State-Overlap Hamiltonians in QAOA:

Performance Advantage for Quantum Machine Learning

Shaun Radgowski

December 18th, 2020

Senior Thesis in Physics

Yale University

Advisor: Dr. Shruti Puri, Assistant Professor

Department of Applied Physics

Contents

Abstract

This paper tackles the problem of using a quantum computer to find ground state energies and eigenstates of many-body quantum systems. Currently, a number of heuristic variational quantum-classical hybrid algorithms—including the Variational Quantum Eigensolver (VQE)[1] and the Quantum Approximate Optimization Algorithm (QAOA)[2]—are available for near-term application to this issue, but their performance capacities are uncertain. This paper first dissects the combinatorial optimization problem addressed by QAOA, then explores a new underlying cost function based on state overlaps instead of energy expectation values to employ in the these algorithms. Then, it uses Python libraries (including Qutip[3] and Cirq[4]) to simulate the performance of this new algorithm on example problems, while benchmarking against traditional QAOA in finding ideal optimizations. We ultimately found an advantage in some situations for layer-by-layer optimization of QAOA, including for $p > 3$ layers in the Ring of Disagrees and $p > 6$ layers in regular MaxCut Three graphs.

[1] Peruzzo et al. A variational eigenvalue solver on a quantum processor. arXiv:1304.3061 [quant-ph]

[2] Farhi et al. A Quantum Approximate Optimization Algorithm. arXiv:1411.4028 [quant-ph]

[3] Johansson et al. QuTiP 2: A Python framework for the dynamics of open quantum systems. [DOI: 10.1016/j.cpc.2012.11.019]

[4] Fingerhuth et al. Open source software in quantum computing. arXiv:1812.09167 [quant-ph]

I. Problem Statement

Quantum States

Many-body quantum states can be conceptualized as a collection of objects, each with a choice between two different "settings." While quantum physicists often reference these settings as spins, they can also be conceptualized as any binary choice: a switch flipped up or down, or a lightbulb turned on or off, or even an ice cream cone with a flavor of chocolate or vanilla. In the classical world, choices from these settings are all necessarily discrete, meaning that each object may exist in only one state at a time. The lightbulb can be either on *or* off (but not both), and the ice cream cone can be either chocolate *or* vanilla (but no combination of the two). This paradigm, however, does not translate to the quantum world; quantum objects are instead allowed to exist in a superposition of the two settings. A quantum switch can be both flipped up *and* flipped down simultaneously until a measurement is made—at that point, the quantum state collapses into one of the two options forevermore.

Through the lens of linear algebra, measuring a quantum state (represented in that world as a column vector) is equivalent to projecting it onto one of its eigenstates. Looking at a quantum lightbulb forces it to choose either on or off, where it will then remain until something changes it. In the canonical example, opening the bunker will make reality decide the fate of Schrödinger's favorite pet, and then however many times we look thereafter, the result will be the same. The probability of which eigenstate the column vector is projected into is a function of the coefficients of each basis state in the superposition. To see how this works, we'll consider the basis states used in Quantum information and computation.

Classical computers use the discrete bit values of 0 (representing low voltage in a wire) and 1 (representing high voltage in a wire) for encoding all information. Quantum computers, contrastingly, have access to a much larger toolbox. By defining our two basis states as another binary set of $|0\rangle$ and $|1\rangle$, we can describe any state $|\psi\rangle$ of a quantum bit (portmanteau-ed as "qubit") as a linear combination of these two basis states.

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle \tag{1}$$

Here, we have adopted the Dirac notation standard of expressing column vectors as kets. The probability of the state collapsing into either of these eigenstates is proportional to the square of their coefficients (a^2 for $|0\rangle$, b^2 for $|1\rangle$). Accordingly, in order to follow the rules of probability, the state must be *normalized* so that the total probability adds up to 1.

$$P\{|0\rangle\} = \frac{a^2}{a^2+b^2}, \quad P\{|1\rangle\} = \frac{b^2}{a^2+b^2} \tag{2}$$

The actual mathematical result of measuring a state is the eigenvalue of the selected eigenstate. To revisit the metaphor from above, the measurement result of checking if a lightbulb is on or off is actually the photons (or lack thereof) reaching your retinas, not the

actual current of electrons moving through the bulb's filament. When $|\psi\rangle$ has collapsed to $|0\rangle$, the measurement result is 1 (the eigenvalue of $|0\rangle$); if the universe instead selects $|1\rangle$, the measurement result is -1 (the eigenvalue of $|1\rangle$).

A collection of these quantum objects, each with a state somewhere on the superposition spectrum between $|0\rangle$ and $|1\rangle$, can be said to have a certain energy value when all collapse to one basis state or the other and are passed through a Hamiltonian. The orientation of these objects that produces the lowest energy value is deemed the "ground state", while all states with higher energies are "excited states." Thus, the act of configuring a many-body quantum system to produce the highest or lowest energy possible for a given Hamiltonian becomes a problem of combinatorial optimization.

This class of optimization problems has actually proven quite challenging for classical computers. For a system of n objects with m settings (in our example, n qubits with 2 basis states), the classical computer will encounter a problem of $O(m^n)$ time complexity in optimizing the system. In other words, the computer would need to attempt every combination from the m^n possible combinations to find the ideal orientation—in the case of 10 qubits, that would mean $2^{10} = 1024$ different combinations of $|0\rangle$ and $|1\rangle$. Obviously, this family of optimizations is a very taxing challenge to be handed off to classical machine learning, and presents an opportunity for a quantum computing advantage.

