

Quantum Computation Algorithms as efficient solution to Optimization Problems

A PROJECT REPORT

Submitted in partial fulfillment of the requirement

for the Award of the Degree

of

BACHELORS OF TECHNOLOGY

in

Computer Science and Engineering

By

Mehil B. Shah

169105105



Department of Computer Science and Engineering

School of Computing and Information Technology

Manipal University Jaipur

Jaipur, Rajasthan

May/2020

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MANIPAL UNIVERSITY JAIPUR, JAIPUR – 303 007 (RAJASTHAN), INDIA

Date
27th April, 2020

CERTIFICATE

This is to certify that the project titled **QUANTUM COMPUTING AS AN EFFICIENT SOLUTION TO OPTIMIZATION PROBLEMS** is a record of the bonafide work done by **Mehil Bimal Shah** (169105105) submitted in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology (B.Tech) in **Computer Science and Engineering** of Manipal University Jaipur, during the academic year 2019-20.

Ms. Bali Devi

*Project Guide, Dept of Computer Science and Engineering
Manipal University Jaipur*

Dr. Sandeep Joshi

*HOD, Dept of Computer Science and Engineering
Manipal University Jaipur*

ACKNOWLEDGMENTS

I would like to express my deep sense of gratitude and regard to my respected and learned guide and mentor, Dr. Yogesh Prasad for his valuable help and guidance. I am extremely thankful to him for the encouragement he has given me in completing the project.

I am also grateful to respected Dr. Sandeep Joshi, Head of Computer Science and Engineering Department, and to our respected Director Dr. Rajveer Singh Shekhawat for providing us with all the necessary facilities required to learn and grow. I am thankful to my mentor, Ms. Bali Devi who constantly supported and encouraged me during the course of my internship, I am sincerely obliged for her support and help, without whom this journey wouldn't have been possible. I am also thankful to all other faculty and staff members of the C.S.E department for their kind cooperation and help.

Lastly, I would like to express my deep appreciation towards my classmates and my indebtedness to parents for providing us with moral support and encouragement.

ABSTRACT

Quantum Computing is the use of quantum-mechanical phenomena to solve problems that classical computers can't solve. We usually enjoy the benefits of classical computing, on a daily basis, however there are certain problems that classical computers are not able to solve, and we don't have enough computational power to tackle the problems. This creates the need for Quantum Computing, which could spur the development of breakthroughs in various domains to solve the problems that have never been solved before.

We began by understanding the basic concepts of Quantum Mechanics, and Mathematics behind the concepts. After the Literature Review Phase, we created the digital circuits, using basic quantum gates, and performed simulations that gave results explaining the behaviour of Quantum Gates, and how they worked. After understanding the basics of Quantum Computing, we decided to attempt solving an NP-Complete Problem, we researched about the various NP-Complete Problems, and decided upon MaxCut Problem. In the next phase, we worked on MaxCut Problem, and came up with an algorithm to efficiently solve the problem using basic algorithmic principles and quantum computing.

The results that we derived from the experiments, helped us understand the behaviour of multi-qubit quantum systems, and quantum gates, that will help us in creating quantum systems from the basic gates. We have also derived an approximation algorithm for estimation of interval and value of MaxCut, which outperforms some of the approximation algorithms by a significant factor.

The tools and software used for the development of the project are as follows :

1. IBM Quantum Experience
2. Qiskit Library
3. Quantum Backends
4. Python
5. Various libraries like Qiskit-Aer, Matplotlib, Scipy and Numpy.

LIST OF TABLES

Table No	Table Title	Page No
2.1	Truth Table for AND Gate	7
2.2	Truth Table for OR Gate	8
2.3	Truth Table for NOT Gate	8
2.4	Truth Table for NAND Gate	9
2.5	Truth Table for NOR Gate	9
2.6	Truth Table for XOR Gate	10
4.1	Final Summary of the Algorithm	35

LIST OF FIGURES

Figure No	Figure Title	Page No
1.1	Quantum Gates, with their Matrices	3
1.2	Superposition and Entanglement	5
2.1	Bloch Sphere	6
2.2	AND Gate	7
2.3	OR Gate	8
2.4	NOT Gate	8
2.5	NAND Gate	9
2.6	NOR Gate	9
2.7	XOR Gate	10
2.8	Classification of Problems as NP, NP-Hard and NP-Complete	11
2.9	Quantum Approximate Optimization Algorithm	12
3.1	8 * 3 Encoder	13
3.2	Truth Table for 8 * 3 Encoder	13
3.3	Circuit Diagram for 8 * 3 Encoder	14
3.4	1 * 4 Demultiplexer	15
3.5	Truth Table for 1 * 4 Demultiplexer	15
3.6	Circuit Diagram for 1 * 4 Demultiplexer	15
3.7	Probability Graph for the Behaviour of Quantum Gates	17
3.8	Formula for Mean Accuracy Value	19
3.9	Defining the Positive and Negative Deviations	19
3.10	Formula for Predicting the Interval of MaxCut	19
4.1	2 Input AND	20
4.2	3 Input AND	20
4.3	2 Input OR	20
4.4	3 Input OR	20

4.5	4 Input OR	20
4.6	Boolean Expression for 8 * 3 Encoder	21
4.7	Code for 8 * 3 Encoder	21
4.8	Quantum Circuit Diagram for 8 * 3 Encoder	22
4.9	Results for 8 * 3 Encoder Experiment	23
4.10	Boolean Expression for 1 * 4 Demultiplexer	23
4.11	Code for 1 * 4 Demultiplexer	24
4.12	Quantum Circuit Diagram for 1 * 4 Demultiplexer	25
4.13	Results for 1 * 4 Demultiplexer	25
4.14	Probability Graph for 1 * 4 Demultiplexer	26
4.15	Code for QAOA Algorithm	27
4.16	QAOA - Single Parameter Optimization	28
4.17	Effect of varying Gamma with Beta	28
4.18	Effect of varying Beta with Gamma	29
4.19	QAOA - Double Parameter Optimization	30
4.20	3-D Graph Plot between Beta, Gamma, and Expectation Value	30
4.21	Parameter Tuning	31
4.22	Accuracy Value, and Mean Accuracy Value	32
4.23	Observing Positive and Negative Deviations	33
4.24	95 th Percentile of Deviations	33
4.25	Confidence Interval of MaxCut - Upper Bound and Lower Bound	34
4.26	Prediction of MaxCut Values, and Computing Accuracy	35

Contents		
		Page No
Acknowledgement		i
Abstract		ii
List of Tables		iii
List of Figures		iii
Chapter 1	INTRODUCTION	
1.1	Introduction to Quantum Computing	1
1.2	Benefits of Quantum Computing	1
1.3	Use of Quantum Computing in Real Life	1
1.4	Quantum Circuit Gates	2
1.5	Quantum Algorithms	3
1.6	Concepts of Quantum Physics	3
Chapter 2	BACKGROUND RESEARCH	
2.1	Mathematical Concepts behind Quantum Mechanics	5
2.2	Basic Digital Logic	6
2.3	NP - Complete Problems	9
2.4	Quantum Approximate Optimization Algorithm	11
Chapter 3	METHODOLOGY	
3.1	Encoder	12
3.2	Demultiplexer	13
3.3	QAOA - Single Parameter Optimization	15
3.4	QAOA - Double Parameter Optimization	17
3.5	Estimation of Interval of MaxCut	17
3.6	Approximation of MaxCut Value	18
Chapter 4	IMPLEMENTATION	
4.1	Quantum Logic Gates	19
4.2	Encoder	20
4.3	Demultiplexer	22
4.4	Maxcut - Parameter Optimization	25
4.5	Parameter Tuning	29
4.6	Interval Construction and Value Estimation	31
Chapter 5	CONCLUSION	35
Chapter 6	FUTURE SCOPE OF WORK	36
REFERENCES		37
ANNEXURES		38-53

1. INTRODUCTION

1.1. *Introduction to Quantum Computing*

- Quantum Computing^[1] is the use of quantum-mechanical phenomena such as superposition and entanglement to perform computation. Computers that perform quantum computation are known as quantum computers.
- Quantum computers are believed to be able to solve certain computational problems, such as integer factorization (which underlies RSA encryption), significantly faster than classical computers.
- Quantum Computing^[2] is a new and exciting field at the intersection of mathematics, computer science and physics. It concerns a utilization of quantum mechanics to improve the efficiency of computation and communication.
- A quantum computer uses quantum phenomena of subatomic particles to compute complex mathematical problems.
- A quantum computer uses qubits to supply information and communicate through the system. It's encoded with quantum information in both states of 0 and 1 instead of classical bits which can only be a 0 or 1. This means a qubit can be in multiple places at once due to superposition.
- A quantum computer^[3] harnesses some of the almost-mystical phenomena of quantum mechanics to deliver huge leaps forward in processing power.

1.2. *Benefits of Quantum Computing*

- Quantum computers^[4] leverage quantum mechanical phenomena to manipulate information.
- In quantum computing qubit is the conventional superposition state and so there is an advantage of exponential speedup which is resulted by handling a number of calculations.
- Quantum computing has the potential to solve some of the world's toughest challenges.
- Quantum computers will enable new discoveries in the areas of healthcare, energy, environmental systems, smart materials, and beyond.
- Quantum computing can bring speed and efficiency to complex optimization problems in machine learning.
- In factories, quantum computers can help deliver optimized solutions.

1.3. *Use of Quantum Computing in Real Life*

- Auto manufacturers^[4] like Volkswagen and Daimler are using quantum computers to simulate the chemical computers can help deliver optimization insights for streamlined output, reduced waste, and lower costs. position of electrical-vehicle batteries to help find new ways to improve their performance.

- Pharmaceutical companies are leveraging quantum computers to analyze and compare compounds that could lead to the creation of new drugs.
- Airbus is using quantum computers to help calculate the most fuel-efficient ascent and descent paths for aircraft.
- ProteinQure is using Quantum Computing to study protein behaviour, using molecular simulation.

1.4. Quantum Circuit Gates

Various Quantum Circuit Gates are as follows :

- Hadamard Gate
- Pauli X, Pauli Y and Pauli Z Gate
- Phase Shift Gate
- Controlled NOT Gate
- Controlled Z Gate
- SWAP Gate
- Toffoli Gate














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

Fig. 1.1 : Quantum Circuit Gates along with their Matrices

1.5. Quantum Algorithms

Various Quantum Algorithms are listed below :

- Deutsch - Jozsa Problem
 - The **Deutsch–Jozsa algorithm** is a deterministic quantum algorithm proposed by David Deutsch and Richard Jozsa in 1992.
 - In the Deutsch–Jozsa problem, we are given a black box quantum computer known as an oracle that implements a function which takes n -digit binary values as input and produces either a 0 or a 1 as output for each such value. We are promised that the function is either constant (0 on all outputs or 1 on all outputs) or *balanced* (returns 1 for half of the input domain and 0 for the other half).^[5] The task then is to determine if the function is constant or balanced by using the oracle.
- Simon's Periodicity Algorithm
 - In the computational complexity theory and quantum computing, **Simon's problem**^[6] is a computational problem that can be solved exponentially faster on a quantum computer than on a classical (or traditional) computer. Although the problem itself is of little practical value, it can be proved that a quantum algorithm can solve this problem exponentially faster than any known classical algorithm.
- Grover's Search Algorithm
 - **Grover's algorithm** is a quantum algorithm that finds with high probability the unique input to a black box function that produces a particular output value.
 - It was devised by Lov Grover in 1996
- Shor's Factoring Algorithm
 - **Shor's algorithm** is a polynomial-time quantum computer algorithm for integer factorization.^[7] Informally, it solves the following problem: Given an integer N , find its prime factors. It was invented in 1994 by the American mathematician Peter Shor.

1.6. Concepts of Quantum Physics

- Superposition
 - Ability of a quantum system to be in multiple states simultaneously.
 - Example : In a coin toss, when the coin is in the air, it exhibits the state of both heads and tails, hence it is said to be in two states at the same time, this is a classical example of Superposition.

- Entanglement
 - **Quantum entanglement** is the physical phenomenon that occurs when a pair or group of particles is generated, interact, or share spatial proximity in a way such that the quantum state of each particle of the pair or group cannot be described independently of the state of the others, even when the particles are separated by a large distance.
- Decoherence
 - It is the physical process that transitions a system from being quantum mechanical to being classical.
 - It can be summarized as loss of quantum coherence.
- Coherence
 - Quantum coherence deals with the idea that all objects have wave-like properties.
 - If an object's wave-like nature is split in two, then the two waves may coherently interfere with each other in such a way as to form a single state that is a superposition of the two states.

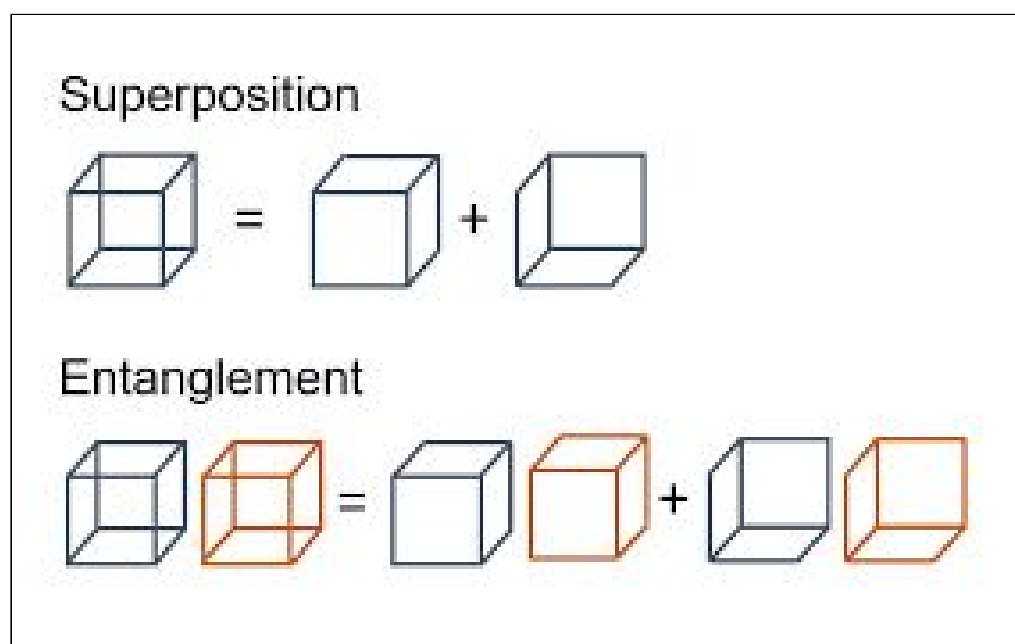


Fig. 1.2 - Superposition and Entanglement - In superposition, two states combine to form a combined state, and in entanglement, there are two combined states, and there exists a state with a given spin and antispin, no matter how far they are

2. BACKGROUND RESEARCH

2.1. *Mathematical Concepts Behind Quantum Mechanics*

- Vector and Vector Spaces
 - A vector has magnitude and direction.
 - Vector Space is a space consisting of vectors, together with the associative and commutative operation of addition of vectors, and the associative and distributive operation of multiplication of vectors by scalars.
- Bloch Spheres
 - In quantum mechanics, the Bloch sphere is a geometrical representation of the pure state space of a two-level quantum mechanical system.

