Xuenan Wang(xw336)

Rutgers University – New Brunswick

CS672 Quantum Computing: Programs and Systems

Programming Assignment 1: QAOA on Cirq

1. Getting familiar with the QAOA algorithm

1.3 *Modify the code to generate the same set of figures for a larger randomly generated graph. You may choose between 8 to 12 nodes for the max-cut graph. Notice how simulating the quantum circuits for QAOA becomes increasingly time consuming as you increase the number of qubits.*

I selected vertices as [3, 5, 7, 9, 11]. For each vertices number, I connect edges from 0 to 1 and 1 to 2 and all the way to n, where n is the number of vertices. This also means that for a graph with vertices number of 5, there are going to be 4 edges. The following are my results.

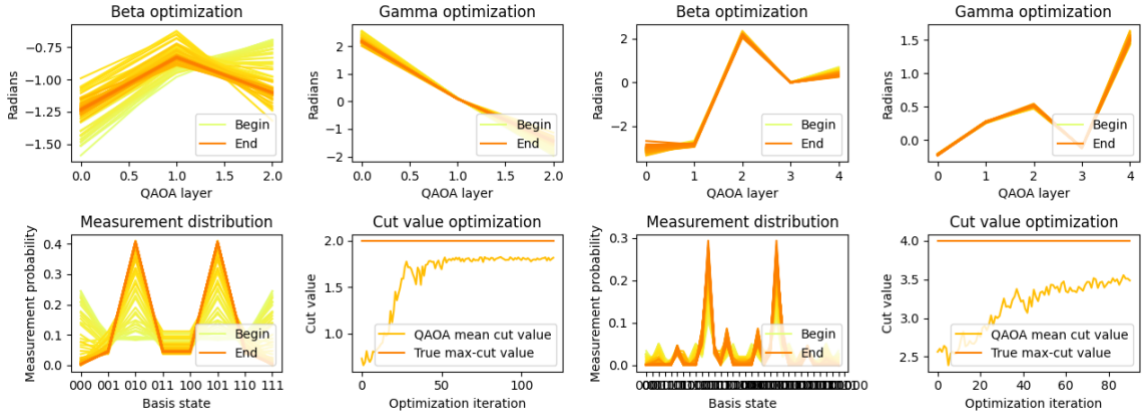


Figure 1. Vertices = 3 (left), Vertices = 5 (right)

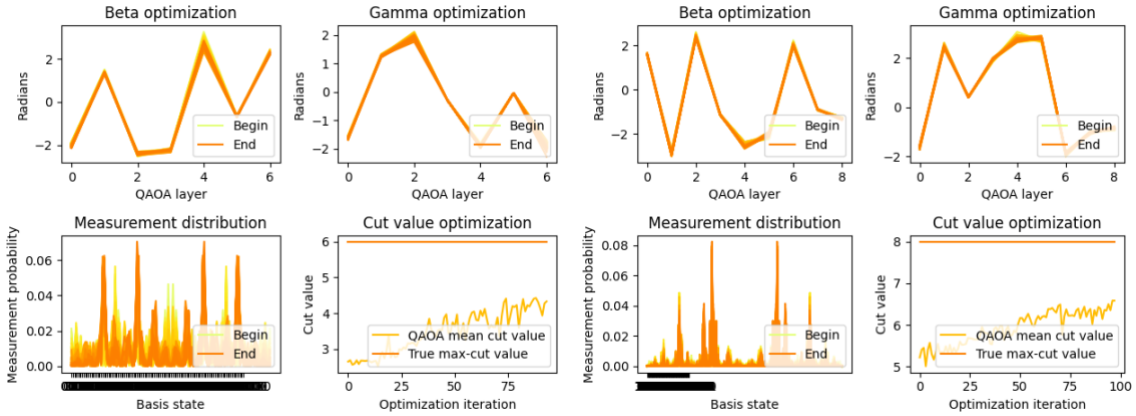


Figure 2. Vertices = 7 (left), Vertices = 9 (right)

