

Thesis Dissertation

**IMPLEMENTATION AND EVALUATION OF ALGORITHMS
ON CLASSICAL AND QUANTUM COMPUTERS**

Sotiris Loizidis

UNIVERSITY OF CYPRUS



DEPARTMENT OF COMPUTER SCIENCE

January 2021

UNIVERSITY OF CYPRUS
DEPARTMENT OF COMPUTER SCIENCE

**Implementation and Evaluation of Algorithms
on Classical and Quantum Computers**

Sotiris Loizidis

Advisor
Associate Professor Demetris Zeinalipour

Thesis submitted in partial fulfilment of the requirements for the award of
degree of Bachelor in Computer Science at University of Cyprus.

January 2021

Acknowledgements

First and foremost, I would like to thank my professor Dr. Zeinalipour, for offering me the chance to work with an academic of his stature and for the high standards he set for this work. He guided me through the hardships and helped me avoid common pitfalls. I would like to thank the special teaching staff of the University of Cyprus and more specifically Dr. Pavlos Antoniou, who reignited my love for Computer Science on many occasions throughout my degree. I would like to thank my family and especially my father for providing for me throughout the course of this project and the entirety of my four years in the university. I would also like to thank my friends, who showed compassion and stayed close to me through the most difficult parts of the writing process. Last but not least I would like to thank Christos Constantinou, who helped me with the mathematical background that was necessary to complete this study and both Marcos-Antonios Charalambous and Andreas Savva, that took the time to provide useful insights concerning the design of this thesis.

We acknowledge the use of IBM Quantum services for this work. The views expressed are those of the authors and do not reflect the official policy or position of IBM or the IBM Quantum team.

Abstract

Quantum computing is an emerging technological breakthrough and it is making strong advancements towards commercialization. Its scope has shifted from the scientific community to a wider audience including major industry organizations such as Google, Amazon, IBM, organized communities interested in Quantum development, and finally independent developers. However, the quantum computing community is still in its early stages and there is a lot more improvement that could be made in the educational aspect concerning resources available and general approach.

This thesis aims to provide a streamlined workflow for Quantum Computing Development with the usage of the IBM Quantum Experience and Qiskit platform. The “Quantum Computing Learning Gate” was developed as a repository on GitHub consisting of four different levels of complexity. The first level is intended for experimenting with Quantum primers. The second level provides an interactive experience for understanding quantum computing behavior through the implementation of “The Exciting Game”, a two-player game that is based on quantum mechanics concepts. The third and fourth levels complete an Implementation and Evaluation of the Deutsch-Jozsa and Bernstein-Vazirani algorithms on quantum and classical systems. In the evaluation of Deutsch-Jozsa, we have found that the quantum implementation shows an exponential speedup over the classical implementation with sufficient input size, however the accuracy of the current quantum systems is not sufficient to produce the correct outcome for the quantum solution since the correct result was found less than 1% of the time. In the evaluation for Bernstein-Vazirani, we have found that the current quantum systems are far from achieving a speedup over the classical implementation, but that the quantum accuracy of the algorithm is much higher than Deutsch-Jozsa, achieving accuracies close to 40% for the largest possible input on a quantum system.

Overall, we conclude that, while Quantum computing is still in its infancy commercially, there can be serious benefits to getting acquainted with the current state-of-the-art quantum frameworks, especially since quantum services are available through the cloud and can be a viable solution to real-world problems.

Contents

Chapter 1	Introduction.....	1
1.1	Motivation.....	1
1.2	Thesis Overview.....	4
1.3	Thesis Contribution.....	5
1.4	Thesis Outline	6
Chapter 2	Related Work and Background.....	7
2.1	Related Work	8
2.1.1	IBM Quantum Experience and Qiskit	8
2.1.2	Amazon Braket	8
2.1.3	D-Wave - Leap.....	8
2.1.4	Quantum Tetris	9
2.1.5	Procedural Game Generation.....	9
2.1.6	Volkswagen Quantum Routing.....	9
2.1.7	Next-Gen Batteries Using Quantum Computers	9
2.2	Background	10
2.2.1	Methodology	10
2.2.2	PyCharm	11
2.2.3	Conda environments	11
2.2.4	Python 3.7	11
2.2.5	NumPy	11
2.2.6	GitHub	12
2.2.7	Matplotlib.....	12
2.2.8	IBM Quantum Platform	12
2.2.9	Qiskit SDK.....	14
2.2.10	Linear Algebra	16
2.2.11	Classical Gates.....	16
2.2.12	Basic Quantum Mechanics	16
Chapter 3	Quantum Computing Primers.....	17
3.1	Key Concepts	17
3.1.1	Superposition	18
3.1.2	Entanglement	18
3.1.3	Quantum Interference – Phase Kickback.....	18
3.2	Qubit.....	18
3.2.1	Bloch Sphere.....	19
3.3	Quantum Circuits	20
3.3.1	Quantum Gates	21
3.3.2	Circuit Composer.....	22

3.3.3	X gate	24
3.3.4	CX “CNOT” Gate	25
3.3.5	Hadamard Gate	26
Chapter 4	Quantum Computing Learning Gate.....	27
4.1	Introduction	27
4.2	Structure Breakdown.....	28
4.2.1	Level 1 – Experimenting with Primers	29
4.2.2	Level 2 – The Exciting Game	29
4.2.3	Level 3 – Implementation of Algorithms	29
4.2.4	Level 4 – Evaluation of Algorithms	30
4.2.5	Independent entities	31
4.3	GitHub Repository	33
Chapter 5	QCLG Level 1 – Implementing Primers.....	34
5.1	Introduction	34
5.2	Achieving Superposition	35
5.2.1	Implementation	36
5.3	The Bell State	37
5.3.1	Implementation	38
5.3.2	Results.....	38
5.4	Phase Kickback	39
5.4.1	Implementation	41
Chapter 6	QCLG Level 2 – The Exciting Game	42
6.1	Concept	42
6.2	Rules.....	43
6.3	Available Cards	43
6.4	Winning Hands	44
6.5	Strategy	49
6.6	Implementation	50
6.6.1	Method Definitions	50
Chapter 7	QCLG Level 3 – Implementation of Algorithms	53
7.1	Deutsch - Jozsa.....	54
7.2	Classical Approach.....	55
7.2.1	Classical Implementation.....	56
7.2.2	Worst-case Scenario for a Classical Solution	59
7.3	Quantum Approach	68
7.3.1	Algorithm’s Structure	69
7.3.2	Proof of Concept with a worked example	70
7.3.3	Quantum Implementation	76

7.4	Bernstein - Vazirani	80
7.5	Classical Approach.....	81
7.5.1	Classical Implementation.....	82
7.6	Quantum Approach	83
7.6.1	Algorithms' Structure	83
7.6.2	Proof of Concept with a worked example	84
7.6.3	Quantum Implementation	87
7.7	Summary	91
Chapter 8	QCLG Level 4 – Evaluation of Algorithms	92
8.1	Evaluation Process	92
8.2	Implementation	93
8.3	Deutsch-Jozsa Measurements	98
8.4	Bernstein-Vazirani Measurements	100
8.5	Additional Assisting Methods.....	101
Chapter 9	Conclusions.....	103
9.1	Conclusions	103
9.2	Limitations	105
9.3	Future Work	106
Bibliography	107
Appendix A	QCLG Level 1 Code	A-1
Appendix B	QCLG Level 2 Code	B-1
Appendix C	QCLG Level 3 Code	C-1
Appendix D	QCLG Level 4 Code	D-1
Appendix E	Supplementary Code	E-1
Appendix F	QCLG Setup	F-1

List of Figures

Figure 1-1 Fugaku Supercomputer - by Fujitsu.....	1
Figure 1-2 IBM Q System One, The first Integrated Quantum Computing System for Commercial Use.	3
Figure 2-1 PyCharm.....	10
Figure 2-2 IBM Quantum.	10
Figure 2-3 GitHub.....	10
Figure 2-4 IBM Quantum Experience Interfaces.	12
Figure 2-5 IBM Quantum Experience Dashboard.....	13
Figure 2-6 The Qiskit Elements.....	14
Figure 3-1 Quantum Optics and Quantum Many-body Systems Andrew Daley's Research Group at the University of Strathclyde [36].....	19
Figure 3-2 Bloch Sphere from the Qiskit Textbook [17].....	20
Figure 3-3 A Circuit Example, Containing Multiple Quantum Gates, and Operations.	20
Figure 3-4 A Qubit State Transformation Through a Quantum Gate Application.	21
Figure 3-5 A More Comprehensive View on the Circuit Composer.....	22
Figure 3-6 A Detailed View of Circuit Construction.	22
Figure 3-7 Measurement Probabilities.....	23
Figure 3-8 Q-Sphere Representation.	23
Figure 3-9 X Gate Behaviour.....	24
Figure 3-10 CX or C-NOT Gate Behaviour.	25
Figure 3-11 Hadamard Gate Behaviours.	26
Figure 4-1 The QCLG Platform.	28
Figure 4-2 The Project Tree.....	32
Figure 4-3 DMSL GitHub Page.....	33
Figure 4-4 QCLG Table of Contents.	33
Figure 5-1 Code for Achieving the Superposition of Three Qubits.	36
Figure 5-2 Bell State Experiment.	37
Figure 5-3 Code for Method run() of Experiment Bell State.	38
Figure 5-4 Results for Experiment Bell State.....	38
Figure 5-5 Phase Kickback effect.....	39
Figure 5-6 Code for the run()Method of the Phase Kickback Experiment.....	41
Figure 5-7 Results for Phase Kickback.....	41
Figure 6-1 The Exciting Game Example 1.	44
Figure 6-2 The Exciting Game Example 2.	44
Figure 6-3 The Exciting Game Example 3.	45
Figure 6-4 The Exciting Game Example 4.	45
Figure 6-5 The Exciting Game Example 5.	46
Figure 6-6 The Exciting Game Example 6.	46
Figure 6-7 The Exciting Game Example 7.	47
Figure 6-8 The Exciting Game Example 8.	47
Figure 6-9 The Exciting Game Example 9.	48
Figure 6-10 Code for the run() Method of The Exciting Game.....	50

Figure 6-11 Code for place_gate() Method of The Exciting Game.....	51
Figure 6-12 Code for the fix_hand() Method of The Exciting Game.....	52
Figure 7-1 Constant-Balanced Function Definition.....	54
Figure 7-2 Deutsch-Jozsa Classical Approach Abstract Form.	55
Figure 7-3 Code for Constructing the Classical Oracle for Deutsch-Jozsa.	57
Figure 7-4 Code for Executing the Classical Solution of Deutsch-Jozsa.	58
Figure 7-5 Worst-Case Scenario for Classical Deutsch-Jozsa.....	59
Figure 7-6 Helper Methods for Constructing the Worst-Case Scenario.....	59
Figure 7-7 All different Combinations for 4 bits.	61
Figure 7-8 Code for Generating n-bit Combinations.....	62
Figure 7-9 The n-bit Combinations Grouped by Number of Ones.	64
Figure 7-10 Code for Rearranging the bit Combinations to Construct Worst Scenario.	65
Figure 7-11 Output of Worst Scenario for 4 bits.	66
Figure 7-12 Execution Times for Classical Solution for 12 bits.	66
Figure 7-13 Execution times for Classical Solution for 23 bits.....	67
Figure 7-14 Execution Times for Classical Solution for 24 bits.	67
Figure 7-15 Deutsch Jozsa Structure.	69
Figure 7-16 Worked Example.....	70
Figure 7-17 Measurement Probabilities for Balanced Function.	75
Figure 7-18 Code for Creating a Balanced Oracle.	77
Figure 7-19 Code for Implementing Deutsch-Jozsa/	78
Figure 7-20 Different States Measured for Deutsch-Jozsa.....	79
Figure 7-21 Results of a Classical and Quantum Execution for Deutsch-Jozsa.....	79
Figure 7-22 Bernstein-Vazirani Abstract Form.	80
Figure 7-23 Bernstein-Vazirani Classical Approach.	81
Figure 7-24 Code for Guessing a Secret Number.....	82
Figure 7-25 Bernstein-Vazirani Quantum Structure.....	83
Figure 7-26 Bernstein-Vazirani Worked Example.	84
Figure 7-27 Code for Bernstein-Vazirani Implementation.....	88
Figure 7-28 Code for Creating the Oracle for the Bernstein-Vazirani Algorithm.....	89
Figure 7-29 State Measurements for Bernstein-Vazirani.	90
Figure 7-30 Execution Results for Bernstein-Vazirani.....	90
Figure 8-1 Code for Redirecting to Algorithm Evaluation.....	93
Figure 8-2 Code for Plotting Evaluation Results.....	94
Figure 8-3 Code for Evaluating the Deutsch-Jozsa Algorithm.....	96
Figure 8-4 Deutsch-Jozsa Evaluation Measurements.....	98
Figure 8-5 Bernstein-Vazirani Evaluation Measurements.....	100
Figure 8-6 Code for Supplementary Methods of Evaluation.....	101

List of Tables

Table 1 XOR and CNOT Truth Tables.....	68
--	----

Chapter 1

Introduction

1.1	Motivation	1
1.2	Thesis Overview.....	4
1.3	Thesis Contribution.....	5
1.4	Thesis Outline	6

1.1 Motivation

The reasoning behind the existence of Computing is solving problems better than humans. Better meaning more efficiently, faster, and with fewer errors. The first “modern” computers were developed in the 1950s and we have made great strides in computation since then. However, as our calculative force increases, so does our curiosity and desire to conquer harder problems. Conventional machines, at their strongest form, can now output a staggering performance of upwards of **10^{17} Floating Point Operations per second** [22],



Figure 1-1 Fugaku Supercomputer - by Fujitsu.

and the time we enter Exascale Computing is only some years away. We have created machines that can ridicule any given every day intensive calculation, yet for some problems, we can still make them calculate until the end of time and then some. A famous example is prime factorization which given a large enough number can prove surprisingly time-consuming. The best-known algorithm that solves the prime factorization problem

scales in sub-exponential time and as we know we can make some pretty big numbers considering the fact they are infinite. Now when we begin to look at some more specific problems it can get somewhat overwhelming. Examples of problems that can't be quite contained in a classical environment are quite more common than we would think, especially in the scientific community.

The simulation of a Quantum System for example or complex molecular structures found in Chemistry are two major examples that show the weakness of Classical Computers. In Newtonian mechanics, we can describe a system in a pretty discrete manner. However when we want to describe the dynamics of a Quantum System and we have to take into account the “magical” concepts of “Superposition” and “Entanglement” we are looking at a behaviour where two quantum objects can be perfectly correlated even though when looked individually can be described as random. An example given to often describe entanglement is that of a coin toss. Suppose we have two double-sided coins. When we measure the first coin it's up roughly half of the times and down the rest. However, the measurement of the second coin is always the opposite of that of the first coin. Already we are dealing with some non-classical properties. If we increase the number of “coins” or particles it can be shown that the number of all possible combinations of such a system can prove to be an intractable problem for a conventional machine [14].

These types of arguments were formally proposed to the Computer Science field by renowned physicist Richard Feynman, who urged the Computer Science community to take advantage of these concepts to try and move from theory to practice [7]. The initial motivation for the notion of Quantum Computing is these complex problems, which cannot yet be fully implemented on a practical scale since Quantum Computing is still in infancy and still heavily debated as to its contribution capabilities as stated here [10] and here [6]. However, the number of people who can experiment with these concepts in their simplest form and potentially contribute to the development is vastly bigger than what it used to be in the fifties.

With emerging technologies such as Cloud Computing Services, which give access to real quantum systems, communities of like-minded people who are curious about the elusive concepts behind what makes Quantum Computing work, we can argue that post-quantum cryptography and all the other applications are closer than we think.

Although it seems to be that these applications are limited it is very likely that there will be some form of commercialized use in areas such as Cybersecurity, Weather Forecasting, Drug Development, and Artificial Intelligence. With behemoths such as IBM, Google, Microsoft, Amazon, JP Morgan Chase, Volkswagen Group already seriously investing in Quantum resources it does not seem like a bad idea to educate ourselves in this field.



Figure 1-2 IBM Q System One, The first Integrated Quantum Computing System for Commercial Use.

For this undergraduate thesis, our goal is to familiarize with the State of the Art Quantum technologies that are available to software developers and share our findings in the methodology of implementing some Quantum Algorithms while making practical use of the physical resources available through the cloud of IBM Quantum Experience. Along the way, we will share our understanding of some famous algorithms and concepts from a programmer's point of view, in our journey to prepare for the upcoming "Quantum Revolution."

1.2 Thesis Overview

Quantum Computing is an elusive concept, not only regarding Information Science but in Physics as well. This thesis is multi-purposed. Our objectives are as follows:

- Demystify some of the “simpler” concepts of Quantum Computing with pragmatic examples.
- Demonstrate a practical workflow that allows a somewhat adequately experienced individual in terms of pure Programming capability and Internet Technologies to start experimenting with Quantum Computing and avoid common pitfalls with the creation of the Quantum Computing Learning Gate which we will introduce in Chapter 4.
- Utilize our findings in order to have a better understanding of some of the more famous Quantum Computing Algorithms in order to implement them and evaluate them against classical implementations.
- Demonstrate the peculiar functionality of Quantum computers by making a minigame that is solely based on the properties of Quantum Mechanics.

Before proceeding with Qiskit experimentation we must state a few disclaimers as well. Below we provide what is **not** a goal of this thesis.

- Qiskit is a very elaborate framework that consists of hundreds of dedicated instructions, [32], and the purpose of this thesis is **not** to provide an introductory course to Qiskit usages and features. The reader is strongly advised to study the official Qiskit material which is provided here [34].
- This thesis is not claiming to **rigorously** prove physical proofs of why these Quantum Systems work or how they work. Whatever worked example we provide, is strictly based on our level of understanding and for the sake of explaining algorithm behaviours.

During our research, it was quickly made clear that quantum computing is not a final solution to all problems of Computer Science [10], [13], [6]. However, since so many behemoths of the industry have chosen to move quantum computing forward, we, as computer scientists need to raise a number of research questions.

The first research question that we raise, concerns the crowd of quantum computing. Is quantum computing for everyone? Is it a subject that should be compulsory in every university's curriculum?

The second research question is about which are the current State-of-the-Art resources available for quantum computing and which one we should choose depending on our level of expertise or needs. During our research, we had to choose a medium in which we could conduct our experiments and acquire valuable information in order to educate ourselves on a completely new subject.

The third and final research question is whether or not a career specialized in quantum computing is a viable option in the next 5 to 10 years. This is of course a question that is dependent on many different factors. Either that be major technological breakthroughs or agenda shifts of major corporations it is still something that we need to ask ourselves.

1.3 Thesis Contribution

With the completion of this thesis we achieve the following milestones:

The first milestone is the completion of a functional education portal which we call “Quantum Computing Learning Gate” or “QCLG”. Through the development of this repository in GitHub, we can offer a no-nonsense approach to quantum computing education through a variety of different experiments, algorithm analysis, evaluation, and interactive tools.

The second milestone is the creation of “The Exciting Game”, an interactive two-player game, incorporated in the QCLG platform, which can help individuals familiarize themselves with basic quantum concepts through a gaming experience.

The third milestone is a comparison study of algorithms on quantum and classical computers. From this, useful data was extracted concerning two major algorithms in the field of quantum computation by simply building upon the basic blocks available through the Qiskit platform.

The fourth milestone is the expansibility of our platform. Not only can one learn from QCLG, but they can also contribute to any of the different levels of development available in QCLG.

The fifth milestone is that our platform can be a central point for further quantum computing exploration since inside QCLG, we list some different sources that we have found extremely useful throughout our own research.

1.4 Thesis Outline

In the first chapter, we roughly presented the incentive of this thesis and talked about the current picture of Quantum Computing. We also raised some research questions that will be answered by the end of this thesis. In the second chapter, we will talk about the various frameworks available for quantum computing and showcase some interesting applications that are closely related to our work. In addition, we will discuss the recommended background knowledge required. In the third chapter, we discuss the basic components that help us construct more complex algorithms. In the fourth chapter, we provide a high-level description of the structure of QCLG. The next chapters will directly expand on the structure given in this chapter. In the fifth chapter, we dive into the first level of QCLG and execute and analyse some simple experiments that underline the major quantum properties we will use in the rest of this thesis. In the sixth chapter, we take an extensive look into the second level of QCLG which is about “The Exciting Game” and how it can help boost the usefulness of the whole project. In the seventh chapter, we showcase level 3 of QCLG by implementing in detail both classical and quantum approaches to two famous algorithms, Deutsch-Jozsa and Bernstein-Vazirani. In the eighth chapter, we showcase the final level of our platform which is the evaluation of algorithms. Here we implement a series of automated tests in order to extract useful data about the two algorithms. In the ninth chapter, we present our technical conclusions concerning the development of programs with Quantum modules and discuss limitations and future work.

Chapter 2

Related Work and Background

2.1	Related Work.....	8
2.1.1	IBM Quantum Experience and Qiskit.....	8
2.1.2	Amazon Braket	8
2.1.3	D-Wave - Leap.....	8
2.1.4	Quantum Tetris	9
2.1.5	Procedural Game Generation.....	9
2.1.6	Volkswagen Quantum Routing.....	9
2.1.7	Next-Gen Batteries Using Quantum Computers.....	9
2.2	Background	10
2.2.1	Methodology	10
2.2.2	PyCharm	11
2.2.3	Conda environments	11
2.2.4	Python 3.7	11
2.2.5	NumPy	11
2.2.6	GitHub.....	12
2.2.7	Matplotlib.....	12
2.2.8	IBM Quantum Platform	12
2.2.9	Qiskit SDK.....	14
2.2.10	Linear Algebra	16
2.2.11	Classical Gates	16
2.2.12	Basic Quantum Mechanics.....	16

2.1 Related Work

Since this thesis is multi-purposed, there is a number of different works that are close to our efforts.

2.1.1 IBM Quantum Experience and Qiskit

The first and major example is the Qiskit framework and textbook, which presents an abundance of information regarding basic quantum concepts, along with worked examples and dives into deeper and more complex problems. The largest portion of this thesis was inspired by the Qiskit textbook. Qiskit is an open-source framework that started in March of 2017. Developed by IBM Research, it now has a strong following with the Qiskit community and has a public repository on GitHub. Qiskit is mainly developed in Python and is cross-platform compatible. The Qiskit textbook is a university-level supplement to quantum computing courses. It offers the ability to work with machine-level code with OpenQASM, [4], as well as high-level abstractions using Python. All of Qiskit is encapsulated in the IBM Q platform, [25]. IBM offers free experimentation on real quantum systems, as well as simulators that work under the Qiskit framework. It also offers interactive tools with no need for coding and a plethora of information through a very well-built documentation library, [26] .

2.1.2 Amazon Braket

Amazon is a giant in the industry and it is not a surprise that it now offers an elaborate cloud quantum computing service for researchers and developers [16]. However, Amazon Web Services in general, require a certain level of specialized expertise, which was impossible to achieve in an academic semester in order to accomplish our goals for this thesis. It is worth noting that Amazon uses D-Wave systems.

2.1.3 D-Wave - Leap

D-Wave was founded in 1999 and is a commercial supplier of quantum computers. D-Wave systems are used by the leading technology organisations including Google, NASA, Volkswagen. The research that stems from D-Wave is very rich with over 100 peer-reviewed papers in scientific journals. In 2018 D-Wave launched the Leap, their own cloud quantum computing service which contains open-source educational material meant for developers, researchers, and businesses. Leap² is directed to people

looking for hybrid solutions that tackle problems with the best mix of quantum and classical resources. It was made for the development of hybrid quantum applications that are not only directed in scientific experimentation but real-world problems, [27],[21].

2.1.4 Quantum Tetris

In the course of work, we managed to implement a quantum game in its infancy. A motivation for this was Quantum Tetris [35]. It is about the famous game but with a quantum spin. It is implemented using IBM resources and Qiskit. While it is quite a different game than what we will present it opened our eyes concerning the simple applications that quantum computing might have.

2.1.5 Procedural Game Generation

As discussed here, [15], there are major efforts to start producing procedural generation for games and more using quantum computing. Concepts such as blur seem to have a better place in the hands of the stronger quantum computers that will come.

2.1.6 Volkswagen Quantum Routing

Volkswagen was the first company to demonstrate successfully a real-world application for quantum computers. Through demonstration on “quantum buses”, Volkswagen presented a live use of a system designed to calculate the fastest routes in almost-real time using D-Wave’s systems [38].

2.1.7 Next-Gen Batteries Using Quantum Computers

In a research paper conducted by joint efforts of IBM and Daimler AG [11], the parent company of Mercedes-Benz, simulations were conducted, of molecules that are formed in lithium-sulfur batteries while operations. This was demonstrated using IBM Q premium-access systems. The goal of this research is to pave the way for the design of the next generation of batteries installed in electric vehicles. We can see that more and more industries have a direct benefit from the development of quantum computing.

2.2 Background

2.2.1 Methodology

For this thesis, we have worked with IBM'S Quantum Experience resources. The operating systems we used are Ubuntu 18.04 and Windows 10, the programming language is Python and the main external library used to implement quantum programs is Qiskit. We have deployed a repository in GitHub and keep all our code. IBM Q offers extensive tutorials in order to set up a local environment capable of accessing IBM'S systems. It is important to note that the default medium for conducting quantum experiments is either's through Quantum Experience's interactive tools such as the Quantum Lab, where the user can design from scratch and execute a Quantum Circuit, or through Jupyter Notebooks where you can write code in normal Python with the necessary instructions from Qiskit in order to implement the Quantum Algorithm. For this thesis, we chose to work with PyCharm since it's a very popular Integrated Development Environment and good practice for testing out IBM's API outside of the recommended frameworks.



Figure 2-1 PyCharm.



Figure 2-2 IBM Quantum.



Figure 2-3 GitHub.

2.2.2 PyCharm

PyCharm is an Integrated Development Environment created by JetBrains. Aside from being a great Python IDE it also possesses multiple versioning control features with built-in GitHub support as well as Conda environments. More information can be found here [30].

2.2.3 Conda environments

In order to maintain universality across Operating Systems, we opted to work with Conda environments. They provide the ability to work with a Python interpreter which can be used in different Operating Systems, [18]. Our Conda virtual environment contains:

- Python 3.7
- Qiskit
- NumPy (version: 1.19.3)
- Matplotlib

This environment can easily be created using the online instructions provided in the conda documentation and then proceed to install the aforementioned dependencies using pip install. Any clarifications for Ubuntu commands can be found here [37].

2.2.4 Python 3.7

Python is an open-source, interpreted-style, programming language that is widely used both in the industry and in academia, because of its simplicity and open-source nature. It is also the underlying language that Qiskit uses. For this thesis, in order to create a stable environment where multiple dependencies coexist, we used Python 3.7, [31].

2.2.5 NumPy

NumPy is one of the dependencies required by Qiskit and one that we had to install manually to ensure a stable working environment. Qiskit being dependant on NumPy makes a lot of sense, since as we will see later, Qubits and System States, in general, are described by vectors and matrices, [29].

2.2.6 GitHub

GitHub is a famous code hosting platform for version control using Git. We used GitHub in order to ensure a clean workflow and easily transfer our work locally to another device, [23].

2.2.7 Matplotlib

Matplotlib is a very strong library for creating visualizations in Python. This was especially useful since Qiskit supports a number of different visualizations and proved vital in order to debug Quantum Circuit construction and help us understand the algorithms' structure in a more interactive way, [28].

2.2.8 IBM Quantum Platform

The IBM Quantum Platform has a lot of useful tools that can help with quantum development, without the use of code. An excellent tool is the circuit composer, [24], which is a very nice way of experimenting with simple experiments, as well as complicated algorithms. We present a more detailed overview of the circuit composer in 3.3.2. The interface of the IBM Quantum Platform is very rich. Besides the circuit composer, the user can view the available quantum systems, access relevant documentation, and develop quantum programs using different tools.

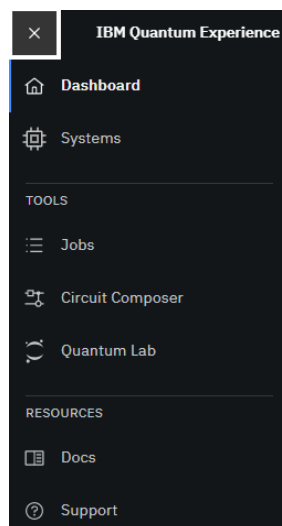


Figure 2-4 IBM Quantum Experience Interfaces.

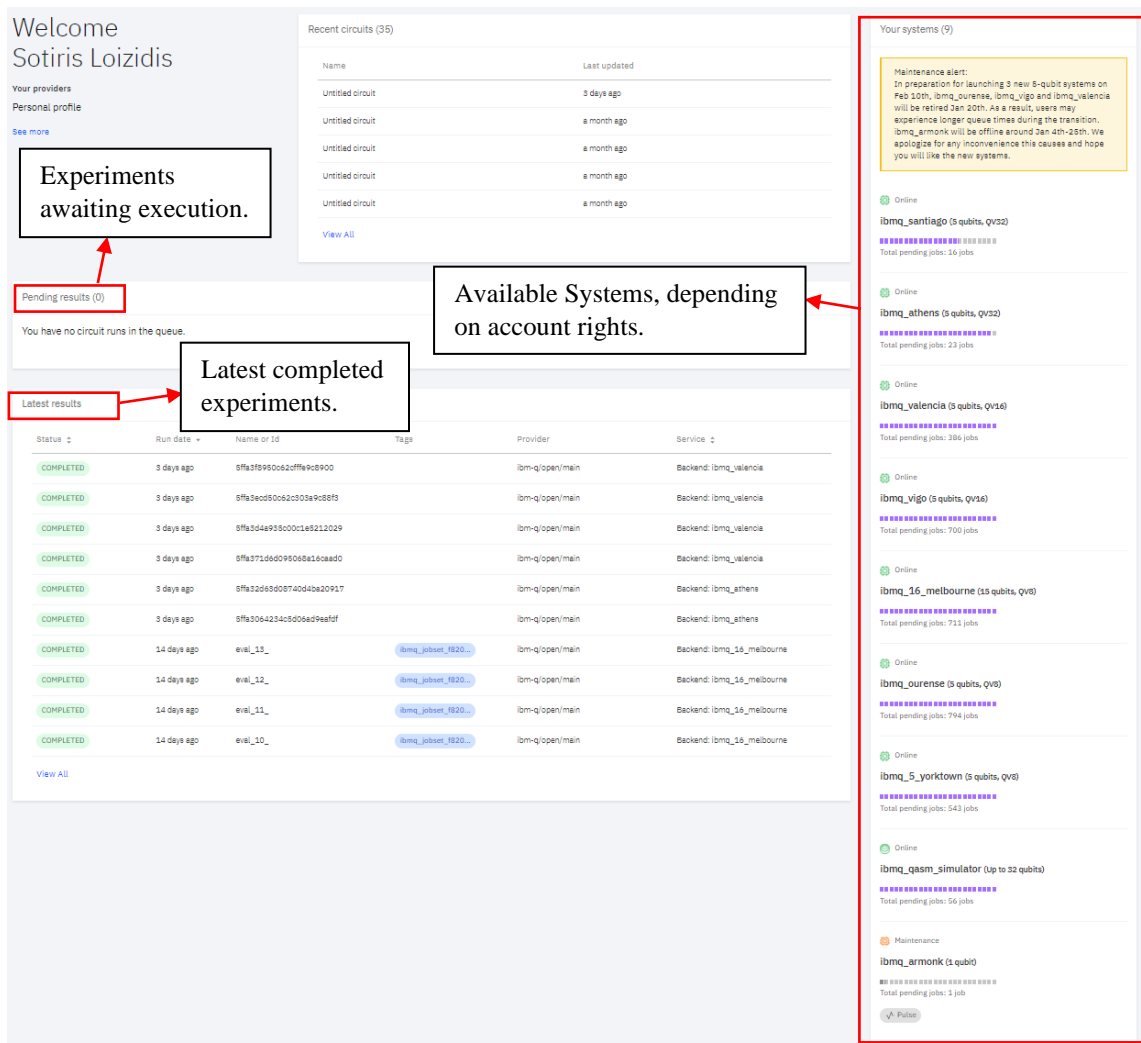


Figure 2-5 IBM Quantum Experience Dashboard

In Figure 2-5 we can see useful information regarding the available systems, as well as a history of all experiments. Clicking on one of the systems, the user can see information regarding a system's hardware.

In the Quantum Lab, shown in Figure 2-4, the user can create Jupyter Notebooks, which is the default way for Quantum development using Qiskit.

In the Jobs label we can see the progress of completed experiments or experiments waiting in the systems queues.

We have implemented necessary methods to bypass long queues by finding the least busy backend capable of executing our experiment. These methods can be found in Appendix E.

2.2.9 Qiskit SDK

Qiskit is an open-source Software Development Kit that is used for development using Quantum Computers. It can be used to develop low-level Quantum Instructions by tinkering with pulses or to develop higher-level applications using Quantum Circuits and complete application modules.

