Quantum Approximate Optimization Algorithm(QAOA)

Qin Yuyang, Ren Tingxu, Chen Ke, Lai Pengyi
CKC College Zhejiang University, Hangzhou 310058
(Dated: June 16, 2023)

Quantum Approximate Optimization Algorithm is a hybrid quantum-classical algorithm for solving combinatorial optimization problems. The depth p and the chosen parameters can greatly influence the algorithm's performance. Based on related papers, we implement unitary matrix operator simulation, deploy three methods to optimize $\vec{\gamma}, \vec{\beta}$ and compare their performance. Then we construct the quantum circuits and use qiskit SDK to simulate them. Besides, we test our quantum circuits' performance under real IBM Quantum Machines noise. Lastly, We study the influence of p on QAOA for a given graph, and lots of results are analyzed. Our codes are available at github

I. BACKGROUND

A. Combinatorial Optimization Problem

Given n bits and m clauses, the combinatorial optimization problem asks for a string $z = z_1 z_2 ... z_n$ that maximize the cost function

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

where $C_\alpha = 1$ if z satisfies clause α and 0 if it doesn't satisfies. Usually, finding optimization solutions is hard and costly using classical algorithms, but it is relatively easier to achieve an approximate solution close to the maximum of $C(z)$. This kind of method is called approximate optimization algorithm. The performance of an approximate optimization algorithm is guaranteed by an approximate ratio of R such that

$$\frac{C(z)}{\max_z C(z)} \geq R \tag{2}$$

B. Maximum Cut

Problem 1 (Maximum Cut (MaxCut)) *Given a graph $G = (V, E)$ with n vertices and m weighted edges, separate V into two disjoint sets S and T to maximize the sum of weights of edges (u, v) such that $u \in S, v \in T$ or $v \in S, u \in T$.*

The MaxCut problem is a well-known combinatorial optimization problem. The problem is NP-hard and APX-hard, meaning that no polynomial-time algorithm has been found and the approximate ratio obtained by polynomial-time algorithms cannot be arbitrarily close to the optimal solution unless $NP = P$.

The quantum computer works under the 2^n dimensional Hilbert space. To solve the combinatorial optimization problem, it is expected to construct a Hamiltonian (phase Hamiltonian) C such that

$$C\,|z\rangle = C(z)\,|z\rangle \tag{3}$$

where $|z\rangle$ is a base in computational basis. In this way, the problem of finding the maximum of $C(z)$ changes into finding the extremal eigenvalue for the phase Hamiltonian.

Definition 1 (Phase Hamiltonian) *The phase Hamiltonian for the MaxCut problem is constructed as*

$$C = \sum_{\langle jk \rangle} \frac{w_{jk}}{2}(1 - \sigma_j^z \sigma_k^z) \tag{4}$$

where w_{jk} is the weight of edge $\langle jk \rangle$.

Let's take the 2^2 dimensional Hilbert space as an example to see how this construction works. If there is an edge between the two vertices, then

$$\sigma_0^z \sigma_1^z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5}$$

$$C = \frac{w}{2}(1 - \sigma_0^z \sigma_1^z) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & w & 0 & 0 \\ 0 & 0 & w & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{6}$$

This means that the result of $C|z\rangle$ is equal to w if and only if the two vertices are in different sets, i.e., $|z\rangle = |01\rangle$ or $|z\rangle = |10\rangle$. Taking the sum over all edges, we can prove that this construction satisfies Equation 3.

C. QAA: Quantum Adiabatic Algorithm

Quantum Adiabatic Algorithm is a method raised around 2000 to solve combinatorial search problems [1]. The algorithm begins with an initial state $|s\rangle$ which is the groudn state of B. Then it constructs a time-dependent Hamiltonian

$$H(t) = (1 - \frac{t}{T})B + \frac{t}{T}C \tag{7}$$

Let the system evolve according to the Schrödinger equation for a sufficiently long time T. The system is expected to stay in the ground state over the smooth evolution. In the end, the ground state of $H(T) = C$ can be obtained.

However, in practical situations, the long evolution time can be intolerable. In addition, the energy levels change continuously over time T, and in some occasions, two energy levels may be pretty near or even cross each other. If this happens to the two lowest energy levels, the system will jump into the other energy level and leave the ground state. This means that a long-time evolution cannot guarantee the maximum answer to be found. These drawbacks greatly limit the performance of QAA.

II. ALGORITHM SPECIFICATION

Quantum Approximate Optimization Algorithm (QAOA) uses a Trotterized approximation of QAA to obtain a quantum gate model algorithm [2]. It separates the continuous evolution process into countable stages and repeatedly applies the short-time evolution of the Phase Operator and the Mix operator to get a state similar to the desired ground state.

A. Operator Definition

Definition 2 (Phase operator) *Define the Phase Operator as*

$$U(C, \gamma) = e^{-i\gamma C} \tag{8}$$

where $\gamma \in [0, 2\pi]$ because C is a diagonal matrix with integar elements. The phase operator can be regarded as applying the Phase Hamiltonian for a short time proportion to γ.

Definition 3 (Mix Hamiltonian) *Define the Mix Hamiltonian as*

$$B = \sum_{i=1}^{n} \sigma_i^x \tag{9}$$

This Hamiltonian is constructed for convenient preparation of the initial state. The ground state of this Hamiltonian is simply a sum over all computational basis

$$|s\rangle = |+\rangle^{\oplus n} = \frac{1}{\sqrt{n}} \sum_z |z\rangle$$

Therefore, this ground state for B is chosen as the initial state for the evolution.

Definition 4 (Mix Operator) *Define the Mix Operator as*

$$U(B, \beta) = e^{-i\beta B}$$

where $\beta \in [0, \pi]$. Similar to the Phase Operator, this operator functions as applying the Mix Hamiltonian for a short time, which is proportioned to β.

B. Envolution

Recall the process of QAA, the long evolution time can be cut into multiple small periods of time-independent evolution. For a small time interval $\Delta t\hbar$ at time t, the evolution operator can be appropriated by

$$\begin{aligned} U \approx e^{-iH(t)\Delta t} &= e^{-i(uB+vC)\Delta t} \\ &= \lim_{N\to\infty} (e^{-iuB\Delta t/N} e^{-ivC\Delta t/N})^N \end{aligned} \tag{10}$$

where $u = (1 - \frac{t}{T}), v = \frac{t}{T}$.