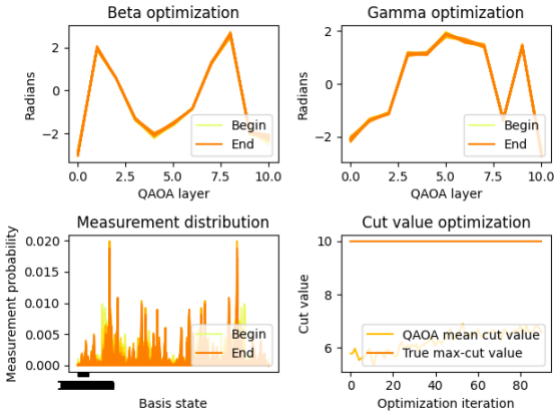


Figure 3. Vertices = 11

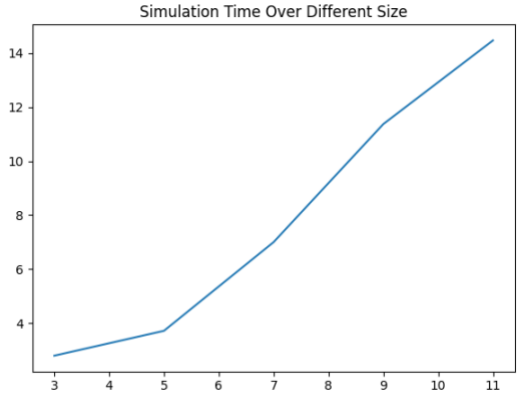


Figure 4. Simulation Time (S) Over Different Vertices

Since we defined our graph as one single line connecting all the vertices, the max-cut value should always be the number of edges. And the distribution should be 010 or 101 and there should be two and only two situations that meet the max-cut.
We can see in bottom-left plot in Figure 1-3, there are always two basis states that have the same and highest measurement probability, and this meets our expectation. However, in the bottom-right plot in Figure 1-3, the QAOA mean cut value is actually less and less accurate with the increase of vertices number. Even though the fact that the QAOA mean cut value is increasing as number of vertices increases, it's not actually reaching our expectation which is the true max cur value.
As we can see in Figure 4, with increase of vertices number, the time we need to simulate this system is increased significantly.

1.4 *Explore the effect of the initial guesses for the beta and gamma parameters, specified in the x0 variable. If you initialize it to constant initial guesses of 0, can the optimization still succeed? Why or why not?*
Same as previous one, I selected the vertices number as [3, 5, 7, 9, 11] and designed the graph as a single linked line. The following are my results.