MaxCut and the Ring of Disagrees

One particular class of problems that this optimization technique can be applied to is known as *MaxCut*[5]. These problems entail a web of n connected nodes, each with a degree d signifying a maximum number of connections to other nodes. For degree three (coined "MaxCut Three"), each node can have either one, two, or three edges connecting to other nodes. For two adjacent nodes j and k (meaning nodes that are at a distance of 1 connection away), there are only three possible subgraphs for the edge $\langle jk \rangle$:

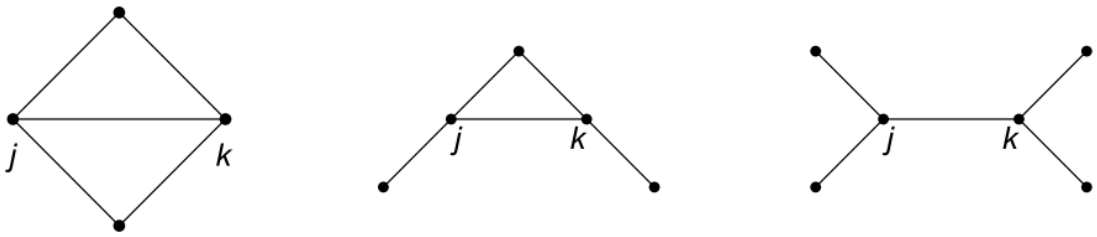


Figure 1: Three subgraphs of MaxCut Three

For graphs with connections like these, we can calculate a cost C from the edge set of $\{\langle jk \rangle\}$ according to a linear sum of each edge's cost:

[5]Eran Halperin et al. MAX CUT in cubic graphs. [*Journal of Algorithms* 53.2]

$$C = \sum_{\langle jk \rangle} \omega_{\langle jk \rangle} \; C_{\langle jk \rangle} \tag{3}$$

where $\omega_{\langle jk \rangle}$ is an optional "weight" parameter for the edge $\langle jk \rangle$ (assumed to all be 1 for all edges moving forward), and the cost of each edge is defined by the function:

$$C_{\langle jk \rangle} = \frac{1}{2}(1 - \delta_{z_j z_k}) \tag{4}$$

where $\delta_{z_j z_k} = 1$ when $z_j = z_k$ and 0 otherwise. This is typically known as the Kronecker Delta function, or the XNOR gate in classical computer science. In order to see what z_j signifies for qubit j, we can translate this cost function into the language of Linear Algebra to form a cost Hamiltonian:

$$H_C = \frac{1}{2} \sum_{\langle jk \rangle} (I - \sigma_j^{(z)} \sigma_k^{(z)}) \tag{5}$$

Here, $\sigma_i^{(z)}$ is the i^{th} cyclic permutation of the Pauli-Z operator in a tensor product chain of length n, where all other terms are the identity matrix. For example, for $n = 6$ qubits,

$$\sigma_1^{(z)} = Z \otimes I \otimes I \otimes I \otimes I \otimes I$$

$$\sigma_2^{(z)} = I \otimes Z \otimes I \otimes I \otimes I \otimes I$$

$$\sigma_3^{(z)} = I \otimes I \otimes Z \otimes I \otimes I \otimes I$$

and so forth, for $i \in \{1, 2, ..., n\}$.

The goal of this class of optimization problems is to *maximize* the cost function given by equation (4). This equates to finding a bitstring $z = z_1 z_2 z_3 ... z_n$ such that the sum C over all edges in the graph is as large as possible. For the cost Hamiltonian in equation (5), z_i would refer to the eigenvalue measured when σ_i^z collapses into one of its eigenstates in this computational basis. Considering that the two-dimensional eigenvectors of the Pauli-Z matrix are $[1, 0]$ and $[0, 1]$ (previously referenced as $|0\rangle$ and $|1\rangle$), z_i can be either +1 if σ_i^z collapses to $|0\rangle$ or -1 if σ_i^z collapses to $|1\rangle$ (or, more precisely, the eigenstate equivalents of these basis states for the higher-dimensional Hilbert space that these matrices are now operating in). The conclusion from this definition is that each individual term of C is maximized when $z_j \neq z_k$, as that would render $1 - \sigma_j^{(z)} \sigma_k^{(z)} = 1 - 0 = 1$. If instead $z_j = z_k$, the matrix product becomes 1 and the term flattens to 0.

For regular MaxCut graphs of degree 2, providing all nodes with exactly 2 edges forms a ring. Again applying the cost function in equation (4) and the insight from the previous paragraph, it is clear that the optimal solution to this graph is a series of antiferromagnetic couplings. Consider a graph like that shown in Figure 2: the cost function there is maximized

when qubits 1, 3, and 5 collapse to one eigenstate and qubits 2, 4, and 6 collapse to the other. This would yield a total cost of $C = 6$ after summing over all edges $\{\langle jk \rangle\}$, or more generally a maximum C value of n when n is even. When n is odd, there must exist an pair of adjacent qubits where $z_i = z_j$, reducing the maximum C value of $n - 1$. This problem is known as the Ring of Disagrees, named after these nearest-neighbor disagreement couplings.

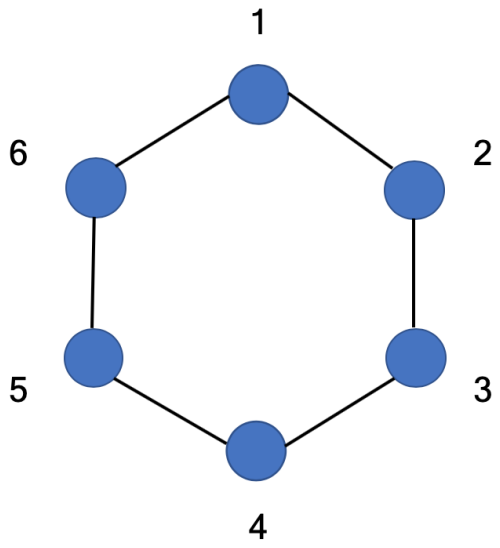


Figure 2: Regular MaxCut2 Ring

II. QAOA

The Existing Hybrid Solution

In their paper, Farhi, Goldstone, and Gutmann devise a variational quantum algorithm known as the Quantum Approximate Optimization Algorithm (QAOA) to address this class of problems. The motivating principle behind the algorithm's design is—to quote Voltaire—"Perfect is the enemy of good." This is to say, they acknowledge that previous efforts have perhaps been too focused on finding perfect, optimal solutions, when an *approximately* optimal solution would work almost as well while being easier to pinpoint.