Notice that the decomposition operators are just in the form of Phase operators or Mix operators, which inspires us to apply these two kinds of operators alternatively on the initial state. For a depth p and $2p$ predetermined parameters $(\vec{\gamma}, \vec{\beta})$, define a quantum state.

$$|\vec{\gamma}, \vec{\beta}\rangle = U(B, \beta_p)U(C, \gamma_p)...U(B, \beta_1)U(C, \gamma_1)|s\rangle \tag{11}$$

The expectation of eigenvalues of C when we measure $|\vec{\gamma}, \vec{\beta}\rangle$ in computational basis is

$$F_p(\vec{\gamma}, \vec{\beta}) = \langle \vec{\gamma}, \vec{\beta}|C|\vec{\gamma}, \vec{\beta}\rangle$$

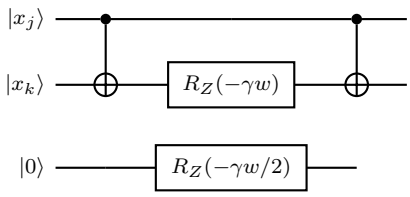


FIG. 1. Quantum circuit for an edge $\langle jk\rangle$ in the Phase Operator

For sufficiently large p and appropriately chosen parameters, the state can evolve into the ground state of C as discussed above. Therefore,

$$\lim_{p\to\infty} \max_{\vec{\gamma},\vec{\beta}} F_p(\vec{\gamma},\vec{\beta}) = C_{max}(z)$$

In reality, we cannot make p as large as infinity, but an approximate solution can be obtained using this state for finite p. To do this, repeatedly measure $|\vec{\gamma},\vec{\beta}\rangle$ in computational basis. For each measure result $|z\rangle$, compute $C(z)$ using traditional computers and keep the maximum number of this value as $\hat{C}(z)$. Over many times of measurements, $\hat{C}(z)$ will be close to or greater than $F_p(\vec{\gamma},\vec{\beta})$. If we can optimize the parameters for this value, then an approximate optimization solution is found.

III. CIRCUIT DESCRIPTION

A. The Phase Operator

The circuit for the Phase Operator $U(C,\gamma)$ varies over different problems and depends on the composition of C. For the MaxCut problem, $U(C,\gamma)$ can be decomposed into a unitary operator for each edge using Trotter decomposition. Therefore, we can design the circuit separately for every edge.

$$\begin{aligned} U(C,\gamma) &= e^{-i\gamma\sum_{\langle jk\rangle} C_{\langle jk\rangle}} \\ &= \Pi_{\langle jk\rangle} e^{-i\gamma\frac{w}{2}(-\sigma_j^z\otimes\sigma_k^z+I)} \\ &= \Pi_{\langle jk\rangle} e^{-i\frac{-\gamma w}{2}(\sigma_j^z\otimes\sigma_k^z)} e^{-i\frac{\gamma w}{2}I} \end{aligned}$$

$e^{-i\frac{-\gamma w}{2}(\sigma_j^z\otimes\sigma_k^z)}$ can be implemented as a CNOT gate, a z-rotation gate and another CNOT gate. while $e^{-i\frac{\gamma w}{2}I}$ is a z-rotation gate applied to $|0\rangle$.

Therefore, an edge $\langle jk\rangle$this operator can be implemented in the circuit like FIG.1.

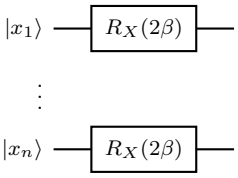


FIG. 2. Quantum circuit for the Mix Operator

B. The Mix Operator

The Mix Operator contains the sum of the σ_x operator of each qubit.

$$U(B,\beta) = e^{-i\beta\sum\sigma_i^x} \tag{12}$$

Therefore, the circuit is to simply add an x-rotation gate $R_{2\beta}^X$ for each qubit as FIG.2, where

$$R_\phi^X = cos(\phi/2)I_2 - sin(\phi/2)i\sigma_x$$

The overall circuit is a combination of these designs.

IV. OPTIMIZING

A complete process for the QAOA algorithm of depth p works as below:

1) Choose a set of $2p$ parameters $(\vec{\gamma},\vec{\beta})$ and use them to build the circuit
2) Use quantum computer to get the state $\vec{\gamma},\vec{\beta}\rangle$
3) Measure the state in computational basis multiple times and record the resulting $|z\rangle$ with maximum $C(z)$
4) Adjust parameters (γ,β) and repeat steps 3 and 4 to get a better solution
5) Output the maximum result

The performance of QAOA greatly depends on the choice of p and the parameters. Usually, p is limited by the hardware and upstream algorithms. Therefore, how to find appropriate parameters to optimize the answer is a core problem. This part is usually solved using classical optimization methods based on iteration. As a result, QAOA is divided into the category of hybrid quantum-classical algorithms.

A. Optimizing methods

The most brute force method is to iterate over a fine grid over the $[0,2\pi]^p \times [0,\pi]^p$ space. Supposing each

interval is divided into k segments, the time complexity will be as large as $O(k^{2p})$. However, the complexity is still independent of n, meaning that this method can still have some application in occasions with small depth.

Generally, the classical optimization method can be classified into two categories. One is gradient-based algorithms, the most classical of which is the gradient descent. In this kind of method, the way of finding the gradient at a given point also varies. These methods may achieve different performances in different situations. The other category is the gradient-free algorithm. In our experiment, we chose Bayesian Optimization as an example. Since the quantum computer can be regarded as a black box for upstream algorithms, there are a great many numerical analysis methods to undertake this work. As long as the search space is $[0, 2\pi]^p \times [0, \pi]^p$, the complexity can remain independent of n. This is where quantum methods differ from traditional algorithms for Combinatorial Optimization Problem.

When p is relatively large, there are also some tricks we can perform on the initial value of $(\vec{\gamma}, \vec{\beta})$. For example, use the optimal values for smaller p to interplot the parameters for larger p. Similar methods can greatly reduce the number of iterations needed for follow-up optimization [3].

V. EXPERIMENT DESIGN

In this section we focus on how we design our experiment, implement it with our code, along with our code structure and some fixed bugs.

We focus on the optimization and simulation of the QAOA algorithm and its implementation on the Max-Cut problem in Python. First, we implement the QAOA utilizing the unitary matrix operator in numpy form. Then we use grid search, Bayesian Optimization and L-BFGS-B to find the best β, γ. With the best β, γ, we construct its quantum circuits and simulate them both with and without noise using IBM qiskit SDK[4].

