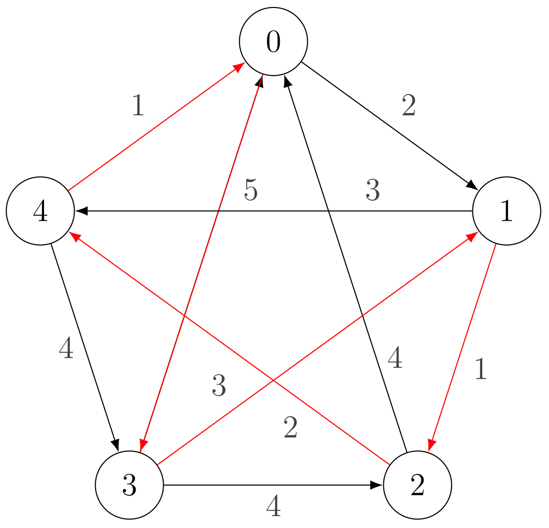
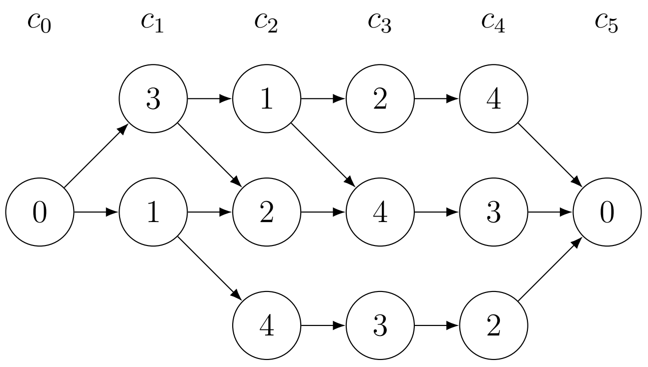


Figure 4.12: Graph G4. The optimal solution with cost 12 (0-3-1-2-4-0) is marked in red.

G4', as well as the *sg* and *nsg* for G4, can be seen in fig. 4.13.



[[0], [1, 3], [2, 4, 1], [2, 4, 3], [4, 2, 3], [0]]

[[0], [1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11], [12]]

Figure 4.13: Graph G4' and its *sg* and *nsg*, trimmed.

The second graph used for testing is G5 (see fig. 4.14).

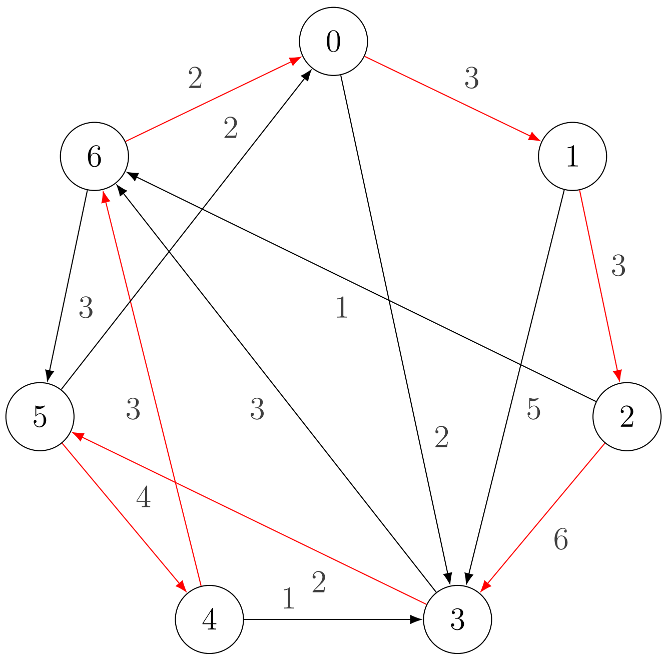
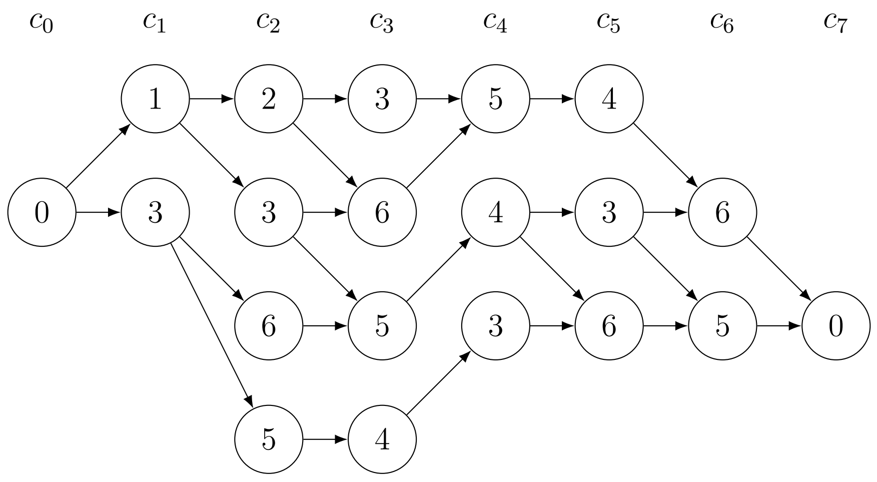


Figure 4.14: Graph G5, used for the TSP. The optimal solution with cost 23 is marked in red.

G5' and the respective *sg* and *nsg* can be seen in fig. 4.15.



[[0], [1, 3], [2, 3, 5, 6], [3, 6, 5, 4], [5, 3, 4], [6, 3, 4], [5, 6], [0]]

[[0], [1, 2], [3, 4, 5, 6], [7, 8, 9, 10], [11, 12, 13], [14, 15, 16], [17, 18], [19]]

Figure 4.15: G5' and G5's *sg* and *nsg*

The results for the two graphs can be seen in table 4.8 for G4 and in table 4.9 for G5.

	optimization cost	optimization result	path
1	12	1-01-001-100-100-1	0-3-1-2-4-0
2	14	1-10-100-010-001-1	0-1-2-4-3-0
3	14	1-10-100-010-001-1	0-1-2-4-3-0
4	14	1-10-100-010-001-1	0-1-2-4-3-0
5	65	1-01-001-010-010-1	0-3-1-4-2-0
6	104	1-10-100-001-100-1	0-1-2-3-4-0
7	104	1-10-100-001-100-1	0-1-2-3-4-0
8	104	1-00-001-100-100-1	0-none-1-2-4-0
9	108	1-01-000-100-100-1	0-3-none-2-4-0
10	109	1-10-100-000-101-1	0-1-2-none-4,3-0

Table 4.8: Results for graph G4 across 10 iterations using the VQE. The optimizer used is SPSA (200), P=50, using an ansatz with 10 repetitions. The optimal solution has been returned once.

It can be seen that across the 10 iterations, 4 valid solutions were returned, with the optimal result among them. The solution shown in line 6 and 7 might be a local optimum.