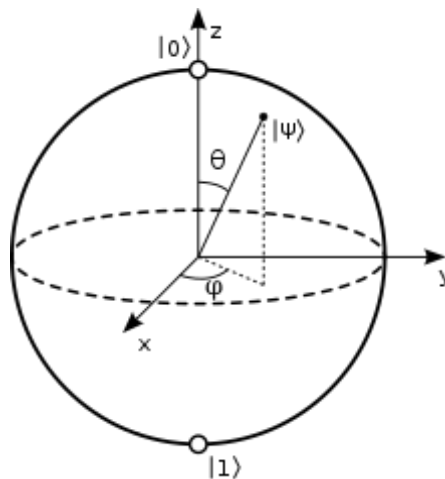


Fig. 2.1 - Bloch Sphere

- Span
 - In linear algebra, the linear span of a set S of vectors in a vector space is the smallest linear subspace that contains the set.
- Basis
 - It is a set B of elements in a vector space V , if every element of V may be written in a unique way as a linear combination of elements of B .
- Hilbert Space
 - A Hilbert space can be thought of as the state space in which all quantum state vectors "live".

- The main difference between a Hilbert space and any random vector space is that a Hilbert space is equipped with an inner product, which is an operation that can be performed between two vectors, returning a scalar.
- Eigenvalues and Eigenvectors
 - In linear algebra, an eigenvector of a linear transformation is a nonzero vector that changes at most by a scalar factor when that linear transformation is applied to it. The corresponding eigenvalue is the factor by which the eigenvector is scaled.

2.2. Basic Digital Logic

Digital, or boolean^[8], logic is the fundamental concept underpinning all modern computer systems. Put simply, it's the system of rules that allow us to make extremely complicated decisions based on relatively simple "yes/no" questions.

Digital logic circuits can be broken down into two subcategories- combinational and sequential. Combinational logic changes "instantly"- the output of the circuit responds as soon as the input changes (with some delay, of course, since the propagation of the signal through the circuit elements takes a little time). Sequential circuits have a clock signal, and changes propagate through stages of the circuit on edges of the clock.

Typically, a sequential circuit will be built up of blocks of combinational logic separated by memory elements that are activated by a clock signal.

Digital Logic Gates

- AND Gate

The output state^[8] of a digital logic AND gate only returns “LOW” again when any of its inputs are at a logic level “0”.

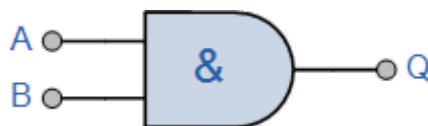


Fig. 2.2 - AND Gate

Table 2.1 - Truth Table for AND Gate

Input A	Input B	Output
0	0	0
0	1	0
1	0	0
1	1	1

- OR Gate

The output^[8], Q of a “Logic OR Gate” only returns “LOW” again when all of its inputs are at a logic level “0”. In other words for a logic OR gate, any “HIGH” input will give a “HIGH”, logic level “1” output.

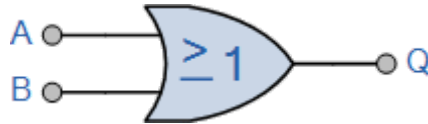


Fig. 2.3 - OR Gate

Table 2.2 - Truth Table for OR Gate

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	1

- NOT Gate

Inverting^[8] NOT gates are single input devices which have an output level that is normally at logic level “1” and goes “LOW” to a logic level “0” when its single input is at logic level “1”, in other words it “inverts” (complements) its input signal.

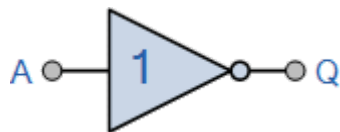


Fig. 2.4 - NOT Gate

Table 2.3 - Truth Table for NOT Gate

Input A	Output
0	1
1	0

- NAND Gate

The NAND^[8] (Not – AND) gate has an output that is normally at logic level “1” and only goes “LOW” to logic level “0” when all of its inputs are at logic level “1”. The Logic NAND Gate is the reverse form of the AND gate which has been mentioned earlier.

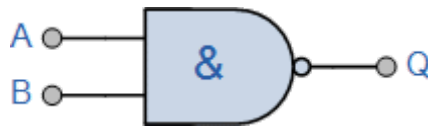


Fig. 2.5 - NAND Gate

Table 2.4 - Truth Table for NAND Gate

Input A	Input B	Output
0	0	1
0	1	0
1	0	0
1	1	0

- NOR Gate

The inclusive NOR (Not-OR) gate has an output that is normally at logic level “1” and only goes “LOW” to logic level “0” when any of its inputs are at logic level “1”. The Logic NOR Gate is the reverse form of OR Gate which has been mentioned earlier.

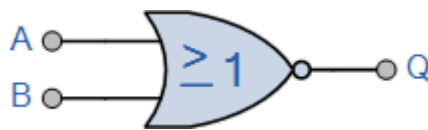


Fig. 2.6 - NOR Gate

Table 2.5 - Truth Table for NOR Gate

Input A	Input B	Output
0	0	1
0	1	0
1	0	0
1	1	0

- XOR Gate

The output^[8] of an Exclusive-OR gate only goes high when its two input terminals are at different logic levels with respect to each other.

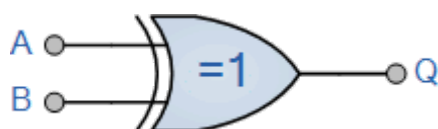


Fig. 2.7 - XOR Gate

Table 2.6 - Truth Table for XOR Gate

Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0

2.3. NP - Complete Problems

In computational complexity theory,^[9] a problem is NP-complete when it can be solved by a restricted class of brute force search algorithms and it can be used to simulate any other problem with a similar algorithm. More precisely, each input to the problem should be associated with a set of solutions of polynomial length, whose validity can be tested quickly (in polynomial time), such that the output for any input is "yes" if the solution set is non-empty and "no" if it is empty.

NP-complete problems are in NP, the set of all decision problems whose solutions can be verified in polynomial time; *NP* may be equivalently defined as the set of decision problems that can be solved in polynomial time on a non-deterministic Turing machine. A problem p in NP is NP-complete if every other problem in NP can be transformed (or reduced) into p in polynomial time.

NP-Complete Problems can be verified in polynomial time, but it is not sure whether they can be solved in polynomial time or not.

Karp's List of NP - Complete Problems

- **Satisfiability:** the boolean satisfiability problem for formulas in conjunctive normal form (often referred to as SAT)
 - 0–1 integer programming
 - Clique
 - Set packing
 - Vertex cover
 - Set covering
 - Feedback node set
 - Feedback arc set
 - Directed Hamilton circuit
 - Satisfiability with at most 3 literals per clause
 - Graph Coloring Problems

- Clique cover
- Exact cover
 - Hitting set
 - Steiner tree
 - 3-dimensional matching
 - Knapsack
 - Job sequencing
 - Partition
 - MaxCut

Solving NP-Complete Problems

The following techniques^[9] can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:

Approximation: Instead of searching for an optimal solution, search for a solution that is at most a factor from an optimal one.

Randomization: Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability.

Restriction: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.

Parameterization: Often there are fast algorithms if certain parameters of the input are fixed.

Heuristic: An algorithm that works "reasonably well" in many cases, but for which there is no proof that it is both always fast and always produces a good result.

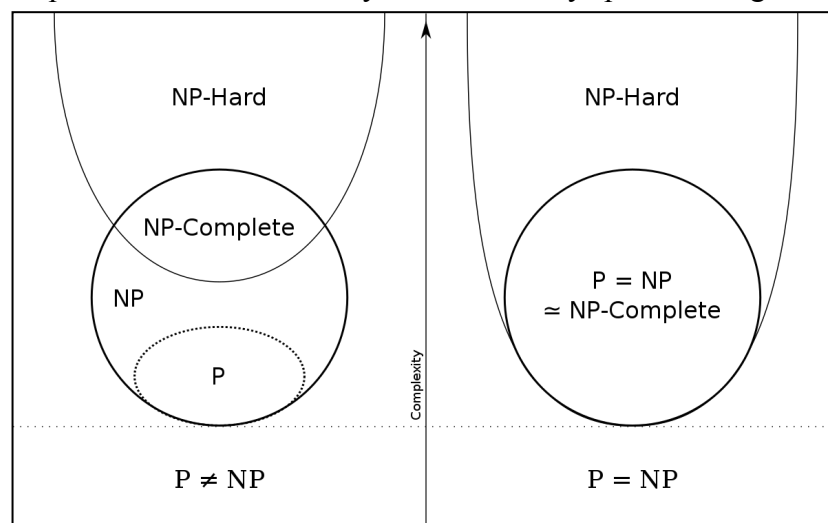


Fig. 2.8 - Classification of Problems as NP, NP-Hard, NP-Complete

2.4. Quantum Approximate Optimization Algorithm

Quantum Approximation Optimization Algorithm (QAOA)^[10] is one of the algorithms that can be implemented in the near-term quantum computer and regarded as one of the most promising algorithms to demonstrate quantum supremacy. QAOA is an approximation algorithm which means it does not deliver the ‘best’ result, but only the ‘good enough’ result, which is characterized by a lower bound of the approximation ratio.

It approximately solves the Maximum Satisfiability Problem, this is the problem of determining the maximum number of clauses of a given Boolean formula that can be made true by an assignment of truth values to the variables of the formula. By approximate, we mean that it “satisfies a high fraction of the maximum number of clauses that can be satisfied”.

QAOA has been successfully demonstrated to apply to graph partitioning problems such as MAX-CUT, number partitioning problems, and has the potential to apply to a range of NP-complete and NP-hard problems.

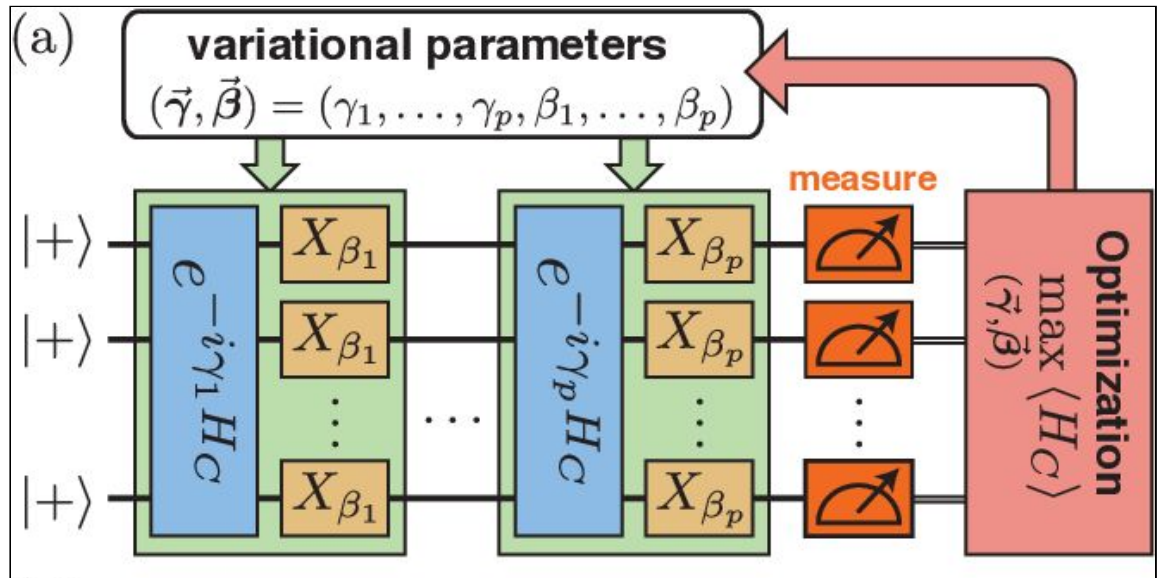


Fig. 2.9 - Quantum Approximate Optimization Algorithm

- In the first phase of the algorithm, parameters β and γ are varied over the fixed intervals, and are generated over evenly spaced intervals.
- Cost Hamiltonians, and Driver Hamiltonian are prepared with the help of variational parameters.
- Hamiltonians generated are applied to each qubit, representing the each node of the graph.
- The resultant value is measured using Quantum Measurement Gates
- We repeat the process for all values of β and γ and select the best parameters which give us the maximum value, and solve the problem for other graphs.

3. METHODOLOGY

3.1. Encoder

An Encoder^[11] is a combinational circuit that performs the reverse operation of Decoder. It has a maximum of 2^n input lines and 'n' output lines. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2^n input lines with 'n' bits.