A. Data Preparation

- **graphic_in.py**: To prepare a random graph with n nodes and get the Max-Cut classic result of it, where **generateGraph()** could new a graph and **graphic(n, edge.ask_min())** provides its Max-Cut answer as well as division scheme in hexadecimal. The graph will be saved into **grapy_in.npy**.
- **graphic_print.py**: draw the graph using networkx SDK.

B. Unitary Matrix Operator Simulation

In the QAOA algorithm, we have

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

where

$$U(C, \gamma) = e^{-i\gamma C}, U(B, \beta) = e^{-i\beta B}$$

which could be achieved by our Python code. There are two methods, eigenvalue decomposition and **np.linalg.eigh** implementation.Eigenvalue decomposition is faster while **np.linalg.eigh** implementation get better results.

1. *Eigenvalue Decomposition*

If C is a Hermitian matrix, then it can be diagonalized as $C = UDU^\dagger$, where U is a unitary matrix and D is a diagonal matrix with diagonal elements being the eigenvalues of C. Therefore, we can write $e^{-i\gamma C}$ as:

$$e^{-i\gamma C} = e^{-i\gamma UDU^\dagger}$$

Since U is a unitary matrix, it satisfies $UU^\dagger = U^\dagger U = I$. According to the definition of matrix exponential function, we have:

$$\begin{aligned} e^{-i\gamma UDU^\dagger} &= \sum_{n=0}^{\infty} \frac{(-i\gamma UDU^\dagger)^n}{n!} \\ &= \sum_{n=0}^{\infty} \frac{(-i\gamma)^n (UDU^\dagger)^n}{n!} \end{aligned}$$

Since $U^\dagger U = I$, $(UDU^\dagger)^n = UD^nU^\dagger$. Therefore, we can write the above formula as:

$$\begin{aligned} e^{-i\gamma UDU^\dagger} &= \sum_{n=0}^{\infty} \frac{(-i\gamma)^n UD^nU^\dagger}{n!} \\ &= U\left(\sum_{n=0}^{\infty} \frac{(-i\gamma)^n D^n}{n!}\right) U^\dagger \\ &= Ue^{-i\gamma D}U^\dagger \end{aligned}$$

Therefore, by performing eigenvalue decomposition on C, we can transform the calculation of the matrix exponential function into a direct calculation of diagonal elements. In this way, we can avoid the error of the matrix exponential function. However, the precision of eigenvalue decomposition obtained by **np.linalg.eig** cannot meet the requirements. Its error is so significant that the result of $e^{-i\gamma C}$ is no longer a unitary matrix anymore. By multiplying these so-called unitary matrices, the magnitude of the final state $|\vec{\gamma},\vec{\beta}\rangle$ is far away from 1. It's the first **precision disasters** we have met. When we use tried different $\vec{\gamma},\vec{\beta}$ in the optimization section to get the best parameters, sometimes we got dramatical results like F_p larger than 10000. To solve the problem, we finally discover this precision bug. So we used **np.linalg.eigh** for eigenvalue decomposition to achieve better precision.

2. *Expm Direct Calculation*

Since the method of eigenvalue decomposition doesn't work, we are forced to seek other solutions, which leads us to **scipy.linalg.expm**. We analyze its result, which turns out to be a nice one. We find $e^{-i\gamma C}$ is a unitary matrix and the magnitude of the final state $|\vec{\gamma},\vec{\beta}\rangle$ is about 0.9999, which is a satisfactory result. So we choose the new method and solve the first **precision disasters**.

C. Parameter Optimization

Now, given an array of $|\vec{\gamma},\vec{\beta}\rangle$, We could simulate the QAOA. So we try to find the best array of $|\vec{\gamma},\vec{\beta}\rangle$ at fixed p. That's exactly what we do in **optimize.ipynb**, finding the best array of $|\vec{\gamma},\vec{\beta}\rangle$ at fixed p and n with various methods.

We use dfs to achieve the grid search, hyperopt package[5] to finish Bayesian Optimization and scipy. optimize to implement the basin-hopping algorithm in the L-BFGS-B method. We compare these three algorithms using its F_p.

$$F_p(\vec{\gamma},\vec{\beta}) = \langle\vec{\gamma},\vec{\beta}|C|\vec{\gamma},\vec{\beta}\rangle$$

According to our test result shown as TABLE.I, it turns out that basin-hopping and L-BFGS-B have done a faster and better job.

We want to find out how F_p changes as p increases in two fixed graphs shown as FIG.4 and FIG.3. So we iterate p in $range(1,11)$ and utilize the basin-hopping algorithm in the L-BFGS-B method to optimize the F_p for each p. Then we get the result of each p and save the $\vec{\gamma},\vec{\beta}$. And the formulation of accuracy we use here is

$$\text{Accuracy} = \frac{F_p(\vec{\gamma},\vec{\beta})}{\text{Answer of classical Algorithm}}$$

We tried to run as large n as possible. However, as n increases 1, the unitary matrix times 4 and it will take more attempts in the parameter fine-tuning process. At last, we chose n=8 and generate a graph as shown in FIG.3. We run on Lab's *Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz* and it takes us several days to get the final result.

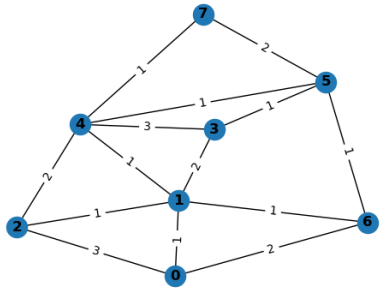


FIG. 3. An Graphic Example of n=8,whose Max Cut = 19

However, when we attempt to simulate the quantum circuit, we find out that we could only simulate a quantum circuit with noise in n=7 utilizing qiskit since the free IBM quantum machine has a maximum qubit of 7 and we use its noise data. Since quantum circuit simulation needs the optimized $\vec{\gamma},\vec{\beta}$, we generate a new graph, shown as FIG.4, and repeat the above process.