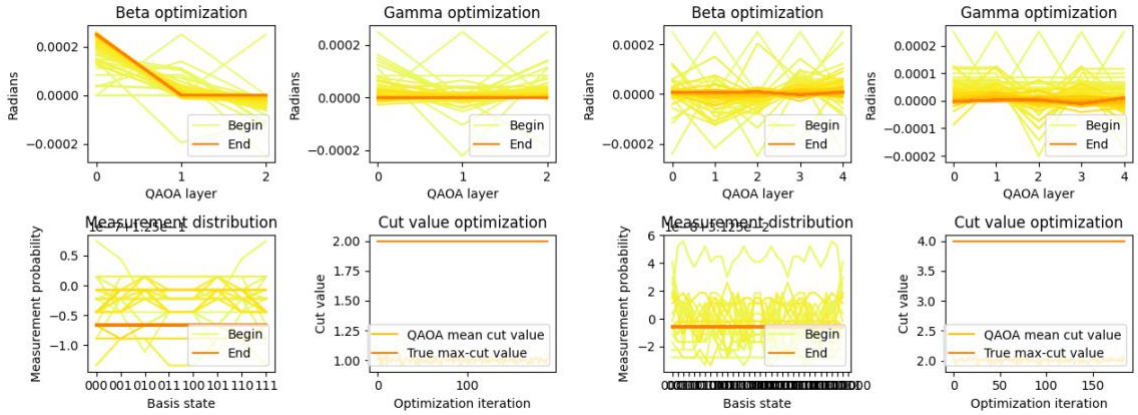


Figure 5. Vertices = 3 (left), Vertices = 5 (right), Started at Zeros

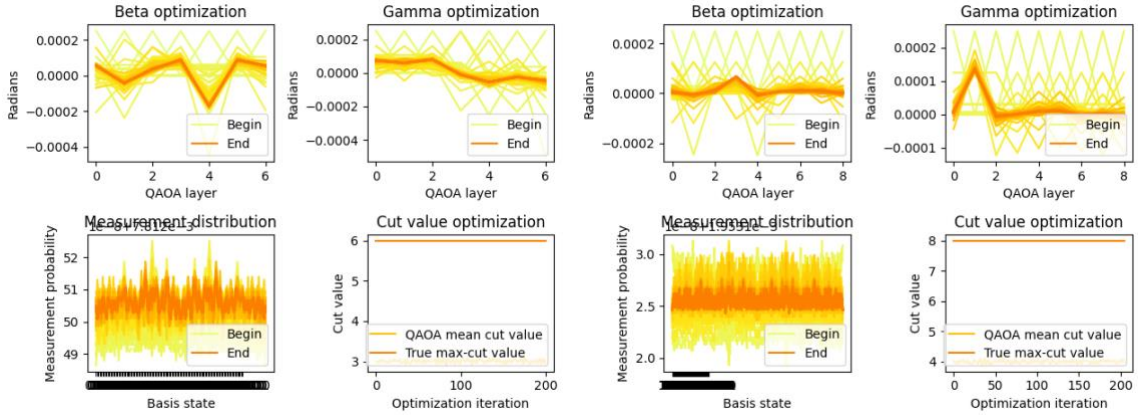


Figure 6. Vertices = 7 (left), Vertices = 9 (right) , Started at Zeros

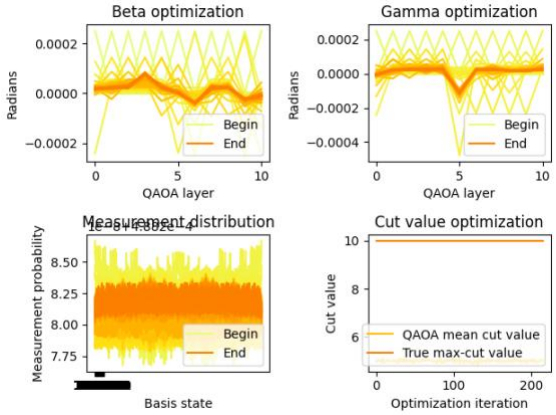


Figure 7. Vertices = 11, Started at Zeros

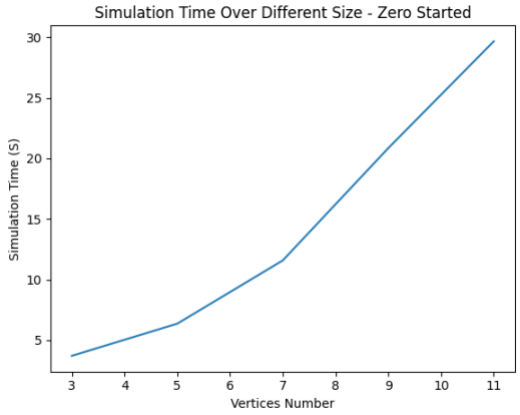


Figure 8. Simulation Time (S) Over Different Vertices, Started at Zeros

We can see from Figure 5-7, when we start at initial guess of zeros, the result are basically noises. This indicates that with constant initial guess as zeros, the simulation cannot success. My guess for this result is that because for a quantum computer, it should always start at a true random point. Even the fact that we are trying to simulate this using pseudo-random numbers, it's still a random starting point. For every qubit, they should be in a entangle state, where each qubit is somehow determined by the others. By initializing the guess to all zeros, we are actually eliminating the possible entangle combinations for qubits. To support my guess, we can observe that with random guesses, the final qubit states are near to their initial states, which indicates that there should be some underlying relationship between the initial states and final states.

And as showed in Figure 8, the time consumption will also increase comparing to random initial guess.

1.5 *You may choose between 8 to 12 nodes for the max-cut graph. Does the approximation ratio improve with increasing number of layers p? Why or why not? What might help improve the approximation ratio?*

I run the original code, and this is the result I got.

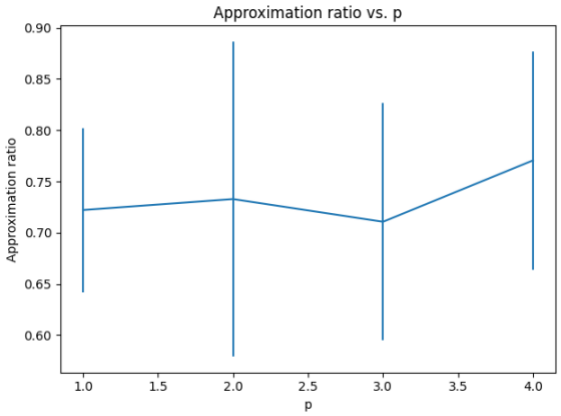


Figure 9. Approximation Ratio VS P (Original Code)

I set n = 8 and max_p = 12. Following are my results.