Given the bitstring $z = z_1 z_2 z_3 ... z_n$ and a MaxCut graph with k edges, we have already seen that the cost function can be rewritten as a sum over all edges

$$C(z) = \sum_{\alpha=1}^{k} C_\alpha(z) \tag{6}$$

where $C_\alpha(z) = 1$ if z satisfies the disagreement clause for edge α and 0 otherwise. This is essentially the same statement as the cost Hamiltonian defined in equation (5). QAOA is

designed to find a string z for which $C(z)$ is as close as possible to the true maximum cost C^*. That is, a string z that maximizes the ratio $\frac{C(z)}{C^*}$. They begin by defining a unitary operator that utilizes the cost Hamiltonian and depends on an angle γ:

$$U(H_C, \gamma) = e^{-i\gamma H_C} = \prod_{\alpha=1}^{k} e^{-i\gamma C_\alpha} \tag{7}$$

Here, they restrict γ to be an angle in the range of 0 to 2π because C has integer eigenvalues. They then define a second operator B (the "Mixer Hamiltonian") which is the sum of all single-bit Pauli-X operators:

$$H_B = \sum_{j=1}^{n} \sigma_j^{(x)} \tag{8}$$

This Hamiltonian is also made into a unitary with a second variable angle, β:

$$U(H_B, \beta) = e^{-i\beta H_B} = \prod_{j=1}^{n} e^{-i\beta\sigma_j^{(x)}} \tag{9}$$

Here, β is restricted to be an angle in the range of 0 to π. Then, to run the algorithm, the quantum computer implements the following steps with p= the number of layers:

1. Begin with n qubits (one for each node in the graph) in the $|0\rangle$ state, for a full-register state of $|s_0\rangle = \otimes^n |0\rangle$
2. Place them all into the superposition state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ via a Hadamard gate on each, for a full-register state of $|s_1\rangle = \frac{1}{\sqrt{2^n}} \otimes^n (|0\rangle + |1\rangle)$
3. For each layer i of p layers:
    a. Select angles γ_i, β_i
    b. Apply the cost unitary $U(H_C, \gamma_i)$ to the state
    c. Apply the mixer unitary $U(H_B, \beta_i)$ to the state

This has defined two angle vectors, each of size p: $\vec{\gamma} = \gamma_1 ... \gamma_p$ and $\vec{\beta} = \beta_1 ... \beta_p$. The final angle-dependent state is now $|s_2\rangle = U(H_B, \beta_p)U(H_C, \gamma_p)...U(H_B, \beta_1)U(H_C, \gamma_1)\,|s_1\rangle$

4. Measure the expectation value $F_p(\vec{\gamma}, \vec{\beta})$ of the cost Hamiltonian H_C in this state:

$$F_p(\vec{\gamma}, \vec{\beta}) = \langle s_2 |\, H_C \,| s_2 \rangle \tag{10}$$

5. Use a classical computer[6] to maximize this expectation value by altering $\vec{\gamma}$ and $\vec{\beta}$:

$$M_p(\vec{\gamma}, \vec{\beta}) = \max F_p(\vec{\gamma}, \vec{\beta}) \tag{11}$$

[6]The algorithm is thus called a "hybrid" algorithm because it utilizes both quantum computation (for state manipulation/measurement) and classical computation (for the machine learning computation).

They then prove that this maximum expectation value M_p approaches the true maximum cost C^* as the number of layers p approaches infinity. While this paper will not dive into their mathematical proof for this, it is worth discussing the two different methods for optimizing $\vec{\gamma}$ and $\vec{\beta}$.

Optimization Methods

While not explicitly delineated in Fahri et al.'s paper, there are two main avenues toward optimization. We will call the first the *collective* method—the method that Fahri uses for QAOA. This involves the classical computer maximizing $F_p(\vec{\gamma}, \vec{\beta})$ all at once, over the $2p$ angles. This is the preferred method for producing the best possible expectation energy value, but it becomes a very taxing optimization calculation for the classical computer once p becomes large.

The second option is what we'll call the *iterative* method. In this method, the classical computer first optimizes $F_1(\gamma_1, \beta_1)$, then plugs those optimized angles (which we'll call γ_1^* and β_1^*) back into the optimization of the next pair of angles, $\max_{\gamma_2,\beta_2} F_2([\gamma_1^*, \gamma_2], [\beta_1^*, \beta_2])$. This continues iteratively, optimizing each $\{\gamma_i, \beta_i\}$ pair of the p pairs of angles separately. This method only involves two-variable optimization at any given time, and consequently runs significantly faster on classical computers. However, the resulting output energies are lower because this method introduces less entanglement, and maximizing each layer individually provides only an approximation for maximization of the entire system. In the next section, we will compare the perfomance of each method in finding optimal states, and explore combining their strengths to find an advantage.

Implementation

Our implementation of both the Ring of Disagrees and randomized MaxCut Three in Python is available in the Appendix. Each is designed as a class, with the core functionalities stated explicitly in the *ROD* class before being inherited into the *MaxCut* class. The arguments passed into the *ROD* class are:

- *N*: (positional) the number of qubits in the circuit/nodes in the ring
- *layers*: (positional) the number of QAOA layers to be applied (previously p)
- *approach*: (positional) which Linear Algebra optimization library from SciPy to use
- *method*: (keyword) as discussed above, the optimization method either "collective" or "iterative" (default "iterative")

The *MaxCut* class takes three additional positional arguments before the arguments above to design the randomized graph, namely:

- *min_conx*: integer specifying each node's minimum number of connections (usually 1)
- *max_conx*: integer specifying each node's maximum number of connections (a.k.a. d)

- *density*: decimal number specifying the average number of connections each node should have, somewhere between the minimum and maximum inclusive

Once an object of either class is initialized, traditional QAOA can be used to calculate its maximum energy by calling the method *analyze()*. This will first print the QAOA circuit to the terminal using the packages Cirq and SymPy, then it will randomly initialize each angle in $\vec{\gamma}$ and $\vec{\beta}$ from a uniform distribution between 0 and 1. It will then employ the Cost Hamiltonian, the Mixer Hamiltonian, and your preferred optimization library to maximize the cost function[7]. It will then print out the ratio of the final energy calculated with the learned parameters to the actual optimal energy (calculated with Qutip's *eigenstates* method on the Cost Hamiltonian). For an ideal algorithm, this ratio will be close to 1.