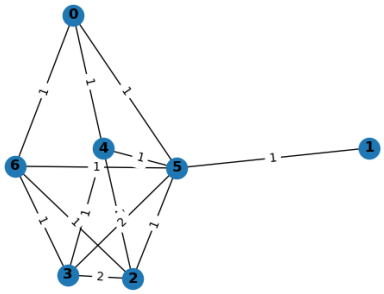


FIG. 4. An Graphic Example of n=7,whose Max Cut = 11

D. Quantum Circuits Simulation

We prepare our qubits with a Hadamard Gate to obtain the mixed state. $U(C,\gamma) = e^{-i\gamma C}$ could be achieved by applying a $Rz(-\gamma)$ and two C_x gate at both ends of it. $U(B,\beta) = e^{-i\beta B}$ could be simply implemented deploying $R_x(2\beta)$ gate. In this way, we succeed in constructing the quantum circuit. Our code could be run on quantum machines, but the long waiting list prevents us. The typical quantum circuit our code generated is shown as FIG.5. The datasheet of IBM nairobi, the real quantum machine we simulate, is shown in FIG.6.

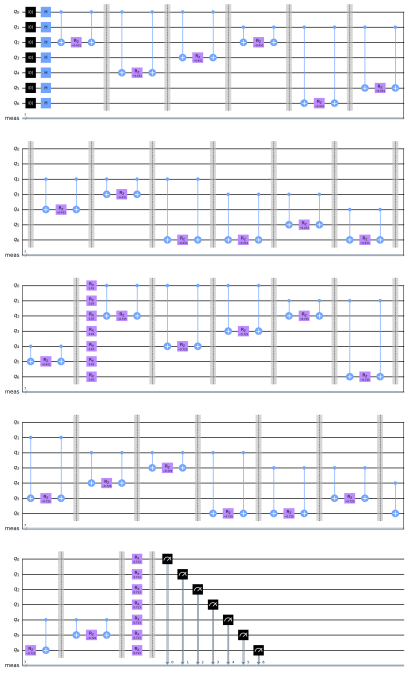


FIG. 5. A quantum circuit example at n=7,p=2.

We obtain our result both with and without noise with the help of **qasm_simulator**. We draw the top 5 states of each situation in the same histogram for p in $range(1,7)$. We try the original 1024 shots shown results in FIG.11 and the large enough 100000 shots shown results in FIG.12, which we aim to observe the real quantum computation result and the expectation of the final states individually. At the same time, we plot the Accuracy-p figure with and without noise, to help us better understand the influence of noise.

$$\text{Accuracy} = \frac{\text{number of desired states}}{\text{total shot number}}$$

Besides, in the experiment, we attempt to construct

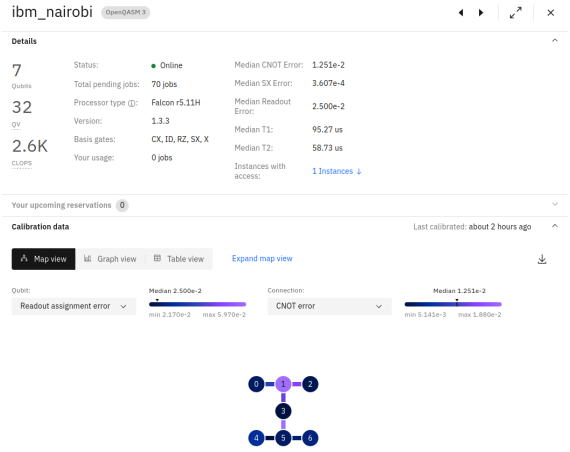


FIG. 6. Overview of IBM nairobi

FIG. 7. A Failed Attempt to Construct the R_z Error

the R_z error ourselves, which turns out to be another **precision disaster**. As shown in the FIG.7, we set the possibility of error to zero, but we still obtain an undesired gate result.It's $P(1)$ is obviously not 0 and its $P(3)$ is not 1. We failed to fix this precision bug. That's why we turn to actual quantum machine noise data.

E. Experiment Design summary

1) We implement QAOA on Max-Cut using a unitary matrix operator.

2) We tried three different ways of parameter optimization and test them with various n and p to find out the pros and cons.

3) We construct the quantum circuit of QAOA.

4) We simulate the quantum circuit using qiskit and use real quantum machine's error data to simulate the noise.

5) We plot our experiment result and discuss the reason behind it.

VI. EXPERIMENT RESULT AND ANALYSE

A. Comparison of different optimizor

TABLE.I compares the performances of different optimizers on data with different attributes. It is shown that when $p = 1, 2$, the result of brute force is relatively not bad. However, as p increases, the brute force algorithm takes too long to execute and the Bayes algorithm meets difficulty in accuracy. BFGS algorithm remains the most stable one.

The result also shows that the accuracy grows significantly with p when p is not so large.

TABLE I. Optimizor

	n=7, p=1, k=20			n=7, p=2, k=10		
real answer	8	3	9	4	15	8
Bayes	6.299	2.357	7.271	3.384	13.209	6.274
BFGS	6.307	2.380	7.336	3.122	13.601	6.696
brute force	6.140	2.337	7.213	3.343	11.860	6.043
BFGS ratio	0.788	0.793	0.815	0.781	0.907	0.837
	n=7, p=3			n=7, p=4		
real answer	12.000	8.000	11.000	15.000	14.000	13.000
Bayes	9.933	5.693	8.154			
BFGS	11.100	6.698	9.218	14.041	12.726	11.817
BFGS ratio	0.925	0.837	0.838	0.936	0.909	0.909

B. Parameter Optimization

We are interested in how the F_p of a certain graph develops as p grows.

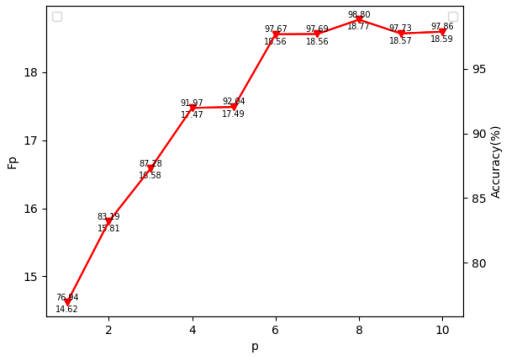


FIG. 8. The relationship between F_p and p , $n = 8$

First, we run our algorithm at n=8, shown as FIG.3 and we got the result as FIG.3. For p larger than 6, we achieve an accuracy of 97%, which is impressive. Some fluctuation occurs when p is large. It's because our optimization algorithm may have stuck in the local minima sometimes.

Then, we run our algorithm at n=7, shown as FIG.4 and we got the result as FIG.4. We even achieve an accuracy of 99.86%! Since n is smaller, we got a smooth monotonically increasing curve this time.