For the second test case, graph G5, no suitable combination of parameters was found within the testing time frame (2 weeks). None of the tested parameter configurations consistently leads to sufficiently good results. Results that incurred $2P$ or $3P$ in penalties were common – for instance, testing with an ansatz with 40 repetitions, SPSA, and a penalty of 50 led to the results in table 4.9.

	optimization cost	optimization result	path
1	114	1-10-1000-0000-010-100-10-1	0-1-2-none-3-6-5-0
2	159	1-00-1000-0001-010-100-10-1	0-none-2-4-3-6-0
3	165	1-10-0001-0010-001-010-10-1	0-1-6-5-4-3-5-0
4	205	1-00-0001-0000-001-010-10-1	0-none-6-none-4-3-5-0
5	206	1-10-0000-0001-010-000-01-1	0-1-none-4-3-none-6-0
6	208	1-10-0000-1001-000-100-10-1	0-1-none-3;4-none-6-5-0
7	209	1-01-0010-0000-000-001-01-1	0-3-5-none-none-4-6-0
8	213	1-10-0010-0001-010-001-01-1	0-1-5-4-3-4-6-0
9	265	1-10-1100-0000-100-101-00-1	0-1-2;3-none-5-6;4-none-0
10	302	1-01-1000-0010-000-100-00-1	0-3-2-5-none-6-none-0

Table 4.9: Results for graph G5 across 10 iterations using the VQE. The optimizer used is SPSA (150), P=50, using an ansatz with 40 repetitions. None of the results is admissible.

Using higher iteration limits for the classical optimizer leads to higher computation times. For this parameter combination, it took 2 hours per run for this parameter combination, meaning that generating a set of 10 results for these parameters around a day. Several tests with iteration limits of 500 and 1000 led to similar results to those in table 4.9 – each result incurring at least $2P$.

Some of the results feature time slices that have not been passed, and some that have been passed several times (line 6 and 9). This is likely due to the constraint that enforces that each vertex be passed once. In avoiding the penalty from that constraint, these results then incur another penalty since only one vertex should be passed in each slice. Featuring an empty slice therefore always results in an overall penalty of at least $2P$ – for the empty slice itself, and then either because another slice is passed twice, or because one vertex was not passed. As has been noted before, it is more cost-efficient to avoid passing two vertices in one slice, since (in addition to the penalty) that can lead to a higher path cost. In the case of result 9, both the path costs 1-2 and 1-3 are included, as are 5-6 and 5-4. There are exceptions, such as in result 6, where the fact that the slices before and after are empty mean that no added path cost is incurred by passing both vertex 3 and 4 in the same slice.

The fact that this problem features 20 qubits, a high number compared to the other cases, has a clear effect on the results. It made finding suitable parameter combinations more complicated and time-intensive. However, testing more parameter combinations will likely lead to a combination that consistently returns valid, if not optimal solutions. Apart from varying the iteration limit of the classical optimizer or increasing the number of repetitions of the ansatz, changing the ansatz itself is also likely to have a substantial effect on the results. Due to time constraints, further testing was not possible, but the calculation of problems with qubit numbers around and over 20 qubits and the difficulties therein present an avenue for further research.

4.4 Collision-Free Multi-Vehicle Routing

In this use case, the objective is to find routes in a given graph for a number of vehicles such that no two vehicles are at the same vertex at the same time (i.e., within the same time slice). This scenario, like the previous one, builds on time-dependent single-vehicle routing, the main difference being the fact that here, there are several vehicles with different destinations. In this implementation, the start vertex is adaptable as well; to turn this into warehouse scenario, the start vertices can simply be set to be the same vertex across the different vehicles.

4.4.1 Implementation

As with the TSP, the preprocessing for this use case is similar to time-dependent single-vehicle routing, with a few differences. For instance, the fact that there are several vehicles with different destinations means that each needs a specifically 'tailored' sliced graph sg_i. For an index i, the sliced graph sg_i corresponds to the vehicle k_i. The numbered graph, on the other hand, needs to consider the whole of the sliced graphs in the indexing, so as not to give duplicate indices to the vertices.

For G1 and two destination vertices 4 and 6, this means that the different sg_i would be collected in an array:

[[[0], [2, 3], [4, 2], [4]],
[[0], [1, 2, 3, 5], [6, 2, 1], [6]]]

[[[0], [1, 2], [3, 4], [5]],
[[6], [7, 8, 9, 10], [11, 12, 13], [14]]]

Figure 4.16: G1's sg and nsg for multi-vehicle routing; here, the destinations are vertex 4 and vertex 6.

In contrast, the cost matrix is generated only once, and not per vehicle. This is because the cost for a given edge remains the same independent of which sg_i it appears in. While the cost matrix contains edges that are likely not present in every sg_i, the structure of the result prevents an edge not present in sg_i from being part of vehicle k_i's route.

The terms for the cost function will again be explained in the following. The terms from equation 4.2 (for path cost calculation) and equation 4.1 (for ensuring that exactly one vertex is passed) are part of this cost function, as they were in use case 2 and 3. However, as per the paper, any reference to a vertex in a slice (c, v) now has to take into account the vehicle index, and as such is substituted by (i, c, v). Likewise, any reference to the set of vertices in a slice c_i also needs to specify vehicle index, and as such is denoted as $V_{(i,c)}$.

Additional terms ensure that no two vehicles collide (i.e., occupy the same vertex in the same time slice). The paper defines a function D, introduced to count all binary variables of a vertex in a given time slice traversed by different vehicles. K is the number of vehicles, and δ takes on a value of 1 if v is present in the time slice c of vehicle k_i, and

0 if not.

$$D(c, v) := \sum_{i=1}^{K} \delta_{v \in V_{i,c}} X_{i,c,v} \tag{4.6}$$

This is then used to formulate a term that ensures that the function value D for any vertex v in any slice c is either 0 (not traversed by any vehicle) or 1 (traversed by exactly one vehicle). As with the other penalty terms, this will incur a penalty if the constraint is not adhered to. Since D can, for any given value, only be equal to or greater than 0, the term below can not evaluate to a negative value.

$$P \cdot \sum_{c=1}^{c_{max}} \sum_{v \in V} (D(c, v) - 1) \cdot D(c, v) \tag{4.7}$$

4.4.2 Results

For the first test, graph G1 was chosen with the destinations 4 and 6. G1' for the destination 9 is shown in fig. 4.5; the *sg* and *nsg* for this multi-vehicle problem are shown in figure 4.16. The optimal result has a cost of 11.

	optimization cost	optimization result	path
1	11	1-01-10-1–1-0100-100-1	0-3-4-4, 0-2-6-6
2	11	1-01-10-1–1-0100-100-1	0-3-4-4, 0-2-6-6
3	11	1-01-10-1–1-0100-100-1	0-3-4-4, 0-2-6-6
4	11	1-01-10-1–1-0100-100-1	0-3-4-4, 0-2-6-6
5	12	1-01-10-1–1-1000-100-1	0-3-4-4, 0-1-6-6
6	13	1-01-10-1–1-0001-001-1	0-3-4-4, 0-5-1-6
7	13	1-01-10-1–1-0001-001-1	0-3-4-4, 0-5-1-6
8	14	1-01-01-1–1-0001-001-1	0-3-2-4, 0-5-1-6
9	56	1-00-10-1–1-0100-100-1	0-none-4-4, 0-2-6-6
10	59	1-01-10-1–1-0001-010-1	0-3-4-4, 0-5-2-6

Table 4.10: Results for graph G1 across 10 iterations using the VQE. The classical optimizer used is SPSA (200), $P = 50$, using an ansatz with 10 repetitions. The optimal solution has been returned 4 times.