III. Power Iteration Cost Function

Underlying Theory

In redesigning the parameter-learning component of QAOA, our aim was to specifically improve the accuracy of the *iterative* optimization method. For any sate $|\psi\rangle = a\,|\lambda_1\rangle + b\,|\lambda_2\rangle$, it is known that applying a Hermitian operator H renders a new state:

$$H(a\,|\lambda_1\rangle + b\,|\lambda_2\rangle) = a\lambda_1\,|\lambda_1\rangle + b\lambda_2\,|\lambda_2\rangle \tag{12}$$

where λ_1 and λ_2 are the eigenvalues of $|\lambda_1\rangle$ and $|\lambda_2\rangle$, respectively. Normalizing this new state yields the probabilities of collapsing into each eigenstate:

$$P\{|\lambda_1\rangle\} = \frac{a^2\lambda_1^2}{a^2\lambda_1^2 + b^2\lambda_2^2}, \quad P\{|\lambda_2\rangle\} = \frac{b^2\lambda_2^2}{a^2\lambda_1^2 + b^2\lambda_2^2} \tag{13}$$

By definition, the higher-energy eigenstate will have a higher eigenvalue. A consequence of this fact is that repeatedly applying an operator to a state has the effect of *biasing* that state towards its highest-energy eigenstate, as the coefficient of that state will grow the fastest. This is to say, because of continued normalization,

$$\lim_{p\to\infty} H^p\,|\psi\rangle = \max\left(\frac{a^2\lambda_1^{2p}}{a^2\lambda_1^{2p} + b^2\lambda_2^{2p}}\,|\lambda_1\rangle\,,\ \frac{b^2\lambda_2^{2p}}{a^2\lambda_1^{2p} + b^2\lambda_2^{2p}}\,|\lambda_2\rangle\right) = \max\left(|\lambda_1\rangle\,, |\lambda_2\rangle\right) \tag{14}$$

For a circuit like QAOA, repeatedly applying the Cost Hamiltonian to the state will accordingly bias the state towards the maximum-energy eigenstate. Unfortunately, a Hermitian operator cannot directly be applied to a state unless it is unitary. To bridge this gap, we employed the same philosophy as Farhi et al. and noted that perhaps a unitary that is

[7] Due to the restraints from SciPy, the optimization function actually *minimizes* the opposite of the cost function. These are mathematically equivalent, so there's no harm done.

close to the Hamiltonian is good enough. In order to create a close-enough unitary, we first need to ensure that the $H|\psi\rangle$ state is normalized. As $H^\dagger = H$, a normalized state can be ensured with:

$$|\psi_N\rangle = \frac{H|\psi\rangle}{\sqrt{\langle\psi| H^\dagger H |\psi\rangle}} = \frac{H|\psi\rangle}{\sqrt{\langle\psi| H^2 |\psi\rangle}} \tag{15}$$

We can find the best unitary approximation $U(\vec{\gamma}, \vec{\beta})|\psi\rangle$ of this state $|\psi_N\rangle$ by minimizing the Fidelity (a measure of distance between two states) between them. If two states $|\phi_1\rangle$ and $|\phi_2\rangle$ are identical, then $|\langle\phi_1|\phi_2\rangle|^2 = 1$; this becomes our new minimization problem. Accounting for an overall phase factor of $e^{i\theta}$, we want this statement this statement to be true:

$$1 \approx e^{i\theta} \langle\psi| U^\dagger(\vec{\gamma}, \vec{\beta}) |\psi_N\rangle \tag{16}$$

To remove the effect of the phase factor, we can take the modulus squared of both sides. Our new cost function[8] consequently becomes, after eliminating the denominator:

$$\sqrt{\langle\psi| H^2 |\psi\rangle} - |\langle\psi| U^\dagger(\vec{\gamma}, \vec{\beta}) H |\psi\rangle| \tag{17}$$

We have dubbed this the **Power Iteration** cost function, due to its effect on the iterative method when the Hamiltonian is raised to a high power p (i.e. when there are multiple QAOA layers).

Implementation

Available in the Appendix after the first code block—under the subheading *Python Scripts for Power Iteration*—is a copy of our implementation of this new cost function. It includes classes *ROD_PI* (inheriting from *ROD*) and *MaxCut_PI* (inheriting from MaxCut). No novel arguments are passed when initializing objects of these classes, and the only updated class methods are *optimize()* and *analyze()*. The core procedure of each class are thus unchanged from their parent classes, aside from the optimization method: the circuit is drawn, the angles are randomly initialized then optimized, and the final energy is printed. Note that the *MaxCut_PI* class is a case of Multiple Inheritance, as it inherits most of its functionality from the original *MaxCut* class, except for substituting in the optimization function from *ROD_PI*.

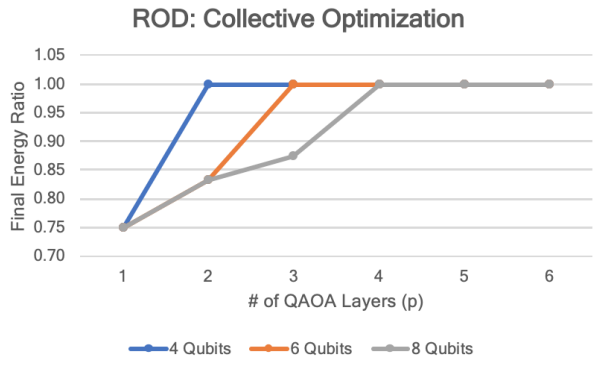
[8]For clarification, this usage of "cost function" is referencing the bottom line of the cost function used in optimization for $\vec{\gamma}$ and $\vec{\beta}$ (eg. lines 145-148 in the Appendix). The cost function of the ensemble—equation (6)—remains unchanged.