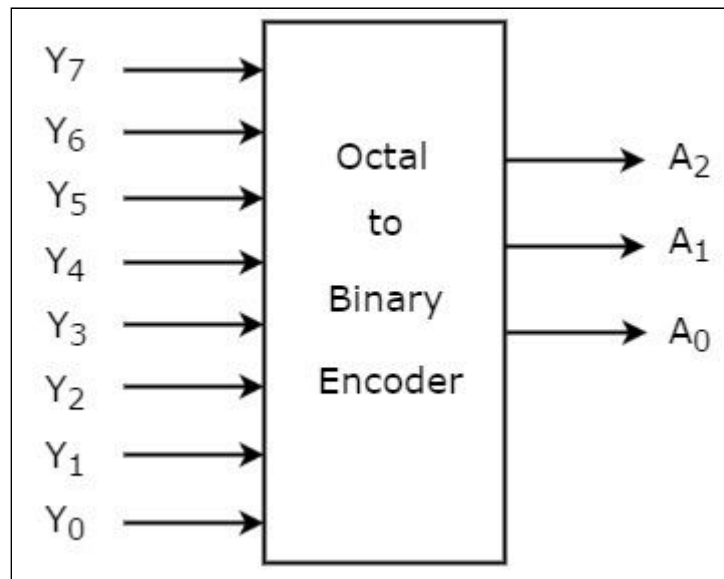


Fig. 3.1 - 8 * 3 Encoder

Inputs								Outputs		
Y ₇	Y ₆	Y ₅	Y ₄	Y ₃	Y ₂	Y ₁	Y ₀	A ₂	A ₁	A ₀
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Fig. 3.2 - Truth Table for Encoder

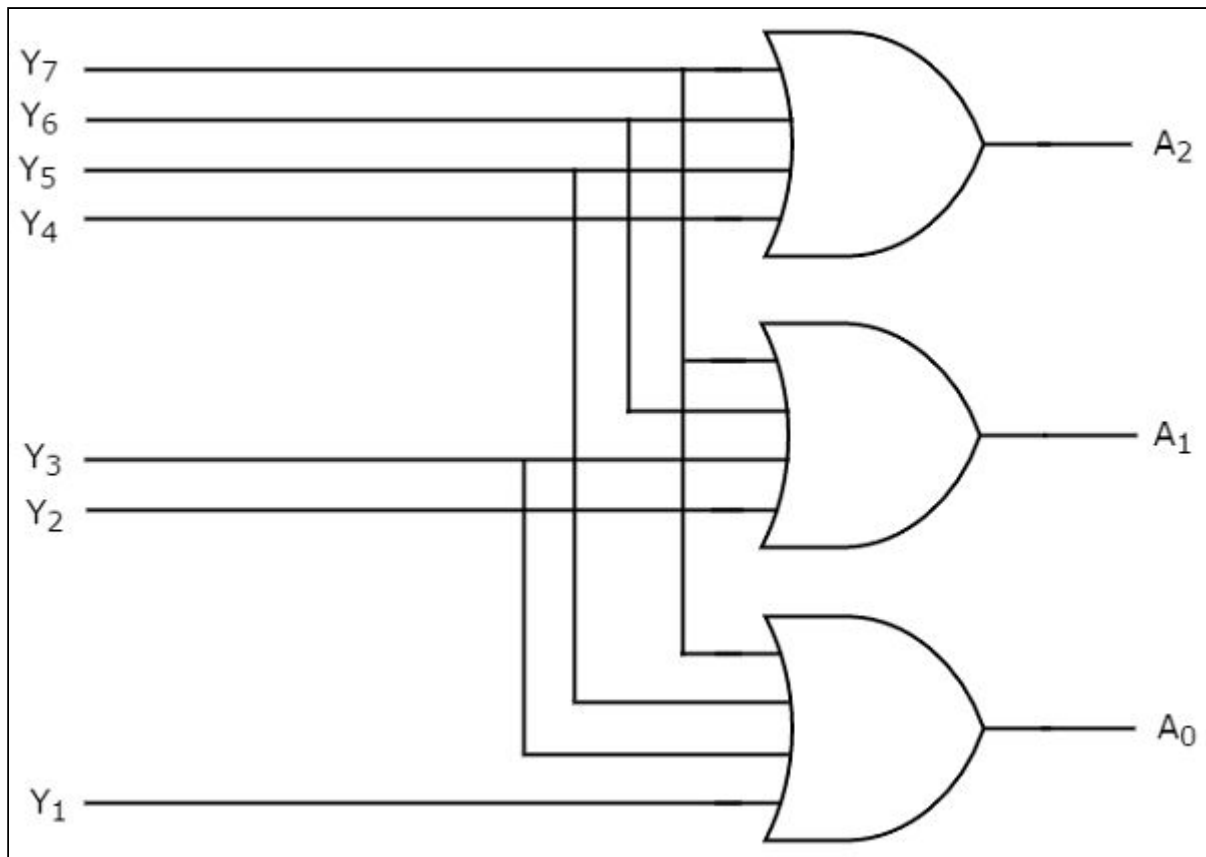


Fig. 3.3 - Circuit Diagram for Encoder

- Using the circuit diagram in Figure 3.3, we derived the boolean expressions for each of A_0 , A_1 and A_2 and then, implemented them using custom-designed Logic Gates using Quantum Gates, which will be discussed later in Chapter 4.

3.2. Demultiplexer

- Demultiplexer^[12] is a combinational circuit that performs the reverse operation of a Multiplexer. It has single input, 'n' selection lines and maximum of 2^n outputs. The input will be connected to one of these outputs based on the values of selection lines.
- Since there are 'n' selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination can select only one output. Demultiplexer is also called De-Mux.
- Using the circuit diagram in Figure 3.6, we derive the boolean expressions and implement them using custom-designed Logic Gates, just like Encoder. The implementation is discussed in Chapter 4.

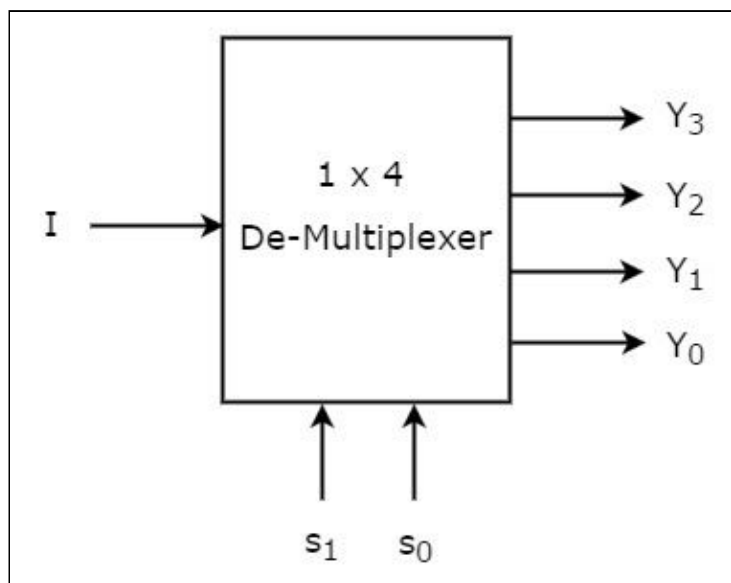


Fig. 3.4 - 1 * 4 Demultiplexer

Selection Inputs		Outputs			
s_1	s_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	I
0	1	0	0	I	0
1	0	0	I	0	0
1	1	I	0	0	0

Fig. 3.5 - Truth Table for 1*4 Demultiplexer

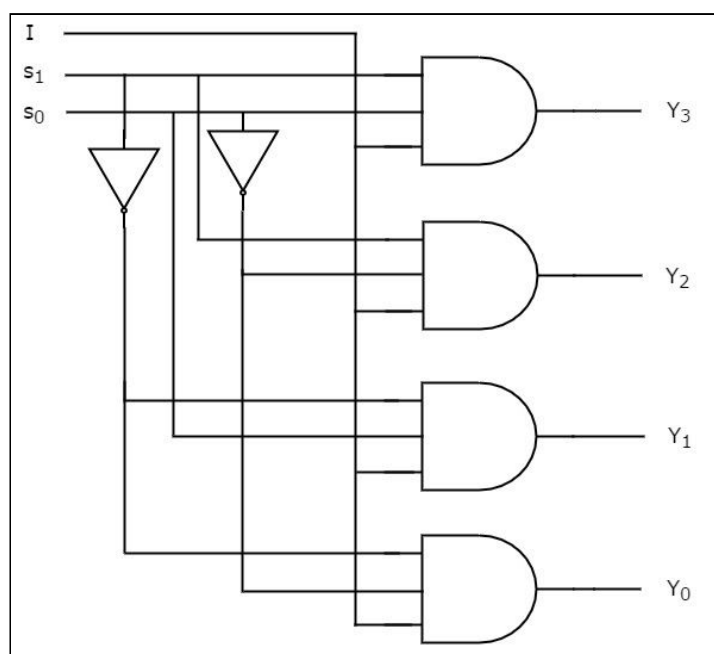


Fig. 3.6 - Demultiplexer Circuit Diagram

3.3. QAOA - Single Parameter Optimization

After completing the simulations of Digital Circuits using Quantum Gates, we decided on solving the MaxCut Problem using Quantum Approximate Optimization Algorithm, which was discussed earlier.

- MaxCut is one of the Karp's 21 NP-Complete Problems.
- To begin with the Graph Problem, we created a class named Graph, which had properties like Nodes, Edges, and methods of generating edges and adding them to the Graph.
- Given a set of vertices and weighted edges connecting some of the vertices, we are interested in separating the vertices into two sets such that the sum of the weights of the edges between the sets is maximized.
- Since the problem is NP-Complete, there are no polynomial time algorithms known for solving the problem.

Quantum Approximate Optimization Algorithm works in 2 stages

- Classical Stage
 - In the classical stage, two parameters gamma and beta are randomly chosen and fed into the quantum stage, getting an expectation value. We then run a classical optimizer over gamma and beta, using the quantum stage as our 'black-box' cost function, until we are satisfied with the expectation value.
- Quantum Stage
 - In Quantum Stage, the parameters beta and gamma derived from classical stage, are used for preparing Cost and Driver Hamiltonian, the newly prepared Hamiltonian are applied to the qubits, which are prepared as superposition of candidate solutions. Once, the circuit is complete, it is run for an arbitrary number of times, and we have the expected value, which is calculated as summation of Probability of the Candidate Solution * Cost of the Candidate Solution.

What is Expectation Value, and why do we want to maximize it?

- The answer of simulation is in the form of bit string consisting of 0's and 1's
- Weighted MaxCut Results are denoted by encoding the graph as a string of 0's and 1's, where the 0's lie in one half of the cut, whereas 1's lie in another half of the cut, it shows how the graph can be divided into two sets which maximize the weight of the edges crossing the cut.
- Expectation Value is the numerical value of the MaxCut, which shows the expected value of the cut. It is a numerical value which denotes the value of the weight/cost of the cut.
- The concepts will be demonstrated with the graph given below in the Figure 3.7

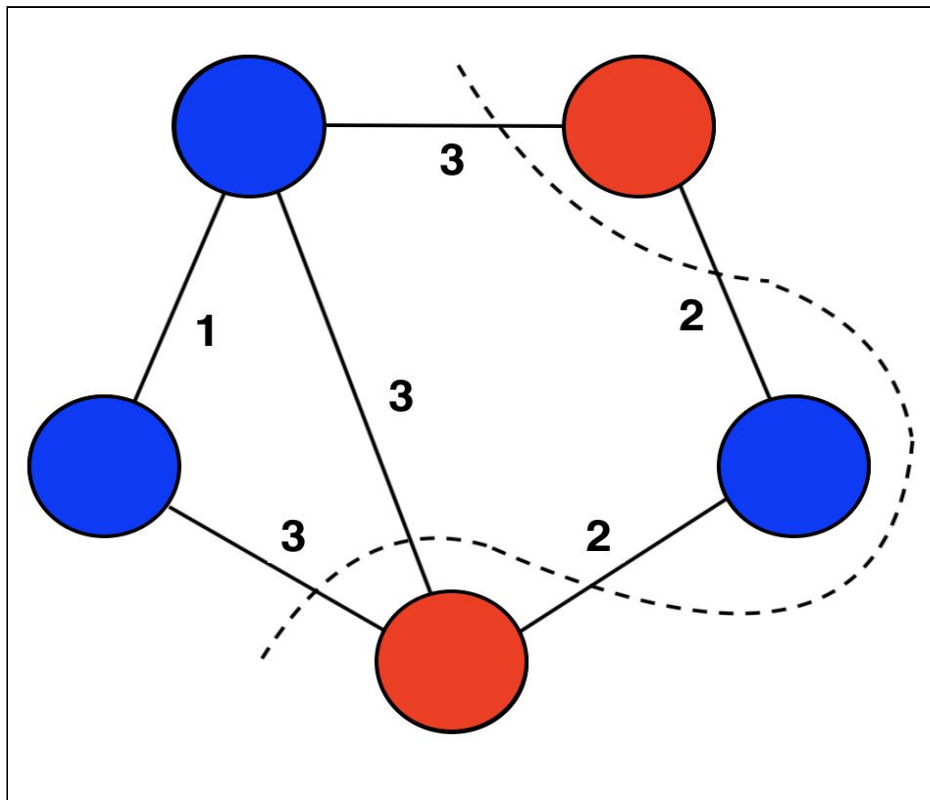


Fig. 3.7 - Graph depicting the MaxCut

- Here, if we include all the blue vertices in one set, and red vertices in another set, we can encode the graph as 00101, where 0 represents the blue vertices, and 1 represents the red vertices starting from the left-most blue vertex.
- The MaxCut value of the graph is $(3+2+2+3+3) = 13$, which should be the expected value of the simulation.

Steps involved in the Quantum Approximate Optimization Algorithm - Single Parameter Optimization

- The parameters are beta and gamma, and they are used to construct the Hamiltonian for the Quantum Stage of QAOA.
- Beta is distributed over 0 to π , and Gamma is distributed over 0 to 2π .
- Select any arbitrary value from the given range for one parameter, and other value is to be varied over the given range, i.e if beta is given a constant value somewhere between 0 to π , we'll create 100 gamma values equally distributed over 0 to 2π
- Now, we find out expected values for all combinations of beta and gamma by running circuit simulations using hamiltonians, and the combination of beta and gamma, which yields the value closest to the expected value is selected to be the optimized value.
- Now, after obtaining the best value of beta and gamma, we try it for various graphs and determine the accuracy of the process.

3.4. QAOA - Double Parameter Optimization

- It is similar to the Single Parameter Optimization, but instead of fixating on one parameter, and varying the second parameter, we vary both the parameters over the given interval, and try to maximize the expectation value.

Steps involved in the Quantum Approximate Optimization Algorithm - Double Parameter Optimization

- Here, instead of keeping one of the parameters constant, we vary both the parameters of beta and gamma, and generate combinations of beta and gamma.
 - Now, we find out expected values for all combinations of beta and gamma by running circuit simulations using hamiltonian, and the combination of beta and gamma, which yields the value closest to the expected value is selected to be the optimized value.
 - Now, after obtaining the best value of beta and gamma, we try it for various graphs and determine the accuracy of the process.
 - After measuring the accuracy, we use the best values of beta and gamma, for our proposed algorithm for prediction of interval of MaxCut value.
-
- It is more accurate than Single Parameter Optimization, as it has more values to check, and both parameters are equally varied over a constant interval.

3.5. Estimation of Interval of MaxCut

In this section, we propose an algorithm for the estimation of interval of MaxCut value i.e we predict the interval in which the expectation value of MaxCut lies, and compare it to the actual MaxCut value in order to predict the accuracy.

This algorithm works on the principle that QAOA performs well on graphs with a lesser amount of nodes, but fails to replicate the same accuracy when it encounters the graph with a higher number of nodes.