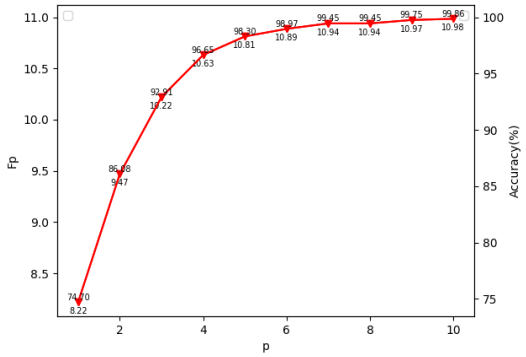


FIG. 9. The relationship between F_p and p on the fixed graph with $n = 7$ vertices

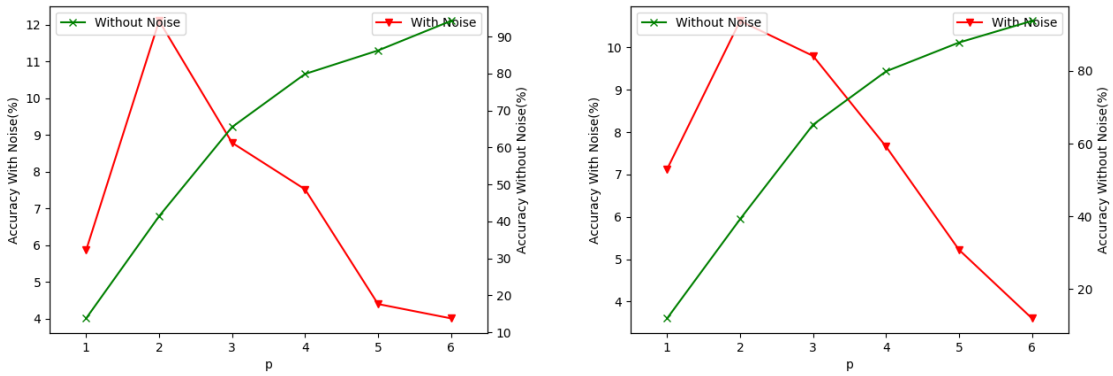
Besides, it's worth mentioning that since p refers to the time our edges could expand to its neighbors, the longest path in the solution graph at p is $2p+1$[2], which means the final solution may be unachievable for small p in theory.

In conclusion, the experiment result shows that F_p increases as p increases and

$$\lim_{p\to\infty} \max_{\vec{\gamma},\vec{\beta}} F_p(\vec{\gamma},\vec{\beta}) = C_{max}(z)$$

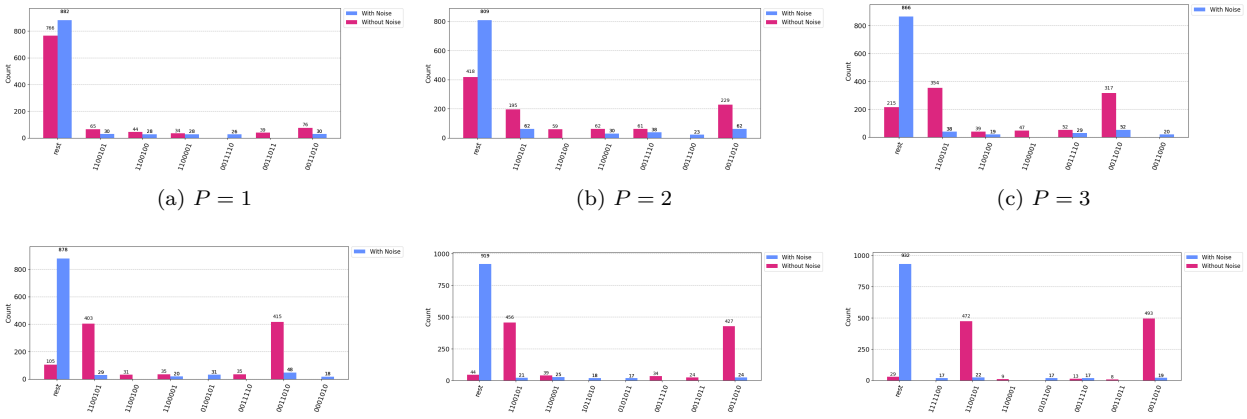
C. Quantum Circuits Simulation

The top-5 quantum circuit's result states histogram pictures are shown in FIG.11 and FIG.12, 1024 shots situation and 100000 shots situation correspondingly.



(a) 1024 shots (b) 100000 shots

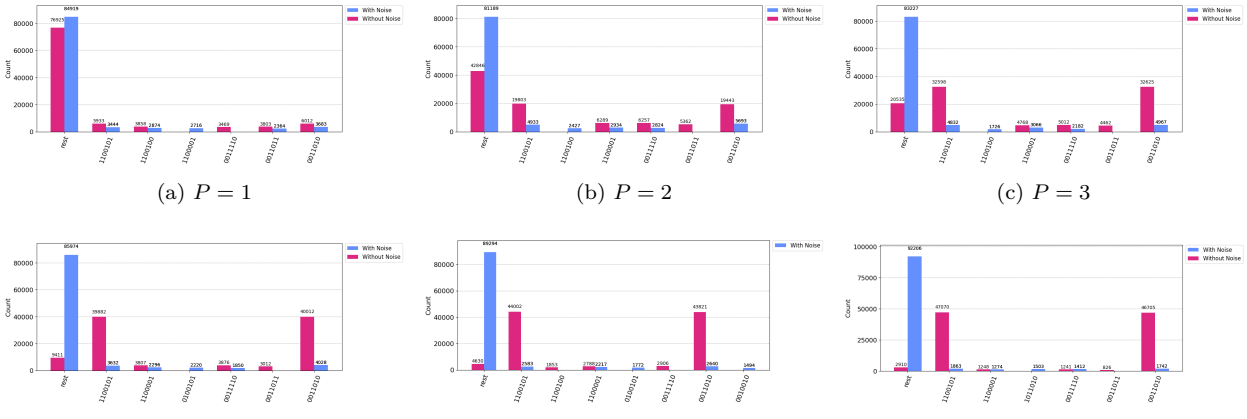
FIG. 10. Accuracy-P digram



(a) $P = 1$ (b) $P = 2$ (c) $P = 3$

(d) $P = 4$ (e) $P = 5$ (f) $P = 6$

FIG. 11. Output states statistics for different p at 1024 shots



(a) $P = 1$ (b) $P = 2$ (c) $P = 3$

(d) $P = 4$ (e) $P = 5$ (f) $P = 6$

FIG. 12. Output states statistics for different p at 100000 shots

We will find that for shots=100000 in FIG.10, the accuracy states could always be distinguished, though hard. So the noise only decreases the possibility of our desired states and randomly attributes them with other states. Since the same process act on every state, the possibility of our desired states is still ahead of others, though the gap is getting smaller as the depth of the quantum circuit increase.