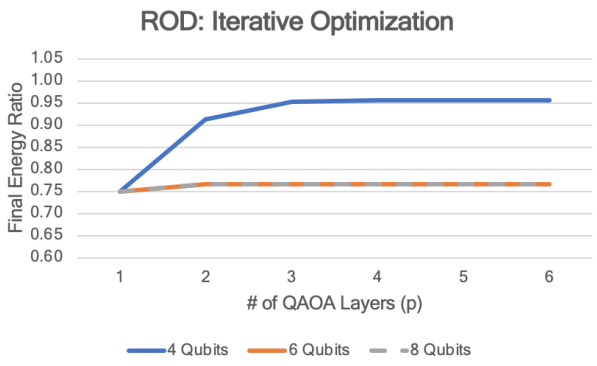
IV. Results

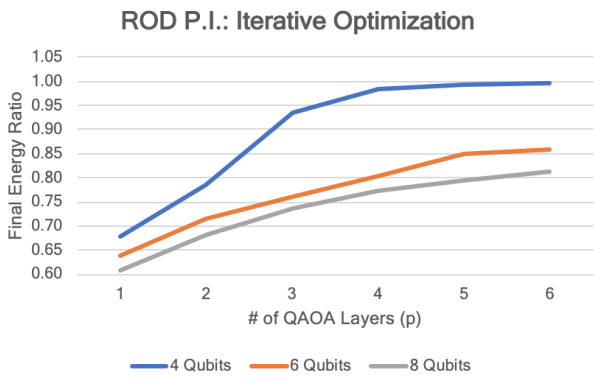
Ring of Disagrees

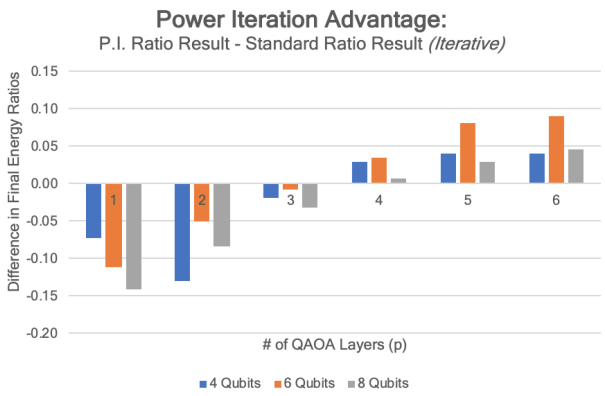
As previously noted, the collective optimization method for standard QAOA consistently yielded results closer to global optimal. Figure 3 shows this superiority—the learned optimal energy reached the theoretical optimal ratio at $p = \frac{n}{2}$ layers for all 3 qubit values within the dataset. Iterative optimization yielded suboptimal results, with each number of qubits approaching a progressively lower asymptote (Figure 4).

By contrast, Power Iteration raised the iterative asymptotes closer to the ideal ratio. Figure 5 illustrates this improvement: all 3 qubit values measured had higher asymptotic limits than with the original algorithm. The exact advantage is quantified in Figure 6, which charts the difference between the Power Iteration's output and the original output by layer. For the Ring of Disagrees, our new cost function demonstrated a positive difference for $p > 3$ layers across all 3 qubit values measured.









In left-to-right order: Fig. 3, Fig. 4, Fig. 5, Fig. 6

MaxCut Three

The iterative method on regular MaxCut Three graphs also saw an improvement with the Power Iteration. These problems are different each time (with different sets of randomized edges as defined in the *make_graph()* method), so these results are calculated across >2000 random runs of the algorithm for different numbers of qubits N and numbers of layers p. The P.I. advantages are expressed here using a histogram for each number of layers ($p \in \{4, 5, ...9\}$), where each positive entry represents a trial where the Power Iteration produced a better optimization result than the standard algorithm. As shown below, we found that the Power Iteration produces an overall optimizational disadvantage for random MaxCut Three graphs at $p \leq 6$ layers, and an overall advantage at $p > 6$ layers.