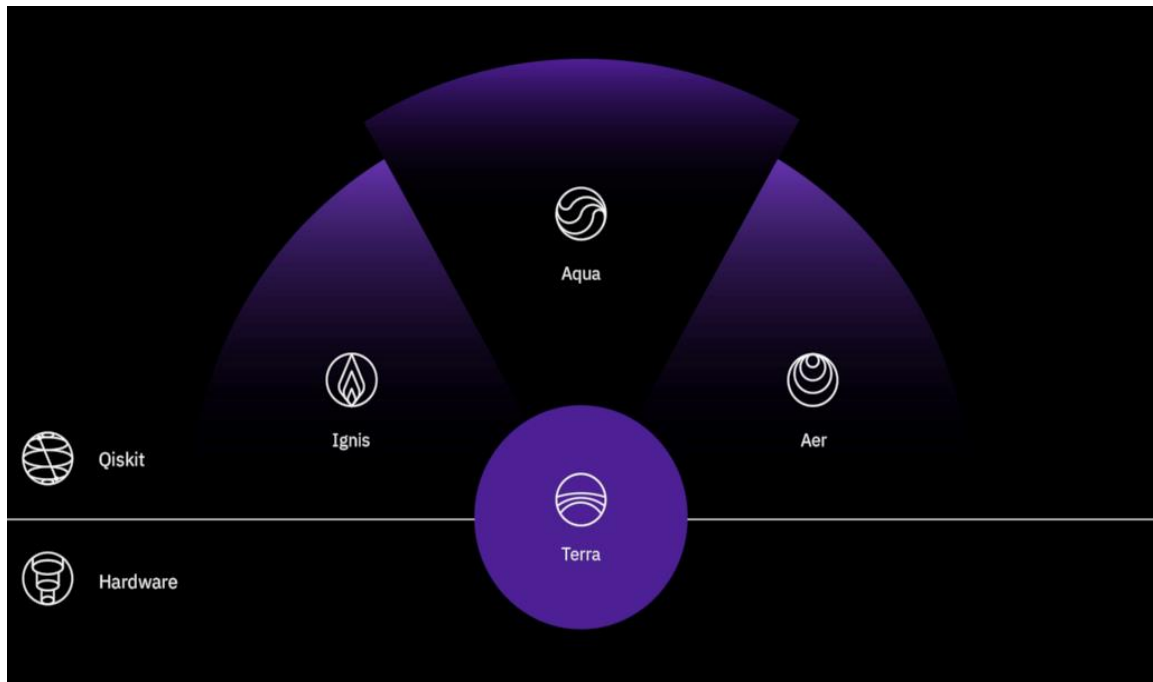


Figure 2-6 The Qiskit Elements.

From the picture above we can see that Qiskit is divided into four elements. Each element has its purpose. Terra is the base for all other elements. It contains the modules necessary to develop on the lowest level, modules necessary for parameter tuning in different hardware as well as modules that are responsible to handle the completed circuits as executable jobs to the designated systems. Aer is responsible helps as a benchmark of the Quantum Computing efforts by providing a realistic simulation of Quantum Systems using classical computers. Ignis is an effort to combat error in quantum systems by providing different ways of examining quantum circuits, gates, or code. Aqua is focused on Quantum Computing Applications. It requires expertise in Quantum Computing and was created for domain-specific experts. More information can be found here [1], here [32], and here [34].

In our thesis, we mainly focus on the modules provided by Terra and Aer. We list the most useful to our thesis:

Terra:

- `qiskit.circuit`, A model where operations or qubits are performed using quantum gates.
- `qiskit.providers`, Useful for interacting with circuits that are currently running.
 - **Provider**: Provides access to different backends, according to provider rights.
 - **Backend**: abstractly represents the real system or simulator and is responsible for executing the quantum circuits and returning the results.
 - **Job**: A job is essentially the key for a certain experiment. It is useful for keeping track of the progress of the experiment, queued, running, etc, and providing the ability to control it.
 - **Result**: An object which keeps the quantum data of a remote backend from a completed experiment. It can be instantiated using `result = job.result()`. There are multiple ways to manipulate this data but the most common is through `result.get_counts(circuit)`.
- `qiskit.visualization`, Contains multiple tools for visualizing quantum components.

Aer:

QasmSimulator: Powerful Simulator that allows for the execution of experiments in ideal quantum computers or even environments close to real systems. For this thesis, we use it for its ability to host ideal circumstances.

2.2.10 Linear Algebra

Quantum computing requires a certain level of experience in linear algebra. From complex numbers to vectors to matrices, to multiplication and addition of different elements, there is a lot to learn if one wants to understand better how quantum systems behave. It is probably the most useful tool in order to have a real understanding of the qubit states and how they change. More information on linear algebra for quantum computers can be found here [2] on page 17.

2.2.11 Classical Gates

Quantum computing uses the quantum gates as abstractions to the actions performed on qubits. Therefore, although classical and quantum computation have major differences, we can see that there is some common ground. Many of the gates used in Qiskit are similar to traditional classical gates.

One example is the classical NOT gate which flips the state of a bit. In quantum computing, there is the X gate which flips the state of a qubit.

Another example is the XOR gate which acts based on two bits. The corresponding gate of quantum computing is the CX gate or C-NOT gate.

An individual would certainly find it easier to experiment with quantum gates if they revisit the classical gates [9].

2.2.12 Basic Quantum Mechanics

The hardware of quantum computing is nothing like a classical computer. It is advised that the aspiring quantum programmer acquires basic knowledge of the materials needed in order for a quantum system to exist. More can be found in [2], chapter 1.

Chapter 3

Quantum Computing Primers

3.1	Key Concepts	17
3.1.1	Superposition	18
3.1.2	Entanglement	18
3.1.3	Quantum Interference – Phase Kickback.....	18
3.2	Qubit.....	18
3.2.1	Bloch Sphere.....	19
3.3	Quantum Circuits	20
3.3.1	Quantum Gates	21
3.3.2	Circuit Composer.....	22
3.3.3	X gate.....	24
3.3.4	CX “CNOT” Gate.....	25
3.3.5	Hadamard Gate	26

3.1 Key Concepts

Quantum Computing has some key concept differences from classical Computation.

More information concerning these concepts can be found here: [2].

- Superposition
- Entanglement
- Quantum Interference

3.1.1 **Superposition**

Superposition is something that is not a property of Newtonian mechanics. Superposition is the ability of a Quantum System to be in a combination of multiple states at once. Suppose we have a Quantum Computer with n qubits, there can be a superposition of all possible 2^n states at once. This property is useful in many Quantum Algorithms.

3.1.2 **Entanglement**

Entanglement is perhaps one of the most curious concepts in the world of Quantum Mechanics. It is closely related to superpositions and it tells us that two or more particles are in such an assortment that their quantum state cannot be described independently of the others in that same “correlated” group. Abstractly speaking, the behaviour, of one particle will directly affect or be affected by the behaviour of some other particles that are somehow related to that particle.

3.1.3 **Quantum Interference – Phase Kickback**

A fundamental idea in Quantum Computing is being able to control the probability of measuring certain states. It is a by-product of the superposition and it allows us to tinker with certain superpositions in order to steer the system in a deterministic outcome which is useful to us.

3.2 **Qubit**

The bit is the simplest single point of information a Classical Computer can describe. If we strip away all ingenious hardware and software designs of the modern machines, a bit is what we are left with. It is the “atom” in the universe of traditional Computation. A bit holds a discrete value of either one or zero. The most used processors today are children of the 64-bit architecture. Explaining 64-bit architecture very roughly we, can say that at least theoretically an architecture of this magnitude that a 64-bit processor can address 2^{64} bytes or 16 exabytes of byte-addressable memory, a memory that can be referenced for each byte with a different address. In Quantum Computing, our “atom” of computation can quite literally be an atom.

While a bit can be described with the functionality of the flip-flop, for example, the qubit is a bit more complicated. There is a number of ways that a qubit can be implemented but let's just imagine an electron. An electron has the property of spin, which we will not delve into, but it can translate to two separate states, spin up and spin down, thus creating a two-state quantum-mechanical system [2].

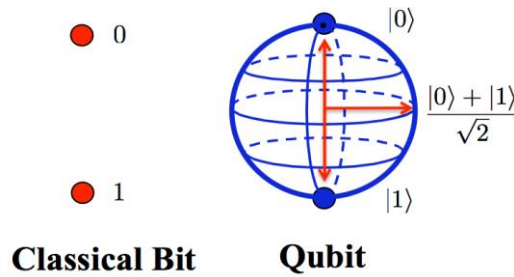


Figure 3-1 Quantum Optics and Quantum Many-body Systems Andrew Daley's Research Group at the University of Strathclyde [36].

What differentiates the qubit from the bit is the additional physical properties acquired by quantum mechanics. Quantum mechanics allow the qubit to exist in a state of both 1 and 0 simultaneously. This property of the qubit is fundamental in quantum computing.

3.2.1 Bloch Sphere

Named after physicist Felix Bloch, the Bloch sphere gives a geometrical representation of the qubit. From the figure below we can observe three axes. X, Y, and Z. The north and south poles of the system as chosen as the ground and excited states of the qubit $|0\rangle$ and $|1\rangle$ and correspond to the spin-down and spin-up of the electron. Points on the surface of the sphere are defined as pure states, states that when are measured can be expected to yield a certain value, and points within the surface of the sphere are defined as mixed states or states that are a statistical ensemble of pure states, which essentially means that they are in a state that is a “superposition” of pure states and we cannot be certain on which of the pure states we will end up measuring. It is worth noting that the “amount” of a pure state in a mixed state configuration is determined by a complex number.

These states are described using Dirac notation. The general case for the two basis states of $|0\rangle$ and $|1\rangle$ is:

$c_0|0\rangle + c_1|1\rangle$ where c_0, c_1 are complex numbers.

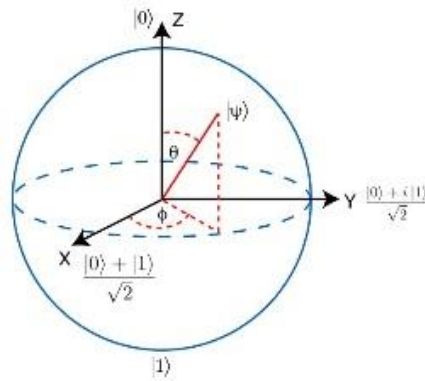


Figure 3-2 Bloch Sphere from the Qiskit Textbook [17].

3.3 Quantum Circuits

Quantum circuits are a model in which a computation is performed with a sequence of quantum gates, which are reversible transformations on a quantum mechanical comparable of an n-bit register. This analogous structure is referred to as an n-qubit register. Further operations can be performed on a quantum circuit like logical operations, barriers in order to control the circuit compilation, and of course measurements in order to get results.

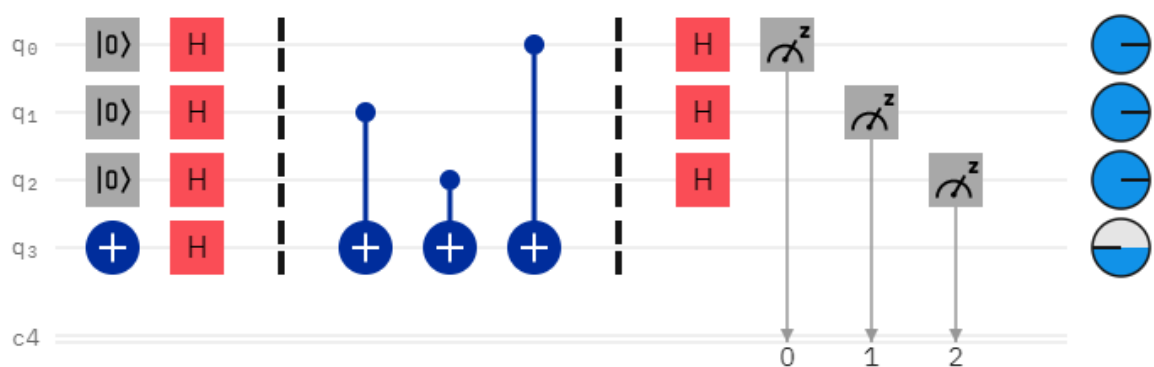


Figure 3-3 A Circuit Example, Containing Multiple Quantum Gates, and Operations.

3.3.1 Quantum Gates

The reader should be familiar with the classical gates of computation such as the AND, OR, XOR, NAND, and so on. The purpose of quantum gates is to set the qubit into a certain physical state so that we can dictate with certainty or to some degree the outcome of the measurement of the qubit or in order to further develop the state of the quantum system as a whole. Quantum gates can be unary operators, acting on a single qubit, or even operators that include multiple qubits in order to be implemented. What quantum gates essentially do, is rotate the qubit to a state. All quantum gates can be constructed by programming them to “rotate” the qubit on any combination of the three axes shown in the Bloch sphere in order to reach a distinct state. There is an infinite number of different physical states that we can shape but below we show a few significant gates that help us on the rest of this thesis. Quantum gates are reversible. Thus, they can be represented as matrices, manipulating the state vector of the Bloch Sphere. Each time a gate operation is performed on a qubit, we can calculate the new state of the qubit by matrix and vector multiplication to find the new state.

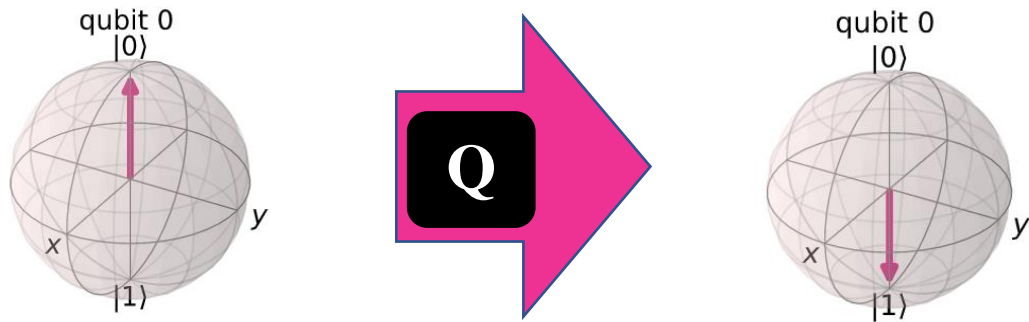


Figure 3-4 A Qubit State Transformation Through a Quantum Gate Application.

One important note is that we can perform operations on different bases, the three mainstream bases being X, Y, and Z.

Looking at

Figure 3-4 we can see that we could have an infinite number of bases by simply using two orthogonal vectors each time. For all experiments shown in this thesis, we will use the Z-basis, which is the most popular.

The two basis states, $|0\rangle$ and $|1\rangle$ are represented in a vector like so:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

3.3.2 Circuit Composer

A very helpful way of understanding and experimenting with quantum gates is the Circuit Composer tool, provided by IBM Q, where we can interactively place quantum gates on a “quantum circuit” and perform all kinds of operations including, gate operations, classical logical operations, measurements and more.

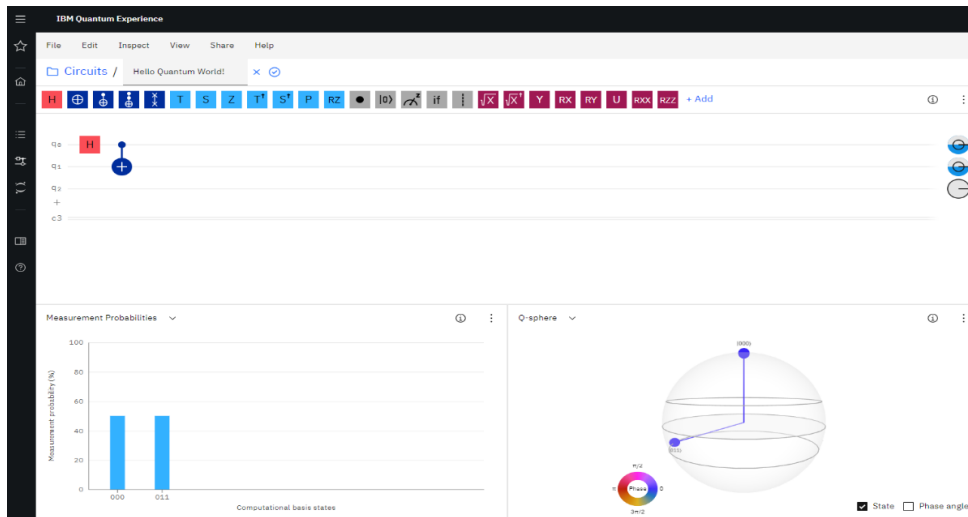


Figure 3-5 A More Comprehensive View on the Circuit Composer.

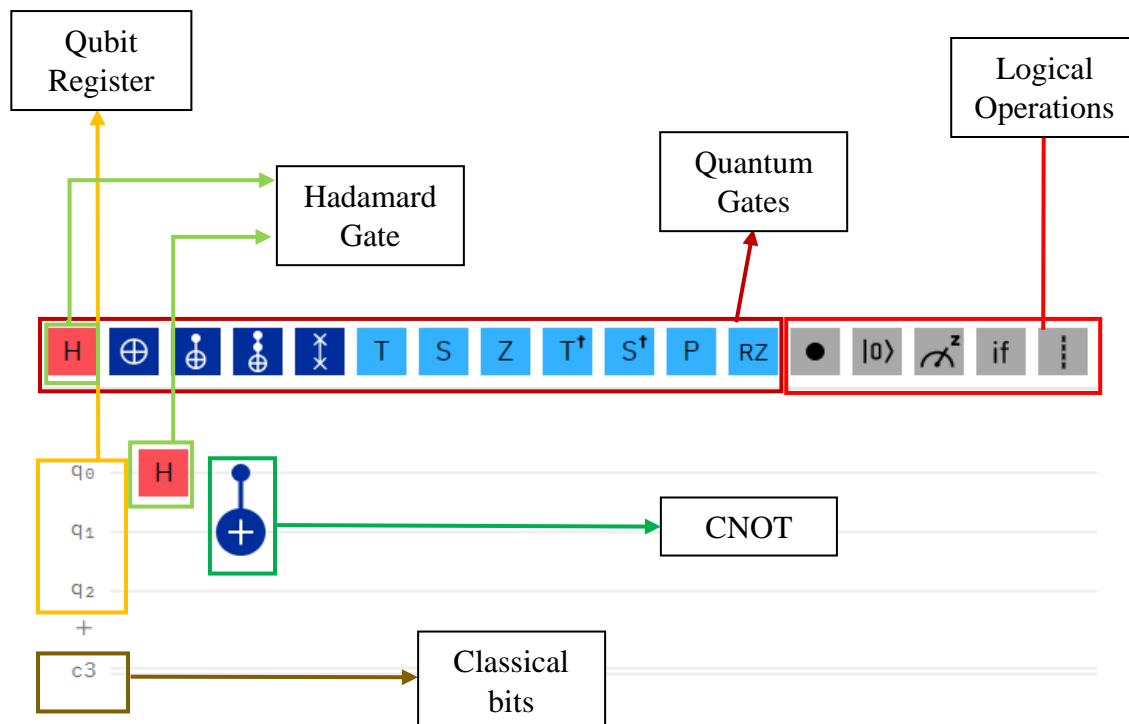
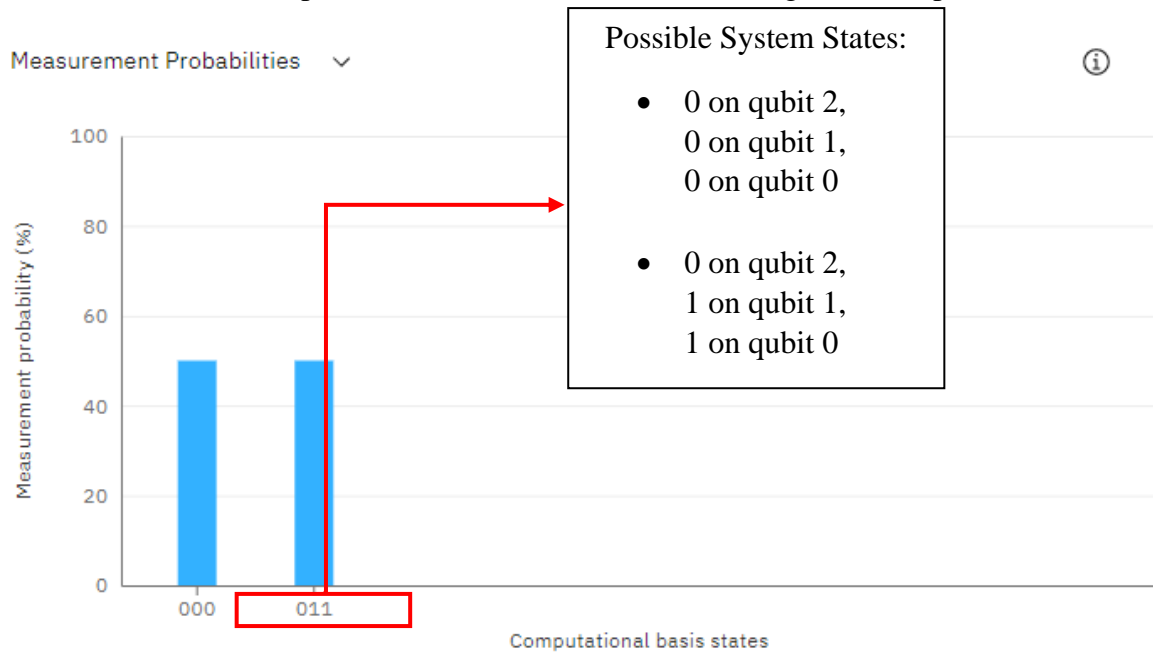


Figure 3-6 A Detailed View of Circuit Construction.

The measurement probabilities section showcase a histogram of all possible states of the



system.

Figure 3-7 Measurement Probabilities.

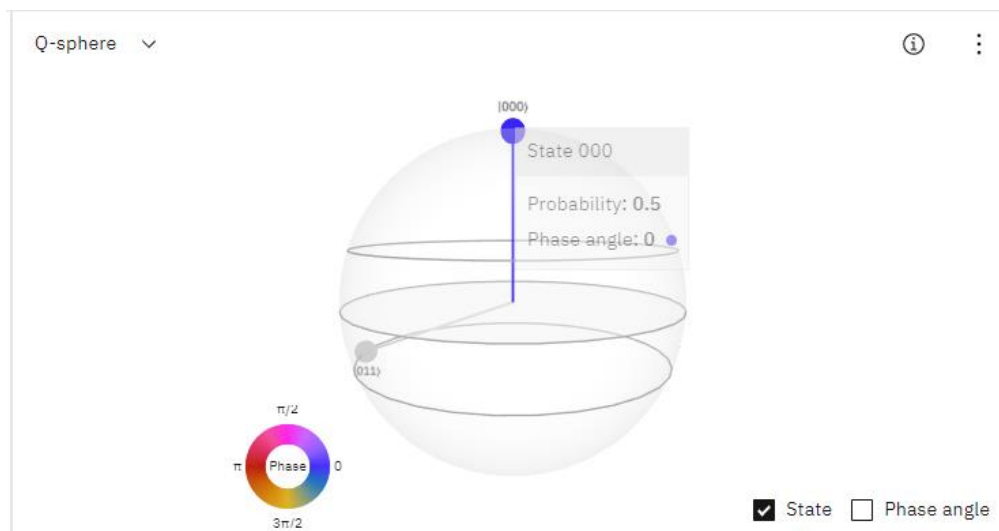


Figure 3-8 Q-Sphere Representation.

The Q- sphere representation is a very nice way of visualizing the global scale of the system. There are a lot of compact details about the system in this single frame. It is different than a Bloch sphere visualisation since it shows the possible states of the system of qubits as a whole. We can also examine the likelihood of each system state and the angle that the phase on the measuring base.

3.3.3 X gate

The NOT gate of Quantum Computing. The X gate acts on a single qubit and switches from state $|0\rangle$ to $|1\rangle$ and vice versa.

The X matrix: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$

Applying an X gate to a qubit that is in state $|0\rangle$ will then have the following outcome:

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$



Figure 3-9 X Gate Behaviour.

On the right side of Figure 3-9, we can see the reversibility of gates since by applying two X gates one after the other we return to the $|0\rangle$ gate.

3.3.4 CX “CNOT” Gate

The controlled-NOT gate is a binary qubit gate since it needs two qubits to operate on. One target qubit and one control qubit. Whenever the control qubit is in the $|1\rangle$ state and X operation is acted on the target qubit. Later we will see how this gate helps us achieve entanglement.

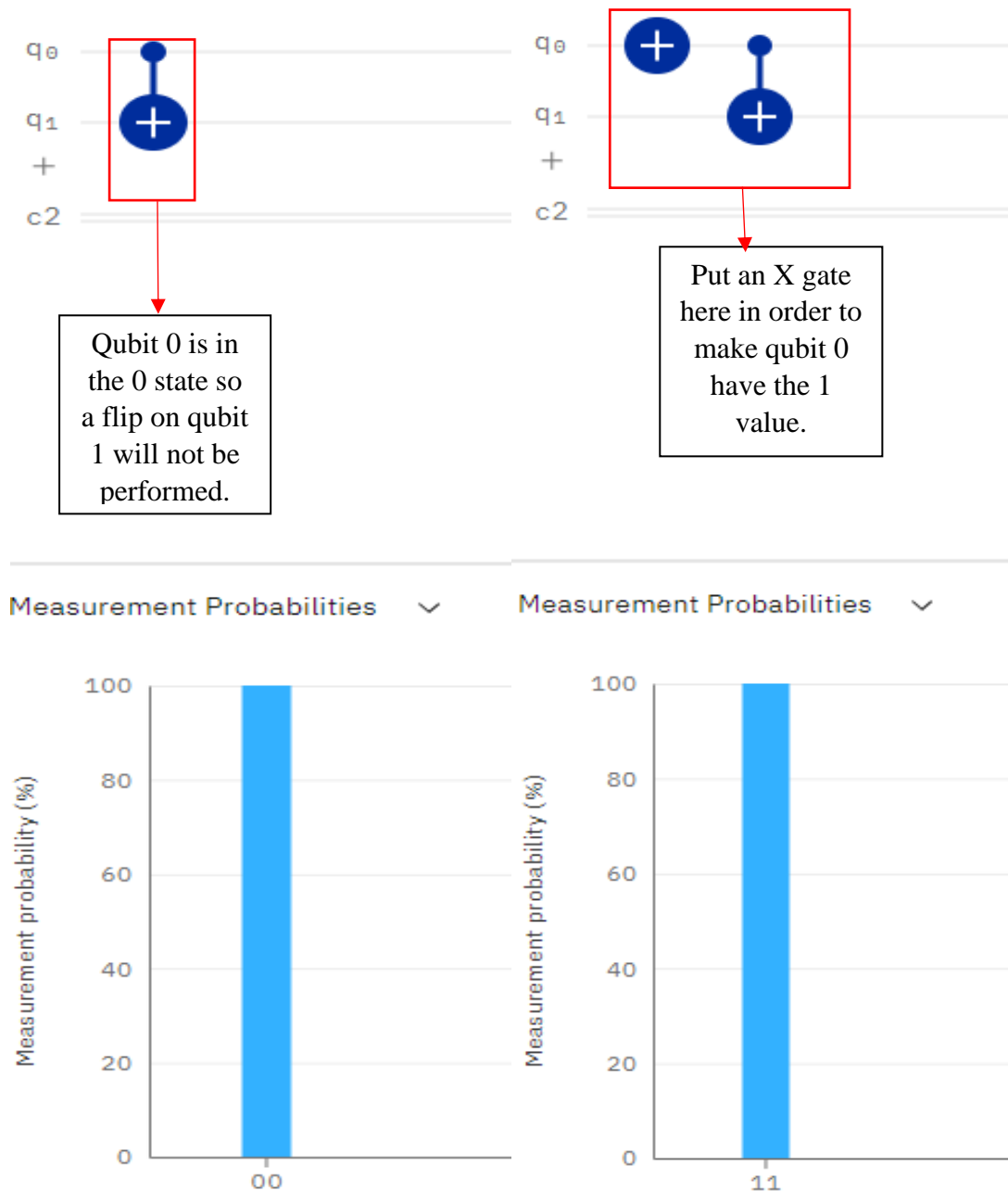


Figure 3-10 CX or C-NOT Gate Behaviour.

3.3.5 Hadamard Gate

The Hadamard gate is a very special operator. It is useful when we want to achieve superpositions entanglement or interference. It will be the cornerstone of all major experiments in this thesis.

The H Matrix: $\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

Applying an H gate to a qubit that is in state $|0\rangle$ will then have the following outcome:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = |+\rangle$$

$$H|1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = |-\rangle$$

Both $|+\rangle$ and $|-\rangle$ are states which are essentially in a superposition of exactly equal probability of yielding 0 or 1.

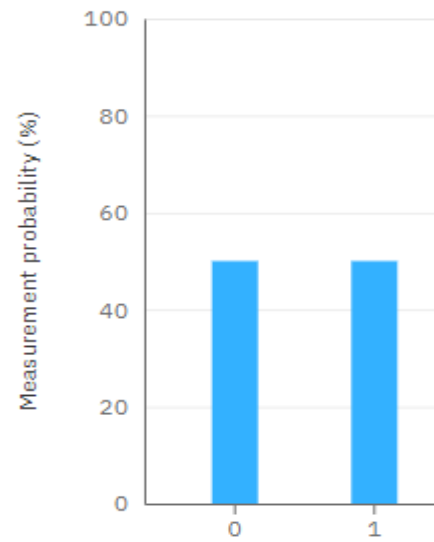
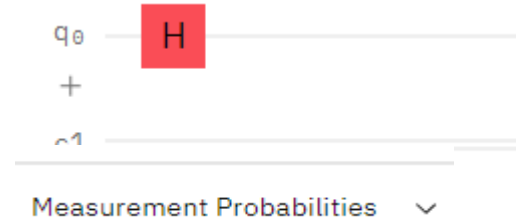
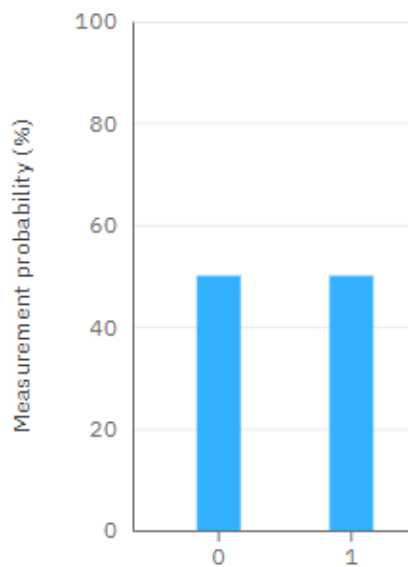
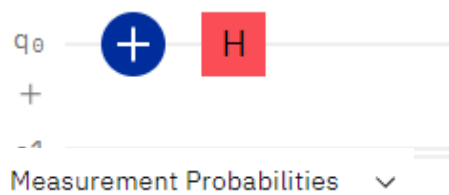


Figure 3-11 Hadamard Gate Behaviours.

Chapter 4

Quantum Computing Learning Gate

4.1	Introduction	27
4.2	Structure Breakdown	28
4.2.1	Level 1 – Experimenting with Primers	29
4.2.2	Level 2 – The Exciting Game	29
4.2.3	Level 3 – Implementation of Algorithms.....	29
4.2.3.1	quantum_algorithms.....	30
4.2.3.2	oracles.....	30
4.2.3.3	classical	30
4.2.4	Level 4 – Evaluation of Algorithms.....	30
4.2.5	Independent entities	31
4.2.5.1	constants	31
4.2.5.2	manager	31
4.2.5.3	credentials.....	31
4.2.5.4	tools	31
4.3	GitHub Repository	33

4.1 Introduction

Once an individual is familiarized with the quantum primers, it is time to move forward. We have developed a repository in GitHub in our efforts to create a standalone platform capable of hosting experiments of different levels. All necessary instructions to deploy this project can be found on Appendix F.

4.2 Structure Breakdown

After many different designs, we have finalized the structure for this platform with simplicity in mind. The idea of QCLG is to allow individuals of different competence levels to experiment with Quantum Computing. It is also possible to contribute to different levels according to the structure rules. We have implemented four different levels of experimentation, each with a different goal. All levels take advantage of Qiskit resources but to different degrees.