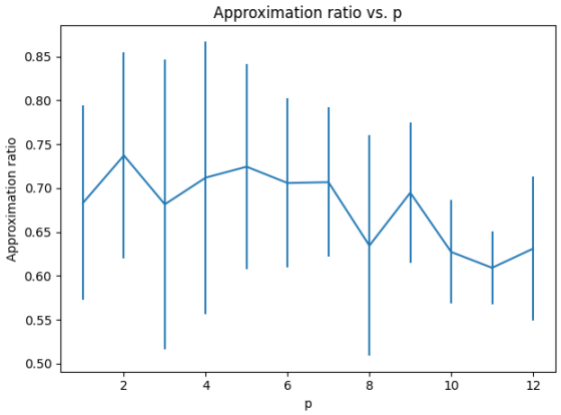


Figure 10. Approximation Ratio VS P (n=8, max_p=12)

The approximation ratio decreases as p increase. I think we can understand this as following: With layer p increases, the complexity and difficulty for simulating this system would be harder and therefore it's reasonable to assume the approximation ratio is lower. Also, for a system with no error correction, more layers mean more errors and with layer number goes up, the errors would be amplified.

As our results above, by increasing the optimization iteration times, the approximation ratio would go up. However, it's reasonable to say that this would be a time-consuming process.

2. Create a plot pertaining to QAOA of your own choice

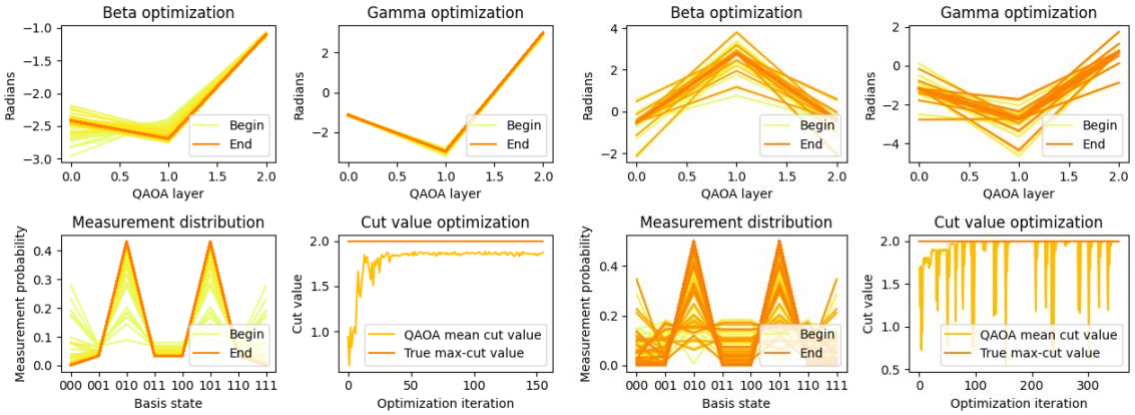
2.1 *State your experiment goals. What parameters are you varying, and what outcomes are you observing?*

My experiment goal: Evaluate the impact of using different optimizers (other than Nelder-Mead) in the scipy.optimize package.

Parameters: Compare optimizers like Nelder-Mead, Powell, CG and BFGS. I'm focusing on cut value approximation curve.

2.2 *Plot the data pertaining to your experiment. Make sure the plot is titled, axes are labeled and properly scaled, and that data series are labeled.*

Following are my results.