The algorithm works as follows :

1. We run the double parameter optimization to get the value of beta and gamma for a 4-node graph.
2. Generate 100 random graphs for a number of vertices, and check it's accuracy value.
3. Find the mean accuracy value and invert it to get the multiplicative factor.
4. Run the code for 100 random graphs and multiply the expected value with the multiplicative factor.
5. Observe the positive and negative deviations, and add them to different lists.
6. Take the 95th percentile value for the list of positive and negative deviations and let's assume it to be x & y.
7. Multiply the multiplicative factor by (1-x) and (1+y) to get the multiplicative factor in which the MaxCut values lie.
8. In this way, we will get an interval with 95% confidence, such that the MaxCut value lies in it.

$$\sum_{1}^n (PredictedValueQAOA / ActualMaxCutValue) * 100$$

Fig. 3.8 - Formula for Deriving Mean Accuracy Value

$$\begin{aligned} PredictedValueQAOA \times MAV &\leq ActualMaxCutValue \rightarrow PositiveDeviation \\ PredictedValueQAOA \times MAV &> ActualMaxCutValue \rightarrow NegativeDeviation \end{aligned}$$

Fig. 3.9 - Defining Positive and Negative Deviations

$$PredictedValue \times MAV \times (1 - 95thPercentileOfPositiveDeviation) \leq MaxCutValue \leq PredictedValue \times MAV \times (1 + 95thPercentileOfNegativeDeviation)$$

Fig. 3.10 - Formula for Predicting the Interval of MaxCut Value

3.6. Prediction of the MaxCut Value

- We modify the algorithm proposed above in order to predict the MaxCut value with a certain approximation value.
- In this algorithm, the first 5 steps remain the same as above.
- After getting the positive and negative deviations, instead of picking the 95th percentile, we pick the highest positive and negative deviations, and also, count the number of positive and negative deviations.
- After obtaining the values, we multiply the highest deviations with the opposite count i.e the maximum positive deviation with the count of negative deviations occurring, and maximum negative deviation with the count of positive deviations occurring, and add them up to get the MaxCut factor, which then multiplied with the multiplicative factor, and the predicted value by the initial algorithm gives the MaxCut value accurately for all the graphs.

$$a \times f \times ((1 - x) \times n + (1 - y) \times m)$$

Fig. 3.11 - Formula for MaxCut Value

- In the above formula
 - a = predicted value by QAOA
 - f = multiplicative factor
 - x = maximum positive deviation
 - n = number of negative deviations
 - y = maximum negative deviation
 - m = number of positive deviations

4. IMPLEMENTATION

4.1. Quantum Logic Gates

We first begin by creating the basic logic gates, using Quantum Gates.

We implemented the multi-input OR and AND Gate using Quantum Gates.

```
def fun_and(qc, q0, q1, q2):  
    qc.ccx(q0, q1, q2)
```

Fig. 4.1 - 2 Input AND Gate

```
def and3(qc, q0, q1, q2, b, q3):  
    fun_and(qc, q0, q1, b)  
    fun_and(qc, b, q2, q3)  
    qc.reset(b)
```

Fig. 4.2 - 3 Input AND Gate

```
def or3(qc, q0, q1, q2, b, q3):  
    fun_or(qc, q0, q1, b)  
    fun_or(qc, b, q2, q3)  
    qc.reset(b)
```

Fig. 4.4 - 3 Input OR Gate

```
def fun_or(qc, q0, q1, q2):  
    qc.x(q0)  
    qc.x(q1)  
    qc.ccx(q0, q1, q2)  
    qc.x(q2)  
    qc.x(q1)  
    qc.x(q0)
```

Fig. 4.3 - 2 Input OR Gate

```
def or4(qc, q0, q1, q2, q3, b1, b2, q4):  
    or3(qc, q0, q1, q2, b1, b2)  
    fun_or(qc, b2, q3, q4)  
    qc.reset(b1)  
    qc.reset(b2)
```

Fig. 4.5 - 4 Input OR Gate

- The 2 Input OR Gate, and 2 Input OR Gate were constructed first, where *CCX Gate* was used to construct AND Gate, and OR Gate.
- In 3 Input OR Gate, we take input $q0$, $q1$, and $q2$ which are input qubits, and there is a buffer qubit b , and output qubit $q3$, and first, we perform OR of $q0$ and $q1$, and save it in the buffer qubit b , and then, we perform OR of b and $q2$, and store it in output qubit of $q3$.
- 4-Input OR Gate works on the same principle of 3-Input OR Gate.
- In 3 Input OR Gate, we take input $q0$, $q1$, and $q2$ which are input qubits, and there is a buffer qubit b , and output qubit $q3$, and first, we perform AND of $q0$ and $q1$, and save it in the buffer qubit b , and then, we perform AND of b and $q2$, and store it in output qubit of $q3$.
- So, these are the basic gates we created for simulating Digital Circuits.

4.2. Encoder

- We implemented the encoder by using Quantum Logic Gates, and the boolean expressions derived from the circuit of the Encoder.
- The boolean expression from the circuit is as follows :

$$Q_0 = \sum(1, 3, 5, 7)$$

$$Q_0 = \sum(\bar{D}_7 \bar{D}_6 \bar{D}_5 \bar{D}_4 \bar{D}_3 \bar{D}_2 D_1 + \bar{D}_7 \bar{D}_6 \bar{D}_5 \bar{D}_4 D_3 + \bar{D}_7 \bar{D}_6 D_5 + D_7)$$

$$Q_0 = \sum(\bar{D}_6 \bar{D}_4 \bar{D}_2 D_1 + \bar{D}_6 \bar{D}_4 D_3 + \bar{D}_6 D_5 + D_7)$$

$$Q_0 = \sum(\bar{D}_6 (\bar{D}_4 \bar{D}_2 D_1 + \bar{D}_4 D_3 + D_5) + D_7)$$

$$Q_1 = \sum(2, 3, 6, 7)$$

$$Q_1 = \sum(\bar{D}_7 \bar{D}_6 \bar{D}_5 \bar{D}_4 \bar{D}_3 D_2 + \bar{D}_7 \bar{D}_6 \bar{D}_5 \bar{D}_4 D_3 + \bar{D}_7 D_6 + D_7)$$

$$Q_1 = \sum(\bar{D}_5 \bar{D}_4 D_2 + \bar{D}_5 \bar{D}_4 D_3 + D_6 + D_7)$$

$$Q_1 = \sum(\bar{D}_5 \bar{D}_4 (D_2 + D_3) + D_6 + D_7)$$

$$Q_2 = \sum(4, 5, 6, 7)$$

$$Q_2 = \sum(\bar{D}_7 \bar{D}_6 \bar{D}_5 D_4 + \bar{D}_7 \bar{D}_6 D_5 + \bar{D}_7 D_6 + D_7)$$

$$Q_2 = \sum(D_4 + D_5 + D_6 + D_7)$$

Fig. 4.6 - Boolean Expression for Encoder

```

simulator = Aer.get_backend('qasm_simulator')
qi=QuantumRegister(8)
qb=QuantumRegister(2)
qo=QuantumRegister(3)
v=QuantumRegister(1)
c=ClassicalRegister(3)
circuit = QuantumCircuit(qi,qb,qo,v,c)
circuit.x(qi[5])
or4(circuit,qi[4],qi[5],qi[6],qi[7],qb[0],qb[1],qo[2])
or4(circuit,qi[2],qi[3],qi[6],qi[7],qb[0],qb[1],qo[1])
or4(circuit,qi[1],qi[5],qi[3],qi[7],qb[0],qb[1],qo[0])
circuit.measure(qo[2],c[2])
circuit.measure(qo[1],c[1])
circuit.measure(qo[0],c[0])
circuit.draw()

```

Fig. 4.7 - Code for Encoder

Explanation of the Code

- We first obtain the simulator from Qiskit.Aer, and we obtain the 'qasm_simulator' backend.
- We then initialize all the registers, as required.
 - qi = Input Register
 - qb = Buffer Register
 - qo = Output Register
 - c = Classical Registers, required for measuring the Quantum Output and mapping it to Classical Output.
- We then decide, which bit should be switched on, and we select the bit 5, for the output, so the expected output should be 101.
- We then implement the classical equations using quantum logic gates.
- After implementing the equations, we measure the output using Classical Registers defined earlier, and then we draw the resultant circuit.

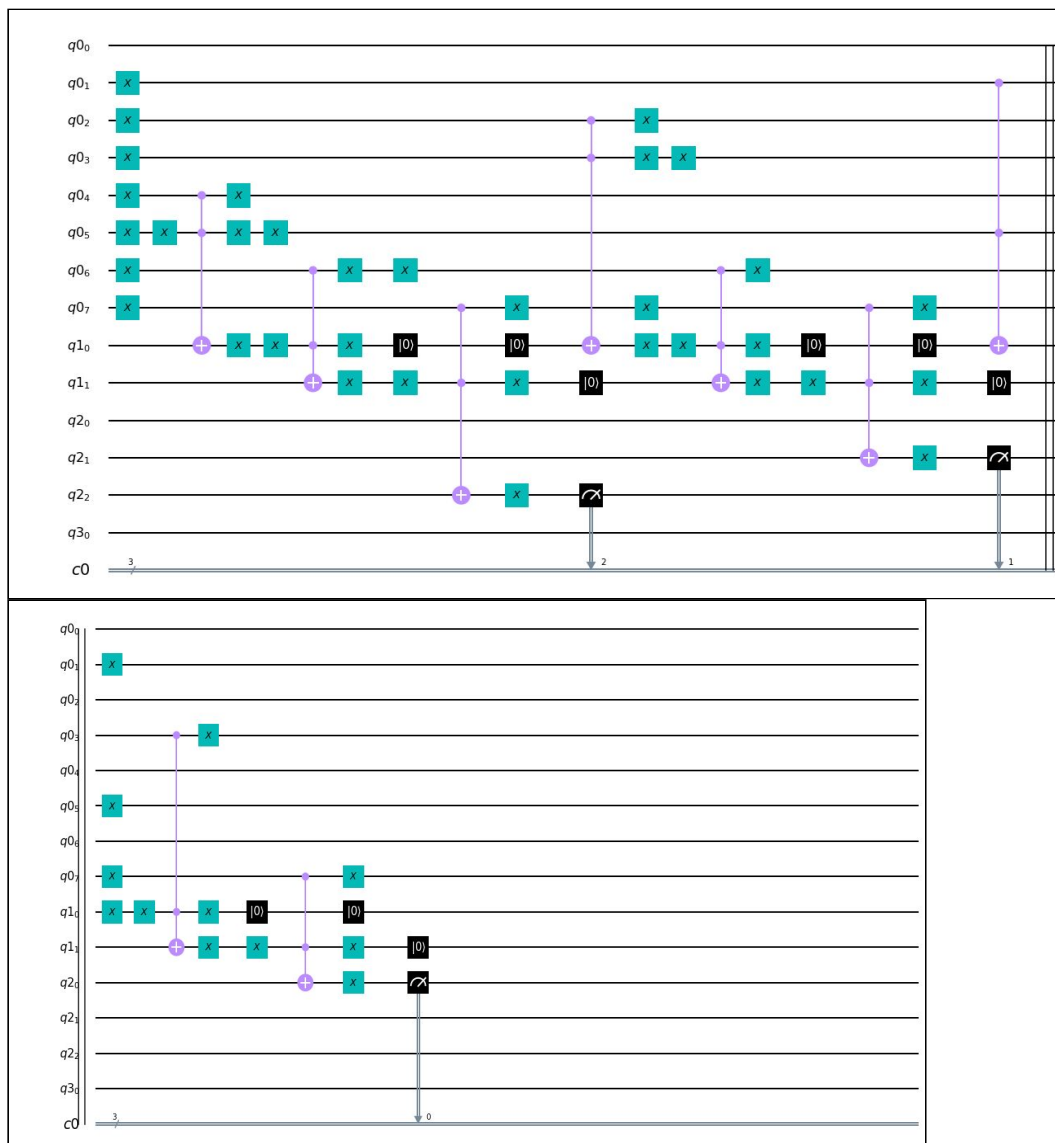


Fig. 4.8 - Circuit Diagram for Encoder

Results of the Experiment

```
job = execute(circuit,simulator,shots = 8192)
result = job.result()
counts = result.get_counts(circuit)
print(counts)
plot_histogram(counts)
```

```
{'101': 8192}
```

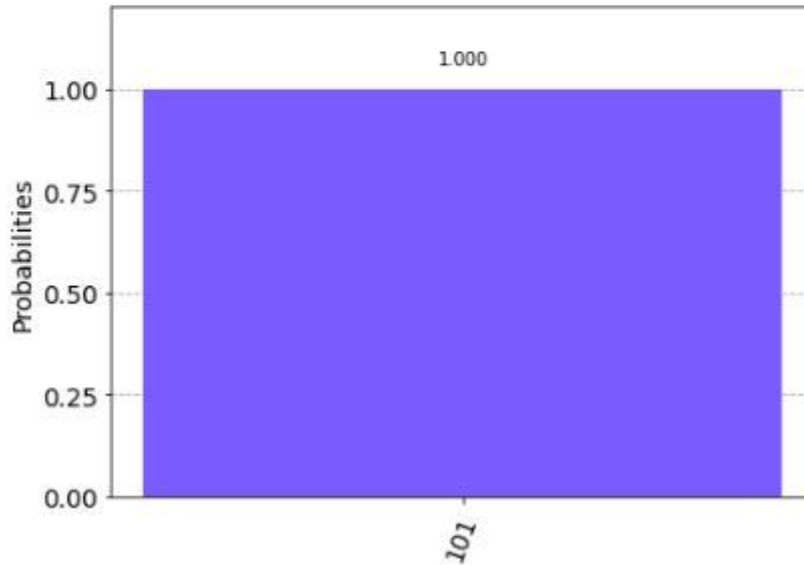


Fig. 4.9 - Results of Encoder Experiment

Explanation and Analysis of the Results

- We execute the given circuit on the simulator obtained, for 8192 times, represented by the number of shots in the function.
- We get the output '101' for all the 8192 times, which was as expected.
- We plot the histogram of the results, and get 101 with the probability 1.00 as expected.
- If we change the input qubit to 4, instead of 5, we get the result '100' as expected.