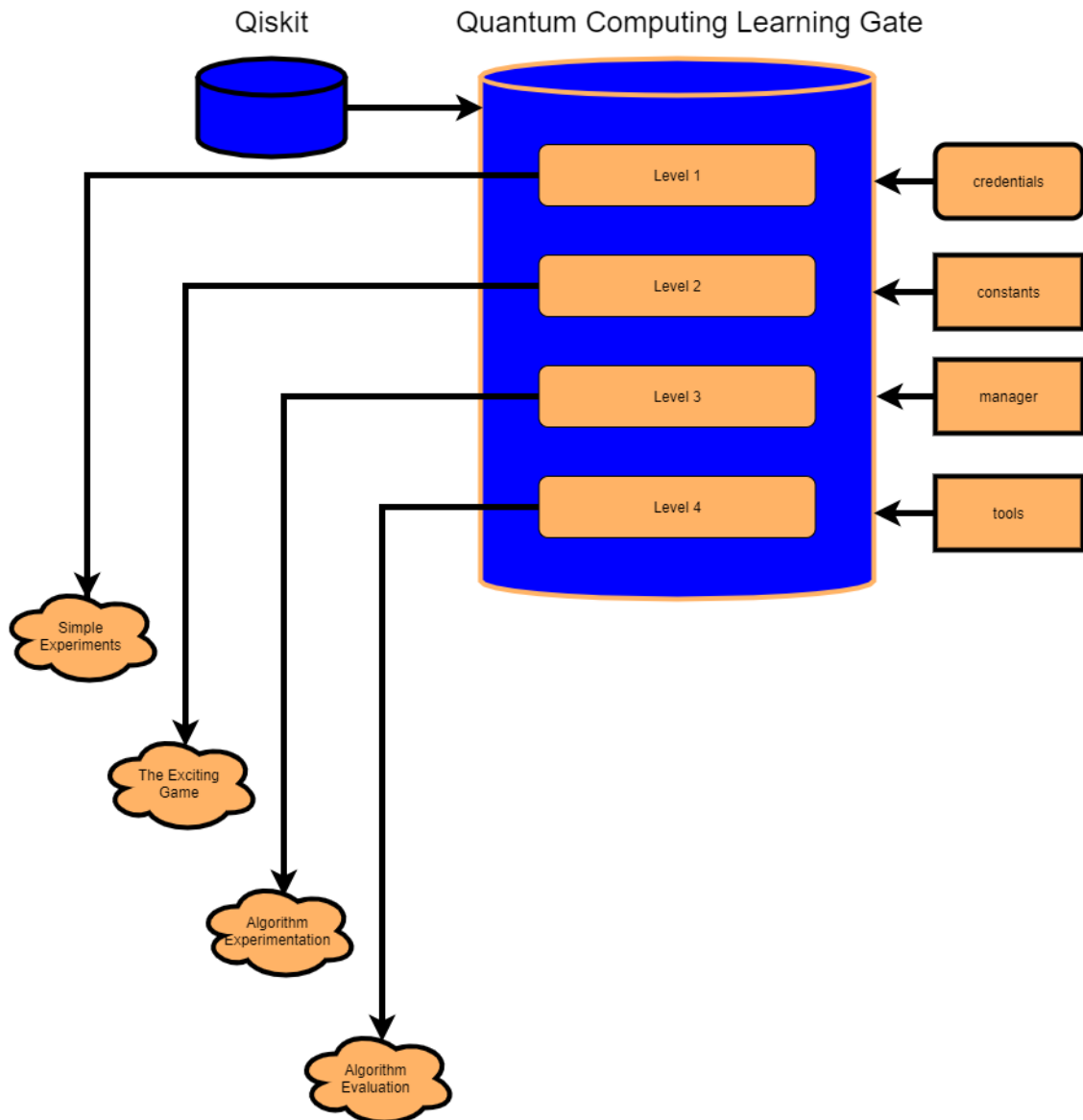


Figure 4-1 The QCLG Platform.

The first level aims to cement the basic knowledge required in order to proceed with more advanced concepts. Based on our learning experience, we believe that an

intermediate step was required before tackling quantum algorithms. Although very interesting, quantum algorithms can prove to be quite uninviting to beginners. We decided to construct a level where the user can fall back on and sharpen their understanding by playing quantum games, hence the creation of Level 2. This way they can try again experimenting on the next level with more confidence. Level 3 analyses the more mainstream quantum algorithms by attacking the problem from both sides of the spectrum. Here we provide implementations for both classical and quantum approaches to solving certain problems. The final level is for evaluating algorithms. Once the user possesses a deeper understanding of an algorithm, they can stress test it and extract useful results performance and accuracy-wise.

4.2.1 Level 1 – Experimenting with Primers

Level 1 is where experimentations concerning the basic concepts of Quantum Computing take place. It is the starting place after getting familiarized with the Quantum Primers we discussed in Chapter 3. Here, the user can explore and add experimentations of very basic quantum circuits in order to test their understanding before moving on to the next levels. The currently available experiments in Level 1 are about Superposition, Entanglement, and Phase Kickback and we showcase them in more detail in Chapter 5.

4.2.2 Level 2 – The Exciting Game

Level 2 is an intermediate layer that aims to motivate the user to learn about Quantum Computing through more interactive mediums. Once the user has become more proficient with the basic quantum concepts, they can test their knowledge with a quantum game for example. The currently available game is “The Exciting Game” which we cover extensively in Chapter 6.

4.2.3 Level 3 – Implementation of Algorithms

Level 3 has two objectives. The first objective is to implement quantum algorithms and the second objective is to implement the classical solution to that algorithm in order to allow the user to achieve a deeper understanding of that algorithm. Level 3 has many choices regarding the execution of experiments. They can choose to solve the problem classically by executing the solution onto their local device and observe

the results, execute the quantum solution of the problem in the IBM backends or solve the problem with both approaches and observe the differences. Below we list the three directories included in Level 3.

4.2.3.1 quantum_algorithms

In this directory, we store the implementations of major quantum Algorithms, as well as a controller class that allows individual calling of each of these algorithms. Up to the current point of development of this thesis, we have implementations for the Deutsch-Jozsa, which can be found in 7.3.3, and Bernstein-Vazirani, which can be found in 7.6.3, offering execution for each algorithm on a simulator, on a real quantum system, on the local machine, executed classically, on both a real quantum system and the local machine for comparison.

4.2.3.2 oracles

In the oracles directory, we store all the oracle functions necessary for implementing the quantum algorithms. Currently, there are implementations for one possible oracle function of Deutsch-Jozsa which can be found in 7.3.3, and the implementation of the oracle function for Bernstein-Vazirani which can be found in 7.6.3.

4.2.3.3 classical

The classical directory contains all required classes for implementing the classical solutions of the Quantum Algorithms. This includes the logic for the classical algorithms, as well as some additional implementations for input generation. These can be found in 7.2.2 and 7.5.1.

4.2.4 Level 4 – Evaluation of Algorithms

Once the user feels comfortable with all the previous levels, they can proceed to Level 4, which is the automated evaluation of an Algorithm based on its classical and quantum solution. Here, besides an adequate understanding of Quantum Computation concepts, the user must also practice with the Qiskit API in order to extract the necessary

data to extract useful results. More information on the evaluation can be found in Chapter 8.

4.2.5 Independent entities

In Figure 4-1 we can see there are four separate entities outside of the QCLG Platform. credentials, constants, manager, and tools.

4.2.5.1 constants

The constants entity is a python file containing a variety of different text messages as well as stored acceptable inputs for many different parts of the project. Its main purpose is to increase readability on the rest of the project.

4.2.5.2 manager

The manager entity is a python file that acts as the main driver of the whole project. From this point, we can branch to every different level available in this project.

4.2.5.3 credentials

This is where the user can store their IBM Q credentials to access the remote resources of IBM. In order to access the IBM resources an account in IBM Q is required. After creating an account, the user can generate a token that allows them to access IBM resources remotely.

4.2.5.4 tools

The tools entity is a python file and is the backbone of the whole project. Here we implement an amalgamation of methods, consisting of remote calls to IBM resources, calls to different parts of the project with customizations, evaluation methods, and more.

The project Tree

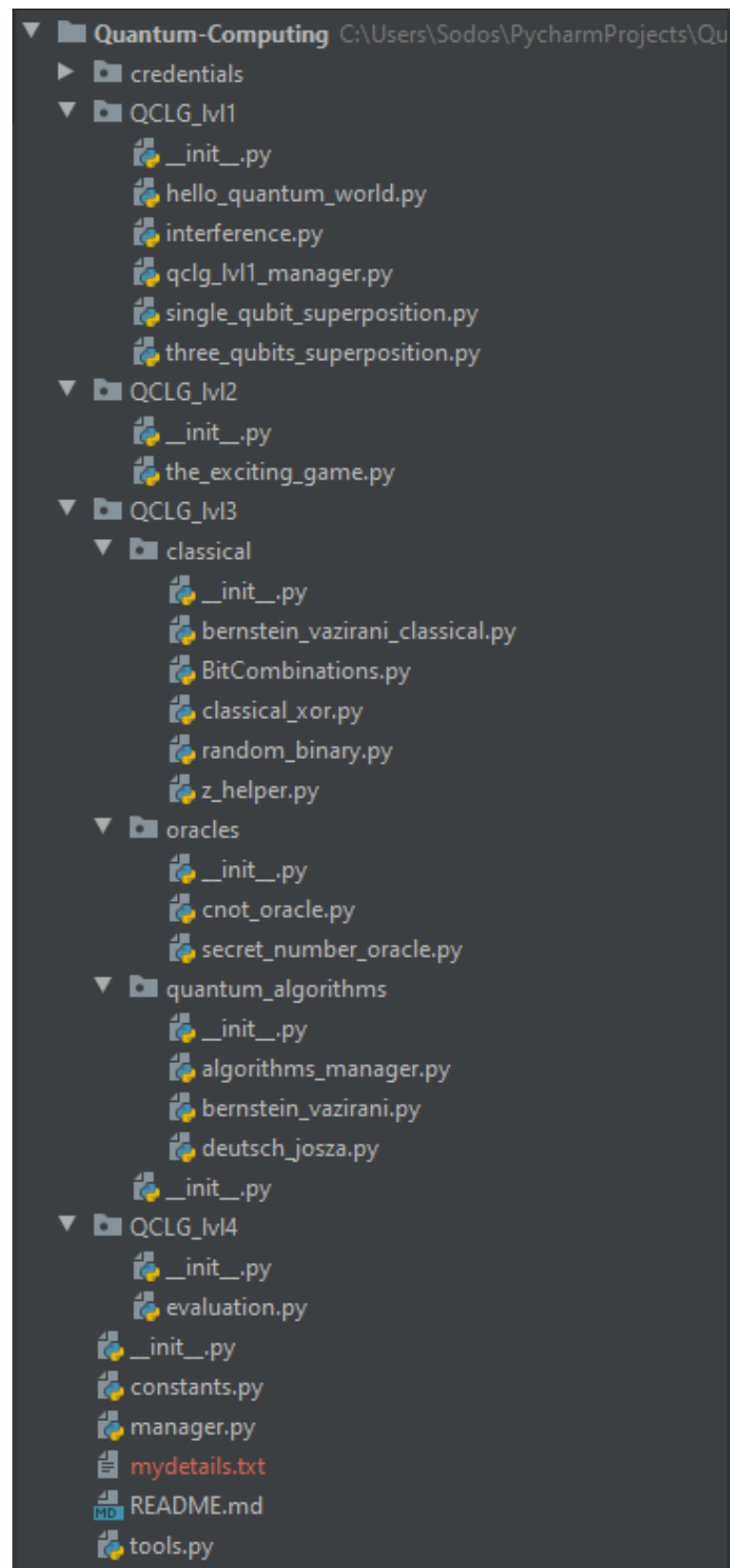


Figure 4-2 The Project Tree.

4.3 GitHub Repository

This project is hosted by the Data Management Systems Laboratory (DMSL) - Department of Computer Science, University of Cyprus, more information about DMSL can be found here [20].

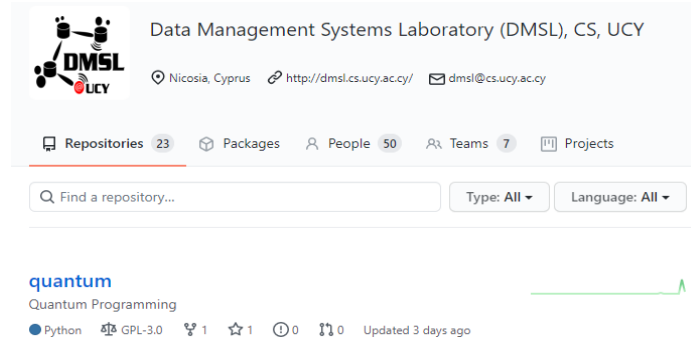


Figure 4-3 DMSL GitHub Page

The Quantum Computing Learning Gate is an ongoing project, and its current state of development, as well as previous or future states of the project, can be found here [19], under the “dmsl/quantum” label. This project is under the “GNU General Public License v3.0”.

We have also supplemented Quantum Computing Learning Gate with a separate README file, containing the essential information about the project, and the deployment guide, shown in Appendix F.

Quantum Computing Learning Gate
Table of contents
<ul style="list-style-type: none">• General info• Technologies• Setup• Structure• Useful Resources

Figure 4-4 QCLG Table of Contents.

We have decided to provide some additional useful resources that helped us during our research, since it can be quite difficult to find a good starting point.

Chapter 5

QCLG Level 1 – Implementing Primers

5.1	Introduction	34
5.2	Achieving Superposition	35
5.2.1	Implementation	36
5.3	The Bell State	37
5.3.1	Implementation	38
5.3.2	Results	38
5.4	Phase Kickback	39
5.4.1	Implementation	41
5.4.1.1	Results	41

5.1 Introduction

At this point, we start implementing a series of simpler experiments to practice on both the primers we discussed in Chapter 3 and methods imported from the Qiskit SDK as we presented briefly in 2.2.9. Some pointers to help out are to always keep a window open with the Qiskit documentation which can be found here [32] and the most important to press the control key + left click to be transferred to the class implementing each method in order to clear any doubt on what the current method achieves. We have supplemented our code with comments where it was deemed necessary, but the user should spend as much time as possible understanding both the concepts presented and the way they were implemented before moving on to the next level of QCLG.

5.2 Achieving Superposition

In this experiment, we will set up a circuit such that all qubits are transcended into a superposition using the Hadamard gate which will configure each qubit into a state with an equal probability of yielding zero or one and then measure the state of the system. We will execute this experiment 1024 times and observe the measurements. For this demonstration, we will use four qubits. The possible states with three are equal to $2^3 = 8$. Since we use the Hadamard gate to set up our system we expect that each time we make a measurement we have an equal probability of measuring one of the 8 possible states:

$|000\rangle, |001\rangle, |010\rangle, |100\rangle, |011\rangle, |101\rangle, |110\rangle, |111\rangle$.

In an errorless quantum computing system, we expect to **approximately** measure each of the above states $\frac{1024}{8} = 128$ times. Of course, even in an errorless system, this should not be the case since what a Hadamard gate guarantees, is an equal probability of measuring either one or zero. Hence, we just expect to see all states measured to a number very close to 128. We can simulate the behaviour of an errorless system using the simulator provided by IBM Q. After assembling a three-qubit circuit with a Hadamard gate attached to each qubit and immediately measuring each qubit we get the two following results. The first result is achieved by executing the circuit on the simulator and the second by executing on a real device.

Simulator execution: Total count all possible states are: {'000': 134, '001': 129, '010': 102, '011': 127, '100': 119, '101': 129, '110': 132, '111': 152}

Real System execution: Total count for all possible states are: {'000': 185, '001': 167, '010': 114, '011': 92, '100': 152, '101': 111, '110': 87, '111': 92}

We can see that indeed we measure roughly as we expected. What we can also see is that while a superposition seems very powerful, it doesn't accomplish much if we are eager to measure qubits. Measurements in quantum computing are irreversible on the state of a system and we should not be measuring whenever.

5.2.1 Implementation

We implement the quantum circuit with the help of Qiskit. Qiskit offers a plethora of different classes, each for a different purpose. Some things to notice in the following code is that we import different classes from Qiskit

```
from qiskit import execute, Aer, QuantumCircuit, IBMQ

class ThreeQubitSuperposition:
    @classmethod
    def run(cls):
        # Use Aer's qasm_simulator
        simulator = Aer.get_backend('qasm_simulator')
        # Create a Quantum Circuit acting on the q register
        circuit = QuantumCircuit(3, 3)
        # Add a H gate on qubit 0
        circuit.h(0)
        circuit.h(1)
        circuit.h(2)
        # Map the quantum measurement to the classical bits
        for i in range(3):
            circuit.measure(i, i)
        # Execute the circuit on the qasm simulator
        job = execute(circuit, simulator, shots=1024)
        # Grab results from the job
        result = job.result()
        # Returns counts
        counts = result.get_counts(circuit)
        print("\nTotal count all possible states are:", counts)
        provider = IBMQ.load_account()
        backend = provider.backends.ibmq_valencia
        # Execute the circuit on a real device
        job = execute(circuit, backend=backend, shots=1024)
        # Grab results from the job
        result = job.result()
        # Returns counts
        counts = result.get_counts(circuit)
        print("\nTotal count for all possible states are:", counts)
```

Figure 5-1 Code for Achieving the Superposition of Three Qubits.

The major processes accomplished in this code is the creation of a circuit with 3 qubits and 3 bits with the call to the QuantumCircuit class. Then we execute the circuit both on the simulator and the ibmq_valencia backend using the necessary Qiskit methods. We then use the `result()` method to acquire and display the results.

5.3 The Bell State

Renowned as the “Hello World” of Quantum Computation, the Bell State demonstrates two of the major concepts of Quantum Computing with the use of just two gates. By executing the Bell State circuit, we can observe both the Superposition and Entanglement of two qubits.

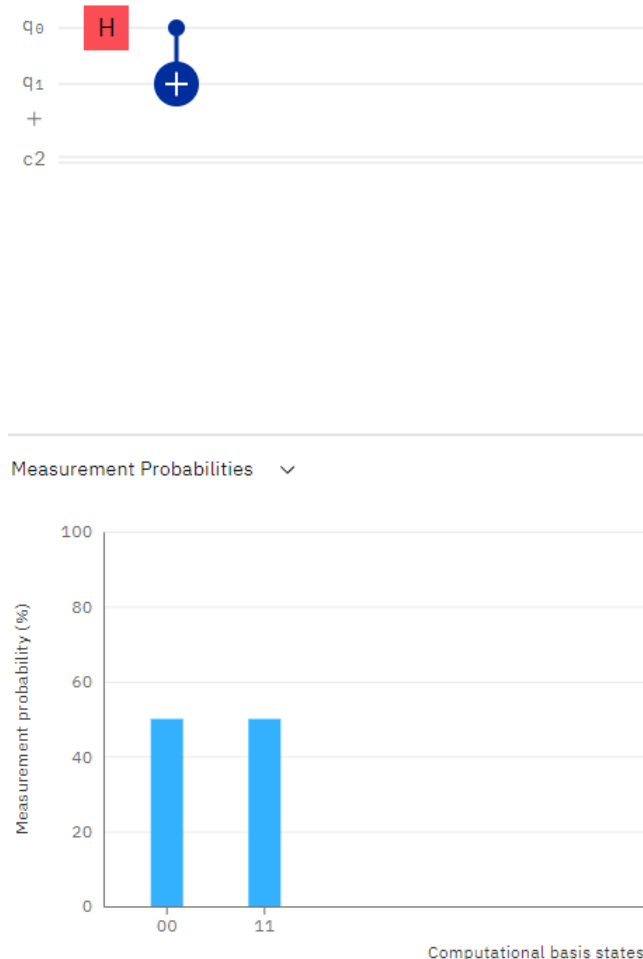


Figure 5-2 Bell State Experiment.

We can see from the measurement probabilities that we have two possible system states. Either both qubits will be zero or both qubits will be one. By applying a Hadamard gate on qubit 0 we transcend it to a state of equally zero and one and by applying a CX gate with the controller qubit being zero, whenever qubit zero is zero, qubit one will be zero and whenever qubit zero is one, qubit one will be one. Hence the two qubits are now entangled, which means that the state of qubit zero is now the major contributing factor of what the state of qubit one will be.

5.3.1 Implementation

```
from qiskit import IBMQ
from qiskit import (
    QuantumCircuit,
    execute,
    Aer)

class HelloWorld:
    @classmethod
    def run(cls):
        with open('./credentials/token', 'r') as file:
            token = file.read()
            IBMQ.save_account(token, overwrite=True)
            # Use Aer's qasm simulator
            simulator = Aer.get_backend('qasm_simulator')
            # Create a Quantum Circuit acting on the q register
            circuit = QuantumCircuit(2, 2)
            # Add a H gate on qubit 0
            circuit.h(0)
            # Add a CX (CNOT) gate on control qubit 0 and target qubit 1
            circuit.cx(0, 1)
            # Map the quantum measurement to the classical bits
            circuit.measure([0, 1], [0, 1])
            # Execute the circuit on the qasm simulator
            job = execute(circuit, simulator, shots=1000)
            # Grab results from the job
            result = job.result()
            # Returns counts
            counts = result.get_counts(circuit)
            print("\nTotal count for 00 and 11 are:", counts)
            # Draw the circuit
            print(circuit)
```

Figure 5-3 Code for Method run() of Experiment Bell State.

5.3.2 Results

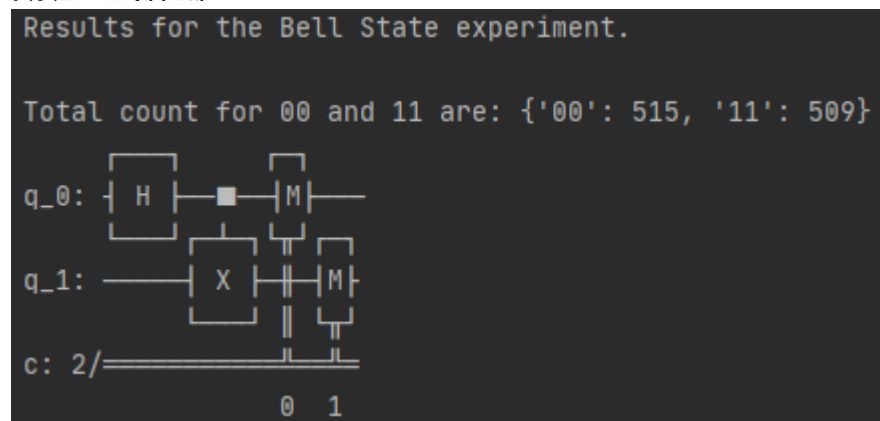


Figure 5-4 Results for Experiment Bell State.

We can indeed see that the only two possible outcomes are the states $|00\rangle$ and $|11\rangle$ with almost equal occurrences.

5.4 Phase Kickback

Phase Kickback is a phenomenon that occurs in Quantum Computation, and the reason why quantum probabilities are different from classical probabilities. If we were to imagine that a Hadamard gate's effect is similar to that of a coin flip, we would expect that two consecutive Hadamard gate operations on the same qubit would be analogous to a coin being flipped twice. Due to phase kickback, that is not the case.

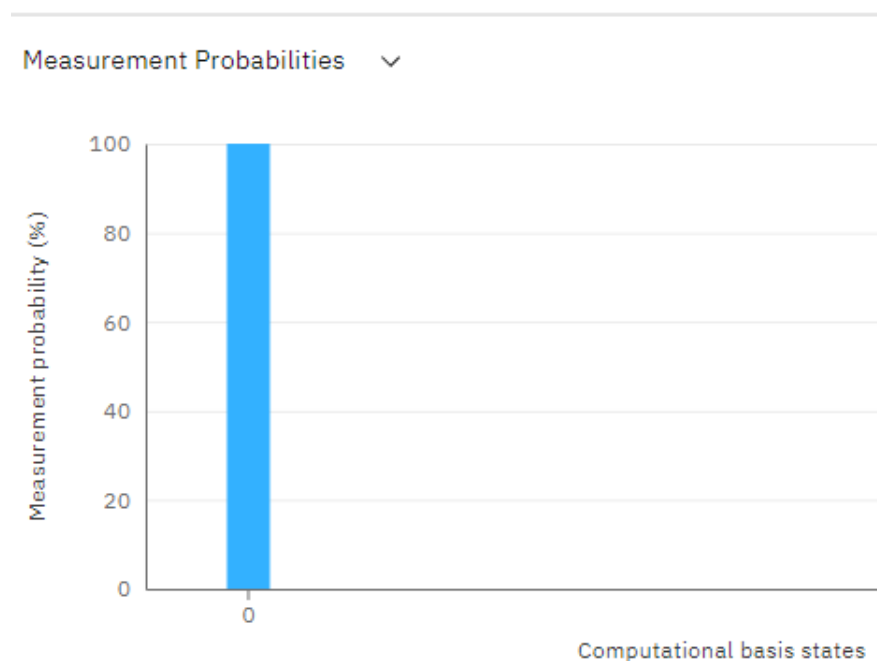
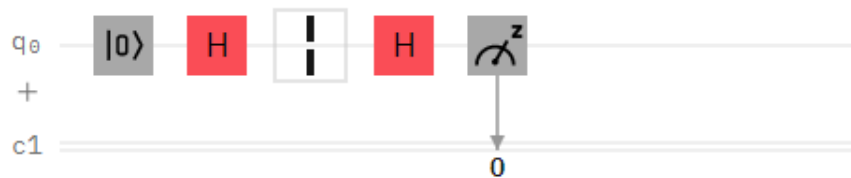


Figure 5-5 Phase Kickback effect.

Why does this happen? How can we have a definite outcome by applying randomness?

Assume a qubit that is instantiated in the $|0\rangle$ state.

Applying a Hadamard gate to this qubit will result in the following complex state:

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = |+\rangle$$

The $|+\rangle$ state, when classically measured in the Z-basis, yields $|0\rangle$ or $|1\rangle$ with equal probability. But we can easily see that when we apply two Hadamard gates to the same qubit we have the following outcome:

$$H|+\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 \cdot 1 + 1 \cdot 1 \\ 1 + (-1 \cdot 1) \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \frac{1}{2} \cdot 2 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle$$

We can see that after two consecutive applications of the Hadamard gate, the amplitudes of the mixed states cancel out and we are left with the definite state of $|0\rangle$.

We can see that if we choose certain linear combinations, some possible states may be destructed, or constructed.

Phase Kickback plays a vital role in Quantum Algorithm Development because it helps to make a transition from a seemingly random system to a useful result. We need to be careful in order to destruct the states that we are not interested in and construct the states that interest us.

5.4.1 Implementation

```
from qiskit import execute, Aer, QuantumCircuit

class Interference:
    @classmethod
    def run(cls):
        # Use Aer's qasm_simulator
        simulator = Aer.get_backend('qasm_simulator')
        # Create a Quantum Circuit acting on the q register
        circuit = QuantumCircuit(1, 1)
        # Add a H gate on qubit 0
        circuit.h(0)
        # Add another H gate to qubit 0
        circuit.h(0)
        # Map the quantum measurement to the classical bits
        circuit.measure(0, 0)
        # Execute the circuit on the qasm simulator
        job = execute(circuit, simulator, shots=1000)
        # Grab results from the job
        result = job.result()
        # Returns counts
        counts = result.get_counts(circuit)
        print("\nTotal count for 0 and 1 are:", counts)
        print(circuit)
```

Figure 5-6 Code for the run() Method of the Phase Kickback Experiment.

5.4.1.1 Results

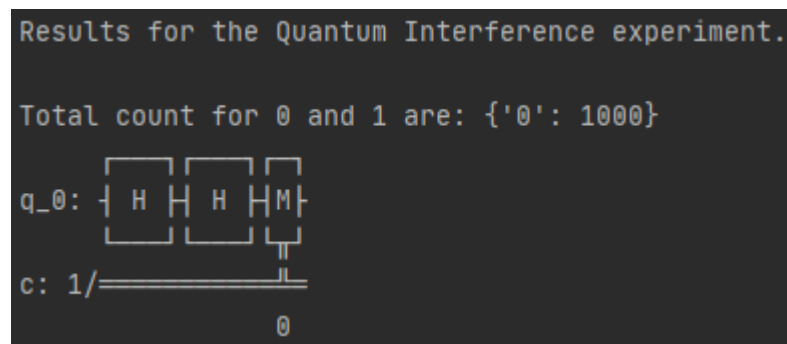


Figure 5-7 Results for Phase Kickback.

Indeed, we see that the only state measured is the $|0\rangle$ state.

This concludes the chapter. It is worth noting that for all these simple experiments we used the simulator in order to observe the behavior of an errorless system without further complications.

Chapter 6

QCLG Level 2 – The Exciting Game

6.1	Concept	42
6.2	Rules.....	43
6.3	Available Cards.....	43
6.4	Winning Hands	44
6.5	Strategy	49
6.6	Implementation	50
6.6.1	Method Definitions	50

6.1 Concept

We have explored a plethora of different concepts in quantum computing and familiarized ourselves with some of the bizarre and unconventional attributes of quantum computation. As a comprehension exercise, we have implemented a quantum minigame, which showcases some of the basic quantum concepts we have learnt.

The game is played like a two-player card game.

In this case, the cards are quantum gates and our playing field is two qubits, on which our cards operate.

The goal of the game is to apply the quantum gates on your possession in the correct order to manage to transcend your qubit from the ground state of $|0\rangle$ to the excited state of $|1\rangle$ before a certain number of turns.

After that, a round is completed, and points are awarded to both players accordingly. Once the number of rounds finished, the player that was “excited” for the most rounds, wins.

Hence the name of the game.

6.2 Rules

The rules are very simple.

1. Finite limit of rounds and the player with the most winning rounds is the winner.
2. For each round, there is a countdown where the player is given the choice to draw the card from the deck and place it on their hand or they can choose to ignore the draw and stay with their current hand.
3. Once the countdown is finished, is player is required to place their cards in the order they want them to be applied on their qubit.
4. The playing field is evaluated and the player who possesses an excited qubit is awarded a point.
5. The hands and playing field are reset for each round.

6.3 Available Cards

In the current version of the Exciting Game, there are four different available cards.

- **THE HADAMARD GATE.** At this point, we are familiar with the Hadamard Gate. The player can use it to insert a randomness effect if they wish to test their luck or come up with tricks using quantum interference.
- **THE X GATE.** The X gate is a very powerful card in the Exciting Game because it can effectively win a round by flipping a qubit from $|0\rangle$ to $|1\rangle$.
- **THE CX GATE.** The CX gate provides a sabotage element to the game. It can flip the qubit of the enemy player.
- **THE RX GATE.** The RX gate has a similar effect to the Hadamard gate with a few key differences. It only acts on the X-axis and applying it twice in a row will result in a qubit state flip instead of negating the effect. For the purposes of the game, it can be thought of as a “half” X gate. It is up to the players to discover all of its usages.

6.4 Winning Hands

Below we provide some examples of winning hands in order to get a better grasp of the game.

Example 1

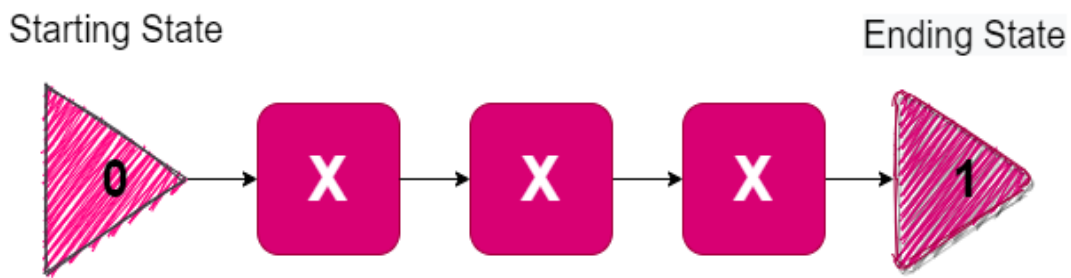


Figure 6-1 The Exciting Game Example 1.

In Figure 6-1 we can easily see how we get to the excited state. Applying the X gate for the first time flips the qubit from the $|0\rangle$ state to the $|1\rangle$ state. Applying the second X gate will lead to another state flip which will lead to the $|0\rangle$ once again. With the third application of the X gate, we end up with the $|1\rangle$ state which is a winning hand.

Example 2

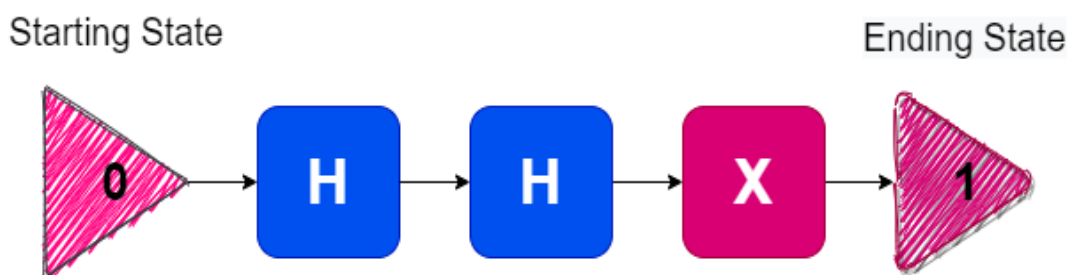


Figure 6-2 The Exciting Game Example 2.

Figure 6-2 showcases a scenario where we can take advantage of interference in order to achieve a winning hand. By putting the two Hadamard gates one after the other we negate their effect and thus keeping intact the initial state of $|0\rangle$. With the application of a single X gate, we once again reach a winning position.