However, for shots=1024, they could only be figured out at $p = 2$ and 3. It's due to the random nature of quantum measurement. As long as the expectation two states are close, they will mix up easily. So in practice, we must control the depth of the quantum circuit since the random nature of quantum measurement may bridge the gap between our desired states and others.

As we analyze the Accuracy-p figure in FIG.11 and FIG.12, it's easy to obtain that noise dramatically decreases the accuracy. As p increases, the depth of the quantum circuit increases, leading to more noise and less accuracy.

Besides, although accuracy increases with p for noiseless situations just as it's theoretically proved, the noise effect will overweigh the accuracy improvement of p and become dominant for a sufficiently large p (in this case $p = 2$). That's why accuracy first increases and then decreases as p increases.

So we should carefully choose our p when we deploy QAOA in the real quantum machine.

VII. CONCLUSION

A. Our Achievement

Quantum Approximate Optimization Algorithm is a hybrid quantum-classical algorithm for solving combinatorial optimization problems. We studied the principle and procedure of this algorithm in comparison with QAA, understood how the parameters work in this algorithm, and learned the circuit representation of each involved operator.

Then we design abundant experiments and deploy it using Python. Our codes are available at https://github.com/qyy2003/QAOA

1) We implement unitary matrix operator simulation with two methods, eigenvalue decomposition and expm direct calculation, and fixed a precision disaster.

2) We deploy three methods, grid search, Bayesian optimization and basin-hopping trick supported L-BFGS-B algorithm, to optimize $\vec{\gamma}, \vec{\beta}$ optimization and compare their performances. It turns out that gradient-based algorithms, L-BFGS-B in particular, have done a way better job.

3) We study the quantum circuits construction theory and implement it with qiskit SDK, which could be run on real quantum machines. And we test our quantum circuits' performance both with and without noise. To better emulate the real noise, we use the data of a real IBM Quantum Machines *ibm_nairob*. Then we analyze the influence of noise in different circumstances.

4) We study the influence of p on QAOA for a given graph. Experiment results show that F_p increases as p grows and some fluctuation may occur due to the limitation of the optimization algorithm. Our noiseless experiments prove that

$$\lim_{p\to\infty} \max_{\vec{\gamma},\vec{\beta}} F_p(\vec{\gamma}, \vec{\beta}) = C_{max}(z)$$

We discuss the influence of p on output states both with and without noise and try some reasoning. Neither too large nor too small, suitable p is needed to achieve better results, according to our experiment.

B. Outlook

- In our work, the optimizers are based on noise-free simulators. The optimized parameters are then passed to a noisy environment in quantum computers to test their performance. However, in real applications, how to optimize the parameters in a noisy environment remains a large problem to solve.
- QAOA is a quantum computing method to replace classical algorithms. In our work, we dug into the performance of QAOA in different depth and using various optimizers, but haven't made a reasonable comparison between QAOA and state-of-art classical algorithms to show how quantum advantage appear in this field. This sort of work requires wider research on relative methods.

- It was shown in our work that noise can greatly reduce the algorithm's performance and that the qubit we need is proportional to vertex number n. It remains to explore how to use as few qubits as possible to obtain similar performances as well as how to reduce the influence of environmental noise and detect the internal error.

Appendix A: Code

. sim.py

```
# set & connect IBM Quantum account
from qiskit import IBMQ
My_token = 'xxxx'#(IBM Quantum account token)
IBMQ.save_account(My_token)
provider = IBMQ.load_account()

import networkx as nx
import numpy as np
from qiskit import QuantumCircuit, Aer, execute, transpile
from qiskit.visualization import plot_histogram, array_to_latex
from qiskit_aer import AerSimulator
from qiskit_aer.noise import (NoiseModel, QuantumError, ReadoutError, pauli_error, depolarizing_error, thermal_relaxation_error)
# beta implement
def create_HD(parameter,G_info):
    nqubits = len(G_info.nodes())
    beta = parameter
    qc_mix = QuantumCircuit(nqubits)
    for i in range(0, nqubits):
        qc_mix.rx(beta*2, i)
    return qc_mix

# gamma implement
def create_Hp(parameter,G_info):
    nqubits = len(G_info.nodes())
    gamma = parameter
    qc_mix = QuantumCircuit(nqubits)
    edge = G_info.edges()
    for pair_nodes in edge:
        # print( pair_nodes)
        i = pair_nodes[0]
        j = pair_nodes[1]
        qc_mix.cx(i,j)
        qc_mix.rz(-1*gamma*G_info.edges[i,j]['weight'],j)
        qc_mix.cx(i,j)
        qc_mix.barrier()
    return qc_mix

# create QAOA quantum circuit
def create_qaoa(parameters_info,G):
    p = int(len(parameters_info)/2)
    nqubits = len(G.nodes())
    circ = QuantumCircuit(nqubits)
    for i in range(0,nqubits):
        circ.reset(i)
        circ.h(i)
    G_info = G

    parameter_Hp = parameters_info[0:p]
    parameter_HD = parameters_info[p:]

    for i in range(0,p):
        parameter1 = parameter_HD[i]
        circ1 = create_HD(parameter1, G_info)
        parameter2 = parameter_Hp[i]
        circ2 = create_Hp(parameter2, G_info)
        new_circ = circ2.compose(circ1)
        circ = circ.compose(new_circ)
    qaoa_circ = circ
    qaoa_circ.measure_all()
    return qaoa_circ

## to init Graph
import matplotlib.pyplot as plt

global G
G = nx.Graph()
n=7
for i in range(n):
    G.add_node(i)
graph = np.load('grapy_in.npy')
for edge in graph:
    if(G.has_edge(edge[0],edge[1])):
        G.edges[edge[0],edge[1]]['weight']=G.edges[edge[0],edge[1]]['weight']+1
    else:
        G.add_edge(edge[0],edge[1], weight=1)

## to simulate the quantum
```

```
An_WoN=[]
An_WN=[]
# get real quantum machine error params
backend = provider.get_backend('ibm_nairobi')
noise_model = NoiseModel.from_backend(backend)
coupling_map = backend.configuration().coupling_map
basis_gates = noise_model.basis_gates

for p in range(1,7):
    parameters=np.load("result7_p/p={}.npy".format(str(p)))
    print(parameters)
    circ = create_qaoa(parameters,G)
    #circ.draw('mpl').savefig("circuit.png")
    ##simulate without Error
    sim = Aer.get_backend('qasm_simulator')         #for 1024 shots
    # result = sim.run(circ,shots=100000).result()  #for 100000 shots
    result = sim.run(circ).result()
    Results = result.get_counts()
    ex=result.results[0]
    print(p,str((ex.data.counts['0x1a']+ex.data.counts['0x65'])/ex.shots*100)+"%")
    An_WoN.append((ex.data.counts['0x1a']+ex.data.counts['0x65'])/ex.shots*100)

    # simulate with real quantum machine error params
    circ = create_qaoa(parameters,G)
    result = execute(circ, Aer.get_backend('qasm_simulator'),
                     coupling_map=coupling_map,
                     basis_gates=basis_gates,
                     noise_model=noise_model).result()                                  #for 1024 shots
                     # noise_model=noise_model,shots=100000).result()  #for 100000 shots
    counts = result.get_counts(0)
    ex=result.results[0]
    print(p,str((ex.data.counts['0x1a']+ex.data.counts['0x65'])/ex.shots*100)+"%")
    An_WN.append((ex.data.counts['0x1a']+ex.data.counts['0x65'])/ex.shots*100)
    # Plot p output
    plot_histogram([counts,Results], legend = ['With Noise', 'Without Noise'],sort='desc',number_to_keep=5,figsize=(12,5)).savefig("figure/[shots=1024]P={}".format(p),bbox_inches='tight')

## plot the Accuracy-p fig
import matplotlib.pyplot as plt
import numpy as np

x = np.arange(1, 7, 1)
print(x)
y1 = np.exp(-x)
y2 = np.log(x)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.plot(x, An_WN,'r',marker='v',label="With Noise");
ax1.legend(loc=1)
ax1.set_ylabel('Accuracy With Noise(%)');
ax1.set_xlabel('p');
ax2 = ax1.twinx()
ax2.plot(x, An_WoN, 'g',marker='x',label = "Without Noise")
ax2.legend(loc=2)
ax2.set_xlim([0.5,6.5]);
ax2.set_ylabel('Accuracy Without Noise(%)');
ax2.set_xlabel('p');
fig.savefig("figure/[shots=1024]Accuracy-P")
plt.show()
```