4.3. Demultiplexer

- We implemented the encoder by using Quantum Logic Gates, and the boolean expressions derived from the circuit of the Demultiplexer.

$$F = \bar{a}\bar{b}A + \bar{a}bB + a\bar{b}C + abD$$

Fig. 4.10 - Boolean Expression for Demultiplexer - here only one term will be selected, depending on the values of a and b

```

simulator = Aer.get_backend('qasm_simulator')
qi = QuantumRegister(1)
qs = QuantumRegister(2)
qxs = QuantumRegister(2)
qb = QuantumRegister(1)
qo = QuantumRegister(4)
c = ClassicalRegister(4)
circuit = QuantumCircuit(qi,qb,qs,qxs,qo,c)
circuit.x(qi[0])
circuit.h(qs[1])
circuit.h(qs[0])
for i in range(0,2):
    circuit.cx(qs[i],qxs[i])
    circuit.x(qxs[i])
and3(circuit,qxs[0],qxs[1],qi[0],qb[0],qo[0])
and3(circuit,qs[0],qxs[1],qi[0],qb[0],qo[1])
and3(circuit,qxs[0],qs[1],qi[0],qb[0],qo[2])
and3(circuit,qs[0],qs[1],qi[0],qb[0],qo[3])

circuit.measure(qo[0],c[3])
circuit.measure(qo[1],c[2])
circuit.measure(qo[2],c[1])
circuit.measure(qo[3],c[0])
circuit.draw()

```

Fig. 4.11 - Code for Demultiplexer

Explanation of the Code

- We first obtain the simulator from Qiskit.Aer, and we obtain the 'qasm_simulator' backend.
- We then initialize all the registers, as required.
 - qi = Input Register
 - qs = Select Lines
 - qxs = Inverted Select Lines
 - qb = Buffer Register
 - qo = Output Register
 - c = Classical Registers, required for measuring the Quantum Output and mapping it to Classical Output.
- We then decide, which bit should be switched on, and we select the bit 0, for the output, so the expected output should be 0001, it follows the big-endian approach for presenting data.
- We get the inverted select lines, for the use of the classical Boolean Equations.
- We then implement the classical equations using quantum logic gates.
- After implementing the equations, we measure the output using Classical Registers defined earlier, and then we draw the resultant circuit.

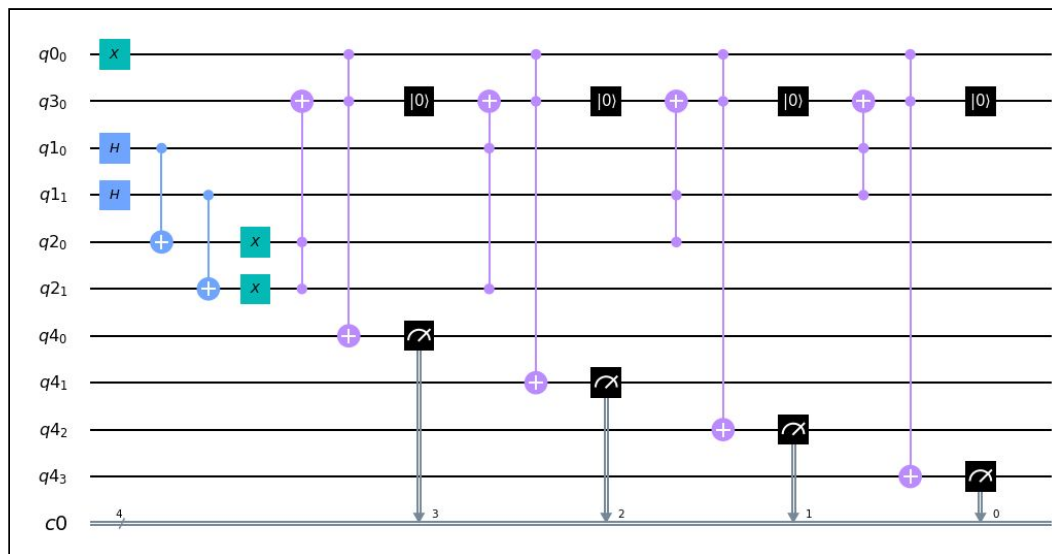


Fig. 4.12 - Circuit Diagram for Demultiplexer

Results of the Experiment

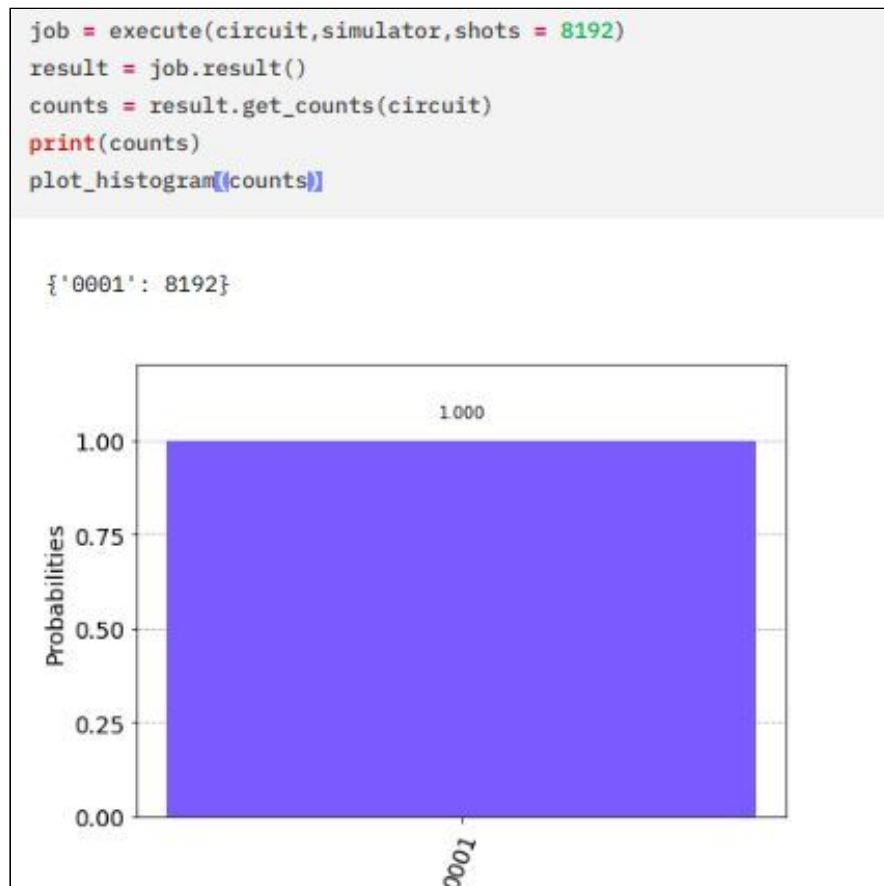


Fig. 4.13 - Results of Demultiplexer Experiment

Explanation and Analysis of the Results

- We execute the given circuit on the simulator obtained, for 8192 times, represented by the number of shots in the function.
- We get the output '0001' for all the 8192 times, which was as expected.
- We plot the histogram of the results, and get '0001' with the probability 1.00 as expected.
- If we change the input qubit to 1, instead of 0 we get the result '0010' as expected.
- We observe the relation between number of runs, and the probability of output states, i.e more the number of runs, more accurate the result. So, we conclude that for quantum gates to perform better, we need to run them for more number of times.
- In the graph given below, the 4 lines indicate the 4 possible outputs and their probabilities. it can be observed that for more number of runs, the probability converges towards the expected 25%.

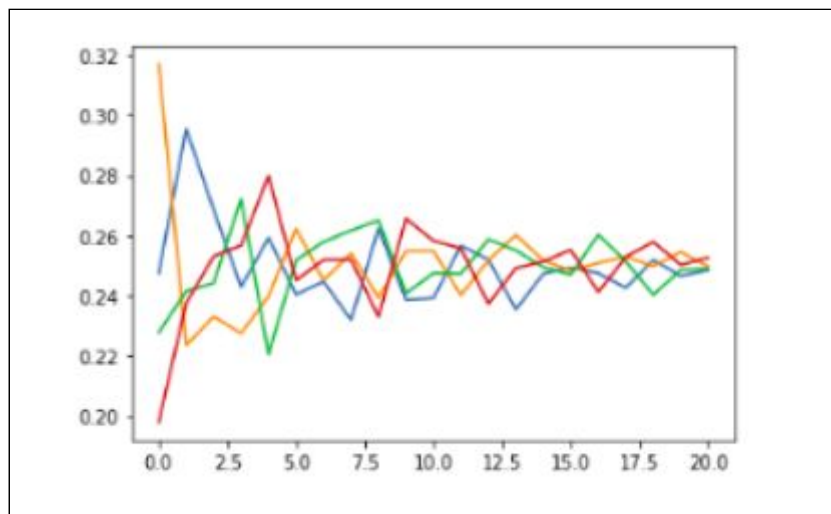


Fig. 4.14 - Probability Graph of the Demultiplexer Experiment.

4.4. MaxCut - Parameter Optimization

- For solving Graph Problems, we defined a Graph Class, which had different methods, and properties of a Graph.
- It included properties like
 - Number of Nodes
 - Number of Edges
 - Adjacency Matrix
- It also included methods like
 - `add_edge()`
 - `get_edges()`
 - `get_score()`
 - `optimal_score()`

- Complete code for the class Graph is provided in the annexure.

Working Procedure of Single Parameter Optimization

1. To begin with Parameter Optimization, we started with Single Parameter Optimization, we created a graph of edges, and added random edges to the graph. We then calculated the optimal cut for the graph, and got the best cut possible from the graph.
2. We then decided on the number of runs for the code, it was decided to be 60.
3. We then fixated on the value of one parameter, which was randomly picked from its respective interval i.e for β , the fixed value must be between 0 to π , and for γ , the fixed value must be between 0 to 2π .
4. We then create a list of lists, which is known as points, where the first parameter is γ , and the second parameter is β .
5. After creating the list of points, we pass the list to Quantum Approximate Optimization Algorithm, which first prepares the Cost Hamiltonian and Driver Hamiltonian, which is a function of the parameters passed earlier.
6. Number of Input Qubits are decided, which is equal to number of nodes in the Input Graph, after Input Qubits are decided, we apply the Hamiltonian to each Node of the Graph.
7. After applying Hamiltonian, we calculate the expectation value for each point of the list, and the parameters yielding the best results are picked and used for further processes.
8. So, in this way, we obtain the best β and γ , and this is how we optimize the parameters in the Single Parameter Optimization.

```
def get_expectation(x, g, NUM_SHOTS=1024):
    gamma, beta = x
    print("Cost of Gamma: %s, beta: %s... " % (gamma, beta))
    q = QuantumRegister(g.N)
    c = ClassicalRegister(g.N)
    qc = QuantumCircuit(q, c)
    for i in range(g.N):
        qc.h(q[i])
    for edge in g.get_edges():
        u, v, w = edge
        qc.cx(q[u], q[v])
        qc.u1(gamma*w, q[v])
        qc.cx(q[u], q[v])
    for i in range(g.N):
        qc.h(q[i])
        qc.u1(-2*beta, q[i])
        qc.h(q[i])
    for i in range(g.N):
        qc.measure(q[i], c[i])
    qc.draw()
```

Fig. 4.15 - Code of the Quantum Approximate Optimization Algorithm

Output of Single Parameter Optimization

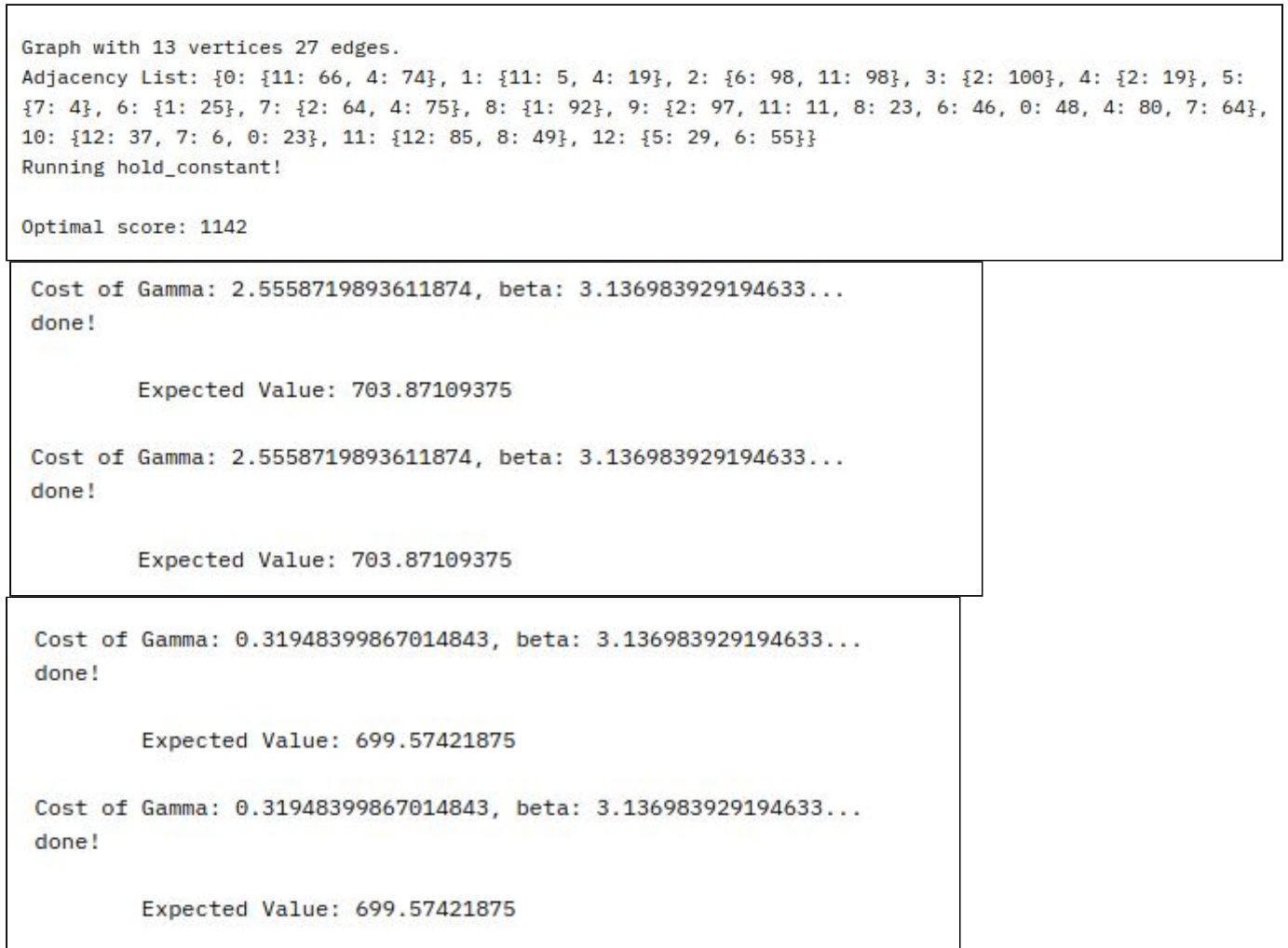


Fig. 4.16 - Output of Single Parameter Optimization

Here, we vary γ and keep the other parameter constant, algorithms yield a lower accuracy than expected.