Example 3

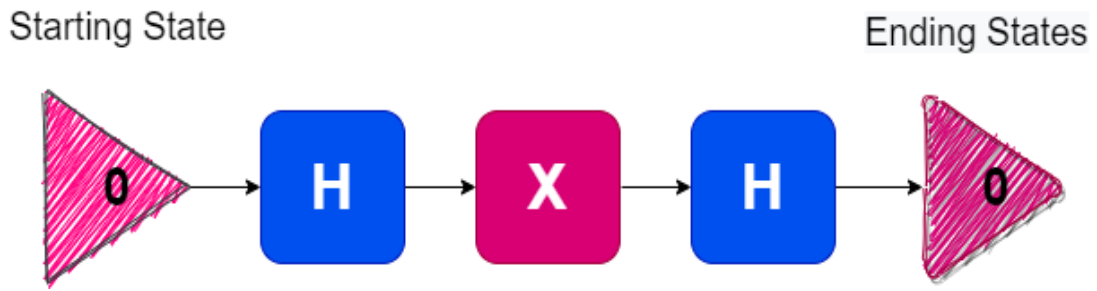


Figure 6-3 The Exciting Game Example 3.

In Figure 6-3, we have the same gates as in example 2. However, notice how the final state is $|0\rangle$ this time. By applying the X gate in the middle of the two Hadamard gates, due to the phase kickback effect, the X gate effect on the qubit is destroyed and we are left with the $|0\rangle$ state.

Example 4

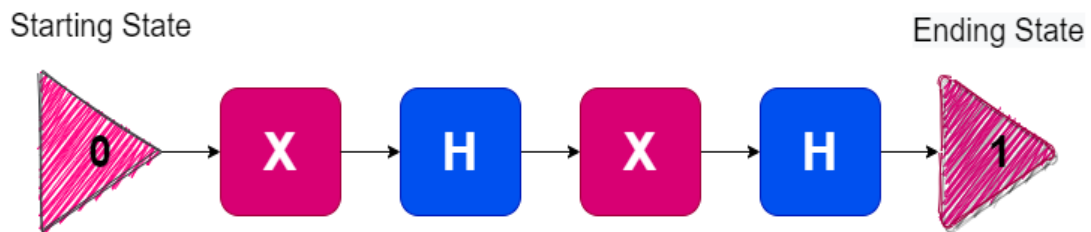


Figure 6-4 The Exciting Game Example 4.

Building on the same logic from Figure 6-3, we can easily see why we reach the $|1\rangle$ state in this example. With the first application of the X gate, we reach the $|1\rangle$ state. Then, by surrounding the second X gate with two Hadamard gates we add nothing new to the $|1\rangle$ state, hence we measure the exciting state.

Example 5

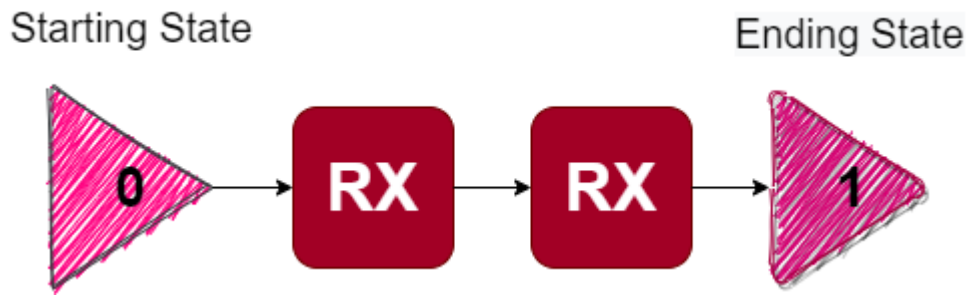


Figure 6-5 The Exciting Game Example 5.

In Figure 6-5, we showcase the simplest winning position possible with two RX gates. With the first application of the RX gate, the qubit is rotated by $\pi/2$ in the X-axis and has an equal probability of measuring $|0\rangle$ or $|1\rangle$. However, with the second application, the qubit is rotated another $\pi/2$ in the X-axis hence formulating the absolute $|1\rangle$ state.

Example 6

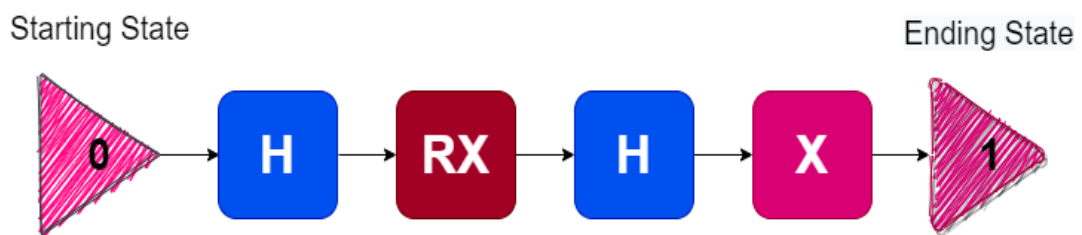


Figure 6-6 The Exciting Game Example 6.

The RX gate is also subjectable to the phase kickback effect. In Figure 6-6, the first three gates effectively do nothing on the state of the qubits, and then by simply appending an X gate, we achieve the exciting state.

Example 7



Figure 6-7 The Exciting Game Example 7.

Following the logic from Figure 6-4, we can see that with the first two applications of the RX Gate the qubit will be in the $|1\rangle$ state. With the application of the third and fourth RX Gate, the qubit will once again reach the $|0\rangle$ state. Hence, by finally appending the X gate we reach the desired state.

Example 8

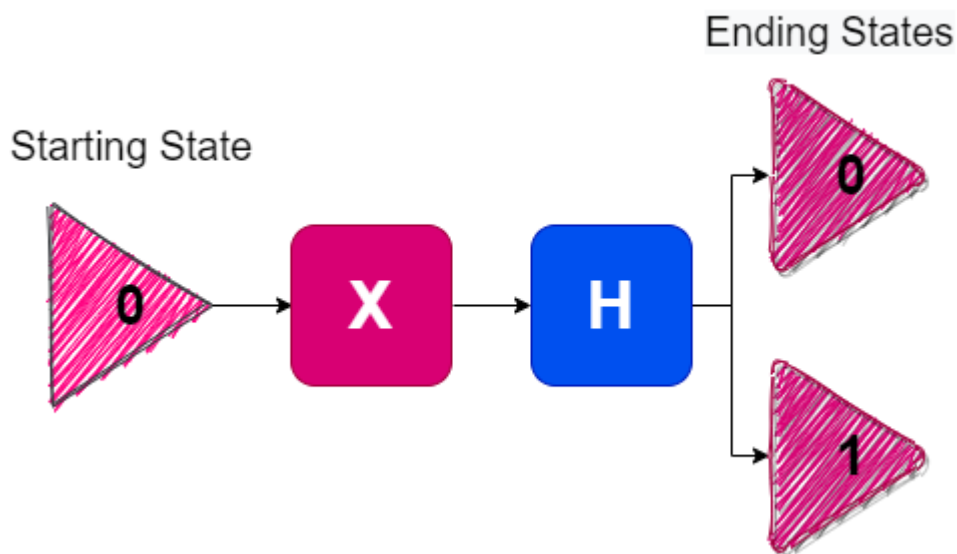


Figure 6-8 The Exciting Game Example 8.

By this point, it should be no surprise that in each round it is not always possible to achieve the excited state. This is an example of a hand where despite having an X gate, the Hadamard gate causes the qubit to behave with uncertainty. There is an element of luck in this case.

Example 9

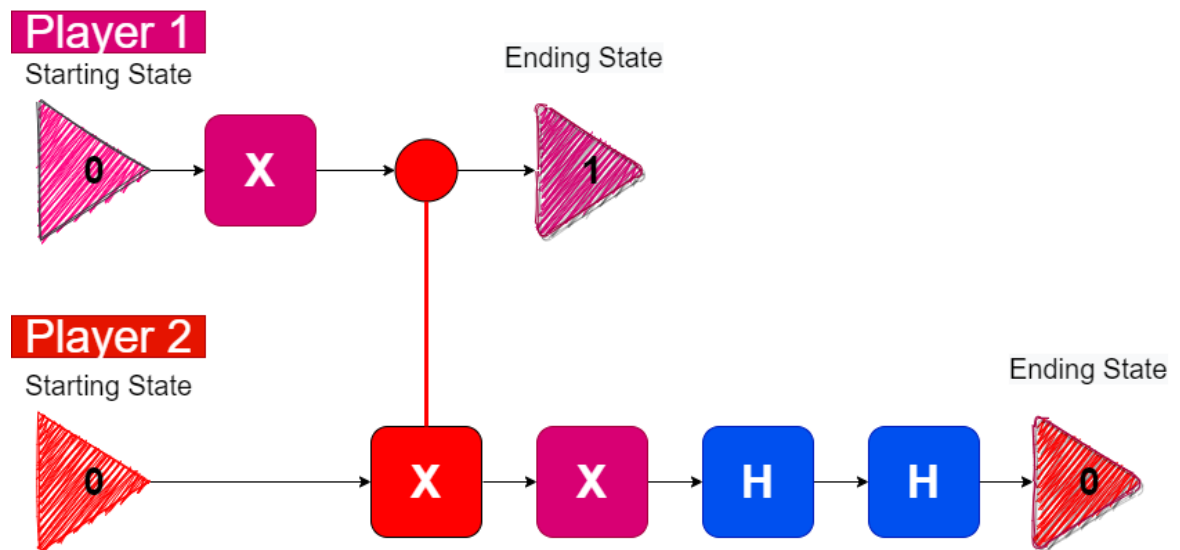


Figure 6-9 The Exciting Game Example 9.

The CX Gate is what makes the game interesting. We can see that in this scenario, player 1 applies an X gate hence putting their qubit to the $|1\rangle$ gate. As we know the CX gate is a Controlled Not gate so when player one applies a CX gate they will manage to flip the starting state of the player-two qubit. Now player 2's evaluation starts with the $|1\rangle$ state as the initial state. Hence, when player 2 applies their X gate they are now to the $|0\rangle$ state. Then with two Hadamard gates, nothing is achieved and player 2 ends up losing the round. We can also consider the scenario where player two had a CX gate in their possession. The best play for player 2 would be to insert their CX gate as their first card to flip the state of player 1 so that neither of them wins the round.

There are a wide range of different combinations and outcomes that each round can follow with just four different gates. We have just shown a few. We have found a few good practices in terms of strategy.

6.5 Strategy

Through experimentation, or by simply playing the game, one can notice there are a few good practices when playing the Exciting Game.

Below we list a few strategies and a brief reasoning behind each strategy.

1. Drawing a card is not always a good idea.

In a surprising number of games, we have found that more often than not, you can construct a winning hand before all cards are dealt for the current round. You need to be careful in order to realize that you possess a hand capable of winning even if you have two or three cards.

2. Phase kickback is your friend.

Even when a hand seems unpromising, there is almost always a way to destroy our less useful cards by using phase kickback. Consider a hand where the cards are Hadamard, Hadamard, X, X. At first glance, this is not a good hand since we have an even number of X gates. However, if we take advantage of phase kickback, we can put the one X gate in-between the two Hadamard gates and destroy its effect leaving us with effectively one X gate which is enough to make our qubit excited. You should always look for a possible phase kickback combo.

3. When all else fails, sabotage your opponent.

This might seem obvious but even if you can't reach the exciting state, you might still be able to sabotage your opponent by applying a CX gate at a point when your qubit is for sure excited.

6.6 Implementation

This is the driver program for the game. All is done in a while-loop until the number of rounds finishes.

Below we list some of the most useful methods implemented.

6.6.1 Method Definitions

```
@classmethod
def run(cls, circuit: QuantumCircuit):
    # use local simulator
    backend = BasicAer.get_backend('qasm simulator')
    results = execute(circuit, backend=backend, shots=1024).result()
    answer = results.get_counts()
    max_value = 0
    max_key = ""
    for key, value in answer.items():
        if value > max_value:
            max_value = value
            max_key = key
    print(answer)
    if max_key == "00":
        print("Both players stay grounded :)")
        return 0
    elif max_key == "01":
        print("Player 1 is excited!")
        return 1
    elif max_key == "10":
        print("Player 2 is excited!")
        return 2
    elif max_key == "11":
        print("Both players are excited!")
        return 3
    return
```

Figure 6-10 Code for the run() Method of The Exciting Game.

Method Definition – run(circuit):

The purpose of this method is to declare the winner after the end of each round.

Input: The circuit containing the two players' qubits along with their placed gates.

Output: A number between 0 and 3 indicating the four different outcomes.

Data Structures Used: In order to find the outcome, we need to traverse through the dictionary of measurements and find the state measurement with the most counts.

```

@classmethod
def place_gate(cls, player, field, qubit):
    card = player.pop()
    print(f"now inserting card {card} from player {qubit + 1}")
    if card == "H":
        field.h(qubit)
    elif card == "X":
        field.x(qubit)
    elif card == "RX":
        field.rx(np.pi / 2, qubit)
    elif card == "CX":
        if qubit == 0:
            field.cx(qubit, qubit + 1)
        else:
            field.cx(qubit, qubit - 1)
    return

```

Figure 6-11 Code for place_gate() Method of The Exciting Game.

Method Definition – place_gate(player, field, qubit):

The purpose of this method is to place the player's gates onto the qubit in the correct order.

Input: The player's hand, the circuit containing the two players' qubits, the qubit of the current player.

Output: A modified field circuit.

Data Structures Used: In order to place the gates to the player's qubit we pop the items in the player list one by one and according to the card's name we append the corresponding quantum gate to the designated qubit of the field circuit, given by the qubit parameter. One thing to note is the way of assigning the CX gate. If are dealing with player one then that means that we want the CX gate to be placed with qubit 0 as the control qubit and qubit 1 as the target qubit and vice-versa.

```

@classmethod
def fix_hand(cls, player: list) -> list:
    new_hand = []
    print("Your current hand is setup like this:")
    print(player)
    i = 0
    while len(player) > 0:
        replacement_choice = input(f"Choose one of your cards to be on position {i} :")
        while replacement_choice not in player:
            replacement_choice = input(f"Choose one of your cards to be on position {i} :")
        new_hand.insert(len(new_hand), replacement_choice)
        player.remove(replacement_choice)
        print("Cards remaining in previous hands")
        print(player)
        i = i + 1

```

Figure 6-12 Code for the fix_hand() Method of The Exciting Game.

Method Definition –fix_hand(player):

The purpose of this method is to sort the player's gates according to their liking.

Input: The player's hand.

Output: The new, modified, hand of the player, as designated by the player's choices.

Data Structures Used: We display the current hand of the player. Then, in a while-loop we make the player choose the next card they want in their ordered final submission for the round. We do this by removing each of the valid choices of the player until their hand is exhausted and we have constructed a new list containing the previous cards of the player but in the order that they would like them to be executed on their qubit.

This concludes the implementation of the Exciting Game. The whole code for this project can be found in the appendix under the Exciting Game section.

Chapter 7

QCLG Level 3 – Implementation of Algorithms

7.1	Deutsch - Jozsa.....	54
7.2	Classical Approach.....	55
7.2.1	Classical Implementation.....	56
7.2.2	Worst-case Scenario for a Classical Solution	59
7.2.2.1	Results for Worst-case scenario	66
7.3	Quantum Approach	68
7.3.1	Algorithm's Structure	69
7.3.2	Proof of Concept with a worked example.....	70
7.3.2.1	Step 1 – Initialization	71
7.3.2.2	Step 2 - Oracle.....	71
7.3.2.3	Step 3 - Interference	74
7.3.2.4	Step 4 - Measurements	75
7.3.3	Quantum Implementation	76
7.4	Bernstein - Vazirani	80
7.5	Classical Approach.....	81
7.5.1	Classical Implementation.....	82
7.6	Quantum Approach	83
7.6.1	Algorithms' Structure	83
7.6.2	Proof of Concept with a worked example.....	84
7.6.2.1	Step 1 – Initialization	85
7.6.2.2	Step 2 - Oracle.....	85
7.6.2.3	Step 3 - Interference	86

7.6.2.4	Step 4 - Measurements	86
7.6.3	Quantum Implementation	87
7.7	Summary	91

7.1 Deutsch - Jozsa

One of the first Quantum algorithms with a substantial speedup over its classical counterpart [5]. This is an algorithm that exploits the ability of a quantum system to support a huge state space in order to calculate the outcome faster than a conventional processor. For the problem this algorithm tackles, let f be a function that accepts as input either one or zero and outputs either one or zero. This is what we will call a Boolean function.

$$f : \{0, 1\} \rightarrow \{0, 1\}$$

Deutsch – Jozsa’s task is to determine, given a function f , if it is given a Balanced Boolean function or a Constant Boolean function. Consider now the two Boolean types of functions: Balanced and Constant. A balanced function is a function where for all possible inputs, the outcome is zero for exactly half the possible inputs and one for the exact remaining half. A constant function is labelled as such when, regardless of input, its outputs are either only zero or only one.

$$\text{balanced: } f(0) = 0, f(1) = 1$$

$$\text{balanced: } f(0) = 1, f(1) = 0$$

$$\text{constant: } f(0) = 1, f(1) = 1,$$

$$\text{constant: } f(0) = 0, f(1) = 0,$$

Figure 7-1 Constant-Balanced Function Definition.

7.2 Classical Approach

Let's go through a scenario for a one-bit input for a classical environment.

We want to determine if a one-bit function f is balanced or constant given the fact that it can only be either balanced or constant and nothing else. We have two possible inputs. Zero and one. We will simply then query the function twice and after accumulating the results we will determine by aggregating the results if the function is balanced. In simpler words by receiving from the output a single zero and a single one or if the function is balanced by receiving two ones or two zeroes.

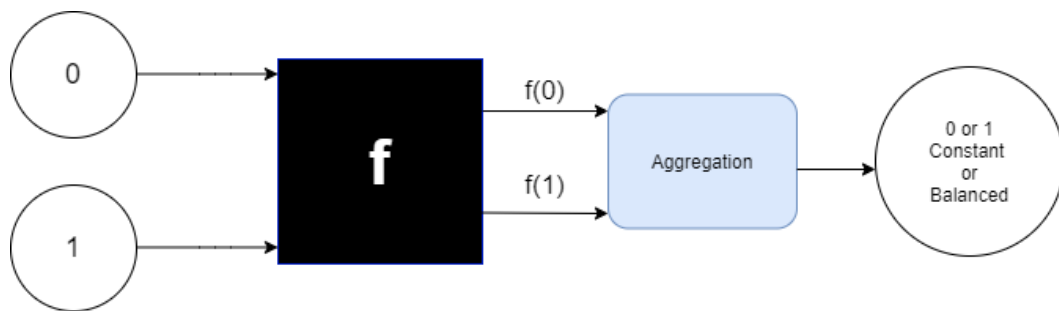


Figure 7-2 Deutsch-Jozsa Classical Approach Abstract Form.

Classical Deutsch-Jozsa Structure

Input: Sequences of bit-strings.

Processing: Based on the oracle function, map each input to an output bit

Output: 'Balanced' if outputted both zero and one **or** 'Constant' if outputted only zero or only one.

7.2.1 Classical Implementation

Below we provide the code for both the aggregation-oracle function and the driver function that calls the algorithm.

Class ClassicalXor:

This class contains three methods and imports DateTime for time measurement and the BitCombinations Class that contains helper methods for generating the worst-case scenario.

Method Definition - `_super_secret_black_box_function_f(list_of_inputs)`:

This function is responsible for executing the full classical version of Deutsch-Jozsa. We provide a list of binary strings and then check all of them. Once we find both a one and a zero, we can say that it is a balanced function and if not, it is a constant. We assume that we are given either a balanced or constant function.

Input: An arbitrary sequence of bit strings.

Output: 'Balanced' or 'Boolean'.

Data Structures Used: The `output_bit`, which is used as a medium for executing multiple xor operations for each input. The `output_zero` and `output_one` variables are used as Boolean flags which are responsible for keeping track of what kind of values the output bit gets after each element. The `counts` variable keeps track of the amount of checks. The method is implemented by taking a list and then with a loop iterate over each element of the list. In an encapsulated loop, we take each element and execute a xor operation to the output bit with all bits of the element. We then check the `output_bit` value and change the value of the flags accordingly. We then reset it to zero and proceed with the conditional statements before continuing with the next iteration. The first conditional statement checks if both flags are set which means we have managed to find both possible values of the output bit hence we terminate and return a 'Balanced' answer. The second conditional statement checks if we have reached $2^{n-1} + 1$ inputs checked. If that is the case that means we still haven't managed to set both flags, hence our function is Constant.

```

from datetime import datetime

from QCLG_lvl3.classical.BitCombinations import BitCombinations

class ClassicalXor:

    @classmethod
    def _super_secret_black_box_function_f(cls, list_of_inputs: list) -> str:
        output_bit = 0
        output_zero = False
        output_one = False
        counts = 0
        for input_bits in list_of_inputs:
            for bit in input_bits:
                output_bit = output_bit ^ int(bit)

            if output_bit == 1:
                output_one = True
            else:
                output_zero = True
            counts = counts + 1
            output_bit = 0
            if output_one and output_zero:
                return f'Balanced After {counts} checks.'
            if counts > (len(list_of_inputs) / 2):
                return f'Constant After {counts} checks.'
        return f'After {counts} checks.'

```

Aggregation of all inputs

Application of the 'oracle function to each input'

Figure 7-3 Code for Constructing the Classical Oracle for Deutsch-Jozsa.

Method Definition - execute_classical_xor(bits):

Input: Number of bits

Output: The nature of the function, 'Balanced' or 'Boolean', and the time it took for aggregating all inputs.

Data Structures Used: The bits parameter is an integer value that determines the length of the inputs that are going to be fed to the super_secret_black_box_function_f. We must first generate a list that contains all possible bit combinations. For this experiment, we have opted to always generate the worst-case scenario of inputs.

We then declare two timestamps using the `DateTime` class for measuring the time it took for the input list to be generated and another two timestamps that measure the time it took our black box function evaluation to be finished. We then return in a list that contains two strings, one with the output of the black box function evaluator method we discussed above and one with the execution times.

```
@classmethod
def execute_classical_xor(cls, bits) -> list:
    start = datetime.now()
    inputs = BitCombinations.produce_worse_scenario(BitCombinations.combinations(bits))
    end = datetime.now()
    elapsed = end - start
    time_to_generate_worst_input = elapsed.total_seconds()
    start = datetime.now()
    function_nature = cls._super_secret_black_box_function_f(inputs)
    end = datetime.now()
    elapsed = end - start
    time_str = elapsed.total_seconds()
    final = [time_to_generate_worst_input, time_str, bits, function_nature]
    return final
```

Figure 7-4 Code for Executing the Classical Solution of Deutsch-Jozsa.

By increasing the number of input bits to n we can see that the number of possible inputs increases exponentially and thus the number of queries to the function f . The reason we chose to pass as input the worst-case scenario every time is that several possible scenarios could occur and drastically change the performance of our classical solution. For example, if chose to randomly generate n -bit sequences, we could query the function f for the first time with our first input and get the answer one and then query it for the second time and get the answer zero. Given the fact we can only expect a balanced or constant function, we immediately know that by receiving both possible inputs, the function cannot be constant, hence it is balanced. These scenarios are mainly affected by the nature of the function -the way it reacts to each input- and by the way, the input is fed to the function. The latter can greatly increase or decrease the number of queries it takes to identify the function. Since the quantum systems that were available to us were not very powerful, we decided to always use the worst-case scenario in order to pursue a meaningful comparison.

Let's say lay down the absolute worst scenario for verifying that a function is balanced. After an arbitrary way of feeding inputs to our function f we manage to force our function to output all its ones at once and all its zeroes, if there are any after the ones have finished.

7.2.2 Worst-case Scenario for a Classical Solution

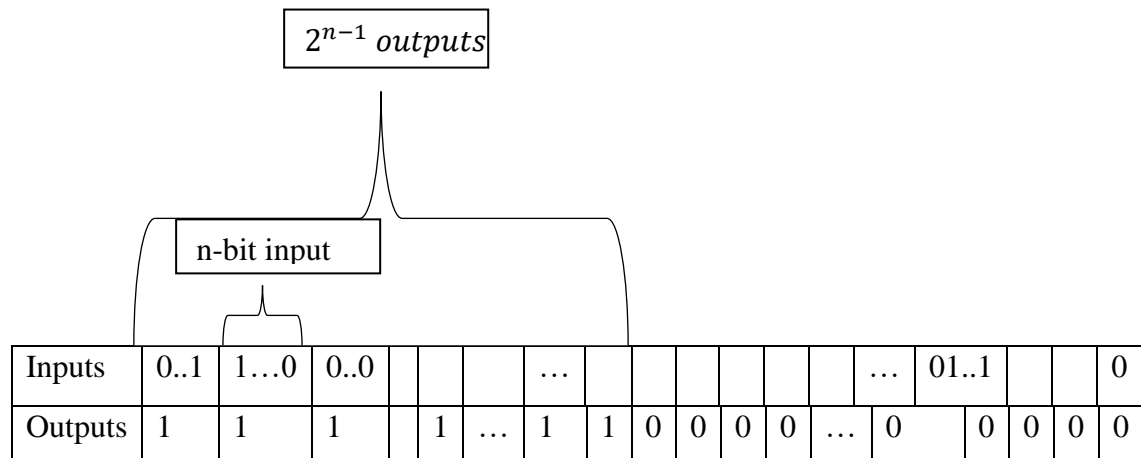


Figure 7-5 Worst-Case Scenario for Classical Deutsch-Jozsa.

It is easy to see that to verify the function is either constant or balanced we will have to query $2^{n-1} + 1$ times in the worst case in order to verify that the function is balanced or constant, in this case balanced.

Below we showcase three helper functions, the main function that generates all possible combinations and the function that groups the output of the main function in a way that simulates the worst-case scenario in the `BitCombinations` Class.

```
class BitCombinations:

    @classmethod
    def split(cls, word: str) -> list:
        return [char for char in word]

    @classmethod
    def get_child(cls, parent: list) -> list:
        return parent.pop()

    @classmethod
    def count_ones(cls, binary_list: list) -> int:
        count = 0
        for bit in binary_list:
            if bit == "1":
                count = count + 1
        return count
```

Figure 7-6 Helper Methods for Constructing the Worst-Case Scenario.

Method Definition - split(word): For translating a binary string into a list of '0' and '1'.

Input: String containing zeroes and ones.

Output: List containing zeroes and ones as characters.

Data Structures Used: The input parameter is a string word and the method returns a list representation of the string that contains all the characters of the string. We do that by iterating over each character of the list and by encapsulating that in square brackets we return a list of those characters.

Method Definition - get_child(parent):

Input: A list containing lists of binary strings

Output: The last element of the list.

Data Structures Used: This method takes a list by reference and pops and returns the last element of the list. The reason for this small method is to make the main method more readable and abstract.

Method Definition - count_ones(binary list): This function is used to count the number of ones contained in a current list. It is helpful when we want to check if we can add any more zeroes in the next iteration.

Input: A list containing ones and zeroes

Output: The number of ones.

Data Structures Used: The parameter is a list containing characters that are '0' or '1'. We initialize a count variable and we iterate through each character and increment whenever we find a '1' and return count.

Given a number of bits n , we want a method that will find all the possible combinations of n -bit numbers with k -bits set to one where $0 \leq k \leq n$. The solution should print all numbers with one set bit first, followed by numbers with two bits set, etc, up to the numbers whose all n -bits are set to one. The only cases where we manually add lists is for the all-zero and all one lists, which are added first and last. The number k symbolizes how many '1' we will have in each iteration.

Tree Diagram for 4-bit combinations.

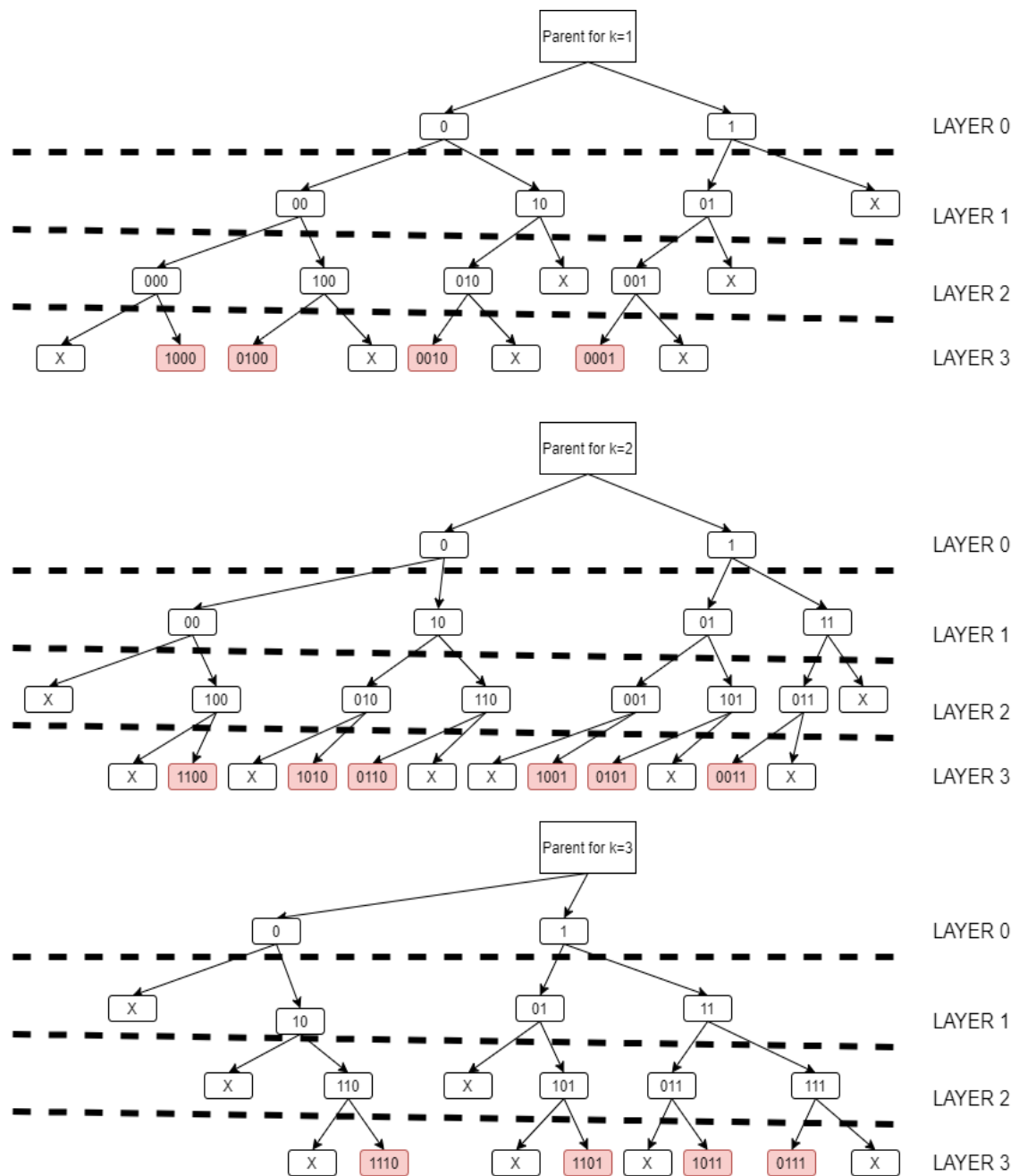


Figure 7-7 All different Combinations for 4 bits.

```

def combinations(bits: int) -> list:
    final_layer = []
    for n_bits in range(1, bits):
        parent = []
        for layer in range(bits - 1):
            new_layer = []
            if layer == 0:
                parent = []
                b0 = ["0"]
                b1 = ["1"]
                parent.append(b0)
                parent.append(b1)
            while len(parent) > 0:
                child = get_child(parent)
                new_child1 = ["1"]
                new_child0 = ["0"]
                remaining_ones_for_child = n_bits - count_ones(child)
                remaining_length = bits - len(child)
                if remaining_ones_for_child == remaining_length:
                    new_child1.extend(child)
                    new_layer.append(new_child1)
                elif 0 < remaining_ones_for_child < remaining_length:
                    new_child1.extend(child)
                    new_layer.append(new_child1)
                    new_child0.extend(child)
                    new_layer.append(new_child0)
                elif remaining_ones_for_child == 0:
                    new_child0.extend(child)
                    new_layer.append(new_child0)
                parent = new_layer
            final_layer.append(parent)
        result = []
        all_zeros = [[split("0" * bits)]]
        all_ones = [[split("1" * bits)]]
        result.extend(all_zeros)
        result.extend(final_layer)
        result.extend(all_ones)
    return result

```

Figure 7-8 Code for Generating n-bit Combinations.

Method Definition - Combinations(bits): This method is responsible to generate all bit combinations and return a list containing all of them.