. optimize.py

```
import numpy as np
import warnings
warnings.filterwarnings("ignore")
# import qutip as qt
import random
class graphic:
```

```
    def __init__(self,n,edges):
        self.n=n
        self.G=np.zeros((self.n,self.n))
        for[i,j] in edges:
            self.G[i][j] = self.G[i][j] +
                1

    def ask_min(self):
        MASK=1<<self.n
        ans=0
        sum=0
        for mask in range(MASK):
            sum=0
            for i in range(self.n):
                for j in range(i+1,self.n
                    ):
                    if(((mask&(1<<i))>>i)
                        +((mask&(1<<j))>>j
                        )==1):
                        sum=sum+self.G[i
                            ][j]
            if(sum>ans):
                ans=sum
        return ans

    def ask_C(self):
        # opr=qt.zero(2)
        # print(opr)
        I=qt.tensor([qt.identity(2) for k
             in range(self.n)])
        for i in range(self.n):
            for j in range(i+1,self.n):
                print(qt.tensor([qt.
                    identity(2) if (k != i
                     and k != j) else qt.
                    sigmaz() for k in
                    range(self.n)]))
                opr=(I-qt.tensor([qt.
                    identity(2) if (k != i
                     and k != j) else qt.
                    sigmaz() for k in
                    range(self.n)]))*self.
                    G[i][j]
        return opr

sigma_z=np.array([[1,0],[0,-1]])
sigma_x=np.array([[0,1],[1,0]])
global p
p=0
n=7
graph = np.load('grapy_in.npy')
state0=np.array([1,0])
state1=np.array([0,1])

C=np.zeros((2**n,2**n))
s=np.zeros(2**n)
for edge in graph:
    tmp_C = 1
    for i in range(n):
        tmp_C = np.kron(tmp_C, sigma_z if
             i in edge else np.eye(2))
    C+=1/2*(np.eye(2**n)-tmp_C)
# print(C.shape)

B=np.zeros((2**n,2**n))
for i in range(n):
    B+=np.kron(np.eye(2**i),np.kron(
        sigma_x,np.eye(2**(n-i-1))))

from scipy.linalg import expm,logm
def QAOA(gamma, beta):
    # print(p)
    qs=np.ones(2**n)/np.sqrt(2**n)
    for i in range(p):
        qs=np.dot(expm(-1j*gamma[i]*C),qs
            )
        qs=np.dot(expm(-1j*beta[i]*B),qs)
    # print(qs)
    return np.matmul(qs.conj(),np.matmul(
        C,qs))


from scipy import optimize as opt

def objective(params):
    #print(params);
    # print(p)
    gamma = params[0:p]
    gamma=np.clip(gamma, 0.01, 2*np.pi)
    beta = params[p:]
    # print("gamma:",gamma)
    beta=np.clip(beta, 0.01, np.pi)
    # print("beta",beta)
    return -QAOA(gamma, beta)

# p=int(input())
for pi in range(1,11):
    p=pi
    print("---\n\n P=",p)
    result=opt.basinhopping(objective,x0=
        np.array([np.random.rand(2*p)]),
        niter=100, niter_success=10,
```

```
        minimizer_kwargs={"method":"L-BFGS
        -B"}, disp=0)
    for i in range(10):
        print("-- Running on : "+str(i))
        result0=opt.basinhopping(
            objective,x0=np.array([np.
            random.rand(2*p)]), niter=100,
             niter_success=10,
            minimizer_kwargs={"method":"L-
            BFGS-B"}, disp=0)
        if(result['fun'].real>result0['
            fun'].real):
            result=result0
        np.save('result7_p/p='+str(pi),
            result['x'])
    print("\n",result)
    # np.save('p='+str(pi),result['x'])
```