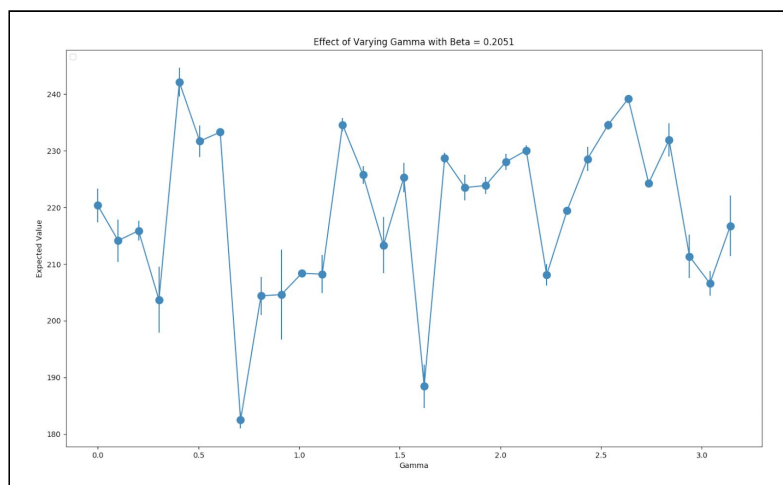


Fig. 4.17 - Effect of Varying Gamma with Beta

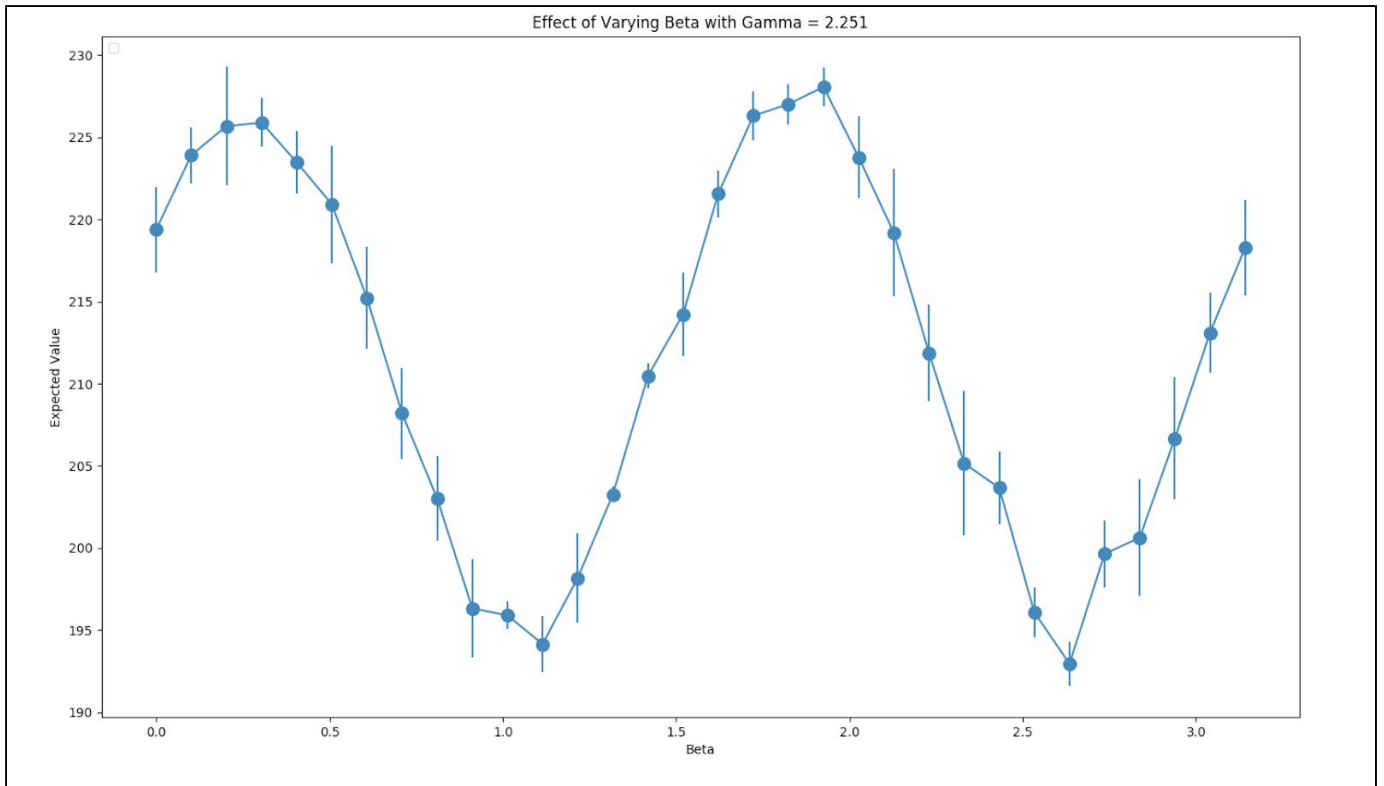


Fig. 4.18 - Effect of Varying Beta with Gamma

In the above graphs, you can observe that how the expectation value (on the y-axis) varies when one parameter is varied while the other parameter is kept constant, the parameters at the local maxima are selected for the further process, but the accuracy of the process was low, so we had to perform double parameter optimization.

Working Procedure of Double Parameter Optimization

1. We started with Double Parameter Optimization in a similar way to Single Parameter Optimization, we created a graph of edges, and added random edges to the graph. We then calculated the optimal cut for the graph, and got the best cut possible from the graph.
2. We then decided on the number of runs for the code, it was decided to be 60.
3. We created two different lists for β and γ , where both are equally varied over their fixed constant intervals.
4. We then create a list of lists, which is known as points, where the first parameter is γ , and the second parameter is β .
5. After creating the list of points, we pass the list to Quantum Approximate Optimization Algorithm, which first prepares the Cost Hamiltonian and Driver Hamiltonian, which is a function of the parameters passed earlier.
6. Number of Input Qubits are decided, which is equal to number of nodes in the Input Graph, after Input Qubits are decided, we apply the Hamiltonian to each Node of the Graph.
7. After applying Hamiltonian, we calculate the expectation value for each point of the list, and the parameters yielding the best results are picked and used for further processes.

8. So, in this way, we obtain the best β and γ , and this is how we optimize the parameters in the Double Parameter Optimization, QAOA Algorithm used is similar to the earlier Single Parameter Optimization.

Output of Double Parameter Optimization

```
Graph with 4 vertices 6 edges.
Adjacency List: {0: {1: 81, 3: 23}, 1: {2: 34}, 2: {0: 55, 3: 4}, 3: {1: 15}}
Running hold_constant!

Optimal score: 159
```

```
148.453125
148.453125
148.453125
Best Cut : 1000
```

Fig. 4.19 - Output of Double Parameter Optimization

Here, we vary both β and γ , this leads to higher accuracy, and hence, we select this process for further steps to optimize the parameter selection. We also plot 3-D Graph to make the selection of beta and gamma, easier to understand.

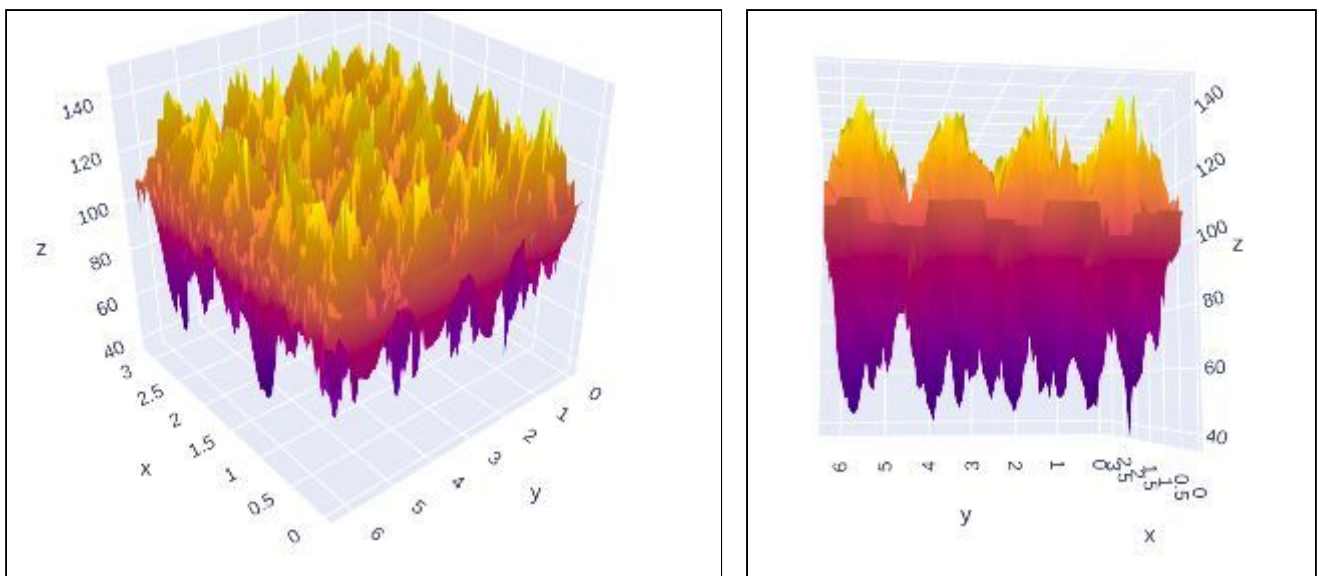


Fig. 4.20 - 3-D Graph plotting Beta, Gamma and Expectation Value

Here, we can observe that the peak of the graph shows the best Beta and Gamma values, and we use that value for the further processes.

4.5. Parameter Tuning

- After deciding on the method to optimize the parameters, we begin the algorithm to solve the NP-Complete MaxCut Problem.
- So, for the first step, we run the graph on a random 4 Node Graph to generate β and γ for the further process.
- In order to speed up the process, we stop the process when the predicted value is greater than 95% of the actual value.
- We then use β and γ for the interval construction process.

```
Graph with 4 vertices 6 edges.
Adjacency List: {0: {}, 1: {0: 83, 2: 95}, 2: {0: 72, 3: 96}, 3: {0: 23, 1: 61}}
Running hold_constant!

Optimal score: 312

0.0
0.0
215.1171875
-----
0.06346651825433926
0.0
221.046875
-----
0.06346651825433926
0.0
221.046875

3.5541250222429985
2.3482611754105527
280.296875
-----
3.5541250222429985
2.3482611754105527
280.296875
-----
0.6346651825433925
2.729060284936588
284.4609375
-----
Accuracy achieved
```

Fig. 4.21 - Parameter Tuning, after the parameter tuning process, we obtain the value of γ and β as 0.634466 and 2.7290 respectively

4.6. Interval Construction and Prediction of MaxCut Value

- This is the implementation of the novel algorithm proposed earlier.
- This algorithm predicts the value of MaxCut, and the Interval in which the value may lie.

Step - 1 Getting the Accuracy Value, and Mean Accuracy Value

- We generate 100 random graphs for a number of vertices, and using the parameters derived earlier, we run the QAOA Algorithm on all the graphs, and measure the Accuracy Value.
- We average out all the values of Accuracy Value to get the Mean Accuracy Value.
- We invert the MAV obtained to get the multiplicative factor.

```
708
485.4066162109375
0.6856025652696858
-----
1274
1018.47021484375
0.7994271702070251
-----
3387
2693.5440673828125
0.7952595415951617
-----
2078
1558.1943359375
0.7498529046859962
-----
1595
1230.3929443359375
0.7714062346933778
-----
74.74232436599634
```

Fig. 4.22 - Obtaining Accuracy Values, and Mean Accuracy Value.

- After obtaining the MAV as 74.74%, we invert the value to get the multiplicative factor, so the multiplicative factor as 1.337971635

Step - 2 : Multiplicative Factor

- After obtaining the multiplicative factor, we run the code with the same graphs, but now we multiply the predicted result with the Multiplicative Factor, to make sure that our answer is more accurate.
- It turns out that the answer is more accurate, but there is a case of overflow, in some cases, the predicted value multiplied by the Multiplicative Factor becomes greater than the actual value, so we need to introduce the concept of deviations in order to normalize the process.

- So, in the next step, we will observe how the predictions deviate from the predicted value, and what we can do to increase the accuracy.

Step - 3 : Deviations

- Using the formula defined earlier, we calculate the positive deviations and negative deviations of the process.
- It has been observed that the positive deviations range from 0.2% to 18%, and the negative deviations range from -0.5% to 9.7%
- So, we track the deviations and maintain the count of the respective deviations.
- After maintaining the list of deviations, we pick out the 95th percentile of positive and negative deviations and perform the next step

1381	1952
1378.068885010495	2023.8164765283188
0.2122458355905124	-3.6791227729671543
-----	-----
2146	500
2210.5583329048372	450.10776181207586
-3.008310014204904	9.978447637584827
-----	-----
1768	2158
1793.5830491703423	2205.1270494267415
-1.4470050435713966	-2.1838299085607717

	3257
	3477.1444473840393
	-6.7591172055277635

Fig. 4.23 - Observing the Positive and Negative Deviations

Step - 4 : Confidence Interval

- After obtaining the 95th percentile of positive and negative deviations, we get the values of $(1 - x)$ and $(1 + y)$ where x is the 95th percentile of positive and y is the 95th percentile of negative deviation.
- After getting the values, we multiply $(1-x)$ and $(1+y)$ to earlier predicted values, to get the confidence interval in which the values lie.

95th Percentile of Deviations
15.503117854853254
-8.294498519990082

Fig. 4.24 - 95th Percentile of Deviations

- As you can observe, the 95th Percentile of Positive Deviation is 15.503, and 95th Percentile of Negative Deviation is -8.29, so we multiply the predicted values with
 - $(1 - 0.15503) = 0.845$
 - $(1 + 0.82944) = 1.082944$
- After multiplying the predicted values, we have a 95% accuracy in predicting the interval of MaxCut Values.

```

True
1874
1571.5031086545014
2014.0996537874123
-----
True
3029
2777.0257765650067
3559.1445058782765
-----
True
3382
3060.6474349172363
3922.645081058765
-----
Correct Predictions : 100/100

```

Fig. 4.25 - Confidence Interval - Upper and Lower Bounds

- In the figure given above, the first value is the actual MaxCut value, the second value is lower bound of interval, and the third value is upper bound of the interval.

Step - 5 : MaxCut Value

- After getting the interval of the MaxCut, we have to predict the accurate value of the maxcut, we have the lower bounds and upper bounds of the interval, and the factors $(1-x)$ and $(1+y)$ from the earlier stages.
- So, we observe the maximum deviations from the lists, and multiply them with the count of the opposite deviation and add them up to get the final factor, which has to be multiplied with the predicted value.
 - If the maximum positive deviation is 20, maximum negative deviation is 10, and the count of positive deviations is 40, and count of negative deviations is 60.
 - So, the final factor will be $(1-0.20)*60 + (1 + 0.10) * 40 = 0.92$
- After getting the final factor, we run the experiment, multiply it with the MAV derived earlier, and multiply it with the factor derived, and that serves as the final prediction value.
- Results from the experiment have been discussed later.