4 out of 10 iterations returned the optimal result of 11, and a further four results were admissible. This problem has a qubit number of 15 – a pattern can be seen in that, similar to other problems with low to medium qubit numbers (see table 4.5 or 4.22), the results here were very clear and there was a low number of inadmissible solutions returned.

The second test case for collision-free multi-vehicle routing is G3 (see fig. 4.10), with destination vertices 4 and 5.

[[[0], [1, 2, 3], [4, 1, 3], [4]],

[[0], [2, 3], [3, 5], [5]]]

[[[0], [1, 2, 3], [4, 5, 6], [7]],

[[8], [9, 10], [11, 12], [13]]]

Figure 4.17: G3's *sg* and *nsg* for multi-vehicle routing; here, the destinations are vertex 4 and vertex 5.

Here, there are two optimal results, both with a cost of 14: either 0-2-3-4 for the route to vertex 4 and 0-3-5 for the route to vertex 5, or 0-3-4 for the route to vertex 4 and 0-2-3-5 for the route to vertex 5.

	optimization cost	optimization result	path
1	14	1-001-100-1–1-10-10-1	0-3-4, 0-2-3-5
2	14	1-010-001-1–1-01-01-1	0-2-3-4, 0-3-5-5
3	15	1-100-100-1–1-01-01-1	0-1-4-4, 0-3-5-5
4	15	1-100-100-1–1-01-01-1	0-1-4-4, 0-3-5-5
5	16	1-100-100-1–1-10-10-1	0-1-4-4, 0-2-3-5
6	57	1-000-100-1–1-01-01-1	0-none-4-4, 0-3-5-5
7	57	1-010-001-1–1-00-01-1	0-2-3-4, 0-none-5-5
8	63	1-010-010-1–1-01-00-1	0-2-1-4, 0-3-none-5
9	104	1-000-100-1–1-10-00-1	0-none-4-4, 0-2-none-5
10	107	1-001-000-1–1-00-10-1	0-3-none-4, 0-none-3-5

Table 4.11: Results for graph G3 across 10 iterations using the VQE. The optimizer used is SPSA (200), $P = 50$, using an ansatz with 20 repetitions. The two optimal results have been returned once each.

Half of the results returned for this test case were admissible, and the optimal cost value was returned twice; in fact, each of the two possible optimal solutions were returned once. The inadmissible solutions each feature at least one time slice where no vertex is passed; this might be due to the fact that in comparison with other constraint violations, this particular constraint violation has the advantage that it avoids adding the edge costs of the edges to and from that slice. Compared to other constraint violations (say, more than one vertex visited in one slice, where edge costs are counted from and to each passed vertex), this is cheaper and therefore seems to occur more often – not only in this case, but across test cases generally.

4.5 Conclusions and Additional Notes on Result Interpretation

In this section, a few notes on the solution process and result evaluation will follow.

- For one, both the VQE and the QAOA are strongly influenced by the choice of parameters. In order to solve a specific problem, some testing with different constellations is required to achieve a useful result. Raising the limit of iterations for the classical optimizer can yield better results, but leads to a significant increase in computing time, for instance. Iteration limits of 200 can lead to computing times of 10 to 15 minutes, while a limit of 1000 can take 2 hours or more for one iteration. The computing time also depends on the number of repetitions of the algorithm itself, where high numbers (as mentioned) generally lead to better results, but also increase the computing time. A high iteration limit, unlike the number of repetitions in the algorithm, does not necessarily correlate with a better result.

- Starting with qubit numbers above a certain threshold (likely about 12-15) the results of several iterations feature more inadmissible results; for higher qubit numbers, the results of the test runs include a fair number of those results. The results for problems with lower qubit numbers, such as can be seen in table 4.5 in section 4.1 (5 qubits) or in table 4.6 in section 4.22 (8 qubits), are consistently identical or feature very little inadmissible results. Problems with qubit numbers at the threshold (see table 4.10, 15 qubits, table 4.11, 14 qubits, or table 4.8, 13 qubits) return about 50% admissible results, while higher qubit numbers (see table 4.7, 17 qubits or 4.9, 20 qubits) return few admissible results and require more ansatz repetitions to yield useful results.

  It therefore seems advisable to, in general, run the algorithm several times and then choose the best result from this selection, rather than computing a single result, especially in cases where the problem has a high qubit number. In most of the cases discussed here this leads to, if not outright the optimal result, then one that is close to it.

- A higher number of repetitions for the VQE ansatz seems to have a positive effect on the results, just as the parameter p is said to do. In both cases, this leads to a longer computing time. If one has sufficient time and computational power, increasing the number of repetitions is one way to increase the odds of achieving good results on a simulator. In time, with larger quantum systems, these problems can be run on real quantum computers, which eliminates the overhead of the simulation.

- There is an abundance of other parameters that can be adapted and changed. The mixer argument for the QAOA is one such example, or the number of steps that are remembered in the classical optimization. Due to the high number of possible combinations, testing these exhaustively was not possible in the scope of this work, though it, too, is an avenue to be pursued when trying to improve on the results. A better understanding and analysis of a given problem and the algorithm used to solve it can make the tuning process easier.

Chapter 5

Passenger Routing Use Case

5.1 Data and Preprocessing

In this chapter, the aviation use case will be outlined, and its implementation and results discussed. As has been briefly described above, the scenario modeled with this use case covers multi-passenger routing. The scenarios are as follows: one or more flights have been cancelled and a number of passengers with different destinations need to be rerouted on different aircrafts. While one passenger might have the original flight's destination as their personal destination, others might need to transfer flights and therefore have a different destination. The objective is to find routes for each passenger to take such that each one reaches their destination, while taking care not to overbook any of the given flights and minimize the cost of the flights.

The cost metrics have been outlined in chapter 3 and will be quickly summarized here; they include:

- monetary cost of the chosen flights,
- monetary cost of the chosen flights and compensation costs paid by the airline for delays, and
- duration of the chosen routes, where the duration per passenger is the time between their planned departure and their actual arrival.

The passengers can be conceptualized similarly to the vehicles in section 4.4 in that they each need a specific route that needs to respect an overall constraint and is part of a holistic minimization of costs. For that reason, this problem can not be broken down into several single-vehicle routing problems. In section 4.4, the overall constraint was the fact that no collisions were allowed. Here, it is the capacity constraint that limits the number of people that can be allocated to a given flight in order to avoid overbooking.