Input: The number of bits n to determine the length of the n-bit combinations

Output: A list that contains all possible combinations with a number of '1' ranging from zero to n

Data Structures Used: We used an iterative approach to tackle this problem. We initialize an empty list, `final_layer`, which in the end will contain all 2^n combinations. We then start a loop that will iterate all K parent lists. In each of this loop's iteration, we want to finish producing all the combinations with K bits set to '1'. In the code, K is named `n_bits`. The loop is started with K equal to one and ends with K equal to `n-1`.

Now, to complete each parent, we initialize an empty list and start to fill each layer. The number of layers needed to complete a parent is always equal to `n`. For layer zero, which is the first layer, we initialize our 'tree' by appending the two possible elements, '0' and '1' to the parent list.

Now for each layer, using a while loop we take each child of the parent and count how many ones, we are allowed to still add to that child according to our current K. We do that by using the variable `remaining_ones_for_child`, which calculates the remaining ones by subtracting the current amount of ones that the child list contains using the `count_ones()` method we described previously and subtracting that from the `n_bits`. We then calculate the remaining length of the child list by subtracting the current length of the child from the number of bits `n`. We are now left with two options. Either append '1' or '0' to the beginning of the list. The way we do that and prepare the child for the next layer is by creating two lists at the beginning of the while loop, one starting with '1' and one starting with '0', named `new_child1` and `new_child0`. We then extend to those newly created lists with the child of the previous layer and append the new child to the new layer thus slowly creating the next layer. We have the option of appending the old child list to the `new_child1` or the `new_child0` or both, to determine to which new children the old child list will be appended and continue its evolution to the new layer, a child must fulfil some properties which we lay down in conditional statements. If the child's remaining ones that need to be added are equal to the number of bits that this child is allowed, then we can only add ones to this child and not zeroes. All these scenarios are shown in Figure 7-7. If the first conditional statement is not true, we check if the remaining ones for the child are more than zero and less than the remaining length that the current child can take. In this case, we can create two new children from the previous child, and we do this by extending both `new_child0` and `new_child1` with the previous child and appending them both to the next layer.

Lastly, for the conditions, if the remaining ones are neither equal to the number of bits the child is allowed nor more than zero, we are only allowed to extend the `new_child0` list, which means appending a new child that contains an additional zero to the front.

Once we have done this for all children of the current parent list, we and emptied it, we are left with the `new_layer` list that contains all possible sequences with one more bit than the previous layer. We assign that new layer as our new parent list and continue until we have reached the final layer iteration and the last parent list pretty much contains all possible combinations and we assign it to the `final_layer` list.

We then create a new list result and manually append all the combinations with zero ones in the sequence, which is only one combination, then all the other combinations we have generated, and then all the combinations with n ones in the sequence which is again only one combination. So, to summarize, the result list which will be returned is a list that contains lists. Each list element result contains, has all combinations of k-bits set to one.

This is depicted nicely in Figure 7-9 for four bits.

```
Enter number of bits: 4
Combinations for K=0 bits set to '1'.
[['0', '0', '0', '0']]

Combinations for K=1 bits set to '1'.
[['0', '0', '0', '1'], ['0', '0', '1', '0'], ['0', '1', '0', '0'], ['1', '0', '0', '0']]

Combinations for K=2 bits set to '1'.
[['0', '0', '1', '1'], ['0', '1', '0', '1'], ['0', '1', '1', '0'], ['1', '0', '0', '1'], ['1', '0', '1', '0'], ['1', '1', '0', '0']]

Combinations for K=3 bits set to '1'.
[['1', '0', '1', '1'], ['1', '1', '0', '1'], ['1', '1', '1', '0']]

Combinations for K=4 bits set to '1'.
[['1', '1', '1', '1']]
```

Figure 7-9 The n-bit Combinations Grouped by Number of Ones.

Now we need to re-sort the result list to have its first 2^{n-1} elements grouped by what they would make the black box function output. In the case of the xor operation, having an odd number of bits will output one, and having an even will output zero.

With that in mind, we simply must choose all k-one combinations that are either of an odd or even sum of ones, for our experiment to put odd sums first and group them

together and only then append the even groups. Looking back to Figure 7-5 we can see that we have successfully simulated the worst possible scenario for the classical computer, which is feeding our algorithm all inputs that make the balanced function output one. Only then supply it with an input that will provide zero to the output, thus confirming a balanced Boolean function. We do this with the help of the `produce_worse_scenario` method:

```
@classmethod
def produce_worse_scenario(cls, combos: list) -> list:
    worse_input_list = []
    even = []
    odd = []
    for i in range(len(combos)):
        if i % 2 == 1: # list with even amount of zeroes and ones
            even.extend(combos[i])
        else:
            odd.extend(combos[i])
    worse_input_list.extend(even)
    worse_input_list.extend(odd)
    return worse_input_list
```

Figure 7-10 Code for Rearranging the bit Combinations to Construct Worst Scenario.

Method definition - `produce_worse_scenario(combos)`: Take the list that contains all lists of lists of ones and zeroes and group them into odd and even groups. Then append all odd groups to one list and the even group afterwards to produces a list containing all the combinations of uneven amounts of ones and zeroes and combinations of ones and zeroes with the same number of ones and zeroes in a row following.

Input: A list containing lists, where each corresponding list contains all combinations for 1 to n-1 ones.

Output: A sorted list that groups even and uneven numbers of ones and zeroes.

Data Structures Used: The `combos` parameter is the result list from the `BitCombinations` method. We initialize an empty list `worse_input_list` and in a loop, we check to see if the current `i` modulo 2 is odd. We do that because we know how the structure of the `combos` list is. We know that for four bits for example when `i` is equal to zero, the list contains all the combinations with zero ones which when given to the black box function will be balanced. For `i` equal to one we have an odd number of ones, 1, for `i` equal to two we have two ones, etc. This is an easy way to group our lists.

Below we can see the worst possible input for four bits.

```
Worst input
['0', '0', '0', '1']
['0', '0', '1', '0']
['1', '0', '0', '0']
['0', '1', '0', '0']
['1', '0', '1', '1']
['0', '1', '1', '1']
['1', '1', '0', '1']
['1', '1', '1', '0']
['0', '0', '0', '0']
['0', '0', '1', '1']
['1', '0', '0', '1']
['0', '1', '0', '1']
['1', '0', '1', '0']
['0', '1', '1', '0']
['1', '1', '0', '0']
['1', '1', '1', '1']
```

Figure 7-11 Output of Worst Scenario for 4 bits.

7.2.2.1 Results for Worst-case scenario

After applying the worst-case scenario, we always need to check for $2^{n-1} + 1$ times.

12 bits = $2^{12} = 4,096$ and for $2^{n-1} + 1$ checks we have $2^{11} + 1$ which is 2049 checks.

```
Enter:
1 for execution on the Local Device Simulator.
2 for execution on the Qasm simulator.
3 for execution on a real device.
4 for execution on the local device and real device!
Your input:1
Enter number of bits for a the classical solution:12
***** FINAL * RESULTS *****
Balanced After 2049 checks.
Determining if xor is balanced for 12 bits took 0.005177 seconds.
Time to generate worse input for 12 bits took 0.03375 seconds.
*****
```

Figure 7-12 Execution Times for Classical Solution for 12 bits.

23 bits = $2^{23} = 8,388,608$ and we have $2^{22} + 1$ which is 4,194,305 checks.

```
Enter:
1 for execution on the Local Device Simulator.
2 for execution on the Qasm simulator.
3 for execution on a real device.
4 for execution on the local device and real device!
Your input:1
Enter number of bits for a the classical solution:23
***** FINAL * RESULTS *****
Balanced After 4194305 checks.
Determining if xor is balanced for 23 bits took 14.249531 seconds.
Time to generate worse input for 23 bits took 51.785056 seconds.
*****
```

Figure 7-13 Execution times for Classical Solution for 23 bits.

24 bits = $2 * 2^{23} = 16,777,216$ and we have $2^{23} + 1$ which is 8,388,609 checks.

```
Enter:
1 for execution on the Local Device Simulator.
2 for execution on the Qasm simulator.
3 for execution on a real device.
4 for execution on the local device and real device!
Your input:1
Enter number of bits for a the classical solution:24
***** FINAL * RESULTS *****
Balanced After 8388609 checks.
Determining if xor is balanced for 24 bits took 30.403439 seconds.
Time to generate worse input for 24 bits took 101.773189 seconds.
*****
```

Figure 7-14 Execution Times for Classical Solution for 24 bits.

We can see that for each additional bit added the calculation time is exponentially increased. For just 32 bits it would take approximately 7 hours to calculate the worst-case scenario and approximately 2 hours to come to a solution.

7.3 Quantum Approach

We have set up a problem that is exponentially more complex. With the Deutsch – Jozsa algorithm we can devise a scheme for calculating all the possible outcomes in polynomial time instead of exponential using the quantum parallelism concept. Now we will work on an example of $n = 2$ qubits. Hadamard gates are used to prepare a superposition of all four possible states, 2^2 . We also need to implement the function f , or oracle function as it is called, in some way. When Deutsch – Jozsa is explained, the oracle function is often interpreted as a black box function that uses quantum parallelism to compute all values of $f(x)$, x being one of the four possible configurations in our example.

$$|\alpha\rangle = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$$

Another concept that makes Deutsch –Jozsa work, is interference, which is achieved with the help of Hadamard gates once again. After applying them for a second time to the values that the oracle function outputted the measurement will be zero for all constant functions and one for balanced functions.

In our experiment, we made use of the oracle function implemented with CNOT gates, which are the quantum equivalent of the traditional XOR gate.

XOR Truth table	
00	0
01	1
10	1
11	0

CNOT Truth table	
00	0
01	1
10	1
11	0

Table 1 XOR and CNOT Truth Tables.

7.3.1 Algorithm's Structure

If we were to describe an abstract form of the algorithm it would comprise four major steps.

Step 1: Prepare the superposition of the qubits.

Step 2: The oracle function f .

Step 3: Interference.

Step 4: Measurements.

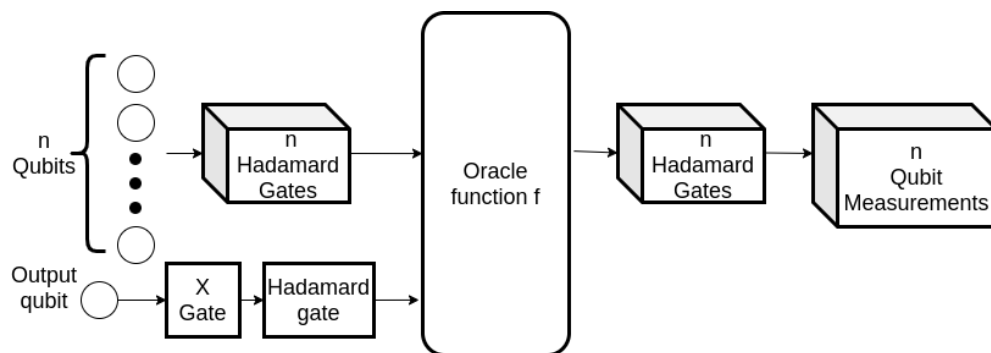


Figure 7-15 Deutsch Jozsa Structure.

This is the general form of the Deutsch-Jozsa for n qubits. For n qubits, we prepare a superpositioned state of 2^{n+1} different states because of the addition of the output qubit. This state is passed into the oracle function and all possible states are evaluated in polynomial time. We then proceed with the decoupling of this quantum state by taking advantage of Hadamard being its inverse and phase kickback.

7.3.2 Proof of Concept with a worked example

Using Dirac notation as a tool we can prove that in an errorless system we can expect to get a consistent output for both types of functions. For this worked example, we have two-qubit input and must show that for a balanced function the output bits measure one and for a constant, zero. It is worth noting that the output qubit is not measured in the final step since we can effectively and reliably determine the function while only measuring the input qubits.

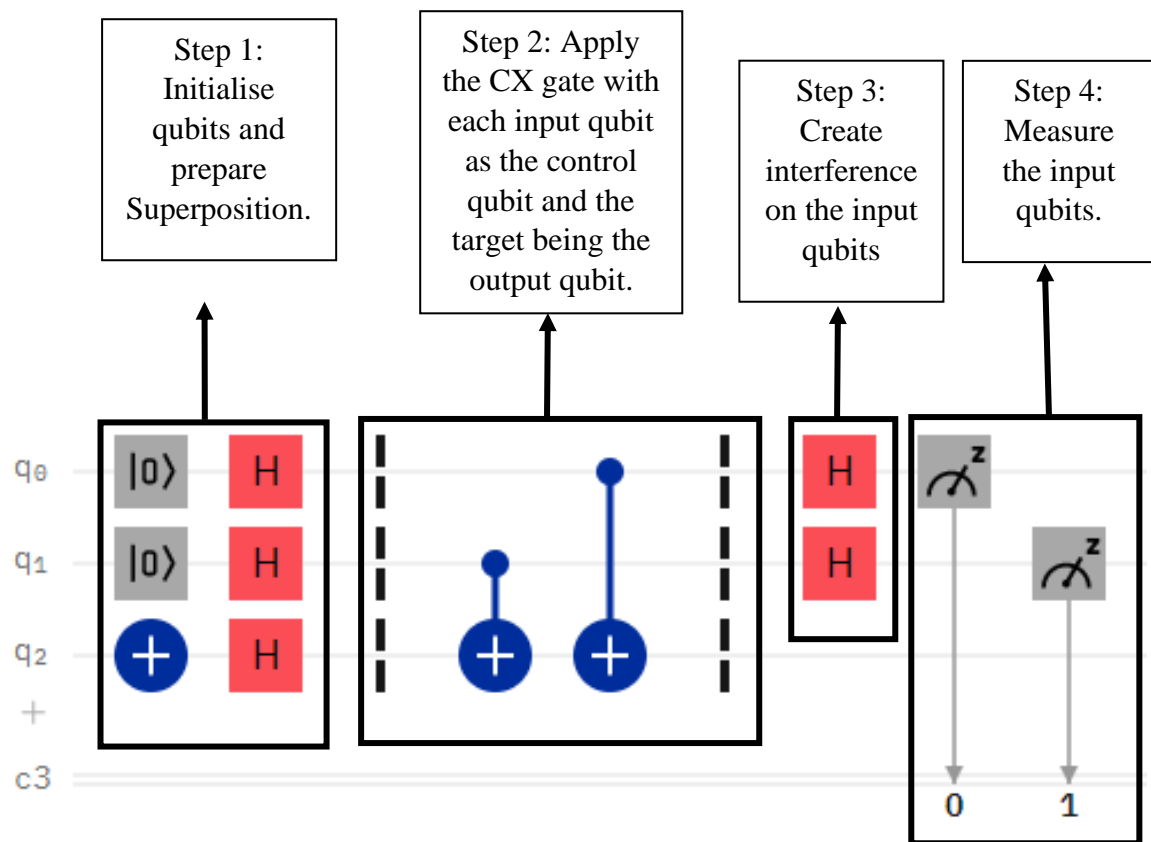


Figure 7-16 Worked Example.

7.3.2.1 Step 1 – Initialization

We initialize two-qubit registers. One contains 2 qubits, q0, and q1, and is initialized to the $|0\rangle$ state and the other one contains one qubit, q2, which is initialized to the $|1\rangle$ state.

$$|\psi_0\rangle = |0_0 0_1\rangle * |1_2\rangle$$

We then create a superposition of the three qubits in the following manner:

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0_0\rangle + |1_0\rangle) * \frac{1}{\sqrt{2}}(|0_1\rangle + |1_1\rangle) * \frac{1}{\sqrt{2}}(|0_2\rangle - |1_2\rangle)$$

Using the distributive property:

$$|\psi_1\rangle = \frac{1}{2}(|0_0 0_1\rangle + |0_0 1_1\rangle + |1_0 0_1\rangle + |1_0 1_1\rangle) * \frac{1}{\sqrt{2}}(|0_2\rangle - |1_2\rangle)$$

$$|\psi_1\rangle = \frac{1}{2\sqrt{2}}(|0_0 0_1 0_2\rangle + |0_0 1_1 0_2\rangle + |1_0 0_1 0_2\rangle + |1_0 1_1 0_2\rangle - |0_0 0_1 1_2\rangle - |0_0 1_1 1_2\rangle - |1_0 0_1 1_2\rangle - |1_0 1_1 1_2\rangle)$$

7.3.2.2 Step 2 - Oracle

We have created a superposition of all possible states and we now pass this ‘variable’ into our function. Our function, which is essentially two C-NOT gates with the input qubits as the control qubits and the output qubit as the target qubit, maps the state $|x\rangle|y\rangle$ to the state $|x\rangle|y \oplus f(x)\rangle$. In order to understand this mapping let’s consider the C-NOT gate’s properties. It does not alter the control qubit’s state. Hence $|x\rangle$ is outputted as itself. Control qubits q0 and q1 follow the $|x\rangle$ mapping. However, the $|y\rangle$ qubit of the function, in our case the output-target qubit q2 will have its value XORed with the outcome of the next CNOT operation on itself. So, for our case, qubits zero and one will remain the same, but qubit two will be transformed according to our oracle’s implementation in a manner the following manner:

From $|\psi_1\rangle$ in Step 1, after passing through the oracle function, the system will be in this state:

$$\begin{aligned} |\psi_2\rangle = & \frac{1}{2\sqrt{2}} \left(|0_0 0_1 \left((0_2 \oplus f(0_0)) \oplus f(0_1) \right)_2 \rangle + |0_0 1_1 \left((0_2 \oplus f(0_0)) \oplus f(1_1) \right)_2 \rangle + \right. \\ & |1_0 0_1 \left((0_2 \oplus f(1_0)) \oplus f(0_1) \right)_2 \rangle + |1_0 1_1 \left((0_2 \oplus f(1_0)) \oplus \right. \\ & \left. \left. f(1_1) \right)_2 \rangle - |0_0 0_1 \left((1_2 \oplus f(0_0)) \oplus f(0_1) \right)_2 \rangle - |0_0 1_1 \left((1_2 \oplus f(0_0)) \oplus f(1_1) \right)_2 \rangle - \right. \\ & \left. |1_0 0_1 \left((1_2 \oplus f(1_0)) \oplus f(0_1) \right)_2 \rangle - |1_0 1_1 \left((1_2 \oplus f(1_0)) \oplus f(1_1) \right)_2 \rangle \right) \end{aligned}$$

As we know, $x \oplus 0 = x$ and $x \oplus 1 = \bar{x}$ so $|\psi_2\rangle$ is simplified to:

$$\begin{aligned} |\psi_2\rangle = & \frac{1}{2\sqrt{2}} \left(|0_0 0_1 (f(0_0) \oplus (0_1))_2 \rangle + |0_0 1_1 (f(0_0) \oplus f(1_1))_2 \rangle + |1_0 0_1 (f(1_0) \oplus \right. \\ & \left. f(0_1))_2 \rangle + |1_0 1_1 (f(1_0) \oplus f(1_1))_2 \rangle - |0_0 0_1 (\bar{f}(0_0) \oplus f(0_1))_2 \rangle - |0_0 1_1 (\bar{f}(0_0) \oplus \right. \\ & \left. f(1_1))_2 \rangle - |1_0 0_1 (\bar{f}(1_0) \oplus f(0_1))_2 \rangle - |1_0 1_1 (\bar{f}(1_0) \oplus f(1_1))_2 \rangle \right) \end{aligned}$$

Before we proceed to Step 3, we need to consider the nature of our function. As discussed above, we can summarize by saying that a constant function is where $f(0) = f(1)$ and a balanced function is where $f(0) = \bar{f}(1)$

If Constant:

$$\begin{aligned} \xRightarrow{|\psi_2\rangle} |\psi_c\rangle = & \frac{1}{2\sqrt{2}} \left(|0_0 0_1 (f(0) \oplus f(0))_2 \rangle + |0_0 1_1 (f(0) \oplus f(0))_2 \rangle + |1_0 0_1 (f(0) \oplus \right. \\ & \left. f(0))_2 \rangle + |1_0 1_1 (f(0) \oplus f(0))_2 \rangle - |0_0 0_1 (\bar{f}(0) \oplus f(0))_2 \rangle - |0_0 1_1 (\bar{f}(0) \oplus \right. \\ & \left. f(0))_2 \rangle - |1_0 0_1 (\bar{f}(0) \oplus f(0))_2 \rangle - |1_0 1_1 (\bar{f}(0) \oplus f(0))_2 \rangle \right) \end{aligned}$$

After factoring:

$$|\psi_C\rangle = \frac{1}{2\sqrt{2}}(|0_00_1\rangle + |0_01_1\rangle + |1_00_1\rangle + |1_01_1\rangle) * ((f(0) \oplus f(0))_2 - (\bar{f}(0) \oplus f(0))_2)$$

Simplifying the contents of qubit zero by replacing $f(x) \oplus f(x)$ with zero and $\bar{f}(x) \oplus f(x)$ with one.

$$|\psi_C\rangle = \frac{1}{\sqrt{2}}(|0_0\rangle + |1_0\rangle) * \frac{1}{\sqrt{2}}(|0_1\rangle + |1_1\rangle) * \frac{1}{\sqrt{2}}(|0_2\rangle - |1_2\rangle)$$

If Balanced:

$$\begin{aligned} \xRightarrow{|\psi_2\rangle} |\psi_B\rangle = & \frac{1}{2\sqrt{2}} \left(|0_00_1(f(0) \oplus f(0))_2\rangle + |0_01_1(f(0) \oplus \bar{f}(0))_2\rangle + |1_00_1(\bar{f}(0) \oplus f(0))_2\rangle \right. \\ & \left. + |1_01_1(\bar{f}(0) \oplus \bar{f}(0))_2\rangle - |0_00_1(\bar{f}(0) \oplus f(0))_2\rangle - |0_01_1(\bar{f}(0) \oplus \bar{f}(0))_2\rangle \right. \\ & \left. - |1_00_1(f(0) \oplus f(0))_2\rangle - |1_01_1(f(0) \oplus \bar{f}(0))_2\rangle \right) \end{aligned}$$

Simplifying the contents of qubit zero by replacing $f(x) \oplus f(x)$ with zero and $\bar{f}(x) \oplus f(x)$ with one.

$$|\psi_B\rangle = \frac{1}{2\sqrt{2}}(|0_00_10_2\rangle + |0_01_11_2\rangle + |1_00_11_2\rangle + |1_01_10_2\rangle - |0_00_11_2\rangle - |0_01_10_2\rangle - |1_00_10_2\rangle - |1_01_11_2\rangle)$$

After factoring:

$$|\psi_B\rangle = \frac{1}{2\sqrt{2}}(|0_00_1\rangle - |0_01_1\rangle - |1_00_1\rangle + |1_01_1\rangle) * (|0\rangle - |1\rangle_2)$$

$$|\psi_B\rangle = \frac{1}{\sqrt{2}}(|0_0\rangle - |1_0\rangle) * \frac{1}{\sqrt{2}}(|0_1\rangle - |1_1\rangle) * \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle_2)$$

7.3.2.3 Step 3 - Interference

Interference is the process of applying a Hadamard gate to each input qubit.

If Constant:

$$\xrightarrow{|\psi_C\rangle} |\psi_{CI}\rangle = \frac{1}{\sqrt{2}}(|0_0\rangle + |1_0\rangle)^H * \frac{1}{\sqrt{2}}(|0_0\rangle + |1_0\rangle)^H * \frac{1}{\sqrt{2}}(|0_2\rangle - |1_2\rangle)$$

$$|\psi_{CI}\rangle = |0_0\rangle * |0_1\rangle * \frac{1}{\sqrt{2}}(|0_2\rangle - |1_2\rangle)$$

If Balanced:

$$\xrightarrow{|\psi_B\rangle} |\psi_{BI}\rangle = \frac{1}{\sqrt{2}}(|0_0\rangle - |1_0\rangle)^H * \frac{1}{\sqrt{2}}(|0_0\rangle - |1_0\rangle)^H * \frac{1}{\sqrt{2}}(|0_2\rangle - |1_2\rangle)$$

$$|\psi_{BI}\rangle = |1_0\rangle * |1_1\rangle * \frac{1}{\sqrt{2}}(|0_2\rangle - |1_2\rangle)$$

We can see that after interference, the input qubits are in the ground state if the function is constant and in the excited state if the function is balanced. The driving force between all this trickery is the Hadamard gate, which as we have previously found out, is its inverse. Matrix multiplication clears the situation quite easily.

$$|\psi_H\rangle = \left(\frac{1}{\sqrt{2}}(|0_0\rangle - |1_0\rangle)\right)^H \quad \Leftrightarrow \quad |\psi_H\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$$|\psi_H\rangle = \frac{1}{2} \begin{bmatrix} 1 * 1 + 1 * (-1) \\ 1 * 1 + (-1)(-1) \end{bmatrix} \quad \Leftrightarrow \quad |\psi_H\rangle = \frac{1}{2} \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

$$|\psi_H\rangle = \frac{1}{2} * 2 * \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \Leftrightarrow \quad |\psi_H\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \Leftrightarrow \quad |\psi_H\rangle = |1\rangle$$

Likewise, if the input qubit is in the state,

$$|\psi_H\rangle = \left(\frac{1}{\sqrt{2}}(|0_0\rangle + |1_0\rangle)\right)^H \quad \Leftrightarrow \quad |\psi_H\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \Leftrightarrow \quad |\psi_H\rangle = |0\rangle$$

7.3.2.4 Step 4 - Measurements

We now apply two measurements to qubit zero and qubit one. If the function is constant, we expect two zeroes and if the function is balanced, we expect two ones. Since we know that f-CNOT is a balanced function we expect to see one from both qubits.

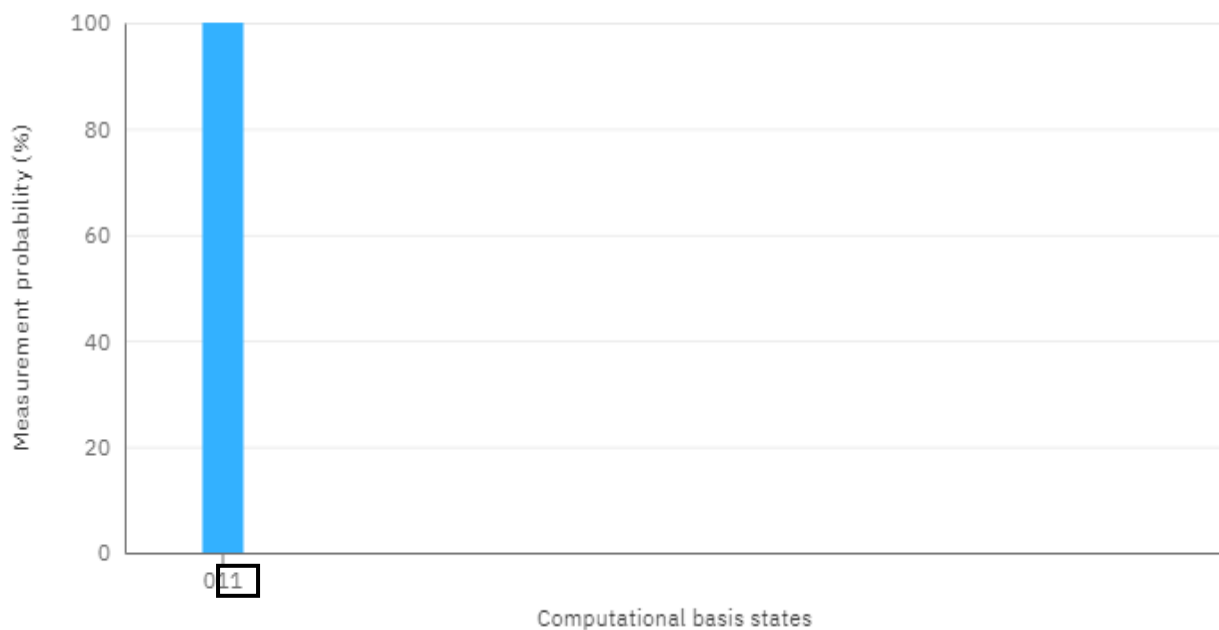


Figure 7-17 Measurement Probabilities for Balanced Function.

From Figure four we see that qubit zero's value is stored in bit zero in the classical bit register and qubit one's value is stored in bit one. We see that indeed one hundred percent of the time the result is that bot input qubits have the value one which indicates that the f-CNOT is a balanced function.

This concludes the logic behind the Deutsch-Jozsa algorithm and why it works. We now devise a plan to exploit this algorithm to determine a function's nature and compare it to a classical computer using the mainstream tools available to the modern software developer.

7.3.3 Quantum Implementation

The code is very straightforward. We have a class called `DeutschJozsa` in which we will create our quantum circuit object.

Method definition – `run_deutsch_josza(bit_string, eval_mode)`:

We first initialize a quantum circuit with $n+1$ qubits since we need to account for the output qubit as well and apply the necessary Hadamard gates to achieve a superpositioned state going into the oracle function appendment. We then apply interference and measure all input qubits.

Input: A bit sequence, a Boolean value dictating whether or not the algorithm will operate on `eval_mode`, which essentially means not to print any display messages.

Output: The `QuantumCircuit` object.

Data Structures Used: In step one we create an empty `QuantumCircuit` object consisting of $n+1$ qubits and n bits. We then apply Hadamard gates to all input qubits using the `h()` method of the `QuantumCircuit` class and in a similar manner append an X gate and then a Hadamard gate to the output qubit.

In step two we create an oracle which is essentially another `QuantumCircuit` object. We do this with the `CnotOracle` Class. The `create_cnot_oracle()` method is responsible for returning an oracle of appropriate dimensions that works with CNOT gates.

Method Definition – `create_cnot_oracle(input_string, input_length, eval_mode)`:

Input: The user-given input string, its length, and the operation mode

Output: A `QuantumCircuit` object

Data Structures Used: We create a `QuantumCircuit` object of `input_length+1` qubits. The first and third for loops are there to initialize the qubits in a different state but in our experiments, we always start with the ground state of all qubits being zero. The second loop creates a CNOT operation of each input qubit with the output qubit. Before and after appending the CNOT operations we use two barrier objects. If the method is on `eval_mode`, then the printing messages will not be executed.

```

from qiskit import QuantumCircuit

class CnotOracle:
    @classmethod
    def create_cnot_oracle(cls, input_string, input_length, eval_mode: bool)
-> QuantumCircuit:
    balanced_oracle = QuantumCircuit(input_length + 1)
    # Place X-gates
    for qubit in range(len(input_string)):
        if input_string[qubit] == '1':
            balanced_oracle.x(qubit)

    # Use barrier as divider
    balanced_oracle.barrier()

    # Controlled-NOT gates
    for qubit in range(input_length):
        balanced_oracle.cx(qubit, input_length)

    balanced_oracle.barrier()

    # Place X-gates
    for qubit in range(len(input_string)):
        if input_string[qubit] == '1':
            balanced_oracle.x(qubit)
    if not eval_mode:
        # Show oracle
        print("This is the oracle function, aka the black box. NORMALLY
THIS WOULD BE HIDDEN!")
        print(balanced_oracle)
    return balanced_oracle

```

Figure 7-18 Code for Creating a Balanced Oracle.

In step 3 we apply Hadamard gates to all input qubits using a loop and then apply an additional barrier before applying measurements from all input qubits to the corresponding measurement bits.