Predicted Value :3531.8579988084484	Predicted Value :2705.1917057431538
Actual Value : 3564	Actual Value : 2731
Accuracy : 99.09814811471516%	Accuracy : 99.05498739447651%
Predicted Value :2127.617908936748	Predicted Value :1857.3387971734837
Actual Value : 2181	Actual Value : 1892
Accuracy : 97.55240297738413%	Accuracy : 98.16801253559639%
Predicted Value :3376.962822854363	Predicted Value :1665.3290577172534
Actual Value : 3408	Actual Value : 1770
Accuracy : 99.08928470816792%	Accuracy : 94.08638744165273%
Predicted Value :1947.0588408751073	
Actual Value : 2066	
Accuracy : 94.24292550218331%	

Fig. 4.22 - Prediction of MaxCut Values, Comparison with Actual Values, and Computing Accuracy.

Results, and Inferences for the Algorithm

1. Best values for γ and β turned out to 0.6346651825433925 and 2.729060284936588.
2. After using these values, we ran the algorithm and found out the accuracy without any modification was 74.72%
3. After we derived the accuracy value, we inverted it to get the MAV, which is 1.337971635.
4. We observed the deviations, and found out that positive deviations were lesser than negative deviations, but the quantity of negative deviations was lesser than positive deviations.
5. For the confidence interval, the values of (1-x) and (1+y) turned out to be 0.823416972 and 1.092848123 respectively.
6. In the last step, we got the final factor, as 0.912495912.
7. So, the most accurate prediction of the MaxCut will occur when QAOA runs for the Graph, and is multiplied by the factor of MAV * Final Factor, which is $1.337971634 * 0.912495912 = 1.2208936464$
8. The accuracy of the Approximation Ratio after all the steps was found out to be 95.87% which was better than some of the existing algorithms.

Table 4.1 - Final Summary of the Algorithm

Parameters (γ and β)	(0.6346, 2.7290)
Initial Accuracy	74.72%
MAV Factor	1.3379
Maximum Positive Deviation	21%
Maximum Negative Deviation	10.2%
95 th Percentile of Positive and Negative Deviations	14.5%, 9.7%
Final Factor	1.22089
Final Accuracy	[92.5%, 95.6%]

5. CONCLUSION

- The field of quantum computing is growing rapidly as many of today's leading computing groups, universities, colleges, and all the leading IT vendors are researching the topic. This pace is expected to increase as more research is turned into practical applications. In the Budget 2020, we observed the Government allocating ₹ 8000 crore for research in Quantum computing which shows the recognition of importance of this area of study.
- In this project, we explored a complete new realm of the computer science field, understood its concepts from the basics and were finally able to produce successful algorithms that show a considerable amount of increase in speed of calculations. We worked on a new powerful platform, that is, IBM Quantum Experience which helped us make the basic circuits, provide us with the quantum capabilities and implement our ideas.
- We got the opportunity of working on a lot of experiments, where we started with implementation of logic gates and simpler circuits, later understanding the *QAOA* and implementing the algorithms under the guidance of our guides.
- Since the domain is fairly younger and has a lot of scope to grow, we will continue to gain more and more knowledge about the subject. This domain has not only given us a wider scope to look at algorithms but also made us curious to look forward and learn more about this field. There is a lot of scope of further optimizations and improvement in our understanding and will continue to do so.

6. FUTURE SCOPE OF WORK

- There is a fairly good amount of future scope in the work done. Since the research on quantum is in a nascent stage and still growing, it leaves a lot of area to be filled, specially optimizing the process even more and fully understanding the finer nuances of *QAOA* Algorithm.
- Future scope also includes certain key areas like improving upon the *QAOA* Algorithm that is more efficient than the current working algorithm. It requires more in depth understanding and formulating the Hamiltonians efficiently, and accurately that makes the algorithm faster by a considerable amount and also, improves its accuracy.
- We also worked on the different uses of MaxCut, and how the MaxCut value can be used in different fields like VLSI Design, and Theoretical Physics.
- We can also extend our solution and approach to different NP-Complete Problems like, Clique Problems, Vertex Cover Problems and Graph Coloring Problems.

REFERENCES

- [1] Grossu, Ioan. (2018). "Introduction to Quantum Computing". 10.13140/RG.2.2.29728.43524.
- [2] Yanofsky, Noson. (2007). "An Introduction to Quantum Computing." 10.1007/978-94-007-0080-2_10.
- [3] Singh, Prashant (2005). "A Study on the basics of Quantum Computing", arXiv:quant-ph/0511061
- [4] Barde, Nilesh & THAKUR, Deepak & Bardapurkar, Pranav & DALVI, Sanjaykumar. (2011). Consequences and Limitations of Conventional Computers and their Solutions through Quantum Computers. Leonardo Electronic Journal of Practices and Technologies. 10. 161-171.
- [5] Aradyamath, Poornima & M, Naghabhushana & Ujjinimatad, Rohitha. (2019). Quantum Computing Concepts with Deutsch Jozsa Algorithm. JOIV : International Journal on Informatics Visualization. 3. 10.30630/joiv.3.1.218.
- [6] Wan, Kwok Ho & Liu, Feiyang & Dahlsten, Oscar & Kim, M.. (2018). Learning Simon's quantum algorithm.
- [7] Tristan, Moore (2016) "Quantum Computing and Shor's Algorithm", https://sites.math.washington.edu/~morrow/336_17/2016papers/tristan.pdf
- [8] Zohuri, Bahman & Moghaddam, Masoud. (2017). What Is Boolean Logic and How It Works. 10.1007/978-3-319-53417-6_6.
- [9] Karp, Richard (1972). "Reducibility Among Combinatorial Problems", <https://people.eecs.berkeley.edu/~luca/cs172/karp.pdf>
- [10] E. Farhi, J. Goldstone, S. Gutmann, "A Quantum Approximate Optimization Algorithm", <https://arxiv.org/abs/1411.4028>
- [11] Sofeoul-Al-Mamun, Md & Miah, Md Badrul & Al Masud, Fuyad. (2017). A Novel Design and Implementation of 8-3 Encoder Using Quantum Technology. European Scientific Journal. 13. 154-164. 10.19044/esj.2017.v13n15p254.
- [12] Khan, Mozammel. (2006). Design of Quantum Ternary Multiplexer and Demultiplexer. Engineering Letters. 13.

ANNEXURES

Code Snippet for Quantum Encoder

```
%matplotlib inline
from qiskit import *
provider = IBMQ.load_account()
simulator = Aer.get_backend('qasm_simulator')

#Register Declaration
qi=QuantumRegister(8)
qb=QuantumRegister(2)
qo=QuantumRegister(3)
v=QuantumRegister(1)
c=ClassicalRegister(3)
circuit = QuantumCircuit(qi,qb,qo,v,c)
circuit.x(qi[5])

#Boolean Expression Simulation
or4(circuit,qi[4],qi[5],qi[6],qi[7],qb[0],qb[1],qo[2])
or4(circuit,qi[2],qi[3],qi[6],qi[7],qb[0],qb[1],qo[1])
or4(circuit,qi[1],qi[5],qi[3],qi[7],qb[0],qb[1],qo[0])

#Output Measurement and Circuit Drawing

circuit.measure(qo[2],c[2])
circuit.measure(qo[1],c[1])
circuit.measure(qo[0],c[0])
circuit.draw()
```

Code Snippet for Quantum Demultiplexer

```
%matplotlib inline
from qiskit import QuantumCircuit, execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from qiskit import *
provider = IBMQ.load_account()
simulator = Aer.get_backend('qasm_simulator')

#Register Declaration
qi = QuantumRegister(1)
qs = QuantumRegister(2)
qxs = QuantumRegister(2)
qb = QuantumRegister(1)
```

```

qo = QuantumRegister(4)
c = ClassicalRegister(4)
circuit = QuantumCircuit(qi,qb,qs,qxs,qo,c)

#Boolean Expression Implementation
circuit.x(qi[0])
circuit.x(qs[1])
circuit.x(qs[0])
for i in range(0,2):
    circuit.cx(qs[i],qxs[i])
    circuit.x(qxs[i])
and3(circuit,qxs[0],qxs[1],qi[0],qb[0],qo[0])
and3(circuit,qs[0],qxs[1],qi[0],qb[0],qo[1])
and3(circuit,qxs[0],qs[1],qi[0],qb[0],qo[2])
and3(circuit,qs[0],qs[1],qi[0],qb[0],qo[3])
circuit.measure(qo[0],c[3])
circuit.measure(qo[1],c[2])
circuit.measure(qo[2],c[1])
circuit.measure(qo[3],c[0])
circuit.draw()

#Visualization
list_0001 = list()
list_0100 = list()
list_1000 = list()
list_0010 = list()
for i in range(100,8200,400):
    job = execute(circuit,simulator,shots=i)
    result = job.result()
    counts = result.get_counts(circuit)
    for j in counts.keys():
        val = float(counts.get(str(j),"0"))
        ans = val/(i+1)
        if j=="0001":
            list_0001.append(ans)
        if j=="0100":
            list_0100.append(ans)
        if j=="0010":
            list_0010.append(ans)
        if j=="1000":
            list_1000.append(ans)
import matplotlib.pyplot as py
py.plot(list_0001)
py.plot(list_0010)
py.plot(list_0100)
py.plot(list_1000)

```

Graph Class

```
class Graph():
    def __init__(self, N, randomize=True):
        """ Initialize a random graph with N vertices. """
        self.N = N
        self.E = 0
        self.adj = {n:dict() for n in range(N)}
        self.currentScore = float('-inf')
        self.currentBest = ""
        self.runs = []
        if randomize:
            self.randomize()

    def randomize(self):
        """ Randomly generate edges for this graph. """

        # Generate list of tuples for all possible directed edges.
        all_possible_edges = set([(x,y) for x in range(self.N) for y in range(self.N) if x != y])

        # Sanity check, ensuring we generated the correct number of edges.
        e_gen = len(all_possible_edges) / 2
        e_shd = self.N * (self.N-1) / 2
        assert e_gen == e_shd, "%d != %d" % (e_gen, e_shd)

        # Choose a random number of edges for this graph to have.
        # Note, we stop at len/2 because we generated directed edges,
        # so each edge counts twice.
        num_edges = randint(1, len(all_possible_edges)/2)
        for i in range(num_edges):
            # Choose an edge, remove it and its directed complement from the list.
            e = choice(list(all_possible_edges))
            all_possible_edges.remove(e)
            all_possible_edges.remove(e[::-1])

            # Unpack tuple into vertex ints.
            u, v = int(e[0]), int(e[1])

            # Choose a random weight for each edge.
            weight = randint(1, 100)

            #weight = 1
            self.add_edge(u, v, weight)

    def add_edge(self, u, v, weight):
        """ Add an edge to the graph. """
```

```

self.E += 1
self.adj[u][v] = weight

def get_edges(self):
    """ Get a list of all edges. """
    edges = []
    for u in self.adj:
        for v in self.adj[u]:
            edges.append((u, v, self.adj[u][v]))
    return edges

def get_score(self, bitstring):
    """ Score a candidate solution. """
    assert len(bitstring) == self.N

    score = 0

    # For every edge u,v in the graph, add the weight
    # of the edge if u,v belong to different cuts
    # given this candidate solution.

    for u in self.adj:
        for v in self.adj[u]:
            if bitstring[u] != bitstring[v]:
                score += self.adj[u][v]
    return score

def optimal_score(self):
    """
    Returns (score, solutions) holding the best possible solution to the
    MaxCut problem with this graph.
    """

    best = 0
    best_val = []

    # Iterate over all possible candidate bitstrings
    # Note: the bitstrings from 0 - N/2 are symmetrically
    # equivalent to those above
    for i in range(ceil((2 ** self.N)/2)):
        # Convert number to 0-padded bitstring.
        bitstring = bin(i)[2:]
        bitstring = (self.N - len(bitstring)) * "0" + bitstring

        sc = self.get_score(bitstring)
        if sc > best:
            best = sc
            best_val = [bitstring]
        elif sc == best:

```



```

        best_val.append(bitstring)
    return best, best_val

def edges_cut(self, bitstring):
    """ Given a candidate solution, return the number of edges that this solution cuts. """
    num = 0
    for u in self.adj:
        for v in self.adj[u]:
            if bitstring[u] != bitstring[v]:
                num += 1
    return num

def update_score(self, bitstring):
    """ Scores the given bitstring and keeps track of best. """
    score = self.get_score(bitstring)
    if score > self.currentScore:
        self.currentScore = score
        self.currentBest = bitstring
        print(self.currentBest)
    return score

def clear_runs(self):
    """ Clear data from past runs. """
    self.currentScore = float('-inf')
    self.currentBest = ""
    self.runs = []

def add_run(self, gamma, beta, expected_value):
    """ Save the data from each run iteration. """
    self.runs.append([gamma, beta, expected_value])

def __str__(self):
    return "Graph with %d vertices %d edges.\nAdjacency List: %s" % (self.N, self.E, self.adj)

```

QAOA - Parameter Optimization

```

def get_expectation(x, g, NUM_SHOTS=1024):
    gamma, beta = x
    print("Cost of Gamma: %s, beta: %s... " % (gamma, beta))
    q = QuantumRegister(g.N)
    c = ClassicalRegister(g.N)
    qc = QuantumCircuit(q, c)
    for i in range(g.N):
        qc.h(q[i])
    for edge in g.get_edges():
        u, v, w = edge
        qc.cx(q[u], q[v])

```

```

qc.u1(gamma*w, q[v])
qc.cx(q[u], q[v])
for i in range(g.N):
    qc.h(q[i])
    qc.u1(-2*beta, q[i])
    qc.h(q[i])
for i in range(g.N):
    qc.measure(q[i], c[i])
qc.draw()
backend = BasicAer.get_backend("qasm_simulator")
job = execute(qc, backend, shots=NUM_SHOTS)
results = job.result()
result_dict = results.get_counts(qc)
print("done!\n")

# Calculate the expected value of the candidate bitstrings.
exp = 0
for bitstring in result_dict:
    prob = np.float(result_dict[bitstring]) / NUM_SHOTS
    score = g.update_score(bitstring)

    # Expected value is the score of each bitstring times
    # probability of it occurring.
    exp += score * prob

print("\tExpected Value: %s\n" % (exp))
g.add_run(gamma, beta, exp)

return exp

def hold_constant(vary="gamma"):
    """ Plots expected value vs. gamma/beta, holding the rest of the variables constant."""
    # Choose some random starting beta/gamma and graph.
    lim = np.pi if vary == "gamma" else 2*np.pi
    constant_var = uniform(0, lim)
    g = Graph(13)
    print(g)
    # RUNS # of runs at each gamma for error bars.
    RUNS = 3

    # Keep track of gammas, expected values, for plotting.
    pts, exp, std = [], [], []

    # The maximum possible expected value is the maximum possible weighted cut.
    opt = g.optimal_score()[0]
    print("Running hold_constant!\n")
    print("Optimal score: %s\n" % (opt))

    # Number of data points to collect.