To approach the problem, first the input needs to be defined. On one hand, one needs a list of passengers; on the other, a list of flights. This is equivalent to using a list of edges to construct a graph as was done in use cases 2–4; here, too, isolated vertices will not be included as they are not relevant to the routing problems at hand.

Passenger data includes:

- Booking ID and name, for identification,
- their destination,
- their originally planned departure time,
- their planned arrival time, and
- their flight distance.

Name and booking ID are not strictly necessary and only serve to better model the scenario. They can be used when returning result routes, but are not necessary for the calculation itself. The other ones, however, are all necessary, though not for all metrics. The flight distance, for example, is of relevance only when including compensation costs, since it has relevance only in the context of compensation costs (which differ based on distance). (The other two metrics take into account the flight cost and and the flight duration, and as such do not need the additional information of a flight's distance.) The start vertex is not listed because in this scenario, the passengers all start from the same location. However, the implementation could be easily adapted to allow for different starting points.

The passenger data is stored in an array that holds objects of the *Passenger* class. The data listed above is then accessible by calling attributes from a given *Passenger* object.

The set of flights also get stored in custom objects. Flights can essentially be treated like edges, but they differ in that flights have a time aspect to them. As such, they have additional properties such as a flight number (acts as a unique identifier) and, apart from monetary cost, a set departure and arrival time. While a route made up of connecting edges is valid in the other scenarios, as long as no two edges are passed at the same time, the same is not true here: the arrival and departure time add another dimension to the validity of a connection. This will therefore appear as one constraint in this use case.

Additionally, flights differ from edges in that they have a capacity property, which refers to the current number of available seats (not the total amount of seats in the plane).

As such, a flight object contains the following data:

- start vertex
- destination vertex
- cost (monetary)
- start time
- arrival time
- capacity (available seats)

5.2 Implementation

These structures are then implemented in the program. Here, the values for start and arrival time are not implemented as timestamps, but simply as integers. Building on this implementation, one could include date and time zone. Here, it was kept simple since this was not the focus of the work.

For different use cases, other parameters can be added. Using an instance of a class *Flight* instead of storing the parameters in an array facilitates this. The flights are stored in a dictionary, a key-value data structure. The key for each flight is its flight number. This allows for easy and unambiguous access. (An entry with an already existing flight number will, due to the nature of a dictionary, overwrite the current entry, which prevents duplicates.) In the initial dictionary, flights with capacity 0 can already be sorted out.

As with the use cases in chapter 4, this scenario presents a number of constraints.

- The routes need to be valid: any connecting flight needs to start from the airport that the previous flight landed at.
- At each leg of a route, the passenger can only take at most one flight. (For instance, a route choosing two flights that are both outgoing from the start node would not constitute a valid solution).
- Any given flight's capacity must not be exceeded.

- The transfer time needs to allow for smooth transfer, that is, it cannot be below a certain threshold. For our purposes, this threshold is a default value (1 hour) that is the same across all vertices/airports. This could also be adapted so as to be specific to a given airport.

In this model, it makes sense for the qubits to represent the edges (flights), not the vertices (airports). Apart from being more convenient for the implementation of the transfer constraint (discussed later), it also allows for the easy handling and distinguishing of flights on the same route (i.e., same start and same destination vertices).

This means that for this use case, it makes sense to use a hybrid model of the cases described before. Among the presented use cases, the first one is the only one that uses qubits to represent edges and not vertices, but it does not implement time slices as the other three scenarios do. Both of these aspects are of use in this scenario.

In the following, the preprocessing for this use case will be explained. The graph used for illustrating steps is the following. It is also used for testing; specific flight data and passenger data is shown in table 5.1 and 5.2. For each passenger, the data includes information on their original (cancelled) flight. For demonstration purposes and testing, the passengers' original flights are different from each other.

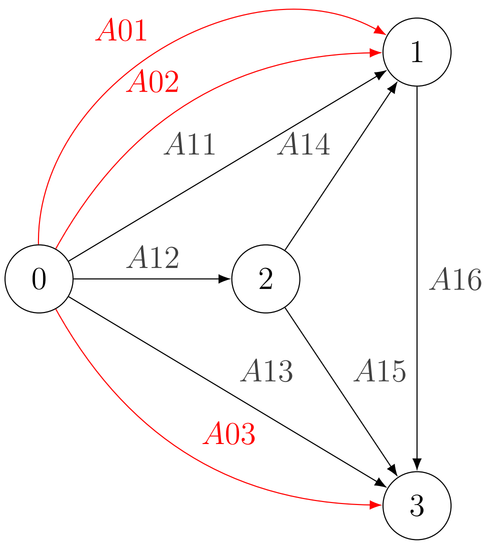


Figure 5.1: Graph G6, used for testing the passenger routing use case. The starting vertex is 0. Edges are marked with their flight numbers. The cancelled flights A01, A02, and A03 are marked red.

Booking Nr.	Name	Flight Nr.	Dest.	Planned Dep.	Planned Arr.	Distance
HX7R2W	A	A01	1	8	9	340km
L5D32X	B	A02	1	8:30	10:30	400km
N7LR19	C	A03	3	7	8:30	610km

Table 5.1: Passenger data for graph G6. Flight distance of the cancelled flights has been included as it is relevant for the calculation of the compensation cost.

Flight Nr.	Start	Dest.	Cost	Start Time	Arrival Time	Capacity
A01	0	1	125	8	9	–
A02	0	1	95	8:30	10:30	–
A03	0	3	285	7	8:30	–
A11	0	1	160	8	10	2
A12	0	2	60	8:30	9:30	2
A13	0	3	215	7:30	10:30	1
A14	2	1	80	10:30	12	2
A15	2	3	75	10	11	2
A16	1	3	40	13	14	3
Z1	1	1	0	0	0	0
Z3	3	3	0	0	0	0

Table 5.2: Flight data for graph G6. Cancelled flights are shown in the first section; non-cancelled flights admissible for rerouting in the second; supplemental flights in the third (in red).

For each passenger, the flight dictionary is updated to contain their specific supplementary flight, which is a flight from their destination to the same, with all other values set to 0. This has the same purpose as the supplementary edges in use cases 2-4: it prevents shorter routes from incurring penalties. The supplementary flights in this example are shown in the lower section of table 5.2 and marked in red.

Then, as in use case 4 (collision-free multi-vehicle routing), a sliced graph is constructed for each passenger, based on the list of flights and the passenger's destination. To this purpose, all unique vertices are extracted from the flight list. This is due to the fact that, as was the case in the other time-dependent scenarios, the length of the individual sliced graphs is limited by the number of unique vertices in the original graph, though here it is $|V| - 1$, since it is the edges that are stored, not the vertices. The first slice of a passenger's sliced graph, accordingly, contains any flight going out from the start vertex. Continuing on, any slice c_i contains potential connecting flights to flights in $c_i - 1$. *Potential* meaning that the connecting flights depart from the vertices that the flights in the previous slice arrive at. They are not necessarily valid connecting flights in that a flight in one slice might depart before a flight in the previous slice arrives at the airport.