The DeutschJozsa class:

```
from qiskit import QuantumCircuit
from oracles.cnot_oracle import CnotOracle

class DeutschJozsa:

    @classmethod
    def deutsch_jozsa(cls, bit_string: str, eval_mode: bool) -> QuantumCircuit:
        n = len(bit_string)

        dj_circuit = QuantumCircuit(n + 1, n)
        # Apply H-gates
        for qubit in range(n):
            dj_circuit.h(qubit)

        # Put output qubit in state |->
        dj_circuit.x(n)
        dj_circuit.h(n)

        # Construct balanced oracle
        balanced_oracle = CnotOracle.create_cnot_oracle(bit_string, n,
eval_mode)

        # Add oracle
        dj_circuit += balanced_oracle

        # Repeat H-gates
        for qubit in range(n):
            dj_circuit.h(qubit)
        dj_circuit.barrier()

        # Measure
        for i in range(n):
            dj_circuit.measure(i, i)
        if not eval_mode:
            print(dj_circuit)

        # return circuit
        return dj_circuit
```

Step 1: Initialisation

Step 2: Oracle

Step 3: Interference

Step 4: Measurements

Figure 7-19 Code for Implementing Deutsch-Jozsa/

Indicative Results:

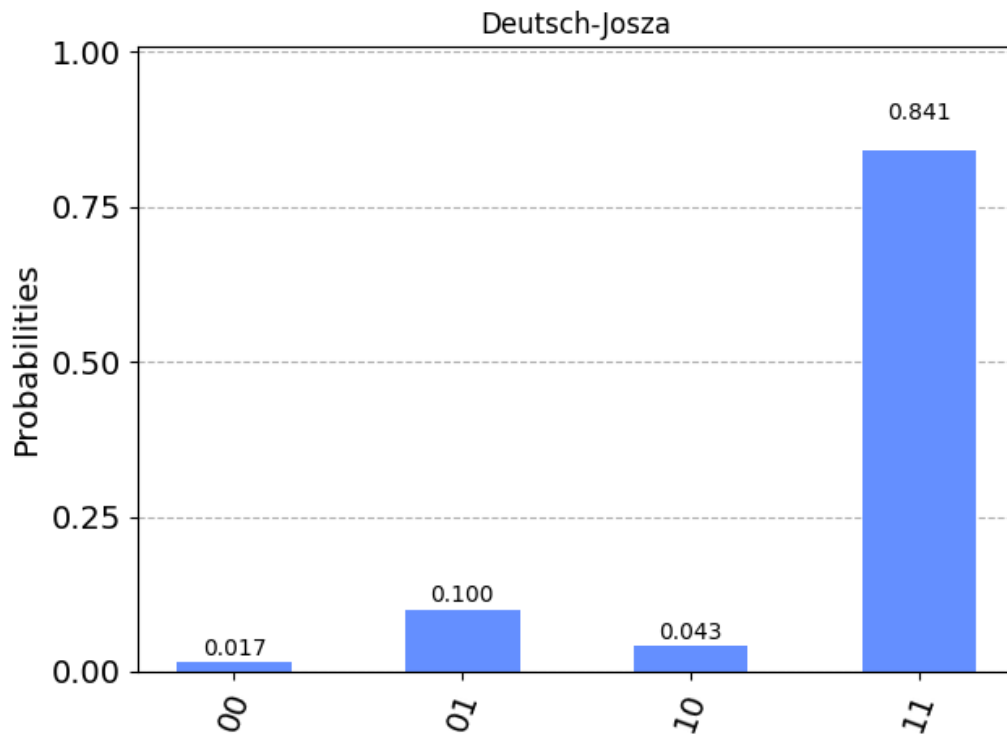


Figure 7-20 Different States Measured for Deutsch-Jozsa.

```
***** FINAL * RESULTS *****  
  
Total counts are: {'00': 17, '01': 102, '10': 44, '11': 861}  
The time it took for the experiment to complete after validation was 5.900763034820557 seconds  
*****  
Results of classical implementation for the Deutsch-Josza Algorithm:  
Function is Balanced After 3 checks.  
Time to generate worse input for 2 bits took 0.0 seconds.  
Determining if xor is balanced for 2 bits took 0.0 seconds.  
*****
```

Figure 7-21 Results of a Classical and Quantum Execution for Deutsch-Jozsa.

We can see that indeed Deutsch-Jozsa outputs the “11” state which dictates that the C-NOT operation is balanced for many executions.

Also, we can see that our classical approach found the answer in 3 attempts, which for 2 bits goes according to the scheme we devised for the worst input which is $2^{n-1} + 1$.

7.4 Bernstein - Vazirani

Suppose a secret binary number X . We want to guess that number as quickly as possible, only knowing the length of the binary. We want to create a function that operates as few times as possible in order to find the secret number, [3].

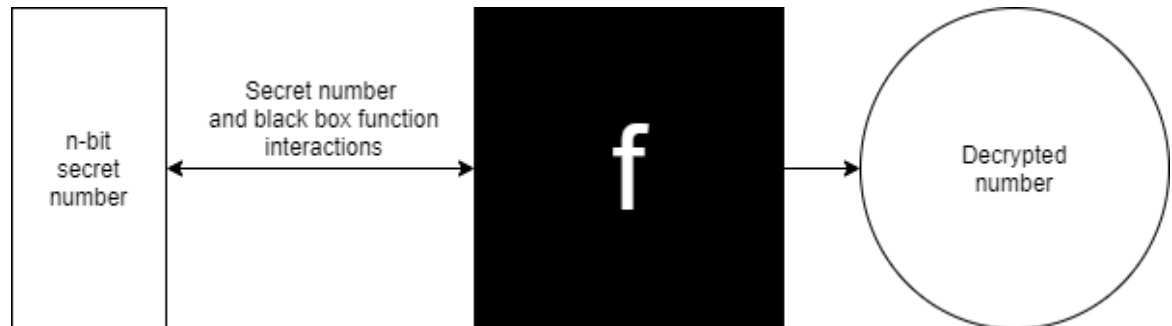


Figure 7-22 Bernstein-Vazirani Abstract Form.

How would a classical computer go to guess the number? One naïve approach would be to guess binary numbers of that length at random until we find it. It's easy to see that for a relatively long binary this would be rather hopeless since there are 2^n possible guesses. This is far from decreasing the number of possible iterations of our algorithm.

7.5 Classical Approach

There is of course a better way. We can take advantage of masking in order to guess an n-bit number in exactly n attempts each time. This is the exact algorithm below.

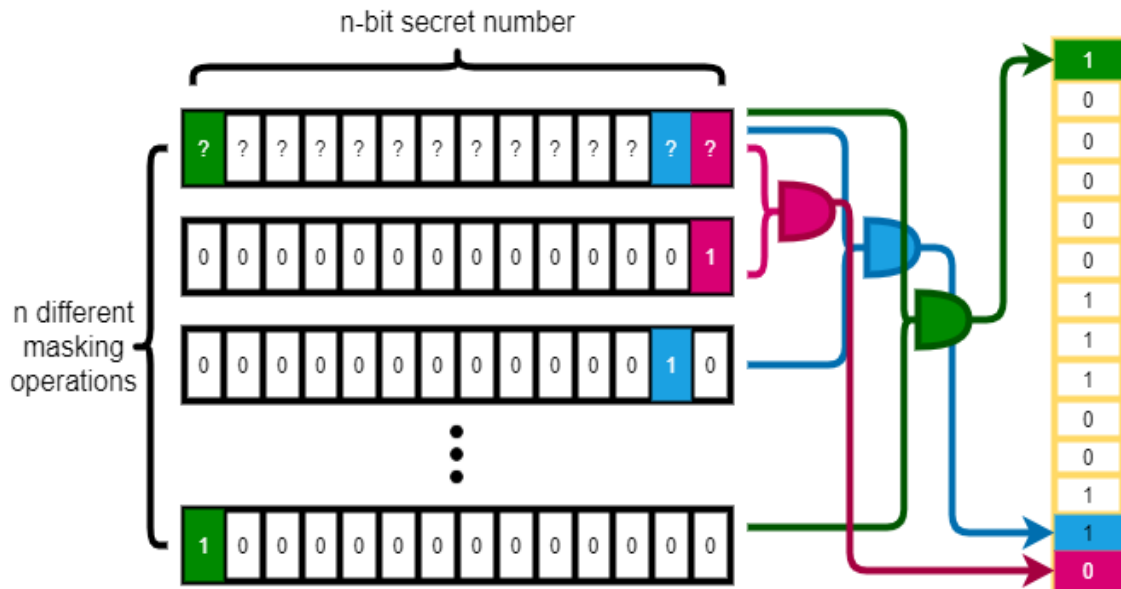


Figure 7-23 Bernstein-Vazirani Classical Approach.

We create n different n-length vectors containing only zero except in one place. We then proceed to take each vector and perform an AND operation with the secret number.

This way we decrypt a bit one by one. We start with the first vector making bit zero have the value one and all other n-1 bits have the value zero. If the zeroth bit is one then the AND operation will return one, which is the correct guess for the value of the secret bit, and if it returns zero then that is again a correct guess since zero AND one is equal to zero. All other bits of the vector will return zero since anything ANDed with zero returns zero. Hence, we can effectively check for each one of the secret bits and build the secret number in n guesses.

7.5.1 Classical Implementation

Class BernsteinVaziraniClassical:

This class contains a single method that guesses the secret number. It may seem unnecessary to proceed with the masking operations since we already receive the secret number as a parameter. This is just for demonstration purposes. In a more realistic scenario, we would not have access to the actual secret number and the masking operation would be necessary since we would send our vector to be ANDed with the secret number, and the vendor which holds the secret number would return to us just the answer of the AND operation. Then this is the procedure we would follow. Since we will generate the secret numbers though, it rightly seems a bit counterintuitive.


Method Definition – guess_number(secret_binary):

Input: A string consisting of '1's and '0's.

Output: A string displaying the guess for the number and how many attempts it took.

Operate the AND operation on each individual bit of the secret number.

In python we can do this very easily without using a vector just by choosing which of the secret number to AND with 1. If the secret number was held by another person, we could do this with the same logic just by asking them to AND the *i*th bit of the number with 1 and tell us the result.



```
class BernsteinVaziraniClassical:
    @classmethod
    def guess_number(cls, secret_binary):
        mask = 1
        guess = ""
        attempts = 0
        for bit in secret_binary:
            hit = int(bit) & mask
            guess += str(hit)
            attempts += 1
        return f"My guess After {attempts} attempts is:\n{guess}."
```

Figure 7-24 Code for Guessing a Secret Number.

7.6 Quantum Approach

The Quantum approach is identical to that of Deutsch-Jozsa, we take advantage of interference while possessing the correct oracle function.

7.6.1 Algorithms' Structure

If we were to describe an abstract form of the algorithm it would again comprise of four major steps.

Step 1: Prepare the superposition of the qubits.

Step 2: The oracle function f .

Step 3: Interference.

Step 4: Measurements.

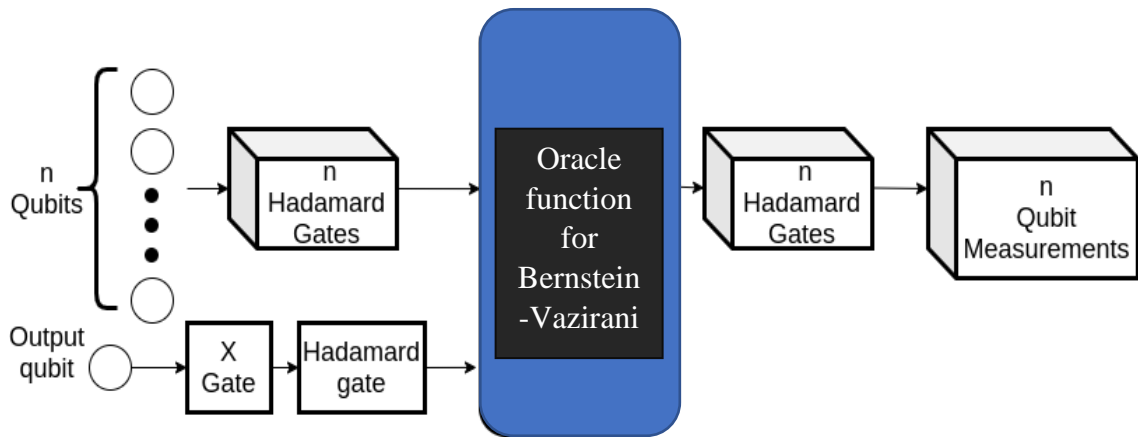


Figure 7-25 Bernstein-Vazirani Quantum Structure.

This is the general form of the Bernstein-Vazirani algorithm for n qubits. For n qubits, we prepare a superpositioned state of 2^{n+1} different states because of the addition of the output qubit. This state is passed into the oracle function and all possible states are evaluated with the oracle function. We then proceed with the decoupling of this quantum state by taking advantage of Hadamard being its inverse and phase kickback. As we can see, the structure of Bernstein-Vazirani is identical to that of Deutsch-Jozsa. The only real difference is the oracle function.

7.6.2 Proof of Concept with a worked example

Using Dirac notation as a tool we can prove that in an errorless system we can expect to get a consistent output for both types of functions. For this worked example, we have a two-bit number so we will need two input qubits and one output qubit and must show that we can successfully guess the number that is inside. It is worth noting that the output qubit is not measured in the final step since we can effectively and reliably determine the number while only measuring the input qubits. We will try and guess the secret number “01”.

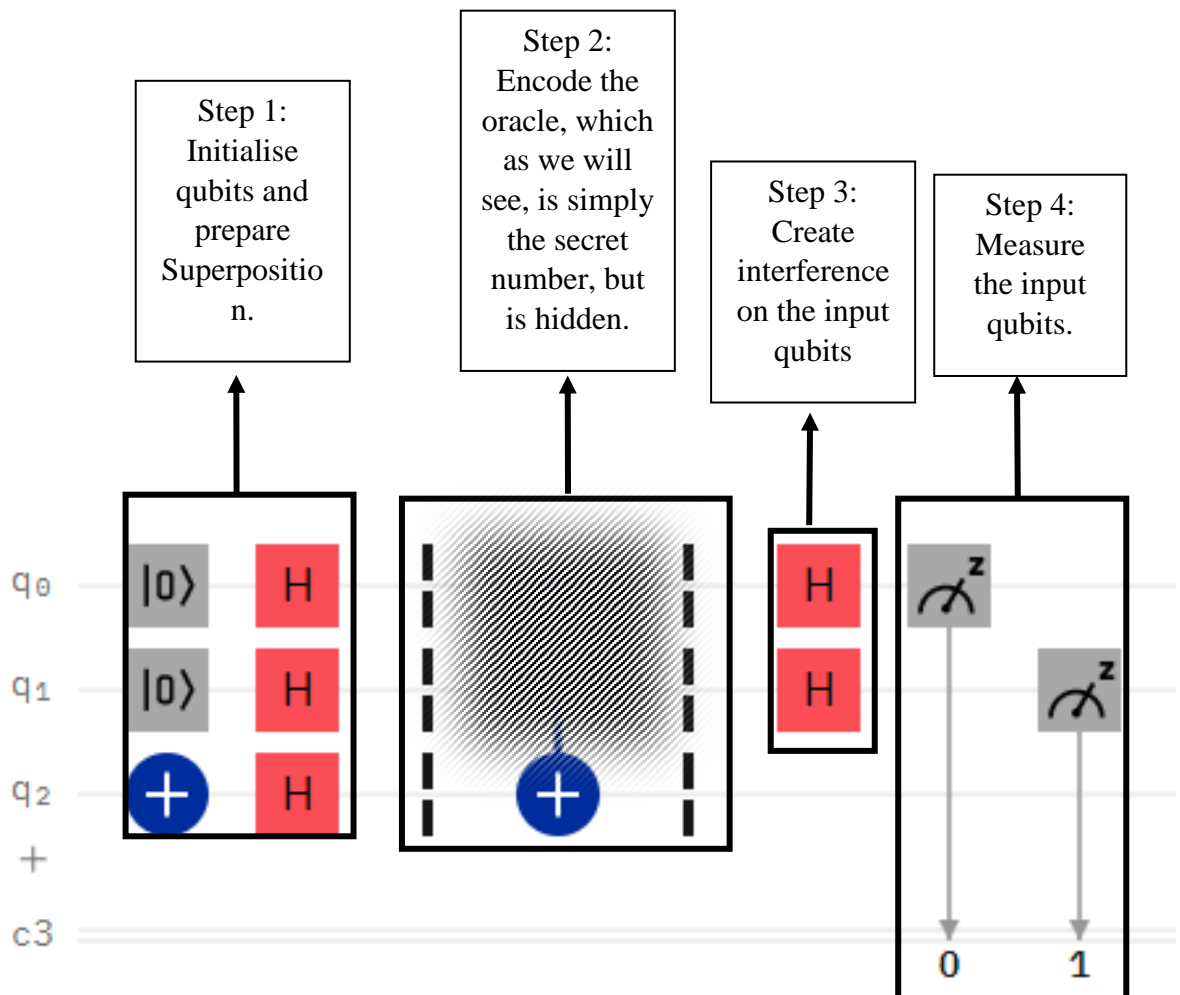


Figure 7-26 Bernstein-Vazirani Worked Example.

7.6.2.1 Step 1 – Initialization

We initialize our system the exact way we did in the Deutsch-Jozsa example.

We end up with the following system:

$$|\psi_1\rangle = \frac{1}{2\sqrt{2}}(|0_0 0_1 0_2\rangle + |0_0 1_1 0_2\rangle + |1_0 0_1 0_2\rangle + |1_0 1_1 0_2\rangle - |0_0 0_1 1_2\rangle - |0_0 1_1 1_2\rangle - |1_0 0_1 1_2\rangle - |1_0 1_1 1_2\rangle)$$

7.6.2.2 Step 2 - Oracle

Our function, which is essentially a C-NOT gate wherever necessary to shape the secret number. The state $|x\rangle|y\rangle$ is once again mapped to the state $|x\rangle|y \oplus f(x)\rangle$. Like in Deutsch-Jozsa, qubits zero and one will remain the same, but qubit two will be transformed according to our oracle's implementation in a manner that we show below:

From $|\psi_1\rangle$ in Step 1, after passing through the oracle function, the system will be in this state:

$$|\psi_2\rangle = \frac{1}{2\sqrt{2}}(|0_0 0_1 (0_2 \oplus f(0_0))_2\rangle + |0_0 1_1 (0_2 \oplus f(0_0))_2\rangle + |1_0 0_1 (0_2 \oplus f(1_0))_2\rangle + |1_0 1_1 (0_2 \oplus f(1_0))_2\rangle - |0_0 0_1 (1_2 \oplus f(0_0))_2\rangle - |0_0 1_1 (1_2 \oplus f(0_0))_2\rangle - |1_0 0_1 (1_2 \oplus f(1_0))_2\rangle - |1_0 1_1 (1_2 \oplus f(1_0))_2\rangle)$$

As we know, $x \oplus 0 = x$ and $x \oplus 1 = \bar{x}$ so $|\psi_2\rangle$ is simplified to:

$$|\psi_2\rangle = \frac{1}{2\sqrt{2}}(|0_0 0_1 f(0_0)_2\rangle + |0_0 1_1 f(0_0)_2\rangle + |1_0 0_1 f(1_0)_2\rangle + |1_0 1_1 f(1_0)_2\rangle - |0_0 0_1 \bar{f}(0_0)_2\rangle - |0_0 1_1 \bar{f}(0_0)_2\rangle - |1_0 0_1 \bar{f}(1_0)_2\rangle - |1_0 1_1 \bar{f}(1_0)_2\rangle)$$

Before we proceed to Step 3, we need to consider the nature of our function. The function f that we use is the CNOT operation which as we now know is balanced.

Hence $f(0) = \bar{f}(1)$ and $\bar{f}(0) = f(1)$.

The equation is transformed as follows:

$$|\psi_2\rangle = \frac{1}{2\sqrt{2}} (|0_0 0_1 \bar{f}(1)_2\rangle + |0_0 1_1 \bar{f}(1)_2\rangle + |1_0 0_1 f(1)_2\rangle + |1_0 1_1 f(1)_2\rangle - |0_0 0_1 f(1)_2\rangle - |0_0 1_1 f(1)_2\rangle - |1_0 0_1 \bar{f}(1)_2\rangle - |1_0 1_1 \bar{f}(1)_2\rangle)$$

After factoring:

$$|\psi_2\rangle = \frac{1}{2\sqrt{2}} (|0_0 0_1\rangle + |0_0 1_1\rangle - |1_0 0_1\rangle - |1_0 1_1\rangle) * (\bar{f}(1) - f(1))_2$$

and once more,

$$|\psi_2\rangle = \frac{1}{2\sqrt{2}} (|0_0\rangle - |1_0\rangle) * (|0_1\rangle + |1_1\rangle) * (\bar{f}(1) - f(1))_2$$

7.6.2.3 Step 3 - Interference

Interference is the process of applying a Hadamard gate to each input qubit, and we can also ignore the third qubit now.

$$|\psi_3\rangle = \frac{1}{\sqrt{2}} (|0_0\rangle - |1_0\rangle)^H * \frac{1}{\sqrt{2}} (|0_1\rangle + |1_1\rangle)^H$$

After interference, the input qubits are excited if there was a CX operation performed on the qubit inside the oracle function, or in the ground state if there was not a CX operation in the corresponding qubit inside the oracle.

Hence our resulting state is as follows:

$$|\psi_3\rangle = |0\rangle * |1\rangle * |state\ of\ qubit\ 2\rangle$$

7.6.2.4 Step 4 - Measurements

We then simply measure the first two input qubits and can expect with certainty to measure the secret number which was indeed 01.

7.6.3 Quantum Implementation

The code is very straightforward. We have a class called BernsteinVazirani in which we will create our quantum circuit object.

Method definition – bernstein_vazirani (random_binary, eval_mode):

Input: A given random binary number and a Boolean value determining if the method will run on `eval_mode`, that is not print any display messages.

Output: A `QuantumCircuit` object

Data Structures Used: This method, like in Deutsch-Jozsa is broken down into four abstract points.

In step one we create a `QuantumCircuit` object of `input_length+1` qubits. Using a for loop, we append Hadamard gates to the input qubits. We then append an X gate and a Hadamard gate immediately after to the output qubit, thus completing the initialization step.

In step 2, we first construct the secret number oracle by calling the `create_secret_number_oracle()` method from the `SecretNumberOracle` class and appending it to the `dj_circuit` which is the primary circuit object.

In step 3, we need to implement interference. We do that by instantiating a new circuit object and with a for loop append Hadamard gate to the first `n` qubits. We then also append a barrier to emphasize the distinct steps of the algorithm, but it is not necessary.

In step 4, using a for loop we assign a measurement to be performed on each of the input qubits and the value of that measurement to correspond to the correct bit in the bits register.

The BernsteinVazirani class:

```
from qiskit import QuantumCircuit
from oracles.secret_number_oracle import SecretNumberOracle

class BernsteinVazirani:

    @classmethod
    def bernstein_vazirani(cls, random_binary, eval_mode: bool) -> QuantumCircuit:

        # Construct secret number oracle

        secret_number_oracle = SecretNumberOracle.create_secret_number_oracle(random_binary=random_binary, eval_mode=eval_mode)
        num_of_qubits = secret_number_oracle.num_qubits

        # Construct circuit according to the length of the number

        dj_circuit = QuantumCircuit(num_of_qubits, num_of_qubits - 1)
        dj_circuit_before_oracle = QuantumCircuit(num_of_qubits, num_of_qubits - 1)
        # Apply H-gates
        for qubit in range(num_of_qubits - 1):
            dj_circuit_before_oracle.h(qubit)

        # Put output qubit in state |->
        dj_circuit_before_oracle.x(num_of_qubits - 1)
        dj_circuit_before_oracle.h(num_of_qubits - 1)
        dj_circuit += dj_circuit_before_oracle

        # Add oracle
        dj_circuit += secret_number_oracle
        dj_circuit_after_oracle = QuantumCircuit(num_of_qubits, num_of_qubits - 1)

        # Repeat H-gates
        for qubit in range(num_of_qubits - 1):
            dj_circuit_after_oracle.h(qubit)
        dj_circuit_after_oracle.barrier()

        # Measure
        for i in range(num_of_qubits - 1):
            dj_circuit_after_oracle.measure(i, i)

        dj_circuit += dj_circuit_after_oracle
        if not eval_mode:
            print("Circuit before the oracle\n")
            print(QuantumCircuit.draw(dj_circuit_before_oracle))
            print("Circuit after the oracle\n")
            print(QuantumCircuit.draw(dj_circuit_after_oracle))
            print(dj_circuit)
        return dj_circuit
```

Step 1:
Initialization

Step 2
Oracle

Interference

Measurements

Figure 7-27 Code for Bernstein-Vazirani Implementation.

The SecretNumberOracle class:

This class is responsible for preparing the oracle function for the algorithm.

```
from qiskit import QuantumCircuit

class SecretNumberOracle:
    @classmethod
    def create_secret_number_oracle(cls, random_binary, eval_mode: bool) ->
    QuantumCircuit:
        n = len(random_binary)
        secret_number_oracle = QuantumCircuit(len(random_binary) + 1,
        len(random_binary))

        # Use barrier as divider
        secret_number_oracle.barrier()

        # Controlled-NOT gates
        for qubit in range(len(random_binary)):
            if random_binary[qubit] == '1':
                secret_number_oracle.cx(qubit, n)

        secret_number_oracle.barrier()
        if not eval_mode:
            # Show oracle
            print("This is the oracle function, aka the black box. NORMALLY
            THIS WOULD BE HIDDEN!")
            print(secret_number_oracle)
        return secret_number_oracle
```

Figure 7-28 Code for Creating the Oracle for the Bernstein-Vazirani Algorithm.

Method definition – create_secret_number_oracle(random_binary, eval_mode):

Input: A random binary and a Boolean value determining the method operation mode.

Output: A QuantumCircuit object

Data Structures Used: We create a QuantumCircuit object that has one more qubit than the length of the secret random binary. In the for-loop, we append a CNOT operation with the control qubit being one of the qubits that are in a position corresponding to that where one exists in the secret random binary. And the target is the output qubit. We then append once again a barrier to create a conceptual box around the oracle and return the circuit in order to complete the algorithm.

Indicative results:

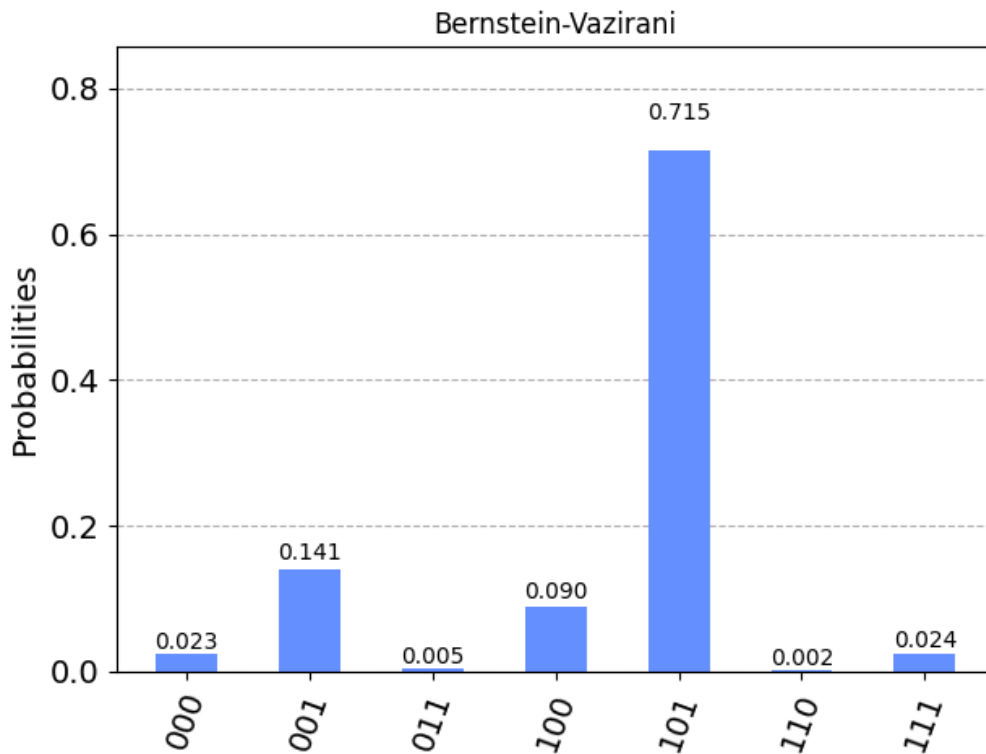


Figure 7-29 State Measurements for Bernstein-Vazirani.

```
***** FINAL * RESULTS *****  
  
Total counts are: {'000': 24, '001': 144, '011': 5, '100': 92, '101': 732, '110': 2, '111': 25}  
The time it took for the experiment to complete after validation was 6.006695747375488 seconds  
*****  
My guess after 3 is: 101  
*****
```

Figure 7-30 Execution Results for Bernstein-Vazirani.

The secret number generated is 101. We can see that for the quantum implementation the correct state was measured for many executions.

The classical solution as we can see takes three guesses to find 101, which is a 3-bit number.

7.7 Summary

It is important that we stress the fact that both Deutsch-Jozsa and Bernstein-Vazirani are algorithms that solve black-box problems. Shor's Algorithms is another, more famous algorithm, that uses this black-box logic in order to factor integers in polynomial time [12].

Another major Algorithm, not covered in this thesis, is the Grover Search Algorithm, which is used to solve unstructured search problems using a phenomenon called amplitude amplification [8], which is not the exact same as Phase-Kickback. We quickly realize that Quantum Algorithm Development is highly dependent on Quantum Systems behavior and not algorithmic thought alone.

However, we need to realize, that, like Classical Computation has its own families of Algorithms, so does Quantum. There is a certain approach to tackling each different problem, and a good quantum intuition, is when we are able to recognise the nature of a problem in order to use the correct algorithm, much like conventional computing.

This concludes QCLG Level 3 / Implementation chapter.

Chapter 8

QCLG Level 4 – Evaluation of Algorithms

8.1	Evaluation Process	92
8.2	Implementation.....	93
8.3	Deutsch-Jozsa Measurements	98
8.4	Bernstein-Vazirani Measurements	100
8.5	Additional Assisting Methods	101

8.1 Evaluation Process

Since quantum computing is not available as a local device in our home computer yet, we need to resort to cloud services. This means that we need to establish a connection with a real Quantum Computer and send our code to be executed in a real Quantum Machine and fetch back the results for evaluation. We also execute the classical solutions of the problems locally to be able to compare the execution times and discuss our findings. For this, we have implemented the Evaluation class which is responsible for executing both approaches multiple times in an automated manner and then displaying the results in a meaningful way.

Some keynotes:

- We compare execution times for the classical execution for 1 bit and the quantum execution for 1 qubit, then for 2 bits and 2 qubits, etc.
- Each classical and quantum experiment is executed 1024 times for each different input size. This is done for two reasons. 1024 times is the default number of executions for a quantum experiment and we can observe more useful results since the execution times for executing each experiment once are very small it would be very difficult to extract useful information.
- For Deutsch-Jozsa specifically:
 - We use a balanced oracle which we explained in section 7.3.2.2.
 - For the classical implementation, we again use the worst-case scenario as explained in section 7.2.2.
- For Bernstein-Vazirani specifically:
 - For the classical implementation, we used the sound classical method, which was explained here in section

After choosing the algorithm for evaluating the process is split as follows:

1. Classical Execution
2. Quantum Execution
3. Comparison

8.2 Implementation

We explain some of the important methods in the evaluation process.

```
from QCLG_lvl3.classical.random_binary import RandomBinary
from tools import Tools
import constants
from qiskit.providers import JobStatus

class Evaluation:

    @classmethod
    def evaluate(cls, algorithm):

        if algorithm == "0":
            cls.evaluate_deutsch_josza()
        elif algorithm == "1":
            cls.evaluate_bernstein_vazirani()
```

Figure 8-1 Code for Redirecting to Algorithm Evaluation.

Method definition – evaluate(algorithm):

Input: Algorithm key

Output: Call to the corresponding evaluating method.

Data Structures Used: We check with an `if, elif` statement whether we are going to evaluate Deutsch-Jozsa or Bernstein Vazirani.

```
@classmethod
def plot_results(cls, inputs: list, quantum: list, classical: list, accuracy:
list):
    import numpy as np
    import matplotlib.pyplot as plt
    X = np.arange(start=1, stop=len(inputs) + 1, step=1)
    plt.figure()
    plt.subplot(211)
    plt.title("Execution Times")
    plt.bar(X + 0.00, classical, color='b', width=0.25)
    plt.bar(X + 0.25, quantum, color='g', width=0.25)
    plt.ylabel("In seconds")
    plt.legend(labels=['Classical', 'Quantum'])
    plt.subplot(212)
    plt.title("Quantum Accuracy")
    plt.bar(inputs, accuracy, color='g')
    plt.ylabel("Percentage")
    plt.legend(labels=['Accuracy Percentage'])
    plt.show()
    return
```

Figure 8-2 Code for Plotting Evaluation Results.

Method definition - plot_results(inputs, quantum, classical, accuracy):

A simple method for constructing two separate plots.

Input: The list containing the range of different inputs, the quantum time results, the classical time results, and the accuracy of the quantum results.

Output: A figure containing two different plots. The first plot is for displaying the execution times for both the classical and quantum executions for all different input sizes and the second plot is for displaying the percentage of correct answers the quantum system achieves for different input sizes.

Data Structures Used: For this method, we need to take advantage of the matplotlib capabilities. We instantiate a figure object and then we will create two subplot objects.

With the `np.arange` method we create a NumPy array that we will use for our X-axis numbers on both subplots. We will then define two new subplots using `plt.subplot(211)` and `plt.subplot(212)`. The first two numbers in the parentheses mean that we are going to have two rows and one column of plots and then 1 for the first subplot and 2 for the second.