. optimize.ipynb

```
import numpy as np
import qutip as qt
class graphic:
    def __init__(self,n,edges):
        self.n=n
        self.G=np.zeros((self.n,self.n))
        for[i,j] in edges:
            self.G[i][j] = self.G[i][j] +
                1

    def ask_min(self):
        MASK=1<<self.n
        ans=0
        sum=0
        for mask in range(MASK):
            sum=0
            for i in range(self.n):
                for j in range(i+1,self.n
                    ):
                    if(((mask&(1<<i))>>i)
                        +((mask&(1<<j))>>j
                        )==1):
                        sum=sum+self.G[i
                            ][j]
            if(sum>ans):
                ans=sum
        return ans

    def ask_C(self):
        # opr=qt.zero(2)
        # print(opr)
        I=qt.tensor([qt.identity(2) for k
             in range(self.n)])
        for i in range(self.n):
            for j in range(i+1,self.n):
                print(qt.tensor([qt.
                    identity(2) if (k != i
                     and k != j) else qt.
                    sigmaz() for k in
                    range(self.n)]))
                opr=(I-qt.tensor([qt.
                    identity(2) if (k != i
                     and k != j) else qt.
                    sigmaz() for k in
                    range(self.n)]))*self.
                    G[i][j]
        return opr

import numpy as np
import random

sigma_z=np.array([[1,0],[0,-1]])
sigma_x=np.array([[0,1],[1,0]])

n=10
p=3

def generateGraph() :
    m = random.randint(1, n*(n-1)/2)
    edges = []
    for i in range(m) :
        x = random.randint(0, n-1)
        y = x
        while x == y:
            y = random.randint(0, n-1)
        edges.append([min(x, y), max(x, y
            )])
    return edges

graph = generateGraph()
np.save('grapy_in',graph)
graph = np.load('grapy_in.npy')
print(ar_load)
state0=np.array([1,0])
state1=np.array([0,1])

C=np.zeros((2**n,2**n))
s=np.zeros(2**n)
for edge in graph:
    tmp_C = 1
    for i in range(n):
        tmp_C = np.kron(tmp_C, sigma_z if
             i in edge else np.eye(2))
    C+=1/2*(np.eye(2**n)-tmp_C)
```

```
print(C.shape)

B=np.zeros((2**n,2**n))
for i in range(n):
    B+=np.kron(np.eye(2**i),np.kron(
        sigma_x,np.eye(2**(n-i-1))))

from scipy.linalg import expm,logm
def QAOA(gamma, beta):
    qs=np.ones(2**n)/np.sqrt(2**n)
    for i in range(p):
        qs=np.dot(expm(-1j*gamma[i]*C),qs
            )
        qs=np.dot(expm(-1j*beta[i]*B),qs)
    # print(qs)
    return np.matmul(qs.conj(),np.matmul(
        C,qs))

import numpy as np
from queue import PriorityQueue as PQ
import math
from scipy.integrate import quad
import matplotlib.pyplot as plt
from scipy import optimize as opt

def objective(params):
    #print(params);
    gamma = params[0:p]
    gamma=np.clip(gamma, 0.01, 2*np.pi)
    beta = params[p:]
    # print("gamma:",gamma)
    beta=np.clip(beta, 0.01, np.pi)
    # print("beta",beta)
    return -QAOA(gamma, beta)

def printQAOA(gamma, beta):
    qs=np.ones(2**n)/np.sqrt(2**n)
    # print(np.dot(qs,qs))
    for i in range(p):
        qs=np.dot(expm(-1j*gamma[i]*C),qs
            )
        qs=np.dot(expm(-1j*beta[i]*B),qs)
    return np.matmul(qs.conj(),np.matmul(
        C,qs))

gamma = np.zeros(p)
beta = np.zeros(p)
grid = 7
max_ret = 0
def dfs(w = 0, i = 0):
    global max_ret
    if i == p:
        if w == 0:
            dfs(1, 0)
            return
        else :
            ret = QAOA(gamma, beta)
            max_ret = max(max_ret, ret)
            return
    if w == 0:
        for j in range(grid) :
            gamma[i] = np.pi * 2 / grid *
                j
            dfs(w, i+1)
    else :
        for j in range(grid) :
            beta[i] = np.pi * 2 / grid *
                j
            dfs(w, i+1)

## using https://github.com/hyperopt/
    hyperopt/wiki/FMin
import math
import numpy as np
from scipy.integrate import quad
from hyperopt import fmin, tpe, hp


def objective1(params):
    #print(params);
    gamma=[]
    beta=[]
    for i in range(p):
        gamma.append(params["g"+str(i)])
        beta.append(params["b"+str(i)])
    return -QAOA(gamma, beta)

space=[]
for i in range(p):
    space.append(("g"+str(i),hp.uniform('
        g'+str(i), 0, 2*np.pi)))
    space.append(("b"+str(i),hp.uniform('
        b'+str(i), 0, 2*np.pi)))
space=dict(space)
print(space)

print(graph)
gra = graphic(n, graph)
real_ans = gra.ask_min()
print(real_ans)

best = fmin(
```

```
    fn=objective1,
    space=space,
    algo=tpe.suggest,
    max_evals=500
)
print(best)


result=opt.basinhopping(objective,x0=np.
    array([np.random.rand(2*p)]), niter
    =100, niter_success=10,
    minimizer_kwargs={"method":"L-BFGS-B"
    }, disp=1)
print(result)

dfs(0, 0)
print(max_ret)
```


[1] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser, Quantum computation by adiabatic evolution, arXiv preprint quant-ph/0001106 (2000).

[2] E. Farhi, J. Goldstone, and S. Gutmann, A quantum approximate optimization algorithm, arXiv preprint arXiv:1411.4028 (2014).

[3] L. Zhou, S.-T. Wang, S. Choi, H. Pichler, and M. D. Lukin, Quantum approximate optimization algorithm: Performance, mechanism, and implementation on near-term devices, Physical Review X **10**, 10.1103/physrevx.10.021067 (2020).

[4] Qiskit contributors, Qiskit: An open-source framework for quantum computing (2023).

[5] J. Bergstra, D. Yamins, and D. Cox, Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures, in International conference on machine learning (PMLR, 2013) pp. 115–123.

[6] V. Akshay, H. Philathong, M. E. Morales, and J. D. Biamonte, Reachability deficits in quantum approximate optimization, Physical review letters **124**, 090504 (2020).

[7] J. Basso, E. Farhi, K. Marwaha, B. Villalonga, and L. Zhou, The Quantum Approximate Optimization Algorithm at High Depth for MaxCut on Large-Girth Regular Graphs and the Sherrington-Kirkpatrick Model, in 17th Conference on the Theory of Quantum Computation, Communic Leibniz International Proceedings in Informatics (LIPIcs), Vol. 232, edited by F. Le Gall and T. Morimae (Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022) pp. 7:1–7:21.