Similar to the trimming in the other use cases, the last slice is then trimmed so that it only contains flights that land at the passenger's destination. Then, going backwards from the end, dead ends are trimmed (flights that will not lead to the destination). Then, superfluous slices, ones that only contain the passenger's supplementary flight, are removed.

What remains is, in the current example, the following overall *sg* that contains the sliced graphs of each of the three passengers:

[[['A11', 'A12'], ['A14', 'Z1']],

[['A11', 'A12'], ['A14', 'Z1']],

[['A11', 'A12', 'A13'], ['A14', 'A15', 'A16', 'Z3'], ['A16', 'Z3']]]

Figure 5.2: *sg* for graph G6. Flights are identified by their flight number. A number beginning with 'Z' indicates a supplementary flight.

Note that the first two passengers have identical sg_i since they have the same destination. As a next step, the individual flights in the time slices are once again indexed:

[[[0, 1], [2, 3]],

[[4, 5], [6, 7]],

[[8, 9, 10], [11, 12, 13, 14], [15, 16]]]

Figure 5.3: *nsg* for graph G6.

The index of the start node of passenger k_i is 0 + the sum of the length of the sliced graphs of every passenger k_j where $j < i$. In this case, there are 17 edges overall in the *sg*, meaning that the number of qubits that is required is also 17.

5.3 Cost Function Terms

The next step is formulating the cost function. In this case, there are three cost functions, one for each metric that can be chosen to optimize. However, they have common terms. These will be introduced first.

For one, since this function builds on the functions of the other time-dependent use cases, it needs to ensure that for each passenger, in each slice, only one flight is taken. This is done by repurposing equation 4.1 as it is used for multi-vehicle routing, with the very minor difference that the element that is checked is edges, not vertices.

$$\sum_{k\in K}\sum_{c\in C_k} P\left(\sum_{e\in E_{k,c}} X_{k,c,e}-1\right)^2 \tag{5.1}$$

For another, any transfer needs to be admissible, which is ensured with the following term. It iterates over every passenger and then, for any flight pair in consecutive slices, checks if these flights constitute a valid transfer. For that to be the case, the first slice needs to land at the airport (vertex) that the second flight departs from. Furthermore, the transfer time needs to be at least one hour. If both conditions are fulfilled, the function t returns 0. If not, it returns the penalty value. This, of course, is only incurred if the two flights are actually part of the route. If one is not, then the respective binary variable $X_{(k,c,e)}$ is zero and so will be the whole product[1]. The reason why t takes the passenger index k as an additional argument is that supplementary flights are not bound by transfer constraints. If the connecting flight to a given flight is a supplementary flight, then the returned value is automatically 0.

$$\sum_{k\in K}\sum_{c\in C_k}\sum_{e_1\in E_{k,c}}\sum_{e_2\in E_{k,c+1}} X_{k,c,e_1}\cdot X_{k,c+1,e_2}\cdot t_{((k,c,e_1),(k,c+1,e_2),k)} \tag{5.2}$$

Next are the metric-specific terms. For the simple monetary cost metric, the flights' costs need to be added up, similarly to how the cost function was handled in the other time-dependent use cases. *cost* is a function that simply returns an edge's flight cost.

$$\sum_{k\in K}\sum_{c\in C_k}\sum_{e\in E_{k,c}} X_{k,c,e}\cdot cost_{k,c,e} \tag{5.3}$$

For the metric that includes compensation cost, one needs the three preceding terms as well as a term that calculates if and how high the passenger's compensation is. Here, r is a function that takes a flight identifier and a passenger index. If the flight leads to the passenger's destination (and is not a supplementary flight) the compensation cost is calculated. It is a simplified version of the European Air Passenger Rights Regulation,

[1]If this model used vertices instead of edges, then to check for two consecutive flights, three binary variables would be needed. This would make the term non-quadratic and would complicate the implementation.

specifically article 7 [33]. The cost numbers used here are:

- For distances under 1500km, 250 if the delay is over 2 hours, and 125 if it is up to 2 hours,
- for distances between 1500 and 3500km, 400 if the delay is over 3 hours, and 200 if it is up to 3 hours, and
- for distances over 3500km, 600 if the delay is over 4 hours, and 300 if the delay is up to 4 hours.

This, of course, does not reflect all of the details of the regulations; it is merely meant to give an example as to how one might go about implementing part of it. For instance, one might additionally implement that, depending on the wait time, additional costs for meals etc. need to be paid. Irrespective of the actual details of the cost regulations, the main term is the following:

$$\sum_{k\in K} \sum_{c\in C_k} \sum_{e\in E_{k,c}} X_{k,c,e} \cdot r_{k,c,e} \tag{5.4}$$

Where r, as described, returns the compensation cost. Once again, by multiplying the compensation with the binary variable $X_{k,c,e}$ of the flight, only flights that are actually taken can incur compensation cost.

Finally, there is the cost function for the time metric. It aims to minimize the overall duration of the passengers' journeys. The individual duration is defined as the interval between the originally planned flight and the arrival of the actually taken flight. As such, not only the flights' durations themselves are relevant – transfer time between flights and waiting time before the first flight needs to be factored in, as well. This is done by adding two terms. The first one covers the waiting time before the first flight. For any flight that starts at the passenger's starting point (i.e. for any flight in slice c_0 of a passenger's sg_i), the waiting time is calculated and returned via the function w.

$$\sum_{k\in K} \sum_{e\in E_{k,0}} X_{k,0,e} \cdot w_{k,0,e} \tag{5.5}$$

The second term adds the transfer time between flights. It mirrors the transfer constraint, though instead of checking if the time is above or below a certain threshold, the value is simply returned in $transfer$. Again, the transfer time between a flight and a supplementary flight is not taken into account.

$$\sum_{k\in K}\sum_{c\in C_k}\sum_{e_1\in E_{k,c}}\sum_{e_2\in E_{k,c+1}} X_{k,c,e_1}\cdot X_{k,c+1,e_2}\cdot transfer_{((k,c,e_1),(k,c+1,e_2),k)} \tag{5.6}$$

Now, one constraint has not been discussed yet: the capacity constraint. It serves to ensure that for a given flight, the number of passengers that take it does not exceed its capacity. This is therefore an inequality constraint. Since the problem is formulated as a QUBO (quadratic unconstrained binary optimization) problem, this needs to be encoded within the formula itself. For inequality constraints <= 1, this is straightforward to implement and can be seen in formula 2.2, for example. Essentially, to ensure that a given sum s is <= 1, one would formulate it as follows:

$$(s-1)\cdot s \tag{5.7}$$

The term is 0 when s is 0 or 1. This can be expanded for constraint values $n > 1$:

$$(s-n)\ldots(s-2)\cdot(s-1)\cdot s, \tag{5.8}$$

essentially constructing a function where all admissible values are roots. However, for values above 1, this term is not quadratic. This is where so-called slack variables come in. They are additional (in this case, binary) variables that can be used to represent inequality constraints for values above 1 while keeping the term quadratic. Inequality constraints don't have to be adapted manually; Qiskit is able to transform given inequality constraints into terms to be added to the cost function.