Deutsch-Josza Evaluation method

```
@classmethod
def evaluate_deutsch_josza(cls):
    print(constants.input_message_3)
    test_range = int(input())
    while test_range > 14 or test_range < 1:
        test_range = int(input("Enter a number between 1 and 14."))

    circuits = []
    n_bits = []
    quantum_execution_times = []
    classical_execution_times = []
    success_rates = []

    print("Evaluating Deutsch - Josza... This might take a while...")

    for number_of_bits in range(1, test_range + 1):
        n_bits.append(number_of_bits)
        circuits.append(Tools.prepare_dj(number_of_bits))
        total_time = 0.0
        for i in range(1024):
            classical_result = Tools.deutsch_josza_classical(number_of_bits)
            total_time = total_time + classical_result[1]
            classical_execution_times.append(total_time)
            completion_percentage = int((number_of_bits / test_range) * 100)
            print(f"{completion_percentage}% of classical executions done.")

        print("Now looking for the least busy backend...")
        least_busy_backend = Tools.find_least_busy_backend_from_open(test_range)

        print("Now waiting for the Quantum Batch Job to finish...\nThis will take a while...")
        quantum_results = Tools.run_batch_job(circuits, least_busy_backend)
        flag = False
        while not flag:
            for status in quantum_results.statuses():
                flag = True
                if status != JobStatus.DONE:
                    flag = False
        print("Quantum experiments finished.")
        print("Preparing plots...")
        count_jobs = 1
        for job in quantum_results.managed_jobs():
            quantum_execution_times.append(job.result().time_taken)
            counts_dict = job.result().get_counts()
            correct_counts = counts_dict.get('1' * count_jobs, 0)
            print(correct_counts)
            success_percentage = (correct_counts / 1024) * 100
            success_rates.append(success_percentage)
            count_jobs = count_jobs + 1
        cls.plot_results(n_bits, quantum_execution_times,
            classical_execution_times, success_rates)
    return
```

Figure 8-3 Code for Evaluating the Deutsch-Jozsa Algorithm.

Method Definition – `evaluate_deutsch_josza()`:

This method encapsulates the abstract process we are going to follow for evaluation. We will first generate all executions for the classical solutions of the algorithm, then proceed with the quantum solutions and finally plot the results to compare the two approaches effectively. This method is also greatly assisted by methods implemented in the `Tools` class which can be found at the end of this chapter.

Input: No input is required

Output: The figure containing the comparisons.

Data Structures Used: Using a while-loop we ensure that the input is a number between 1 and 14 since we are limited by the hardware available by IBM Q. We then define five lists. The `circuits` list will store the different quantum circuits that will be constructed for each different number of qubits. The `n_bits` list stores the number of qubits for each iteration. The `quantum_execution_times` list will store all the execution times for each different input size for the quantum circuits. The `classical_execution_times` list will store all the execution times for each different input size for the classical solutions. We then use a for-loop that will execute the classical solution for each size of the input range and append the execution time on the `classical_execution_times` list. We then display a completion percentage which is calculated easily by dividing the current iteration number, by the total number of iterations and multiplying by 100.

After we conclude with the classical executions, we need to execute the quantum solutions. This is done by finding the least busy backend capable of supporting the number of qubits required for the largest input we are going to evaluate and then submitting a batch job using the `run_batch_job()` from the `Tools` class and then using a while-loop we wait for all the circuits to finish executing in order to continue with the final assembly.

For each finished quantum experiment, we append the time to the `quantum_execution_times` list. We also retrieve the dictionary containing all the different measurements acquired from running each experiment 1024 times. We then search for the number of times the correct answer was measured. In the case of Deutsch-Jozsa and because we are running it with a balanced oracle, the correct answer is all input qubits measuring to 1. Hence, we are looking for the number of times the ('1' * amount of input qubits) is measured. If it was not measured, we assign the value 0. We then find the accuracy for the given input size by dividing by 1024, which is the time the experiment was executed for that input size and multiplying by 100. We append that percentage to the `success_rates` list and continue with the rest of the input sizes. With the conclusion of this while-loop, we call our plotting method and the `evaluate_deutsch_josza()` method is completed.

The method definition for Bernstein-Vazirani is almost identical with the only difference being how to find how many times the correct answer was measured by the quantum system. To do this, we need to search in the dictionary of measurements the key which is equal to the secret binary number and find how many times that secret number was measured. We do this by storing the different random binaries in a separate list called `random_binaries` and then checking to see the count of the experiment for 3 qubits where the measurement is equal to the 3-bit secret number that was generated beforehand. The code for the Bernstein-Vazirani method can be found on Appendix D.

8.3 Deutsch-Jozsa Measurements

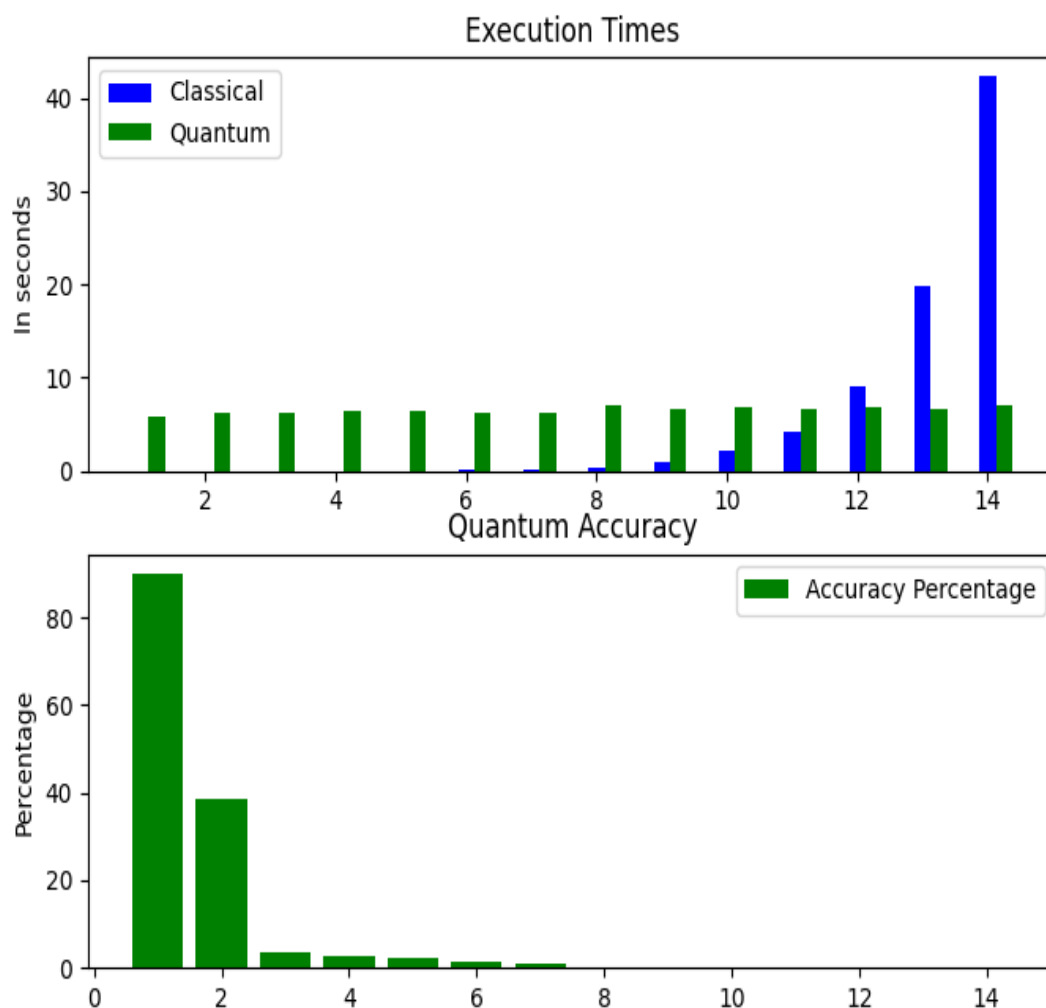


Figure 8-4 Deutsch-Jozsa Evaluation Measurements.

We can extract some very important information from the above figure. In the first subplot, we observe the execution times for input sizes of 1 to 14. We can see that for an input size of up to 11 bits/qubits, the classical computer is much faster. However, as we discussed in 7.1, Deutsch-Jozsa is an exponential problem. The exponential increase in time taken is obvious for input sizes 12, 13, and 14 with execution times increasing from 12 seconds to 24 seconds to 44 seconds respectively. At the same time, the execution time for the quantum system remains almost constant, a phenomenon that agrees with the fact that the quantum solution finds the answer with “one guess”.

However, we cannot yet celebrate. While the quantum computer shows a lot of promise, timewise, we can see that we run into a lot of problems as the input size increases, accuracy wise.

Due to physical errors in quantum systems, the amount of times the quantum system measures the correct state is greatly decreased as we advance into greater input sizes. We can maybe empirically recognize the correct answer for up to 6 qubits, but from that point on, the correct measurement was sometimes not even measured once.

Error correction is another major section of quantum computing as discussed in detail here [10] and we can see from our findings, why it is so important.

8.4 Bernstein-Vazirani Measurements

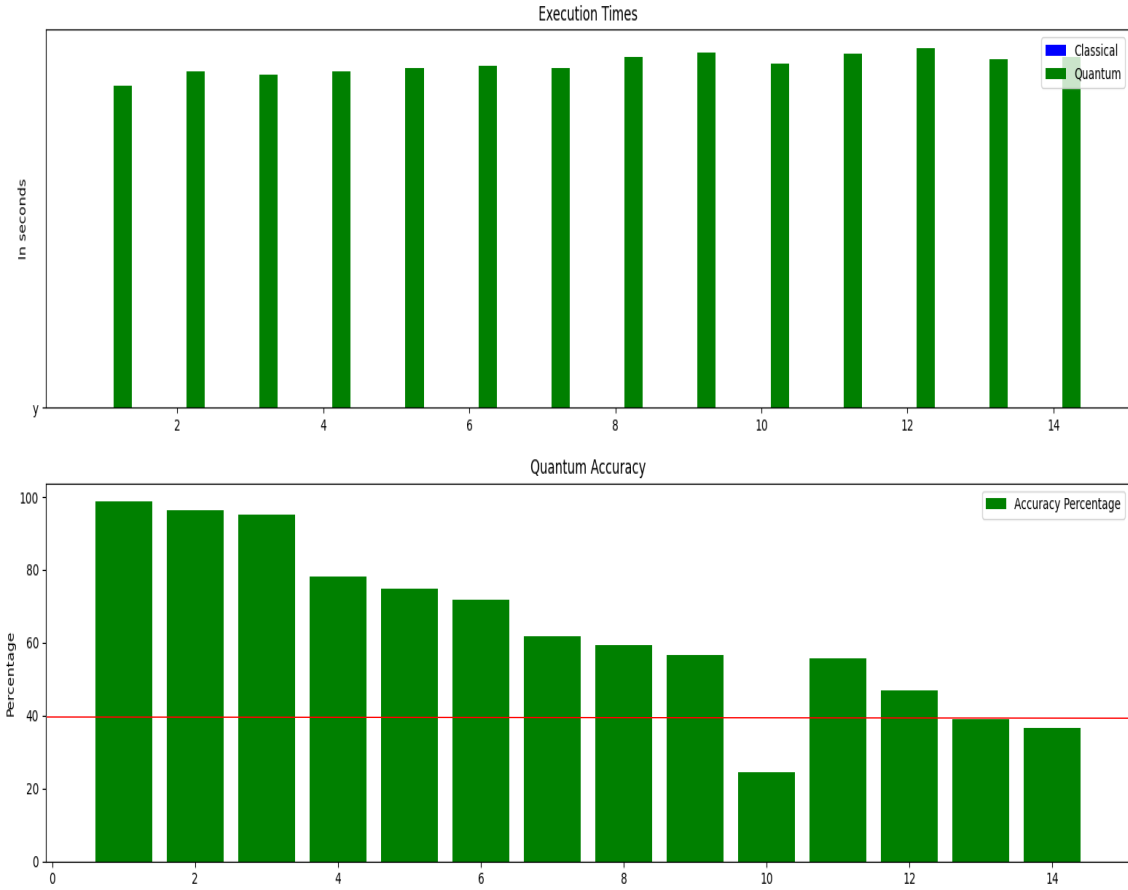


Figure 8-5 Bernstein-Vazirani Evaluation Measurements.

Continuing with the Bernstein-Vazirani algorithm, in the first subplot, we can see that the classical execution times are negligible even for plotting. That is because the Bernstein-Vazirani is solved in linear time in our classical solution and not in exponential time as Deutsch-Jozsa. The accuracy of Bernstein-Vazirani however, is much better. We can see that even for a 14-qubit secret number, the correct answer was measured almost 40% of the time which is substantially more accurate. We think that this is because of how the oracle function is implemented. Since Bernstein-Vazirani requires less C-NOT gates than Deutsch-Jozsa, there should be fewer physical errors, due to the smaller amount of gates.

8.5 Additional Assisting Methods

```
# Evaluation methods

@classmethod
def prepare_dj(cls, bits: int):
    bit_sequence = "0" * bits
    dj_circuit = DeutschJosza.deutsch_josza(bit_sequence, eval_mode=True)
    return dj_circuit

@classmethod
def prepare_bv(cls, random_binary: str):
    bj_circuit = BernsteinVazirani.bernstein_vazirani(random_binary,
eval_mode=True)
    return bj_circuit

@classmethod
def deutsch_josza_classical(cls, bits: int):
    return ClassicalXor.execute_classical_xor(bits=bits)

@classmethod
def bernstein_vazirani_classical(cls, bits: int):
    random_binary = RandomBinary.generate_random_binary_v2(bits)
    return BernsteinVaziraniClassical.guess_number(random_binary)

@classmethod
def run_batch_job(cls, circuits: list, least_busy_backend) -> ManagedJobSet:
    transpiled_circuits = transpile(circuits, backend=least_busy_backend)
    # Use Job Manager to break the circuits into multiple jobs.
    job_manager = IBMQJobManager()
    job_set_eval = job_manager.run(transpiled_circuits,
backend=least_busy_backend, name='eval',
max_experiments_per_job=1) #
max_experiments_per_job = 1 very important to get
# individual execution times
    return job_set_eval
```

Figure 8-6 Code for Supplementary Methods of Evaluation.

The purpose of these methods is to use pre-existing methods from the Tools class but call them differently in order to automate some inputs and speed up the evaluation process. The methods `prepare_dj()`, `prepare_bv()`, `deutsch_josza_classical()`, are almost exact replicas with the slight difference that they run on eval mode, which just means that the printing messages are dismissed.

The `bernstein_vazirani_classical()` method differs only in the way of generating the random binary, which is by randomly generating a random binary number of length 'bits' directly and not by asking the user. The implementation for

`generate_random_binary_v2(bits)` can be found in the appendix, along with the rest of the code.

Method definition – `run_batch_job(circuits, least_busy_backend)`:

Input: A list of quantum circuits and the list busy backend capable of executing the largest circuit in the given list.

Output: A `ManagedJobSet` object that holds all the managed jobs/experiments along with all their information. Further information about the `ManagedJobSet` object can be found here [32].

Data Structures Used: We first need to use the `transpile` method which will transpile the experiments into a manageable batch job for the backend to execute in parallel. We then run this batch job which returns each finished experiment asynchronously. This means that it does not wait for all experiments to finish to return the results. This is why we created a while-loop which waited for all experiments to finish in the `evaluate_deutsch_josza()` and `evaluate_bernstein_vazirani()` methods previously. The `max_experiments_per_job` was a very important parameter for successfully evaluating the algorithms. By limiting the number of experiments per job to 1, we can have discrete execution times for each different input size. When we tried to run the batch job with the default settings, this was not possible, and we could not extract discrete execution times.

This concludes the QCLG Level 4 / Evaluation chapter.

Chapter 9

Conclusions

9.1	Conclusions	103
9.2	Limitations	105
9.3	Future Work	106

9.1 Conclusions

Concluding this thesis, we first need to say that quantum computing is a marvelous attempt of humankind to push the limits of knowledge to unprecedented new grounds and we are thrilled that we could be a part of the Quantum revolution that is afoot. Most importantly, we completed our goals and can answer the four research questions that we beset at the beginning of this thesis in section 1.2.

Firstly, we completed our goals with the development of the “Quantum Computing Learning Gate”. Through this platform, we managed to demonstrate how a software developer can take advantage of the abundant resources available and initiate their own quantum journey. We have demonstrated the most useful examples that helped us proceed with more advanced concepts on the matter. We have developed a quantum game that capitalizes on the capabilities of a quantum system in order to provide a joyful and interactive experience whilst keeping a consistent learning environment. We have completed a case study of algorithms on quantum and classical systems to showcase an additional point of view of understanding in the emerging scene of quantum research. We have implemented a practical workflow for evaluating algorithms using IBM Q’s systems. Finally, we have gathered useful material that can be a jumping-off point for any aspiring quantum developers to initiate their journey and contribute in their own way.

Secondly, we managed to find answers to our three questions.

The first question was about who should be interested in quantum computing. Should it be reserved for researchers? Is it a subject that should be made compulsory in all universities across the world? The answer to both of these questions is no. Quantum computing is moving forward through a joint effort of a passionate community that is its borders are way beyond the scientific community. We are in the age of quantum computing and significant contributions have been made through the power of open-source projects. However, it is our personal belief that it should not be a compulsory course in computer science degrees. It differs in many ways from traditional computation, and its impact, although significant is not yet at the stage of refinement that is required to be taught to undergraduate students. We would advocate the notion of it being available as an elective course to more undergraduate or post-graduate curriculums since in 10 years it could change the ways we approach machine learning, optimization problems, and security.

The second question was is regarding the choice between the available resources online for initiating quantum endeavors. The answer to this is not absolute. It largely depends on the goals of the individual or organization. We, as students have chosen Qiskit since it is a framework that puts education high in its list of priorities by offering a variety of tutorials through the Qiskit textbook, online videos, and even has a community on Slack for any Qiskit related issue. For professionals or organizations that are interested in quantum solutions, the decision is a lot tougher. Since quantum computing is offered through cloud services there are a lot of factors to consider. If they have worked with Amazon Web Services for conventional solutions previously, they might find it easier to make a transition to Amazon Braket. If, however, we are talking about a startup that wants to try out the new technology it might be best to explore other options that may offer cheaper rates.

The third and final question that we can answer is whether pursuing a career specialized in quantum computing is a viable career path. This question was the main inducement for completing this thesis. Quantum computing is a fascinating subject but when it comes up in conversations, even in professional software environments, especially in Cyprus, it is thought of as a strictly theoretical section of Computer Science and deemed as a purely academic career path. In our opinion, this is not the case. Although in its early industrial stages, quantum computing could offer solutions to companies, and

why not, Cypriot companies. Maybe not in the next 3 to 5 years but definitely into the foreseeable future, a quantum computing specialization could be as valuable as a software engineering or cybersecurity specialization. The cloud is much closer than we realize and with it the quantum or hybrid solutions available. We are optimistic and for sure keeping the opportunity of working as a quantum developer high in the list of career paths.

9.2 Limitations

During the completion of this thesis, we encountered many obstacles. Our first obstacle was the absence of previous friction with quantum computing. Quantum computing is not a course that is offered in the standard curriculum or any of the elective courses at the University of Cyprus. Therefore, we had to start with very small steps which were time-consuming.

Another limitation was the quantum physics aspect of quantum computing. Although there have been many leaps with quantum frameworks, there is still a large benefit in having previous experience with complicated concepts such as the superposition, entanglement, amplitudes, and all other phenomena found in the world of quantum mechanics.

The third limitation had to do with the resources that were available to us and were twofold. IBM Q generously offers quantum systems for free but for this thesis, it was difficult to extract the data that we hoped for. Quantum systems are still error-prone, and while we were able to plan legitimate workflows for implementing and evaluate algorithms, we still could not achieve an advantage with the quantum systems because of the very low accuracy. The other issue regarding resources was the queue time required for executing experiments. We started working on this thesis in early January of 2020 and the traffic on the IBM Q systems was much lower than what it is today. At times, we needed to wait for hours to successfully execute jobs on the desired backends. This is of course a good thing as well because it means that more and more people are interested in quantum research, but it certainly slowed down the progress of experimentation.

An additional limitation was getting familiar with the Qiskit framework in a short amount of time. Qiskit is a very rich framework that is open-source and is three years into development. Its public GitHub repository has over 100 contributors [33]. A framework

of the magnitude demands thorough research through documentation study and multiple trial and error experimentations. We continuously discovered more and more features available that were vital to the development of a more complete version of QCLG. We were only able to experience a glimpse of what Qiskit can offer through its different elements and hundreds of Python classes. This brings us neatly into our next section which is our aspirations for QCLG.

9.3 Future Work

Quantum computing, due to its difficulty, and studying demands, is challenging. This also leaves a lot of space for improvement in our current work. Concerning the QCLG there is a lot of ground to be covered. The four levels could be enhanced by additional experiments, additional algorithms for analysis, and more evaluation options. Also, there can be a fifth level, which will contain real-world applications of algorithms. Furthermore, “The Exciting Game” is a very promising project that could evolve into a standalone application, much like Quantum Tetris [35]. Our idea is that the current code can be modified to work as a backend-side application, and we can deploy a front-end graphical interface that takes the game to the next level in terms of interaction and playability. Another idea is the creation of a separate domain from GitHub that presents key points of QCLG and references material relative to quantum computing in order to host a portal capable of generating traffic. Finally, we believe that it would be a good idea to explore other quantum resources besides IBM and Qiskit to expand our horizons.

Bibliography

- [1] Héctor Abraham et al. (2019). Qiskit: An Open-source Framework for Quantum Computing.
- [2] Bernhardt, C. (2019). Quantum Computing for Everyone / Chris Bernhardt. London: The MIT Press.
- [3] Bernstein Ethan, Vazirani Umesh (1993).“Quantum complexity theory”, Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing (STOC '93), pp. 11–20, DOI:10.1145/167088.167097.
- [4] Andrew W. Cross, Lev S. Bishop, John A. Smolin, & Jay M. Gambetta. (2017). Open Quantum Assembly Language.
- [5] David Deutsch and Richard Jozsa (1992). “Rapid solutions of problems by quantum computation”. Proceedings of the Royal Society of London A. 439: 553. Bibcode:1992RSPSA.439..553D. DOI:10.1098/rspa.1992.0167.
- [6] Chris Edwards. 2019. Questioning quantum. Commun. ACM 62, 5 (May 2019), 15–17. DOI:<https://doi.org/10.1145/3317673>
- [7] Feynman, R.P. Simulating physics with computers. Int J Theor Phys 21, 467–488 (1982). <https://doi.org/10.1007/BF02650179>
- [8] Grover, L. (1996). A Fast Quantum Mechanical Algorithm for Database Search. In Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing (pp. 212–219). Association for Computing Machinery.

- [9] Chuan-Zheng Lee. (2017). Logic Gates - ENGR 40M lecture notes. Stanford University. Available: <https://web.stanford.edu/class/archive/engr/engr40m.1178/slides/logicgates.pdf>
- [10] Don Monroe. 2019. Closing in on quantum error correction. *Commun. ACM* 62, 10 (October 2019), 11–13. DOI:<https://doi.org/10.1145/3355371>
- [11] Julia E. Rice, Tanvi P. Gujarati, Tyler Y. Takeshita, Joe Latone, Mario Motta, Andreas Hintennach, & Jeannette M. Garcia. (2020). Quantum Chemistry Simulations of Dominant Products in Lithium-Sulfur Batteries.
- [12] Shor, P. (1999). Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer *SIAM Review*, 41(2), 303-332.
- [13] Moshe Y. Vardi. 2019. Quantum hype and quantum skepticism. *Commun. ACM* 62, 5 (May 2019), 7. DOI:<https://doi.org/10.1145/3322092>
- [14] Michael L. Wall, Arghavan Safavi-Naini, and Martin Gärtnner. 2016. Many-body quantum mechanics: too big to fail? *XRDS* 23, 1 (Fall 2016), 25–29. DOI:<https://doi.org/10.1145/2983537>
- [15] James R. Wootton. 2020. Procedural generation using quantum computation. In *International Conference on the Foundations of Digital Games (FDG '20)*. Association for Computing Machinery, New York, NY, USA, Article 98, 1–8. DOI:<https://doi.org/10.1145/3402942.3409600>
- [16] Amazon Braket. Available: <https://aws.amazon.com/braket/>
- [17] Bloch Sphere Representation Picture. Available: <https://qiskit.org/textbook/ch-states/images/bloch.png>

- [18] Conda Environments Documentation. Available:
<https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html>
- [19] Data Management Systems Laboratory GitHub Page. Available:
<https://github.com/dmsl>
- [20] Data Management Systems Laboratory Website. Available:
<https://dmsl.cs.ucy.ac.cy/>
- [21] D-Wave Systems. Available: <https://www.dwavesys.com/our-company/meet-d-wave>
- [22] Fugaku Supercomputer. Available:
<https://www.fujitsu.com/global/about/innovation/fugaku/specifications/>
- [23] GitHub Documentation. Available: <https://docs.github.com/en>
- [24] IBM Circuit Composer. Available: <https://www.ibm.com/quantum-computing/experience/>
- [25] IBM Quantum. Available: <https://www.ibm.com/quantum-computing/>
- [26] IBM Quantum Documentation. Available: <https://quantum-computing.ibm.com/docs/>
- [27] Leap 2 YouTube video from D-Wave Systems. Available:
https://www.youtube.com/watch?v=4-ctzeBQkTc&ab_channel=D-WaveSystems

- [28] matplotlib Documentation. Available: <https://matplotlib.org/3.3.3/contents.html>
- [29] NumPy Documentation. Available: <https://numpy.org/doc/stable/>
- [30] PyCharm Documentation. Available: <https://www.jetbrains.com/pycharm/documentation/>
- [31] Python 3.7 Documentation. Available: <https://docs.python.org/3.7/>
- [32] Qiskit Documentation. Available: <https://qiskit.org/documentation/>
- [33] Qiskit Repository on GitHub. Available: <https://github.com/Qiskit/qiskit>
- [34] Qiskit Textbook. Available: <https://qiskit.org/textbook/preface.html>
- [35] Quantum Tetris. Available: <http://quantumtetris.com/>
- [36] Qubit Representation Picture. Available: <https://qoqms.phys.strath.ac.uk/figures/qubit.png>
- [37] Ubuntu Basic Commands. Available: <https://ubuntu.com/tutorials/command-line-for-beginners#1-overview>
- [38] Volkswagen Quantum Routing. Available: <https://www.volkswagen-newsroom.com/en/press-releases/volkswagen-optimizes-traffic-flow-with-quantum-computers-5507>

Appendix A

QCLG Level 1 Code

```
import constants
from QCLG_lv11.hello_quantum_world import HelloWorld
from QCLG_lv11.interference import Interference
from QCLG_lv11.single_qubit_superposition import SingleQubitSuperposition
from QCLG_lv11.three_qubits_superposition import ThreeQubitSuperposition

class SimpleExperimentsManager:
    @classmethod
    def showcase(cls):
        print("Hi, these are the available experiments")

        for i in range(len(constants.experiments)):
            print(f"{i}. {constants.experiments[i]}")

        choice = input(f"Which experiment would you like to try from 0 to {len(constants.experiments)-1}? ")

        while choice not in constants.acceptable_experiment_inputs:
            choice = input(f"Which experiment would you like to try from 0 to {len(constants.experiments) - 1}? ")
            if choice == "0":
                HelloWorld.run()
            elif choice == "1":
                SingleQubitSuperposition.run()
            elif choice == "2":
                ThreeQubitSuperposition.run()
            elif choice == "3":
                Interference.run()
```

```
from qiskit import execute, Aer, QuantumCircuit

class Interference:
    @classmethod
    def run(cls):
        # Use Aer's qasm_simulator
        simulator = Aer.get_backend('qasm_simulator')
        # Create a Quantum Circuit acting on the q register
        circuit = QuantumCircuit(1, 1)
        # Add a H gate on qubit 0
        circuit.h(0)
        # Add another H gate to qubit 0
        circuit.h(0)
        # Map the quantum measurement to the classical bits
        circuit.measure(0, 0)
        # Execute the circuit on the qasm simulator
        job = execute(circuit, simulator, shots=1000)
        # Grab results from the job
        result = job.result()
        # Returns counts
        counts = result.get_counts(circuit)
        print("Results for the Quantum Interference experiment.")
        print("\nTotal count for 0 and 1 are:", counts)
        print(circuit)
```

```

from qiskit import IBMQ
from qiskit import (
    QuantumCircuit,
    execute,
    Aer)

class HelloWorld:
    @classmethod
    def run(cls):
        with open('./credentials/token', 'r') as file:
            token = file.read()
        IBMQ.save_account(token, overwrite=True)
        # Use Aer's qasm simulator
        simulator = Aer.get_backend('qasm_simulator')
        # Create a Quantum Circuit acting on the q register
        circuit = QuantumCircuit(2, 2)
        # Add a H gate on qubit 0
        circuit.h(0)
        # Add a CX (CNOT) gate on control qubit 0 and target qubit 1
        circuit.cx(0, 1)
        # Map the quantum measurement to the classical bits
        circuit.measure([0, 1], [0, 1])
        # Execute the circuit on the qasm simulator
        job = execute(circuit, simulator, shots=1024)
        # Grab results from the job
        result = job.result()
        # Returns counts
        counts = result.get_counts(circuit)
        print("Results for the Bell State experiment.")
        print("\nTotal count for 00 and 11 are:", counts)
        # Draw the circuit
        print(circuit)

```

```

from qiskit import execute, Aer, QuantumCircuit, IBMQ

class SingleQubitSuperposition:
    @classmethod
    def run(cls):
        # Use Aer's qasm simulator
        simulator = Aer.get_backend('qasm_simulator')
        # Create a Quantum Circuit acting on the q register
        circuit = QuantumCircuit(1, 1)
        # Add a H gate on qubit 0
        circuit.h(0)
        # Map the quantum measurement to the classical bits
        circuit.measure(0, 0)
        # Execute the circuit on the qasm simulator
        job = execute(circuit, simulator, shots=1000)
        # Grab results from the job
        result = job.result()
        # Returns counts
        counts = result.get_counts(circuit)
        print("\nTotal count for 0 and 1 are:", counts)
        provider = IBMQ.load_account()
        backend = provider.backends.ibmq_valencia
        # Execute the circuit on a real device
        job = execute(circuit, backend=backend, shots=1000)
        # Grab results from the job
        result = job.result()
        # Returns counts
        counts = result.get_counts(circuit)
        print("\nTotal count for 0 and 1 are:", counts)
        # Draw the circuit
        print(circuit)

```

```

from qiskit import execute, Aer, QuantumCircuit, IBMQ

class ThreeQubitSuperposition:
    @classmethod
    def run(cls):
        # Use Aer's qasm simulator
        simulator = Aer.get_backend('qasm_simulator')
        # Create a Quantum Circuit acting on the q register
        circuit = QuantumCircuit(3, 3)
        # Add a H gate on qubit 0
        circuit.h(0)
        circuit.h(1)
        circuit.h(2)
        # Map the quantum measurement to the classical bits
        for i in range(3):
            circuit.measure(i, i)
        # Execute the circuit on the qasm simulator
        job = execute(circuit, simulator, shots=1024)
        # Grab results from the job
        result = job.result()
        # Returns counts
        counts = result.get_counts(circuit)
        print("\nTotal count all possible states are:", counts)

        provider = IBMQ.load_account()
        backend = provider.backends.ibmq_valencia
        # Execute the circuit on a real device
        job = execute(circuit, backend=backend, shots=1024)
        # Grab results from the job
        result = job.result()
        # Returns counts
        counts = result.get_counts(circuit)
        print("\nTotal count for all possible states are:", counts)
        # Draw the circuit
        print(circuit)

```

All the above are the entirety of the code for QCLG Level 1. Here there are implementations for experimenting with superposition, interference and entanglement. All these experiments are managed by a standalone QCLG Level 1 manager class.