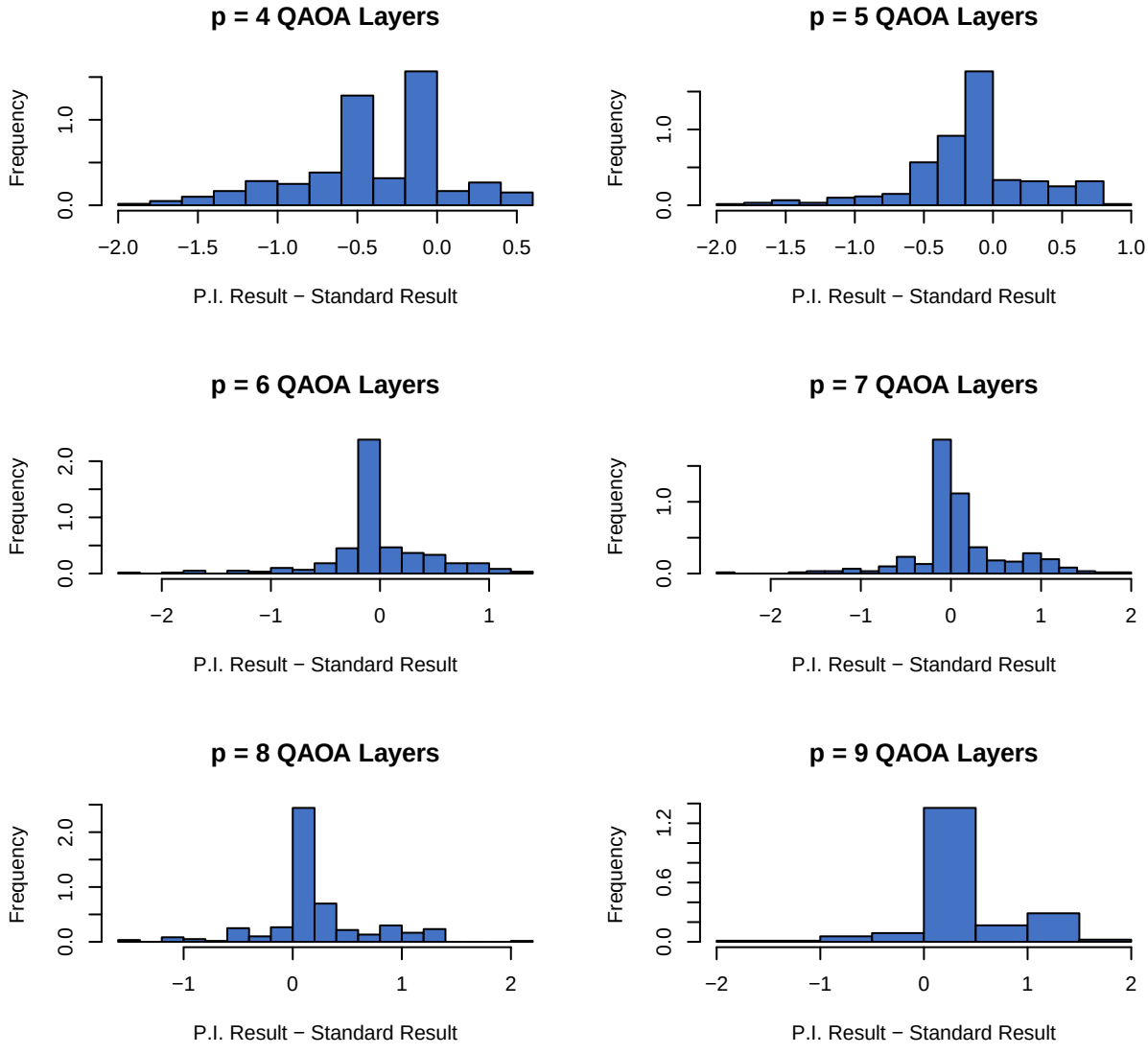


Figure 7: P.I. Advantage at each p value

Conclusion

Our state-overlap cost function ultimately provided an advantage in maximizing the cost found in equation (3) iteratively for some situations. This advantage begins at a critical number of QAOA layers ($p^* = 4$ for the Ring of Disagrees and $p^* = 7$ for regular MaxCut Three graphs), and increases with higher values of p. No advantage has been found for the collective optimization method, which is expected given the theoretical foundation of the Power Iteration cost function—the advantage comes from repeated application of a unitary that is similar to the cost Hamiltonian, biasing the quantum state towards its highest-energy eigenstate. In terms of quantum machine learning, this presents a middle ground between the expediancy of traditional iterative QAOA and the accuracy of traditional collective QAOA. For any problem that can be represented with a series of nodes with different states (e.g. switchboards and circuits, traffic grids, neural nets, epigenetics, etc.), this new cost function can be utilized on a quantum computer to more quickly find an approximately optimal allocation. This research is far from complete; outstanding inquiries worthy of further study include the algorithm's performance on higher values of N and p, advantage differences between different densities of MaxCut graphs, and the relation between N, p, and the variance[9] of optimization outcomes.

Acknowledgements

This research would not have been possible without the help of several important individuals and organizations. Thank you to the Yale Quantum Institute for their institutional and organizational support, and to the Yale Center for Research Computing for their assistance in utilizing the HPC cluster. Thank you to Aidan and Dustin for your relentless support of my success and well-being, I am deeply indebted for your help these past several months in ways both large and small. Thank you to my parents and Poppy for being my biggest fans, and for believing in me since day one. Finally, thank you to my advisor extraordinaire, Dr. Puri—your endless patience, commitment to my understanding, and brilliance have been absolutely essential to this research. You have made it possible for me to succeed in this field, and for that I am very grateful.

[9]For more complex graphs and maximization problems, the opportunity for the optimization library or function to get stuck in a local maximum increases. This translates into increasingly variant results at higher values of N and p, where the optimized outcome is increasingly sensitive to the initialized values of $\vec{\gamma}$ and $\vec{\beta}$ and the specific library employed.