It should be noted that no matter if they are implemented through Qiskit or directly in the cost function, inequality constraints increase the problem size with each slack variable. However, in this case, there are ways to lower that impact. The original constraint value is the capacity of a flight, though this can be further reduced. (The sum of occurrences of a flight in the overall sg will be referred to as s.)

If the number of passengers is lower than the capacity, then the constraint does not need to be enforced. This is because in a valid solution, a passenger cannot take a flight more than once, and as such, having a total of three passengers could not lead to overbooking a flight with a capacity of 4. Even though a flight can appear several times within a sg_i (see flight $A16$ in sg_3 in fig. 5.2), a single passenger cannot take a flight several times in a valid solution – it would incur penalties through the transfer time constraint.

Going one step further, one can simply check the number of passengers that can take the flight. With three passengers, if a flight only appears in two of their sg, then a capacity of two need not be enforced, because any valid solution could not result in violating the capacity constraint. (Any inadmissible result is again caught by the transfer constraint.) This means that if the number of passengers that can potentially take the flight is higher than the capacity, then then the capacity constraint needs to be enforced. Since that number is equal to or lower than the total number of passengers, checking first whether the number of passengers is lower can be foregone.

In summary, this means that the capacity constraint only needs to be enforced in cases where the number of passengers that can take the flight is higher than the flight's capacity.

As an example: $A15$ has a capacity of 2, which is the constraint value. The number of passengers is 3 and therefore not lower. However, flight $A15$ can only be taken by the third passenger (see fig. 5.2), meaning it can only be taken once overall and could not exceed a capacity of 2. Therefore the constraint is automatically adhered to.

For flight $A11$, which has a capacity of two but can be taken by all three passengers, the constraint needs to be enforced.

5.4 Results

Testing was done with one graph, G6. This problem features 17 qubits for the problems itself, and a number of additional qubits needed in the background as slack variables for the capacity constraint. The total number of qubits in this case amounts to 23, a significant increase. Therefore, though this graph only features 4 vertices and 6 edges, a larger example would fall out of scope.

Similar to the second test case of the TSP, here, too, no suitable combination of parameters could be found that consistently returns valid results. Therefore, instead of discussing the VQE results, the optimal solutions as calculated by the CplexOptimizer will be discussed.

5.4.1 Basic Cost Function

Here, the goal is simply to minimize the overall cost of the flights. For three passengers with destinations 1, 1, and 3, the classical optimizer returns the following result.

Optimization Cost	Optimization Result	Path
480	10-01–01-10–010-1000-10	A11-Z1; A12-A14; A12-A14-A16

Table 5.3: Result for the basic cost function for G6, with three passengers with destinations 1, 1, and 3. P=5000.

Passenger 1 and 2 each have two possible routes, A11 or A12-A14. Passenger 3 has four possible routes, A13, A12-A15, A11-A16, and A12-A14-A16. Of those, the second one is not admissible, since it does not allow for an hour of transfer time, and A13 is the most expensive option. The solution routes for the passengers therefore all include vertex 1. The reason why passenger 1's route features A11 instead of the cheaper A12-A14 connection is because both A12 and A14 have a capacity of 2. One passenger therefore gets assigned the more expensive flight A11 – in this case, passenger 1.

The optimization cost of 480 is the sum of the chosen flights' costs.

5.4.2 Extended Cost Function (Including Compensation Cost)

Optimization Cost	Optimization Result	Path
890	10-01–01-10–001-0001-01	A11-Z1; A12-A14; A13-Z3-Z3

Table 5.4: Result for the extended cost function for G6 which includes compensation costs, with three passengers with destinations 1, 1, and 3. P=5000.

The routes that each passenger can take remain the same. However, the different cost function introduced a different routing solution. For one, passenger 3 now takes the more expensive flight A13 instead of the less expensive routes. This is due to the fact that flight A16, which would be the last leg if any of the other admissible routes were chosen), arrives at 14 o'clock – five and a half hours later than passenger 3 was to arrive originally. Flight A13, however, arrives at 10 o'clock. A delay of two hours or under means that the compensation cost is 125 – 250 if it was over 2 hours. Any alternative flights do not offer a cost difference higher than 125 (the difference between A12-A14-A16, the cheapest alternative, and A13 is 35). The same reasoning lies behind the choice of A11 for passenger 1 – A14 would arrive three hours later than the original arrival (which, since the original flight was <1500km, would incur the whole compensation amount). Passenger 2, who would have arrived at 10:30, has a delay of under 2 hours when taking the cheaper flight, and so is assigned the cheaper option of A12-A14. The cost of 890 now includes both flight cost and the compensation cost paid by the airline.

5.4.3 Time Cost Function

Optimization Cost	Optimization Result	Path
8	10-01–10-01–001-0001-01	A11-Z1; A11-Z1; A13-Z3-Z3

Table 5.5: Result for the time cost function for G6, with three passengers with destinations 1, 1, and 3.

Here, the fastest option for passengers 1 and 2 is flight A11. The fastest option for passenger is A13, which arrives at 10:30 o'clock – A16 arrives at 14 o'clock. The nature of the transfer time constraint means that here, direct flights are more likely to get chosen, since the transfer adds at least one hour to the overall travel duration. This, of course, does not apply for cases when the direct flight departs much later than the alternative route. Here, the optimization cost of 8 means the overall travel duration across the three passengers is 8 hours.

5.5 Results using the VQE

This use case has been tested with a variety of parameter combinations. While none of the tested combinations returned suitable results, some will be shown in the following.

	optimization cost	optimization result	path
1	5,340	00-01–10-01–010-1000-10	none-Z1, A11-Z1, A12-A14-A16
2	5,375	00-01–10-01–001-0001-01	none-Z1, A11-Z1, A13-Z3-Z3
3	5,440	00-10–10-01–100-0010-01	none-A14, A11-Z1, A11-A16-Z3
4	5,540	01-10–10-01–100-0010-10	A12-A14, A11-Z1, A11-A16-A16
5	10,140	00-01–01-10–000-0001-01	none-Z1, A12-A14, none-Z3-Z3
6	10,240	00-01–01-10–010-0000-10	none-Z1, A12-A14, A12-none-16
7	10,260	00-01–01-01–100-0010-01	none-Z1, A12-A14, A11-A16-Z3
8	10,275	00-01–01-00–001-0001-01	none-Z1, A12-none, A13-Z3-Z3
9	10,275	00-01–01-01–001-0001-01	none-Z1, A12-Z1, A13-Z3-Z3
10	10,375	00-00–10-01–001-0001-01	none-none, A11-Z1, A13-Z3-Z3

Table 5.6: Results for graph G6 for the basic cost function across 10 iterations using the VQE. The optimizer used is SPSA (1000), P=5000, using an ansatz with 40 repetitions.