Appendix B

QCLG Level 2 Code

```
from random import randint

import numpy as np
from qiskit import execute, BasicAer
from qiskit.circuit.quantumcircuit import QuantumCircuit

class Game:

    @classmethod
    def run(cls, circuit: QuantumCircuit):
        # use local simulator
        backend = BasicAer.get_backend('qasm_simulator')
        results = execute(circuit, backend=backend, shots=1024).result()
        answer = results.get_counts()
        max_value = 0
        max_key = ""
        for key, value in answer.items():
            if value > max_value:
                max_value = value
                max_key = key
        print(answer)
        if max_key == "00":
            print("Both players stay grounded :)")
            return 0
        elif max_key == "01":
            print("Player 1 is excited!")
            return 1
        elif max_key == "10":
            print("Player 2 is excited!")
            return 2
        elif max_key == "11":
            print("Both players are excited!")
            return 3
        return

    @classmethod
    def place_gate(cls, player, field, qubit):
        card = player.pop()
        print(f"now inserting card {card} from player {qubit + 1}")
        if card == "H":
            field.h(qubit)
        elif card == "X":
            field.x(qubit)
        elif card == "RX":
            field.rx(np.pi / 2, qubit)
        elif card == "CX":
            if qubit == 0:
                field.cx(qubit, qubit + 1)
            else:
                field.cx(qubit, qubit - 1)
        return
```

```

@classmethod
def create_playing_field(cls, player1: list, player2: list) ->
QuantumCircuit:
    field = QuantumCircuit(2, 2)
    player1.reverse()
    player2.reverse()
    while len(player1) > 0:
        cls.place_gate(player1, field, 0)
    while len(player2) > 0:
        cls.place_gate(player2, field, 1)
    field.measure(0, 0)
    field.measure(1, 1)
    return field

@classmethod
def generate_deck(cls) -> list:
    cards = ["H", "H", "X", "X", "CX", "RX", "RX"]
    deck = []
    for j in range(4):
        for i in range(len(cards)):
            deck.append(cards[i])
    return deck

@classmethod
def shuffle_deck(cls, deck: list):
    for i in range(len(deck) * 5):
        j = randint(0, len(deck) - 1)
        k = randint(0, len(deck) - 1)
        temp = deck[j]
        deck[j] = deck[k]
        deck[k] = temp
    return

@classmethod
def deal_starting_hands(cls, player1: list, player2: list, deck: list):
    for i in range(0, 4, 2):
        player1.append(deck.pop())
        player2.append(deck.pop())
    return

@classmethod
def draw_from_deck(cls, deck: list) -> str:
    return deck.pop()

@classmethod
def replace(cls, replacement_choice, card, player):
    player.remove(replacement_choice)
    player.append(card)
    return

```

```

@classmethod
def fix_hand(cls, player: list) -> list:
    new_hand = []
    print("Your current hand is setup like this:")
    print(player)
    i = 0
    while len(player) > 0:
        replacement_choice = input(f"Choose one of your cards to be on
position {i} :")
        while replacement_choice not in player:
            replacement_choice = input(f"Choose one of your cards to be on
position {i} :")
        new_hand.insert(len(new_hand), replacement_choice)
        player.remove(replacement_choice)
        print("Cards remaining in previous hands")
        print(player)
        i = i + 1

    print("New hand")
    print(new_hand)
    print()
    return new_hand

```

```

@classmethod
def play_the_exciting_game(cls):
    deck = cls.generate_deck()
    cls.shuffle_deck(deck)
    player1 = []
    player1_wins = 0
    player2 = []
    player2_wins = 0
    rounds = int(input("Enter number of rounds: "))

    print("The exciting game begins!")
    current_round = 0
    while current_round <= rounds:
        countdown = 4
        print("#" * (current_round + 1), end="")
        print(f"ROUND {current_round}", end="")
        print("#" * (current_round + 1))
        print()
        cls.deal_starting_hands(player1, player2, deck)
        while countdown != 0:
            print("\nPlayer 1")
            print(player1)
            cls.draw(player1, deck)
            print("\nPlayer 2")
            print(player2)
            cls.draw(player2, deck)
            countdown = countdown - 1
            print(f"{countdown} dealings remain before the players have to
see who's Excited!")
            if countdown == 0:
                print("Next turn is going to be Exciting!!!")

        print("Both players get to fix their hands in the order they
desire!")
        player1 = cls.fix_hand(player1)
        player2 = cls.fix_hand(player2)
        playing_field = cls.create_playing_field(player1, player2)
        print(playing_field.draw())
        round_result = cls.run(playing_field)
        if round_result == "1":
            player1_wins = player1_wins + 1
        elif round_result == "2":
            player2_wins = player2_wins + 1
        current_round = current_round + 1

    if player1_wins > player2_wins:
        print("PLAYER ONE WAS MOST EXCITED!")
    elif player2_wins > player1_wins:
        print("PLAYER TWO WAS MOST EXCITED!")
    else:
        print("PLAYERS WERE EQUALLY EXCITED!")

```

This code is for QCLG Level 2 and is the entire implementation of “The Exciting Game”. There is a plethora of helper functions and the driver method `play_the_exciting_game()` which runs the game.

Appendix C

QCLG Level 3 Code

Classical Directory

```
class BernsteinVaziraniClassical:

    @classmethod
    def guess_number(cls, secret_binary):
        mask = 1
        guess = ""
        attempts = 0
        for bit in secret_binary:
            hit = int(bit) & mask
            guess += str(hit)
            attempts += 1
        return f"My guess After {attempts} attempts is:\n{guess}."
```

```
import random

class RandomBinary:

    @classmethod
    def generate_random_binary(cls, limit: int) -> str:
        rand = int(random.uniform(0, limit))
        print(f"random binary in decimal:{rand}")
        random_bin = bin(rand)[2:]
        print(random_bin)
        return random_bin

    @classmethod
    def generate_random_binary_v2(cls, binary_length: int) -> str:
        random_bin = ""
        for i in range(binary_length):
            rand = int(random.uniform(0, 1))
            if rand >= 0.5:
                random_bin = random_bin + "1"
            else:
                random_bin = random_bin + "0"
        return random_bin
```



```

from datetime import datetime

from QCLG_lvl3.classical.BitCombinations import BitCombinations

class ClassicalXor:

    @classmethod
    def _super_secret_black_box_function_f(cls, list_of_inputs: list) -> str:
        output_bit = 0
        output_zero = False
        output_one = False
        counts = 0
        for input_bits in list_of_inputs:
            for bit in input_bits:
                output_bit = output_bit ^ int(bit)

            if output_bit == 1:
                output_one = True
            else:
                output_zero = True
            counts = counts + 1
            output_bit = 0
            if output_one and output_zero:
                return f'Balanced After {counts} checks.'
            if counts > (len(list_of_inputs) / 2):
                return f'Constant After {counts} checks.'
        return f'After {counts} checks.'

    @classmethod
    def execute_classical_xor(cls, bits) -> list:
        start = datetime.now()
        inputs =
        BitCombinations.produce_worse_scenario(BitCombinations.combinations(bits))
        end = datetime.now()
        elapsed = end - start
        time_to_generate_worst_input = elapsed.total_seconds()
        start = datetime.now()
        function_nature = cls._super_secret_black_box_function_f(inputs)
        end = datetime.now()
        elapsed = end - start
        time_str = elapsed.total_seconds()
        final = [time_to_generate_worst_input, time_str, bits,
        function_nature]
        return final

```

```

class BitCombinations:

    @classmethod
    def split(cls, word: str) -> list:
        return [char for char in word]

    @classmethod
    def get_child(cls, parent: list) -> list:
        return parent.pop()

    @classmethod
    def count_ones(cls, binary_list: list) -> int:
        count = 0
        for bit in binary_list:
            if bit == "1":
                count = count + 1
        return count

    @classmethod
    def produce_worse_scenario(cls, combos: list) -> list:
        worse_input_list = []
        even = []
        odd = []
        for i in range(len(combos)):
            if i % 2 == 1: # list with even amount of zeroes and ones
                even.extend(combos[i])
            else:
                odd.extend(combos[i])
        worse_input_list.extend(even)
        worse_input_list.extend(odd)
        return worse_input_list

```

```

@classmethod
def combinations(cls, bits: int) -> list:
    final_layer = []
    for n_bits in range(1, bits):
        parent = []
        for layer in range(bits - 1):
            new_layer = []
            if layer == 0:
                parent = []
                b0 = ["0"]
                b1 = ["1"]
                parent.append(b0)
                parent.append(b1)
            while len(parent) > 0:
                child = cls.get_child(parent)
                new_child1 = ["1"]
                new_child0 = ["0"]
                remaining_ones_for_child = n_bits - cls.count_ones(child)
                remaining_length = bits - len(child)
                if remaining_ones_for_child == remaining_length:
                    new_child1.extend(child)
                    new_layer.append(new_child1)
                elif 0 < remaining_ones_for_child < remaining_length:
                    new_child1.extend(child)
                    new_layer.append(new_child1)
                    new_child0.extend(child)
                    new_layer.append(new_child0)
                elif remaining_ones_for_child == 0:
                    new_child0.extend(child)
                    new_layer.append(new_child0)
            parent = new_layer
        final_layer.append(parent)
    result = []
    all_zeros = [[cls.split("0" * bits)]]
    all_ones = [[cls.split("1" * bits)]]
    result.extend(all_zeros)
    result.extend(final_layer)
    result.extend(all_ones)
    return result

```

```

from qiskit import QuantumCircuit

class CnotOracle:
    @classmethod
    def create_cnot_oracle(cls, input_string, input_length, eval_mode: bool)
-> QuantumCircuit:
        balanced_oracle = QuantumCircuit(input_length + 1)
        # Place X-gates
        for qubit in range(len(input_string)):
            if input_string[qubit] == '1':
                balanced_oracle.x(qubit)

        # Use barrier as divider
        balanced_oracle.barrier()

        # Controlled-NOT gates
        for qubit in range(input_length):
            balanced_oracle.cx(qubit, input_length)

        balanced_oracle.barrier()

        # Place X-gates
        for qubit in range(len(input_string)):
            if input_string[qubit] == '1':
                balanced_oracle.x(qubit)
        if not eval_mode:
            # Show oracle
            print("This is the oracle function, aka the black box. NORMALLY
THIS WOULD BE HIDDEN!")
            print(balanced_oracle)
        return balanced_oracle

```

```

from qiskit import QuantumCircuit

class SecretNumberOracle:
    @classmethod
    def create_secret_number_oracle(cls, random_binary, eval_mode: bool) ->
QuantumCircuit:
        n = len(random_binary)
        secret_number_oracle = QuantumCircuit(len(random_binary) + 1,
len(random_binary))
        # Use barrier as divider
        secret_number_oracle.barrier()
        # Controlled-NOT gates
        for qubit in range(len(random_binary)):
            if random_binary[qubit] == '1':
                secret_number_oracle.cx(qubit, n)
        secret_number_oracle.barrier()
        if not eval_mode:
            # Show oracle
            print("This is the oracle function, aka the black box. NORMALLY
THIS WOULD BE HIDDEN!")
            print(secret_number_oracle)
        return secret_number_oracle

```

```

import constants
from tools import Tools

def print_answers(answer_of_simulation, answer_of_real, least_busy_backend,
classical_answer, algorithm):
    print("***** FINAL * RESULTS *****")
    if answer_of_simulation is not None:
        Tools.print_simul(answer_of_simulation, algorithm)
        print("*****")
    if answer_of_real is not None:
        Tools.print_real(answer_of_real, least_busy_backend, algorithm)
        print("*****")
    if classical_answer is not None:
        Tools.print_classical_answer(classical_answer, algorithm)
        print("*****")
    return

class AlgorithmsManager:
    @classmethod
    def showcase(cls):
        answer_of_simulation = None
        answer_of_real = None
        least_busy_backend = None
        classical_answer = None

        algorithm = input(constants.input_message_1)
        while algorithm not in constants.acceptable_algorithm_inputs:
            algorithm = input(constants.input_message_1)

        execution = input(constants.input_message_2)
        while execution not in constants.acceptable_execution_inputs:
            execution = input(constants.input_message_2)

        if execution == "0":
            classical_answer = Tools.execute_classically(algorithm)
        elif execution == "1":
            answer_of_simulation = Tools.execute_in_simulator(algorithm)
        elif execution == "2":
            answer_of_real = Tools.execute_in_real_device(algorithm)
        elif execution == "3":
            combined = Tools.execute_both(algorithm)
            classical_answer = combined[0]
            answer_of_real = combined[1]

        print_answers(answer_of_simulation, answer_of_real,
least_busy_backend, classical_answer, algorithm)

```

The manager for handling algorithm execution choices. The different choices are to run the algorithm classically, on a real quantum device, on the IBM simulator or both on a quantum real device and classically on the local device.

```

from qiskit import QuantumCircuit

from QCLG_lvl3.oracles.cnot_oracle import CnotOracle

class DeutschJozsa:

    @classmethod
    def deutsch_jozsa(cls, bit_string: str, eval_mode: bool) ->
QuantumCircuit:
        n = len(bit_string)

        dj_circuit = QuantumCircuit(n + 1, n)
        # Apply H-gates
        for qubit in range(n):
            dj_circuit.h(qubit)

        # Put output qubit in state |->
        dj_circuit.x(n)
        dj_circuit.h(n)

        # Construct balanced oracle
        balanced_oracle = CnotOracle.create_cnot_oracle(bit_string, n,
eval_mode)

        # Add oracle
        dj_circuit += balanced_oracle

        # Repeat H-gates
        for qubit in range(n):
            dj_circuit.h(qubit)
        dj_circuit.barrier()

        # Measure
        for i in range(n):
            dj_circuit.measure(i, i)
        if not eval_mode:
            print(dj_circuit)

        # return circuit
        return dj_circuit

```

```

from qiskit import QuantumCircuit

from QCLG_lvl3.oracles.secret_number_oracle import SecretNumberOracle

class BernsteinVazirani:

    @classmethod
    def bernstein_vazirani(cls, random_binary, eval_mode: bool) ->
QuantumCircuit:
        # Construct secret number oracle

        secret_number_oracle =
SecretNumberOracle.create_secret_number_oracle(random_binary=random_binary,
eval_mode=eval_mode)
        num_of_qubits = secret_number_oracle.num_qubits

        # Construct circuit according to the length of the number

        dj_circuit = QuantumCircuit(num_of_qubits, num_of_qubits - 1)
        dj_circuit_before_oracle = QuantumCircuit(num_of_qubits,
num_of_qubits - 1)

        # Apply H-gates
        for qubit in range(num_of_qubits - 1):
            dj_circuit_before_oracle.h(qubit)

        # Put output qubit in state |->
        dj_circuit_before_oracle.x(num_of_qubits - 1)
        dj_circuit_before_oracle.h(num_of_qubits - 1)

        dj_circuit += dj_circuit_before_oracle

        # Add oracle
        dj_circuit += secret_number_oracle

        dj_circuit_after_oracle = QuantumCircuit(num_of_qubits, num_of_qubits
- 1)

        # Repeat H-gates
        for qubit in range(num_of_qubits - 1):
            dj_circuit_after_oracle.h(qubit)
        dj_circuit_after_oracle.barrier()

        # Measure
        for i in range(num_of_qubits - 1):
            dj_circuit_after_oracle.measure(i, i)

        dj_circuit += dj_circuit_after_oracle
        if not eval_mode:
            print("Circuit before the oracle\n")
            print(QuantumCircuit.draw(dj_circuit_before_oracle))
            print("Circuit after the oracle\n")
            print(QuantumCircuit.draw(dj_circuit_after_oracle))
            print(dj_circuit)
        return dj_circuit

```

Appendix D

QCLG Level 4 Code

```
from QCLG_lvl3.classical.random_binary import RandomBinary
from tools import Tools
import constants
from qiskit.providers import JobStatus

class Evaluation:

    @classmethod
    def evaluate(cls, algorithm):

        if algorithm == "0":
            cls.evaluate_deutsch_josza()
        elif algorithm == "1":
            cls.evaluate_bernstein_vazirani()

    @classmethod
    def plot_results(cls, inputs: list, quantum: list, classical: list,
accuracy: list):
        import numpy as np
        import matplotlib.pyplot as plt
        X = np.arange(start=1, stop=len(inputs) + 1, step=1)
        plt.figure()
        plt.subplot(211)
        plt.title("Execution Times")
        plt.bar(X + 0.00, classical, color='b', width=0.25)
        plt.bar(X + 0.25, quantum, color='g', width=0.25)
        plt.ylabel("In seconds")
        plt.legend(labels=['Classical', 'Quantum'])
        plt.subplot(212)
        plt.title("Quantum Accuracy")
        plt.bar(inputs, accuracy, color='g')
        plt.ylabel("Percentage")
        plt.legend(labels=['Accuracy Percentage'])
        plt.show()
        return
```

Appendix D contains all code necessary for evaluating algorithms in level 4 of the QCLG.


```

@classmethod
def evaluate_deutsch_josza(cls):
    print(constants.input_message_3)
    test_range = int(input())
    while test_range > 14 or test_range < 1:
        test_range = int(input("Enter a number between 1 and 14."))

    circuits = []
    n_bits = []
    quantum_execution_times = []
    classical_execution_times = []
    success_rates = []

    print("Evaluating Deutsch - Josza... This might take a while...")

    for number_of_bits in range(1, test_range + 1):
        n_bits.append(number_of_bits)
        circuits.append(Tools.prepare_dj(number_of_bits))
        total_time = 0.0
        for i in range(1024):
            classical_result = Tools.deutsch_josza_classical(number_of_bits)
            total_time = total_time + classical_result[1]
        classical_execution_times.append(total_time)
        completion_percentage = int((number_of_bits / test_range) * 100)
        print(f"{completion_percentage}% of classical executions done.")

    print("Now looking for the least busy backend...")
    least_busy_backend = Tools.find_least_busy_backend_from_open(test_range)

    print("Now waiting for the Quantum Batch Job to finish...\nThis will take a while...")
    quantum_results = Tools.run_batch_job(circuits, least_busy_backend)
    flag = False
    while not flag:
        for status in quantum_results.statuses():
            flag = True
            if status != JobStatus.DONE:
                flag = False
    print("Quantum experiments finished.")

    print("Preparing plots...")
    count_jobs = 1
    for job in quantum_results.managed_jobs():
        quantum_execution_times.append(job.result().time_taken)
        counts_dict = job.result().get_counts()
        correct_counts = counts_dict.get('1' * count_jobs, 0)
        print(correct_counts)
        success_percentage = (correct_counts / 1024) * 100
        success_rates.append(success_percentage)
        count_jobs = count_jobs + 1

    cls.plot_results(n_bits, quantum_execution_times,
classical_execution_times, success_rates)
    return

```

```

@classmethod
def evaluate_bernstein_vazirani(cls):
    print(constants.input_message_3)
    test_range = int(input())
    while test_range > 14 or test_range < 1:
        test_range = int(input("Enter a number between 1 and 14."))

    n_bits = []
    circuits = []
    quantum_execution_times = []
    classical_execution_times = []
    success_rates = []
    random_binaries = []
    print("Evaluating Bernstein - Vazirani... This might take a while...")

    for number_of_bits in range(1, test_range + 1):
        n_bits.append(number_of_bits)
        classical_result = Tools.bernstein_vazirani_classical(number_of_bits)
        random_binary =
RandomBinary.generate_random_binary_v2(number_of_bits)
        random_binaries.append(random_binary)
        circuits.append(Tools.prepare_bv(random_binary))
        classical_execution_times.append(classical_result[1])
        completion_percentage = int((number_of_bits / test_range) * 100)
        print(f"{completion_percentage}% of preparation done.")

    print("Now looking for the least busy backend...")
    least_busy_backend = Tools.find_least_busy_backend_from_open(test_range)

    print("Now waiting for the Quantum Batch Job to finish...\nThis will take
a while...")
    quantum_results = Tools.run_batch_job(circuits, least_busy_backend)
    flag = False
    while not flag:
        for status in quantum_results.statuses():
            flag = True
            if status != JobStatus.DONE:
                flag = False
    print("Quantum experiments finished.")

    print("Preparing plots...")
    count = 0
    for job in quantum_results.managed_jobs():
        quantum_execution_times.append(job.result().time_taken)
        counts_dict = job.result().get_counts()
        correct_counts = counts_dict.get(random_binaries[count], 0)
        print(correct_counts)
        success_percentage = (correct_counts / 1024) * 100
        success_rates.append(success_percentage)
        count = count + 1

    cls.plot_results(n_bits, quantum_execution_times,
classical_execution_times, success_rates)
    return

```

Appendix E

Supplementary Code

Credentials Directory

```
# Put your research token here and additional names for:
# hub, group and project if you want to access research backends.

# Normal account
account_token_open = ''

# Research account
account_token_research = None
hub = None
group = None
project = None
```

Account_details.py. This file needs to be filled with the appropriate user information in order for the QCLG to function properly and be able to make calls to the remote IBM systems.

For this to happen the user needs to create an IBM Q account and generate their token on the starting page of login.

```

algorithms = ["Deutsch-Josza", "Bernstein-Vazirani"]

experiments = ["Bell State - Hello World", "Superposition with one Qubit",
               "Superposition with three Qubits",
               "Interference"]

acceptable_execution_inputs = ['0', '1', '2', '3', '4']

acceptable_algorithm_inputs = ['0', '1']

acceptable_experiment_inputs = ['0', '1', '2', '3']

input_message_1 = "Choose an Algorithm: " \
                  "\n0 for Deutsch-Josza." \
                  "\n1 for Bernstein-Vazirani" \
                  "\nYour input:"

input_message_2 = "Enter:" \
                  "\n\n0 for execution on the Local Device Simulator." \
                  "\n\n1 for execution on the Qasm simulator." \
                  "\n\n2 for execution on a real device." \
                  "\n\n3 for execution on the local device and real device." \
                  "\n\n4 for evaluating the algorithm" \
                  "\nby running comparisons of classical and quantum inputs \
with different sized inputs." \
                  "\n\nYour input:"

input_message_3 = "\nEnter the maximum size of input you would like to \
evaluate" \
                  "\n(in number of bits). The maximum is 14."

cards = ["H", "H", "X", "X", "CX", "SX", "SXDG"]

acceptable_choice_inputs = ['1', '2', '3', '4']

experimentation_level = "Enter:" \
                        "\n\n1 For Level 1 experimentation. (Simple \
experiments)" \
                        "\n\n2 For Level 2 experimentation. (The Exciting \
Game)" \
                        "\n\n3 For Level 3 experimentation. (Algorithm \
Experimentation)" \
                        "\n\n4 For Level 4 experimentation. (Algorithm \
Evaluation)" \
                        "\n\nYour input:"

```

Useful display messages.

```

import constants
from QCLG_lvl1.qclg_lvl1_manager import SimpleExperimentsManager
from QCLG_lvl2.the_exciting_game import Game
from QCLG_lvl3.quantum_algorithms.algorithms_manager import AlgorithmsManager
from QCLG_lvl4.evaluation import Evaluation

class Manager:
    if __name__ == '__main__':
        flag = "Y"
        while flag == "Y":

            choice = input(constants.experimentation_level)
            while choice not in constants.acceptable_choice_inputs:
                choice = input(constants.experimentation_level)

            if choice == '1':
                SimpleExperimentsManager.showcase()
            elif choice == '2':
                Game.play_the_exciting_game()
            elif choice == '3':
                AlgorithmsManager.showcase()
            elif choice == '4':
                algorithm = input(constants.input_message_1)
                while algorithm not in constants.acceptable_algorithm_inputs:
                    algorithm = input(constants.input_message_1)
                Evaluation.evaluate(algorithm)

            flag = input("Would you like to keep experimenting? (Y/N)")
            while flag is not "Y" and flag is not "N":
                flag = input("Would you like to keep experimenting? Please
give a valid response. (Y/N)")
            print("\nYour experimenting session is now concluded. \nThank you.")

```

Code for the manager class responsible for branching to the four levels of QCLG.

```

from datetime import datetime
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit, BasicAer, execute, IBMQ, transpile
from qiskit.providers import BaseJob
from qiskit.providers.ibmq import least_busy, IBMQJobManager
from qiskit.providers.ibmq.managed import ManagedJobSet
from qiskit.visualization import plot_histogram

import constants
from QCLG_lvl3.classical.bernstein_vazirani_classical import BernsteinVaziran-
iClassical
from QCLG_lvl3.classical.classical_xor import ClassicalXor
from QCLG_lvl3.classical.random_binary import RandomBinary
from QCLG_lvl3.quantum_algorithms.bernstein_vazirani import BernsteinVazirani
from QCLG_lvl3.quantum_algorithms.deutsch_josza import DeutschJosza
from credentials import account_details

class Tools:
    @classmethod
    def calculate_elapsed_time(cls, first_step: datetime, last_step:
datetime):
        difference = last_step - first_step
        return difference.total_seconds()

    @classmethod
    def run_on_simulator(cls, circuit: QuantumCircuit):
        # use local simulator
        backend = BasicAer.get_backend('qasm_simulator')
        shots = 1024
        results = execute(circuit, backend=backend, shots=shots).result()
        answer = results.get_counts()
        max_value = 0
        max_key = ""
        for key, value in answer.items():
            if value > max_value:
                max_value = value
                max_key = key
        return max_key[::-1]

    @classmethod
    def run_on_real_device(cls, circuit: QuantumCircuit, least_busy_backend):

        from qiskit.tools.monitor import job_monitor
        shots = int(input("Number of shots (distinct executions to run this
experiment: )"))
        job = execute(circuit, backend=least_busy_backend, shots=shots, opti-
mization_level=3)
        job_monitor(job, interval=2)
        return job

```

```

@classmethod
def find_least_busy_backend_from_open(cls, n):
    if account_details.account_token_open is None:
        account_token = input("Insert your account token: ")
    else:
        account_token = account_details.account_token_open
    if IBMQ.active_account() is None:
        IBMQ.enable_account(account_token)
    provider = IBMQ.get_provider(hub='ibm-q')
    return least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= (n + 1) and
                                                                    not x.configuration().simulator and x.status().operational == True))

@classmethod
def find_least_busy_backend_from_research(cls, n):
    if account_details.account_token_research is None \
        or account_details.hub is None \
        or account_details.group is None \
        or account_details.project is None:
        account_token = input("Insert your account token: ")
        hub = input("Insert your hub: ")
        group = input("Insert your group: ")
        project = input("Insert your project: ")
    else:
        account_token = account_details.account_token_research
        hub = account_details.hub
        group = account_details.group
        project = account_details.project

    IBMQ.enable_account(account_token)
    provider = IBMQ.get_provider(hub=hub, group=group, project=project)
    print(provider)
    return least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= (n + 1) and
                                                                    not x.configuration().simulator and x.status().operational == True))

@classmethod
def print_simul(cls, answer_of_simul, algorithm: str):
    print(constants.algorithms[int(algorithm)])
    print("\nMeasurements: ", answer_of_simul)
    return

@classmethod
def print_real(cls, job: BaseJob, least_busy_backend, algorithm: str):
    results = job.result()
    answer = results.get_counts()
    print("\nTotal counts are:", answer)
    elapsed = results.time_taken
    print(f"The time it took for the experiment to complete after validation was {elapsed} seconds")
    plot_histogram(data=answer, title=f"{constants.algorithms[int(algorithm)]} on {least_busy_backend}")
    plt.show()
    return

```

```

@classmethod
def execute_classically(cls, algorithm):
    if algorithm == "0":
        return cls.execute_deutsch_josza_classically()
    elif algorithm == "1":
        return cls.execute_bernstein_vazirani_classically()

@classmethod
def execute_in_simulator(cls, algorithm):
    dj_circuit = None
    if algorithm == "0":
        bits = str(input("Enter a bit sequence for the quantum circuit:"))
        dj_circuit = DeutschJosza.deutsch_josza(bits, eval_mode=False)
    elif algorithm == "1":
        decimals = int(input("Give the upper limit of the random number: "))
        random_binary = RandomBinary.generate_random_binary(decimals)
        dj_circuit = BernsteinVazirani.bernstein_vazirani(random_binary,
eval_mode=False)
    return cls.run_on_simulator(dj_circuit)

@classmethod
def execute_in_real_device(cls, algorithm):
    if algorithm == "0":
        answer = cls.execute_dj_in_real_device()
        return answer
    elif algorithm == "1":
        decimals = int(input("Give the upper limit of the random number: "))
        random_binary = RandomBinary.generate_random_binary(decimals)
        answer = cls.execute_bv_in_real_device(random_binary)
        return answer

@classmethod
def execute_dj_in_real_device(cls):
    bits = str(input("Enter a bit sequence for the quantum circuit:"))
    least_busy_backend = Tools.choose_from_provider(len(bits) + 1)
    dj_circuit = DeutschJosza.deutsch_josza(bits, eval_mode=False)
    answer_of_real = Tools.run_on_real_device(dj_circuit, least_busy_backend)
    print(f"least busy is {least_busy_backend}")
    return answer_of_real

@classmethod
def execute_bv_in_real_device(cls, random_binary: str):
    dj_circuit = BernsteinVazirani.bernstein_vazirani(random_binary,
eval_mode=False)
    least_busy_backend = Tools.choose_from_provider(dj_circuit.qubits)
    answer_of_real = Tools.run_on_real_device(dj_circuit, least_busy_backend)
    print(f"least busy is {least_busy_backend}")
    return answer_of_real

```



```

@classmethod
def choose_from_provider(cls, size: int):
    least_busy_backend = None
    research = input("Do you want to run this experiment on the research
backends? (Y/N)")
    while research != "Y" and research != "N":
        research = input("Do you want to run this experiment on the research
backends? (Y/N)")
    if research == "N":
        least_busy_backend = Tools.find_least_busy_backend_from_open(size)
    elif research == "Y":
        least_busy_backend = Tools.find_least_busy_backend_from_re-
search(size)
    return least_busy_backend

@classmethod
def execute_deutsch_josza_classically(cls):
    number_of_bits = int(input("Enter number of bits for a the classical so-
lution:"))
    return ClassicalXor.execute_classical_xor(bits=number_of_bits)

@classmethod
def execute_bernstein_vazirani_classically(cls):
    decimals = int(input("Give the upper limit of the random number: "))
    random_binary = RandomBinary.generate_random_binary(decimals)
    return BernsteinVaziraniClassical.guess_number(random_binary)

@classmethod
def print_classical_answer(cls, classical_answer, algorithm):
    time_to_generate_worst_input = classical_answer[0]
    execution_time = classical_answer[1]
    bits = classical_answer[2]
    function_nature = classical_answer[3]
    print(f"Results of classical implementation for the {constants.algo-
rithms[int(algorithm)]} Algorithm:")
    print(f"Function is {function_nature}")
    print(f"Time to generate worse input for {bits} bits took {time_to_gener-
ate_worst_input} seconds.")
    print(f"Determining if xor is balanced for {bits} bits took {execu-
tion_time} seconds.")
    print(classical_answer)

@classmethod
def execute_both(cls, algorithm):
    answer = []
    if algorithm == "0":
        classical = cls.execute_deutsch_josza_classically()
        real = cls.execute_dj_in_real_device()
        answer.append(classical)
        answer.append(real)
    elif algorithm == "1":
        classical = cls.execute_bernstein_vazirani_classically()
        real = cls.execute_bv_in_real_device(classical)
        answer.append(classical)
        answer.append(real)
    return answer

```

```

# Evaluation methods

@classmethod
def prepare_dj(cls, bits: int):
    bit_sequence = "0" * bits
    dj_circuit = DeutschJozsa.deutsch_josza(bit_sequence, eval_mode=True)
    return dj_circuit

@classmethod
def prepare_bv(cls, random_binary: str):
    bj_circuit = BernsteinVazirani.bernstein_vazirani(random_binary,
eval_mode=True)
    return bj_circuit

@classmethod
def deutsch_josza_classical(cls, bits: int):
    return ClassicalXor.execute_classical_xor(bits=bits)

@classmethod
def bernstein_vazirani_classical(cls, bits: int):
    random_binary = RandomBinary.generate_random_binary_v2(bits)
    return BernsteinVaziraniClassical.guess_number(random_binary)

@classmethod
def run_batch_job(cls, circuits: list, least_busy_backend) -> ManagedJobSet:
    transpiled_circuits = transpile(circuits, backend=least_busy_backend)
    # Use Job Manager to break the circuits into multiple jobs.
    job_manager = IBMQJobManager()
    job_set_eval = job_manager.run(transpiled_circuits,
backend=least_busy_backend, name='eval',
max_experiments_per_job=1) #
max_experiments_per_job =1 very important to get
# individual execution times
    return job_set_eval

```

The supplementary code is required to support all actions involved in the QCLG levels. There is a plethora of backend calls to IBM resources, and calls to local implementations of classical solutions.

Appendix F

QCLG Setup

We demonstrate the procedure of setting up QCLG in a new Windows 10 machine.

The instructions are very similar for Ubuntu with slight modifications required for paths.

Clone the repository locally by doing the following:

1. Launch PyCharm.
2. From the “VCS” tab, choose the option “Get from Version Control...”
3. From the pop-up window paste the https URL of the dmsl/quantum repository:
<https://github.com/dmsl/quantum.git>
4. Press the Clone button and choose to open on a new window.

The project will not yet work. We need to create and setup the conda environment and setup a Run Configuration in PyCharm.

For the conda environment.

Install Anaconda.

Open an Anaconda cmd.

We will now need to create the environment using these instructions:
<https://docs.anaconda.com/anacondaorg/user-guide/tasks/work-with-environments/>

We have uploaded our environment into our personal account, (sooodos/quantum) and we can activate it with the following command.

conda create --name my_env_name sooodos/quantum

After the installation finishes, go back to PyCharm. In the bottom-right corner click on the Project Interpreter box.

Choose the “Add Interpreter...” option.

From the pop-up window choose “Conda Environment” and then click on the “Existing environment” option.

Now you need to search for the python.exe script responsible for the Interpreter.

By default, it should be located in `C:\Users\NameOfUser\Anaconda3\envs\my_env_name\python.exe`.

Save changes.

Now we need to add a Run Configuration from PyCharm.

Go to the upper-right corner of the window, the Run Configurations box is located to the left of the “run” icon.

Click on it and then click “Edit Configurations”.

On the pop-up window click the “+” icon and choose “Python”. Now we need to add the script path of manager.py in order to create our Run Configuration. Just click on the folder icon which is on the far-right corner of the “Script path” place holder and find manager.py from the project hierarchy.

Click Apply and OK.

Now add the token generated by your IBM Account and insert into the account_details.py file in the corresponding token placeholder.

QCLG is now operational.