```

```

NUM_RUNS = 60
MIN = 0
MAX = 2*np.pi if vary == "gamma" else np.pi

points = np.linspace(MIN, MAX, NUM_RUNS)
for point in points:
    pts.append(point)

    # Calculate expected values.
    vals = []
    for i in range(RUNS):
        # Params are passed in as gamma, beta, so order matters.
        params = [point, constant_var] if vary == "gamma" else [constant_var, point]
        vals.append(get_expectation(params, g))

    # Calculate mean, standard deviation.
    exp.append(mean(vals))
    std.append(stdev(vals))

print("Best Cut : " + g.currentBest)
fig, ax = plt.subplots()

ax.errorbar(x=pts, y=exp, yerr=std, fmt='o-', markersize=10)
ax.legend(loc=2)

# Names for plotting.
vary_name = "Gamma" if vary == "gamma" else "Beta"
const_name = "Beta" if vary_name == "Gamma" else "Gamma"

ax.set_title("Effect of Varying %s with %s = %s" % (vary_name, const_name, constant_var))
ax.set_xlabel("%s" % (vary_name))
ax.set_ylabel("Expected Value")
plt.show()

hold_constant()

```

QAOA - Double Parameter Optimization

```

maxExp = 0.0
def get_expectation(x, g, NUM_SHOTS=128):
    # Look for progress bar as a global variable.

    gamma, beta = x
    # Construct quantum circuit.
    q = QuantumRegister(g.N)

```

```

c = ClassicalRegister(g.N)
qc = QuantumCircuit(q, c)

# Apply hadamard to all inputs.
for i in range(g.N):
    qc.h(q[i])

# Apply V for all edges.
for edge in g.get_edges():
    u, v, w = edge

    # Apply CNOTs.
    qc.cx(q[u], q[v])

    qc.u1(gamma*w, q[v])

    # Apply CNOTs.
    qc.cx(q[u], q[v])

# Apply W to all vertices.
for i in range(g.N):
    qc.h(q[i])
    qc.u1(-2*beta, q[i])
    qc.h(q[i])

# Measure the qubits (avoiding ancilla).
for i in range(g.N):
    qc.measure(q[i], c[i])

# Run the simulator.
backend = BasicAer.get_backend("qasm_simulator")
job = execute(qc, backend, shots=NUM_SHOTS)
results = job.result()
result_dict = results.get_counts(qc)
# Calculate the expected value of the candidate bitstrings.
exp = 0
global maxExp
for bitstring in result_dict:
    prob = np.float(result_dict[bitstring]) / NUM_SHOTS
    score = g.update_score(bitstring)

    # Expected value is the score of each bitstring times
    # probability of it occurring.
    exp += score * prob
    if(exp>maxExp):
        maxExp = exp
g.add_run(gamma, beta, exp)
print(maxExp)

```

```

    return exp
ptsB = []
ptsG = []
vals = []
pointsB = []
pointsG = []
def hold_constant():
    """ Plots expected value vs. gamma/beta, holding the rest of the variables constant."""
    # Choose some random starting beta/gamma and graph.
    global ptsB
    global ptsG
    global pointsB
    global pointsG
    MINB = 0
    MAXB = np.pi

    g = Graph(4)
    print(g)
    # RUNS # of runs at each gamma for error bars.
    RUNS = 3

    # Keep track of gammas, expected values, for plotting.
    pts, exp, std = [], [], []

    # The maximum possible expected value is the maximum possible weighted cut.
    opt = g.optimal_score()[0]
    print("Running hold_constant!\n")
    print("Optimal score: %s\n" % (opt))

    # Number of data points to collect.
    NUM_RUNS = 100
    ans = list(list())
    MIN = 0
    MAX = 2*np.pi
    pointsB = np.linspace(MINB, MAXB, NUM_RUNS)
    pointsG = np.linspace(MIN, MAX, NUM_RUNS)
    for i in range(100):
        for j in range(100):
            ptsB.append(pointsB[i])
    for i in range(100):
        for j in range(100):
            ptsG.append(pointsG[j])
    for i in itertools.product(pointsB, pointsG):
        pts.append(i)
    for i in range(len(pts)):
        # Calculate expected values.
        params = pts[i]
        vals.append(get_expectation(params, g))
    print("Best Cut : " + g.currentBest)

```

```
hold_constant()
```

Solution to the NP - Complete Problem of MaxCut - Part 1 - Parameter Tuning

```
maxB = 0.0
maxG = 0.0
maxExp = 0.0
def get_expectation(x, g, opt, NUM_SHOTS=128):
    # Look for progress bar as a global variable.
    global maxB
    global maxG
    gamma, beta = x
    # Construct quantum circuit.
    q = QuantumRegister(g.N)
    c = ClassicalRegister(g.N)
    qc = QuantumCircuit(q, c)
    # Apply hadamard to all inputs.
    for i in range(g.N):
        qc.h(q[i])
    # Apply V for all edges.
    for edge in g.get_edges():
        u, v, w = edge
        # Apply CNOTs.
        qc.cx(q[u], q[v])
        qc.u1(gamma*w, q[v])
        # Apply CNOTs.
        qc.cx(q[u], q[v])
    # Apply W to all vertices.
    for i in range(g.N):
        qc.h(q[i])
        qc.u1(-2*beta, q[i])
        qc.h(q[i])
    # Measure the qubits (avoiding ancilla).
    for i in range(g.N):
        qc.measure(q[i], c[i])
    # Run the simulator.
    backend = BasicAer.get_backend("qasm_simulator")
    job = execute(qc, backend, shots=NUM_SHOTS)
    results = job.result()
    result_dict = results.get_counts(qc)
    # Calculate the expected value of the candidate bitstrings.
    exp = 0
    global maxExp
    for bitstring in result_dict:
        prob = np.float(result_dict[bitstring]) / NUM_SHOTS
        score = g.update_score(bitstring)
        exp += score * prob
    if(exp>maxExp):
```

```

        maxB = beta
        maxG = gamma
        maxExp = exp
    print(maxB)
    print(maxG)
    print(maxExp)
    print("-----")
    g.add_run(gamma, beta, exp)
    return exp
ptsB = []
ptsG = []
vals = []
pointsB = []
pointsG = []
def hold_constant():
    """ Plots expected value vs. gamma/beta, holding the rest of the variables constant."""
    global ptsB
    global ptsG
    global maxExp
    global pointsB
    global pointsG
    MINB = 0
    MAXB = np.pi

    g = Graph(4)
    print(g)
    # RUNS # of runs at each gamma for error bars.
    RUNS = 3

    # Keep track of gammas, expected values, for plotting.
    pts, exp, std = [], [], []

    # The maximum possible expected value is the maximum possible weighted cut.
    opt = g.optimal_score()[0]
    print("Running hold_constant!\n")
    print("Optimal score: %s\n" % (opt))

    # Number of data points to collect.
    NUM_RUNS = 100
    ans = list(list())
    MIN = 0
    MAX = 2*np.pi
    pointsB = np.linspace(MINB, MAXB, NUM_RUNS)
    pointsG = np.linspace(MIN, MAX, NUM_RUNS)
    for i in range(100):
        for j in range(100):
            ptsB.append(pointsB[i])
    for i in range(100):
        for j in range(100):

```

```

        ptsG.append(pointsG[j])
    for i in itertools.product(pointsB,pointsG):
        pts.append(i)
    for i in range(len(pts)):
        # Calculate expected values.
        params = pts[i]
        exp = vals.append(get_expectation(params,g,opt))
        if(maxExp>=0.9*opt):
            print("Accuracy achieved")
            break
    hold_constant()

```

Solution to the NP - Complete Problem of MaxCut - Part 2 - Interval Construction and Value Prediction

```

count = 0
meanAcc = 0.0
def get_expectation(x,n):
    global count
    global meanAcc
    g = Graph(n)
    beta, gamma = x
    q = QuantumRegister(g.N)
    c = ClassicalRegister(g.N)

    qc = QuantumCircuit(q, c)
    for i in range(g.N):
        qc.h(q[i])
    for edge in g.get_edges():
        u, v, w = edge
        qc.cx(q[u], q[v])
        qc.u1(gamma*w, q[v])
        qc.cx(q[u], q[v])
    for i in range(g.N):
        qc.h(q[i])
        qc.u1(-2*beta, q[i])
        qc.h(q[i])
    for i in range(g.N):
        qc.measure(q[i], c[i])
    backend = BasicAer.get_backend("qasm_simulator")
    job = execute(qc, backend, shots=8192)
    results = job.result()
    result_dict = results.get_counts(qc)
    exp = 0
    for bitstring in result_dict:
        prob = np.float(result_dict[bitstring]) / 8192
        score = g.update_score(bitstring)

```



```

    exp += score * prob
# Calculate the expected value of the candidate bitstrings.
v=list(result_dict.values())
k=list(result_dict.keys())
print(g.optimal_score()[0])
print(exp)
print(float(exp/g.optimal_score()[0]))
meanAcc+=float(exp/g.optimal_score()[0])
print("-----")
if(g.get_score(k[v.index(max(v))])==g.optimal_score()[0]):
    count+=1
params = [5.711986642890533,0.8250647373064104]
for i in range(100):
    n = random.randint(5,15)
    get_expectation(params,n)
print(meanAcc)
print("-----")
print("Correct Answer Count by Approach 1 : " + str(count))

count = 0
meanAcc = 0.0
pDev = list()
nDev = list()
def get_expectation(x,n,val):
    global pDev
    global nDev
    global count
    global meanAcc
    g = Graph(n)
    beta, gamma = x
    q = QuantumRegister(g.N)
    c = ClassicalRegister(g.N)
    qc = QuantumCircuit(q, c)
    for i in range(g.N):
        qc.h(q[i])
    for edge in g.get_edges():
        u, v, w = edge
        qc.cx(q[u], q[v])
        qc.u1(gamma*w, q[v])
        qc.cx(q[u], q[v])
    for i in range(g.N):
        qc.h(q[i])
        qc.u1(-2*beta, q[i])
        qc.h(q[i])
    for i in range(g.N):
        qc.measure(q[i], c[i])
    backend = BasicAer.get_backend("qasm_simulator")
    job = execute(qc, backend, shots=8192)
    results = job.result()

```

```

result_dict = results.get_counts(qc)
exp = 0
for bitstring in result_dict:
    prob = np.float(result_dict[bitstring]) / 8192
    score = g.update_score(bitstring)
    exp += score * prob
# Calculate the expected value of the candidate bitstrings.
v=list(result_dict.values())
k=list(result_dict.keys())
print(g.optimal_score()[0])
print(exp*val)
o = (float((g.optimal_score()[0]-exp*val)/g.optimal_score()[0])*100)
if o>=0:
    pDev.append(o)
else:
    nDev.append(o)
print(o)
print("-----")
if(g.get_score(k[v.index(max(v))])==g.optimal_score()[0]):
    count+=1
params = [5.711986642890533,0.8250647373064104]
factor = 1/(0.7477343509947454)
for i in range(100):
    n = random.randint(5,15)
    get_expectation(params,n,factor)
print(pDev)

count = 0
def get_expectation(x,n,val,a,b):
    global count
    global pDev
    global nDev
    global count
    global meanAcc
    g = Graph(n)
    beta, gamma = x
    q = QuantumRegister(g.N)
    c = ClassicalRegister(g.N)
    qc = QuantumCircuit(q, c)
    for i in range(g.N):
        qc.h(q[i])
    for edge in g.get_edges():
        u, v, w = edge
        qc.cx(q[u], q[v])
        qc.u1(gamma*w, q[v])
        qc.cx(q[u], q[v])
    for i in range(g.N):
        qc.h(q[i])
        qc.u1(-2*beta, q[i])

```

```

    qc.h(q[i])
for i in range(g.N):
    qc.measure(q[i], c[i])
backend = BasicAer.get_backend("qasm_simulator")
job = execute(qc, backend, shots=8192)
results = job.result()
result_dict = results.get_counts(qc)
exp = 0
for bitstring in result_dict:
    prob = np.float(result_dict[bitstring]) / 8192
    score = g.update_score(bitstring)
    exp += score * prob
# Calculate the expected value of the candidate bitstrings.
v=list(result_dict.values())
k=list(result_dict.keys())
if (exp*val*a<=g.optimal_score()[0] and g.optimal_score()[0]<=(exp*val*b)):
    print("-----")
    print("True")
    print(g.optimal_score()[0])
    print(exp*val*a)
    print(exp*val*b)
    count+=1
else:
    print("-----")
    print("False")
    print(g.optimal_score()[0])
    print(exp*val*a)
    print(exp*val*b)
params = [5.711986642890533,0.8250647373064104]
factor = 1/(0.7444540376466091)
for i in range(100):
    n = random.randint(10,16)
    get_expectation(params,n,factor,0.84496882145,1.0829449852)
print("-----")
print("Correct Predictions : " +str(count) +"/100")

count = 0
ans = 0
import random
def get_expectation(x,n,f):
    global ans
    global count
    global pDev
    global nDev
    global count
    global meanAcc
    g = Graph(n)
    beta, gamma = x
    q = QuantumRegister(g.N)

```

```

c = ClassicalRegister(g.N)
qc = QuantumCircuit(q, c)
for i in range(g.N):
    qc.h(q[i])
for edge in g.get_edges():
    u, v, w = edge
    qc.cx(q[u], q[v])
    qc.u1(gamma*w, q[v])
    qc.cx(q[u], q[v])
for i in range(g.N):
    qc.h(q[i])
    qc.u1(-2*beta, q[i])
    qc.h(q[i])
for i in range(g.N):
    qc.measure(q[i], c[i])
backend = BasicAer.get_backend("qasm_simulator")
job = execute(qc, backend, shots=8192)
results = job.result()
result_dict = results.get_counts(qc)
exp = 0
for bitstring in result_dict:
    prob = np.float(result_dict[bitstring]) / 8192
    score = g.update_score(bitstring)
    exp += score * prob
# Calculate the expected value of the candidate bitstrings.
v=list(result_dict.values())
k=list(result_dict.keys())
if((exp*val*f/g.optimal_score()[0])>1.00999999):
    print("Violation")
ans=ans + exp*val*0.917475 /g.optimal_score()[0]
print("Predicted Value : " + str(exp*val*f))
print("Actual Value : " + str(g.optimal_score()[0]))
print("Accuracy : " + str(exp*val*f*100/g.optimal_score()[0])+"%")
params = [5.711986642890533,0.8250647373064104]
factor = 1/(0.7444540376466091)
for i in range(50):
    n = random.randint(10,16,0.917475)
    get_expectation(params,n,factor)
print("-----")
print("Mean Accuracy " + str(ans*2)+" %")

```