Here, each result has an optimization cost that is higher than P, meaning each result is inadmissible. This can also be seen when looking at the paths – all results but one have a time slice that is not traversed. This is often accompanied with the other slice in that route being a supplementary flight, which is only logical – it has a cost of zero, and choosing a non-supplementary flight would only add to the cost. Interestingly, the first route is the most affected by this. The only result where this does not happen (see line 4) suggests that a passenger take flight A16 twice. As has been mentioned before, this automatically incurs a penalty as it violates the transfer constraint.

Even though the overall results are all inadmissible, the subresults can sometimes show valid routes for a passenger. This is the case for instance in the first two routes in line 4, or in the last two routes in line 7.

Chapter 6

Conclusion and Outlook

6.1 Result Discussion and Conclusion

For problems with low qubit numbers (such as the test cases using graph G2 – see table 4.1 and 4.4), the VQE and the QAOA both return the optimal result with very high probability. For larger problems, several runs of the test case are necessary, since – depending on the problem size – the rate of inadmissible or non-optimal results returned can be quite high, up to 50-80% (see table 4.7, 4.8, and 4.11). For problems that require around 20 qubits or more, the process of finding a solution becomes more complicated and time-intensive, since more (and more precise) parameter tuning is needed in order to achieve useful results. Additionally, for a high number of repetitions of the ansatz (in the QAOA, this is equivalent to the parameter p), which can improve results, computation times can get quite high. This, it should be said, is an effect of simulating a quantum computer on a classical one – the simulation overhead increases the time and power needed to calculate the solution for a problem.

The QAOA and the VQE are algorithms usable on current quantum computers and can both be used to good effect when the parameters suit the problem, which presents the core challenge in their use. Determining which ansatz to use for the VQE is one example where the choice of parameter is not obvious, but requires deeper analysis and testing.

Regarding this work, the time frame allocated for testing (2 weeks) was sufficient for finding suitable parameter combinations for most problems (all test cases for SV and MV routing as well as the first test case for the TSP), though not for all of the problems. For the larger problems (the second test case of the TSP with 20 qubits, see table 4.9, and the passenger routing use case with 23 qubits, see section 5.4), the results consistently incurred penalties and will need further testing and analysis. Both the reduction of the problem, if feasible, and the careful selection of parameters therefore play an integral role in the use of the QAOA and the VQE.

The usage of Qiskit in this implementation can be assessed as positive; the comprehensive capabilities and documentation facilitates implementation change and adaptations, and the tutorials offered convey further information on algorithms and their usage. The fact that Qiskit handles the transformation of the cost function into a cost hamiltonian, for instance, and the support for inequality constraints via slack variables were both useful features in the context of this work.

6.2 Further Avenues for Research

This work focused on the implementation of five routing use cases with quantum algorithms using a quantum computer simulator. The results open up a number of further questions and possible alternative approaches:

- Comparison of the results gained by using an actual quantum computer compared to a simulator;
- similarly, comparison of the results of a mock instance of an actual quantum computer – a simulator that incorporates noise, though this again claims more resources – to a simulator;
- analysis of the effect of using a different mixer parameter for the QAOA, instead of the default one;
- parameter studies for the classical optimizer beyond the iteration limit – such as the learning rate for the SPSA;
- "warm-starting" the QAOA [34], i.e. initializing it with a solution close to the optimal solution,
- comparison of the performance and results of the QAOA and VQE compared to variations of the same, such as ADAPT-QAOA [35] and ADAPT-VQE [36];
- and analysis of parameter combinations and their effectiveness and emerging patterns thereof for the QAOA and the VQE.

6.3 Outlook

As mentioned, the advantage of the QAOA and the VQE is that they can be used on current quantum computers, quantum computers that have limited qubit numbers and non-trivial error rates (NISQ – Noisy Intermediate-Scale Quantum technology). Using a simulator removes noise, but simulators are inherently limited in their capacity to simulate large problems on quantum computers, requiring a substantial amount of both time and computational power; the simulation overhead increases with problem size.

The current research is occupied with reducing error rates, achieving higher qubit numbers, and improving quantum computer performance in general. With the growing capabilities of quantum computers, so will their usefulness increase, making the QAOA and the VQE usable for larger and more relevant problems in the future. On the other hand, the algorithms themselves also see improvements – variations on the current algorithms (like the above-mentioned ADAPT-QAOA) are one example. As new approaches for problem formulation and new algorithms emerge, the possibilities for solving optimization problems increase and so does their quality. The hardware constraints, in some way, necessitates this: Adapting the software to work on limited hardware has always been an important driver in streamlining and optimizing approaches for problem solving.

Quantum computing might be a young field – but it is a field that will both see and effect great changes over the coming years and decades, and the current research paves the way for this new way of computing.

Bibliography

[1] Daniel JAROSZEWSKI, Fabian KLOS, and Benedikt STURM. *Ising formulations of routing optimization problems.* 2020. arXiv: `2012.05022 [quant-ph]` [cit. on pp. 1, 2, 16, 17, 19, 21, 23–25, 35].

[2] Jairo R. MONTOYA-TORRES et al. "A literature review on the vehicle routing problem with multiple depots." In: *Computers and Industrial Engineering* 79 [2015], pp. 115–129. ISSN: 0360-8352. DOI: `https://doi.org/10.1016/j.cie.2014.10.029`. URL: `https://www.sciencedirect.com/science/article/pii/S036083521400360X` [cit. on p. 5].

[3] Jamal ELHACHMI and Ziyad BAHOU. "Fleet management through Vehicle Routing Problem and Multi Depot Vehicle Scheduling Problem: A Literature Review." In: *Proceedings of the International Conference on Industrial Engineering and Operations Management* [2019], pp. 394–402. ISSN: 2169-8767. URL: `http://ieomsociety.org/ieom2019/papers/112.pdf` [cit. on p. 5].

[4] Jeffrey YEPEZ. *Lecture notes: Qubit representations and rotations.* Jan. 2013. URL: `https://www.phys.hawaii.edu/~yepez/Spring2013/lectures/Lecture1_Qubits_Notes.pdf` [visited on 08/31/2021] [cit. on p. 7].

[5] Siyavus ACAR. *A Presentation: Group of Unitary Operators.* Dec. 2009. URL: `https://www.uwo.ca/math/faculty/khalkhali/files/PresentationAcar.pdf` [cit. on p. 10].

[6] Han-Sen ZHONG et al. "Quantum computational advantage using photons." In: *Science* 370.6523 [2020], pp. 1460–1463 [cit. on pp. 11, 12].

[7] Julian KELLY. *A Preview of Bristlecone, Google's New Quantum Processor.* Mar. 2018. URL: `https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html` [cit. on p. 11].

[8] Catherine McGeoch and Pau Farré. *The D-Wave Advantage System: An Overview.* Sept. 2020. URL: `https://www.dwavesys.com/media/s3qbjp3s/14-1049a-a_the_d-wave_advantage_system_an_overview.pdf` [visited on 08/08/2021] [cit. on p. 11].