Appendix

Python Scripts for Standard QAOA

```
import cirq
import numpy as np
import qutip
import random
import sympy
from scipy.optimize import minimize


class ROD():
    def __init__(self, N, layers, approach, method="iterative"):
        self.N = N
        if N % 2 == 0:
            self.nrows = int(N / 2)
            self.ncols = 2
        else:
            self.nrows = N
            self.ncols = 1

        if layers < 1:
            raise ValueError("Layers must be a positive integer.")
        self.layers = layers
        if approach not in ["BFGS", "Nelder-Mead", "Powell", "CG"]:
            raise ValueError("Optimization approach must be from approved list.")
        self.approach = approach
        if method not in ["collective", "iterative"]:
            raise ValueError("Layers method approach must be from approved list.")
        self.method = method
        self.qubits = [[cirq.GridQubit(i, j) for i in range(self.nrows)] \
            for j in range(self.ncols)]

        p = list(range(N))
        self.cyclic_perm = [[p[i - j] for i in range(N)] for j in range(N)]

    def beta_layer(self, beta):
        """Generator for U(beta, B) mixing Hamiltonian layer of QAOA."""
        for row in self.qubits:
            for qubit in row:
                yield cirq.X(qubit)**beta

    def gamma_layer(self, gamma):
        """Generator for U(gamma, C) cost Hamiltonian layer of QAOA."""
        for i in range(self.nrows):
            for j in range(self.ncols):
                if i < self.nrows - 1:
                    yield cirq.ZZ(self.qubits[i][j], self.qubits[i + 1][j])**gamma
                if j < self.ncols - 1:
                    yield cirq.ZZ(self.qubits[i][j], self.qubits[i][j + 1])**gamma
                yield cirq.Z(self.qubits[i][j])**(gamma)

    def qaoa(self):
        """Returns a QAOA circuit."""
        circuit = cirq.Circuit()
        symbols = []
        for i in range(self.layers):
            symbols.append(sympy.Symbol("\u03B3" + str(i + 1)))
            symbols.append(sympy.Symbol("\u03B2" + str(i + 1)))

        circuit.append(cirq.H.on_each(*[q for row in self.qubits for q in row]))
        for i in range(self.layers):
            circuit.append(self.gamma_layer(symbols[2 * i]))
            circuit.append(self.beta_layer(symbols[2 * i + 1]))

        return circuit
```

```
64
65      def cost_hamiltonian(self):
66          """Generates the Cost Hamiltonian for spins on a ring."""
67
68          ZZ = qutip.tensor(qutip.sigmaz(), qutip.sigmaz())
69          H_p = qutip.qeye(2)
70          for k in range(self.N - 3):
71              H_p = qutip.tensor(qutip.qeye(2), H_p)
72
73          H1 = (1 - qutip.tensor(ZZ, H_p)) / 2
74          H = [H1.permute(cycle) for cycle in self.cyclic_perm]
75          return sum(H)
76
77      def mixer_hamiltonian(self):
78          """Generates the Mixer Hamiltonian."""
79
80          H_mix = qutip.sigmax()
81          for k in range(self.N - 1):
82              H_mix = qutip.tensor(qutip.qeye(2), H_mix)
83
84          H_B = [H_mix.permute(self.cyclic_perm[i]) for i in range(self.N)]
85          return sum(H_B)
86
87      def vector_input(self):
88          """Generates the Vector Input."""
89          vec_0 = (qutip.basis(2, 0) + qutip.basis(2, 1)) / np.sqrt(2)
90          vec_input = vec_0
91          for k in range(self.N - 1):
92              vec_input = qutip.tensor(vec_input, vec_0)
93
94          return vec_input
95
96      def optimization(self):
97          """Computes the optimal energy from the circuit."""
98
99          H = self.cost_hamiltonian()
100         H_B = self.mixer_hamiltonian()
101         vec_input = self.vector_input()
102
103         # Optimize layers simultaneously
104         if self.method == "collective":
105             def cost(angles):
106                 U = 1
107                 for i in range(self.layers)[::-1]:
108                     U *= (1j * angles[2 * i] * H_B).expm()
109                     U *= (1j * angles[2 * i + 1] * H).expm()
110
111                 # Cost = |<psi|U' H U|psi>|
112
113                 vec_var = (U * vec_input)
114                 return -abs((vec_var.dag() * H * vec_var).tr())
115
116             angles = []
117             print("\n\n")
118             for i in range(2 * self.layers):
119                 angle = random.random()
120                 print(f"Initialized angle {i + 1}: {angle}")
121                 angles.append(angle)
122
123             print(f"\nOptimizing angles with {self.approach}...\n")
124
125             results = minimize(cost, angles, method=self.approach)
126             for i in range(2 * self.layers):
127                 print(f"Optimized angle {i + 1}: {results.x[i]}")
128
129             return results.x
```

```
        # Optimize layers individually
        else:
            all_angles = []
            print("\n\n")

            def cost(angles):
                U = (1j * angles[0] * H_B).expm()
                U *= (1j * angles[1] * H).expm()

                # Cost = |<psi|U' H U|psi>|

                vec_var = (U * vec_input)
                return -abs((vec_var.dag() * H * vec_var).tr())

            for i in range(self.layers):
                new_angles = [random.random(), random.random()]
                print(f"Initialized Gamma {i + 1}: {new_angles[0]}")
                print(f"Initialized Beta {i + 1}: {new_angles[1]}")

                results = minimize(cost, new_angles, method=self.approach)
                U1 = (1j * results.x[0] * H_B).expm()
                U2 = (1j * results.x[1] * H).expm()
                vec_input = U1 * U2 * vec_input

                all_angles.append(results.x[0])
                all_angles.append(results.x[1])

            print("\n")
            print(f"Optimizing angles with {self.approach}...\n")
            for i in range(self.layers):
                print(f"Optimized Gamma {i + 1}: {all_angles[2 * i]}")
                print(f"Optimized Beta {i + 1}: {all_angles[2 * i + 1]}")

            return all_angles

    def final_energy(self):
        """Calculates the final energy ratio from learned angles."""

        H = self.cost_hamiltonian()
        eigenvalues, _ = H.eigenstates()

        H_B = self.mixer_hamiltonian()
        vec_input = self.vector_input()
        results = self.optimization()

        U = 1
        for i in range(self.layers)[::-1]:
            U *= (1j * results[2 * i] * H_B).expm()
            U *= (1j * results[2 * i + 1] * H).expm()

        vec_var = (U * vec_input)

        # Returns the ratio of the found solution/the optimal solution
        return abs((vec_var.dag() * H * vec_var).tr())/eigenvalues[-1]

    def analyze(self):
        print(f"\n********** QAOA with {self.N} Qubits, {self.layers} Layers **********")
        print(self.qaoa())
        if self.method == "collective":
            print(f"\n\n********** Optimizing Layers Simultaneously **********")
        else:
            print(f"\n\n********** Optimizing Layers Individually **********")

        energy = round(self.final_energy(), 4)
        print(f"\nFinal Energy Ratio: {energy}\n")
        return(energy)
```

```python
class MaxCut(ROD):
    def __init__(self, min_conx, max_conx, density, *args, **kwargs):
        if min_conx > max_conx:
            raise ValueError("Maximum connections must be at least minimum connections.")
        self.min_conx = min_conx
        self.max_conx = max_conx

        # Density = avg number of connections per node
        if density < min_conx or density > max_conx:
            raise ValueError("Density must be between the minimum and maximum connections.")
        self.density = density
        super().__init__(*args, **kwargs)

        self.graph_edges = self.make_graph()

    def make_graph(self):
        """Produces an random edge dictionary for a regular graph within specifications."""

        nodes = list(range(self.N))
        edges = {
            node: [] for node in nodes
        }
        edges_count = {
            node: 0 for node in nodes
        }

        # Add minimum value of connections to each point:
        for i in range(self.min_conx):
            for node in nodes:
                if edges_count[node] == self.max_conx:
                    pass
                else:
                    while True:
                        x = np.random.choice(nodes, size=1)[0]
                        if x != node and edges_count[x] < self.max_conx:
                            if x not in edges[node]:
                                break

                    edges[node].append(x)
                    edges[x].append(node)
                    edges_count[node] += 1
                    edges_count[x] += 1

        current_density = sum(edges_count.values())/self.N

        # If at or above current density, it's finished
        if current_density >= self.density:
            return edges

        # Otherwise, add extra connections for density:
        for i in range(round(self.N * (self.density - current_density)/2)):
            while True:
                pair = np.random.choice(nodes, size=2, replace=False)
                a = pair[0]
                b = pair[1]

                if a in edges[b]:
                    continue

                if edges_count[a] < self.max_conx and edges_count[b] < self.max_conx:
                    break

            edges[a].append(b)
            edges[b].append(a)
            edges_count[a] += 1
            edges_count[b] += 1

        return edges
```

```
    def cost_hamiltonian(self):
        """Generates the Cost Hamiltonian for spins on a max-cut graph."""

        Zs = []
        for i in range(self.N):
            if i == 0:
                ZZ = qutip.tensor(qutip.sigmaz(), qutip.qeye(2))
                for i in range(self.N - 2):
                    ZZ = qutip.tensor(ZZ, qutip.qeye(2))
                Zs.append(ZZ)

            else:
                ZZ = qutip.qeye(2)
                for j in range(1, self.N):
                    if i == j:
                        ZZ = qutip.tensor(ZZ, qutip.sigmaz())
                    else:
                        ZZ = qutip.tensor(ZZ, qutip.qeye(2))
                Zs.append(ZZ)

        H = 0
        all_edges = self.graph_edges
        for node, edges in all_edges.items():
            Zi = Zs[node]
            for edge in edges:
                Zj = Zs[edge]
                H += (1 - (Zi * Zj)) / 2

        return H

    def analyze(self):
        title = f"Max-Cut {self.max_conx} with {self.N} Qubits, {self.layers} Layers"
        print(f"\n\n********** {title} **********")
        print(self.qaoa())
        print(self.graph_edges)
        if self.method == "collective":
            print(f"\n\n********** Optimizing Layers Simultaneously **********")
        else:
            print(f"\n\n********** Optimizing Layers Individually **********")

        energy = round(self.final_energy(), 4)
        print(f"\nFinal Energy Ratio: {energy}\n")
        return(energy)
```