[9] Paul Benioff. "The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines." In: *Journal of statistical physics* 22.5 [1980], pp. 563–591 [cit. on p. 11].

[10] Richard P. Feynman. "Simulating Physics with Computers." In: *International Journal of Theoretical Physics* 21.6/7 [1982], pp. 467–488. DOI: `10.1007/BF02650179` [cit. on p. 11].

[11] Jonathan A. Jones, Michele Mosca, and Rasmus H. Hansen. "Implementation of a quantum search algorithm on a quantum computer." In: *Nature* 393.6683 [May 1998], pp. 344–346. ISSN: 1476-4687. DOI: `10.1038/30687`. URL: `http://dx.doi.org/10.1038/30687` [cit. on p. 11].

[12] Isaac L. Chuang, Neil Gershenfeld, and Mark Kubinec. "Experimental Implementation of Fast Quantum Searching." In: *Physical Review Letters* 80 [15 Apr. 1998], pp. 3408–3411. DOI: `10.1103/PhysRevLett.80.3408`. URL: `https://link.aps.org/doi/10.1103/PhysRevLett.80.3408` [cit. on p. 11].

[13] John Preskill. *Why I Called It 'Quantum Supremacy'.* Oct. 2019. URL: `https://www.quantamagazine.org/john-preskill-explains-quantum-supremacy-20191002/` [visited on 08/05/2021] [cit. on p. 11].

[14] Edwin Pednault et al. *On "Quantum Supremacy"s.* Oct. 2019. URL: `https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy` [visited on 08/31/2021] [cit. on p. 12].

[15] Frank Arute et al. "Quantum supremacy using a programmable superconducting processor." In: *Nature* 574.7779 [2019], pp. 505–510 [cit. on p. 12].

[16] IBM Corporation. *The Quantum Decade – A playbook for achieving awareness, readiness, and advantage.* July 2021, p. 19. URL: `https://www.ibm.com/downloads/cas/J25G350K` [visited on 08/11/2021] [cit. on p. 12].

[17] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. *A Quantum Approximate Optimization Algorithm.* 2014. arXiv: `1411.4028 [quant-ph]` [cit. on pp. 13, 14].

[18] Qiskit TEXTBOOK. *Solving combinatorial optimization problems using QAOA.* Aug. 2021. URL: `https://qiskit.org/textbook/ch-applications/qaoa.html` [visited on 08/09/2021] [cit. on p. 13].

[19] The Qiskit TEAM. *Solving combinatorial optimization problems using QAOA.* Aug. 2021. URL: `https://qiskit.org/textbook/ch-applications/qaoa.html` [visited on 08/09/2021] [cit. on p. 13].

[20] QISKIT. *qaoa_program.py.* July 2021. URL: `https://github.com/Qiskit/qiskit-optimization/blob/main/qiskit_optimization/runtime/qaoa_program.py` [visited on 08/11/2021] [cit. on pp. 14, 15].

[21] Michael JD POWELL. "A direct search optimization method that models the objective and constraint functions by linear interpolation." In: *Advances in optimization and numerical analysis.* Springer, 1994, pp. 51–67 [cit. on p. 15].

[22] J.C. SPALL. "Multivariate stochastic approximation using a simultaneous perturbation gradient approximation." In: *IEEE Transactions on Automatic Control* 37.3 [1992], pp. 332–341. DOI: `10.1109/9.119632` [cit. on p. 15].

[23] Alberto PERUZZO et al. "A variational eigenvalue solver on a photonic quantum processor." In: *Nature Communications* 5.1 [July 2014]. ISSN: 2041-1723. DOI: `10.1038/ncomms5213`. URL: `http://dx.doi.org/10.1038/ncomms5213` [cit. on p. 15].

[24] Google Quantum AI. *Quantum variational algorithm.* Aug. 2021. URL: `https://quantumai.google/cirq/tutorials/variational_algorithm` [visited on 08/31/2021] [cit. on p. 15].

[25] MD SAJID ANIS et al. *Qiskit: An Open-source Framework for Quantum Computing.* 2021. DOI: `10.5281/zenodo.2573505` [cit. on p. 21].

[26] Aaron MEURER et al. "SymPy: symbolic computing in Python." In: *PeerJ Computer Science* 3 [Jan. 2017], e103. ISSN: 2376-5992. DOI: `10.7717/peerj-cs.103`. URL: `https://doi.org/10.7717/peerj-cs.103` [cit. on p. 21].

[27] Wes MCKINNEY. "Data Structures for Statistical Computing in Python." In: *Proceedings of the 9th Python in Science Conference.* Ed. by Stéfan van der WALT and Jarrod MILLMAN. 2010, pp. 51–56 [cit. on p. 21].

[28] J. D. HUNTER. "Matplotlib: A 2D graphics environment." In: *Computing in Science & Engineering* 9.3 [2007], pp. 90–95. DOI: `10.1109/MCSE.2007.55` [cit. on p. 22].

[29] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX." In: *Proceedings of the 7th Python in Science Conference.* Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15 [cit. on p. 22].

[30] Qiskit. *qiskit.providers.aer.QasmSimulator.* Aug. 2021. url: `https://qiskit.org/documentation/stubs/qiskit.providers.aer.QasmSimulator.html` [visited on 08/31/2021] [cit. on p. 22].

[31] Fabian Klos. *Routing in a directed graph.* Aug. 2020. url: `https://github.com/PlanQK/Routing/blob/master/Routing%20in%20directed%20graph.ipynb` [visited on 08/09/2021] [cit. on p. 25].

[32] Qiskit. *Qiskit Aqua (NOW DEPRECATED).* Apr. 2021. url: `https://github.com/Qiskit/qiskit-aqua#migration-guide` [visited on 08/20/2021] [cit. on p. 26].

[33] European Union. *Regulation (EC) No 261/2004 of the European Parliament and of the Council of 11 February 2004 establishing common rules on compensation and assistance to passengers in the event of denied boarding and of cancellation or long delay of flights, and repealing Regulation (EEC) No 295/91 (Text with EEA relevance) - Commission Statement.* Feb. 2004. url: `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A32004R0261&qid=1629721615189` [visited on 08/23/2021] [cit. on p. 56].

[34] Daniel J. Egger, Jakub Mareček, and Stefan Woerner. "Warm-starting quantum optimization." In: *Quantum* 5 [June 2021], p. 479. issn: 2521-327X. doi: `10.22331/q-2021-06-17-479`. url: `http://dx.doi.org/10.22331/q-2021-06-17-479` [cit. on p. 63].

[35] Linghua Zhu et al. *An adaptive quantum approximate optimization algorithm for solving combinatorial problems on a quantum computer.* 2020. arXiv: `2005.10258 [quant-ph]` [cit. on p. 63].

[36] Harper R. Grimsley et al. "An adaptive variational algorithm for exact molecular simulations on a quantum computer." In: *Nature Communications* 10 [July 2019]. issn: 2041-1723. doi: `10.1038/s41467-019-10988-2`. url: `https://doi.org/10.1038/s41467-019-10988-2` [cit. on p. 63].