Python Scripts for Power Iteration

```python
class ROD_PI(ROD):
    def optimization(self):
        """Computes the optimal energy from the circuit."""

        H = self.cost_hamiltonian()
        H_B = self.mixer_hamiltonian()
        vec_input = self.vector_input()

        # Optimize layers simultaneously
        if self.method == "collective":
            def cost(angles):
                U = 1
                for i in range(self.layers)[::-1]:
                    U *= (1j * angles[2 * i] * H_B).expm()
                    U *= (1j * angles[2 * i + 1] * H).expm()

                # Cost = |sqrt(<psi|H^2|psi>)| - |<psi|U' H|psi>|
                vec_var = U * vec_input
                term_one = (vec_input.dag() * (H**2) * vec_input).tr()
                term_two = (vec_var.dag() * H * vec_input).tr()
                return -abs(abs(np.sqrt(term_one)) + abs(term_two))

            angles = []
            print("\n\n")
            for i in range(2 * self.layers):
                angle = random.random()
                print(f"Initialized angle {i + 1}: {angle}")
                angles.append(angle)

            print(f"\nOptimizing angles with {self.approach}...\n")
            results = minimize(cost, angles, method=self.approach)
            for i in range(2 * self.layers):
                print(f"Optimized angle {i + 1}: {results.x[i]}")

            return results.x

        # Optimize layers individually
        else:
            all_angles = []
            print("\n\n")

            def cost(angles):
                U = (1j * angles[0] * H_B).expm()
                U *= (1j * angles[1] * H).expm()

                # Cost = |sqrt(<psi|H^2|psi>)| - |<psi|U' H|psi>|
                vec_var = (U * vec_input)
                term_one = (vec_input.dag() * (H**2) * vec_input).tr()
                term_two = (vec_var.dag() * H * vec_input).tr()
                return -abs(abs(np.sqrt(term_one)) + abs(term_two))

            for i in range(self.layers):
                new_angles = [random.random(), random.random()]
                print(f"Initialized Gamma {i + 1}: {new_angles[0]}")
                print(f"Initialized Beta {i + 1}: {new_angles[1]}")

                results = minimize(cost, new_angles, method=self.approach)
                U1 = (1j * results.x[0] * H_B).expm()
                U2 = (1j * results.x[1] * H).expm()
                vec_input = U1 * U2 * vec_input

                all_angles.append(results.x[0])
                all_angles.append(results.x[1])

            print("\n")
            print(f"Optimizing angles with {self.approach}...\n")

```

```
            for i in range(self.layers):
                print(f"Optimized Gamma {i + 1}: {all_angles[2 * i]}")
                print(f"Optimized Beta {i + 1}: {all_angles[2 * i + 1]}")

            return all_angles

    def analyze(self):
        title = f"Power Iteration QAOA with {self.N} Qubits, {self.layers} Layers"
        print(f"\n******* {title} *******")
        print(self.qaoa())
        if self.method == "collective":
            print(f"\n\n********** Optimizing Layers Simultaneously **********")
        else:
            print(f"\n\n********** Optimizing Layers Individually **********")

        energy = round(self.final_energy(), 4)
        print(f"\nFinal Energy Ratio: {energy}\n")
        return(energy)


class MaxCut_PI(MaxCut, ROD_PI):
    def __init__(self, *args, **kwargs):
        super(MaxCut).__init__(*args, **kwargs)

    def cost_hamiltonian(self):
        return super(MaxCut).cost_hamiltonian()

    def optimization(self):
        return super(ROD_PI).optimization()

    def analyze(self):
        title = f"P.I. Max-Cut {self.max_conx} with {self.N} Qubits, {self.layers} Layers"
        print(f"\n\n********** {title} **********")
        print(self.graph_edges)
        print(self.qaoa())
        if self.method == "collective":
            print(f"\n\n********** Optimizing Layers Simultaneously **********")
        else:
            print(f"\n\n********** Optimizing Layers Individually **********")

        energy = round(self.final_energy(), 4)
        print(f"\nFinal Energy Ratio: {energy}\n")
        return(energy)
```