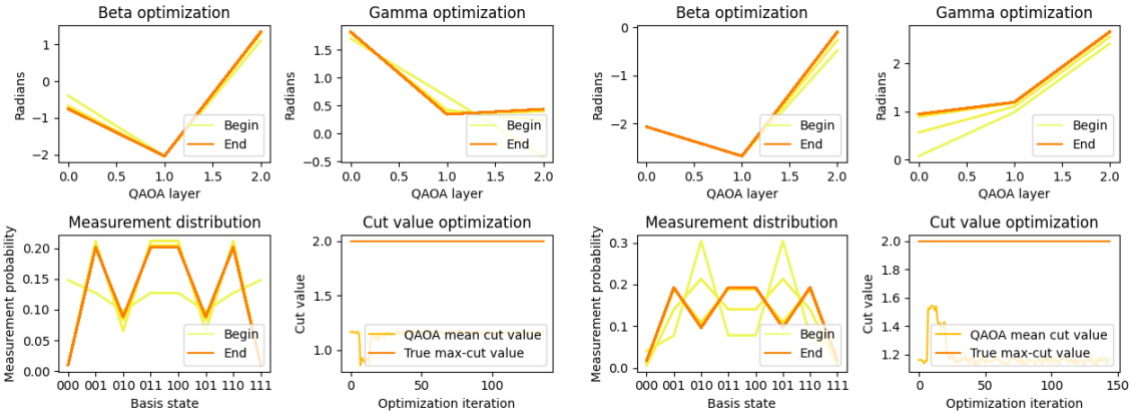


Figure 11. Vertices = 3: Nelder-Mead(top-left), Powell(top-right), CG(bottom-left), BFGS(bottom-right)

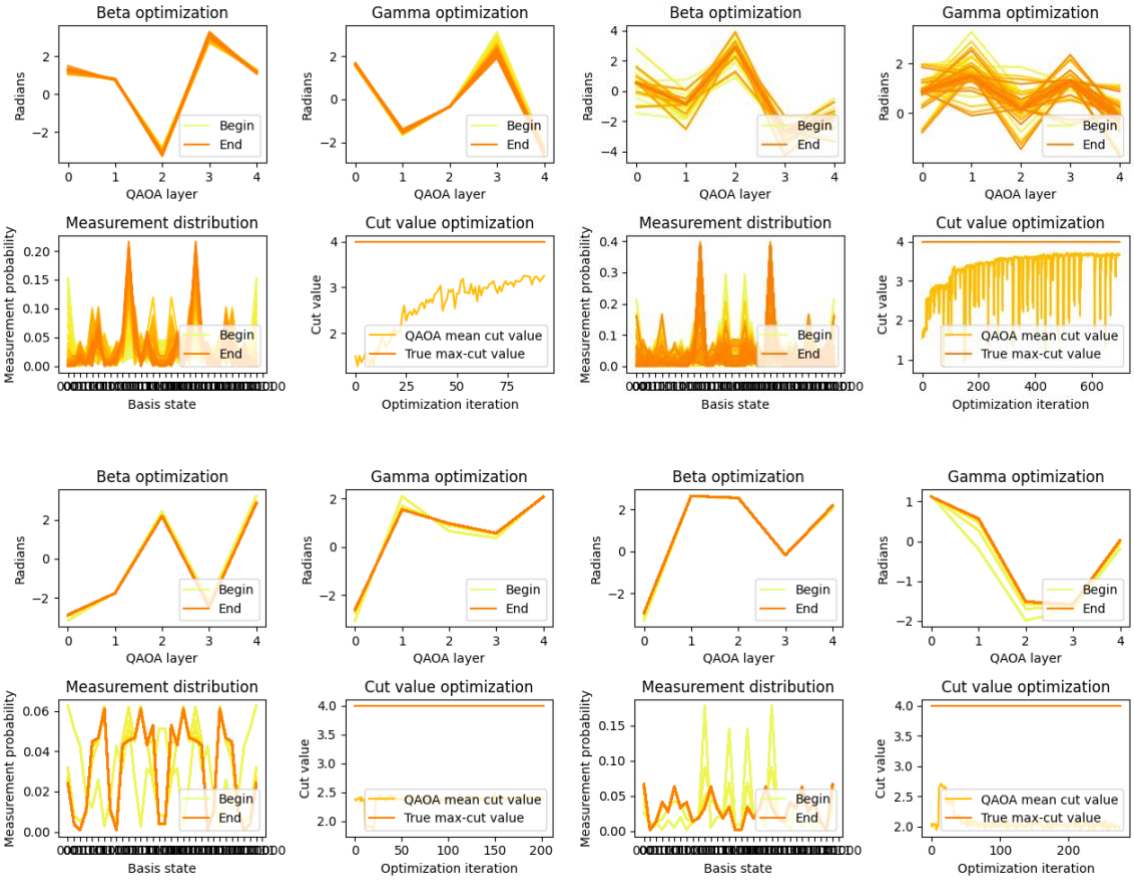


Figure 12. Vertices = 5: Nelder-Mead(top-left), Powell(top-right), CG(bottom-left), BFGS(bottom-right)

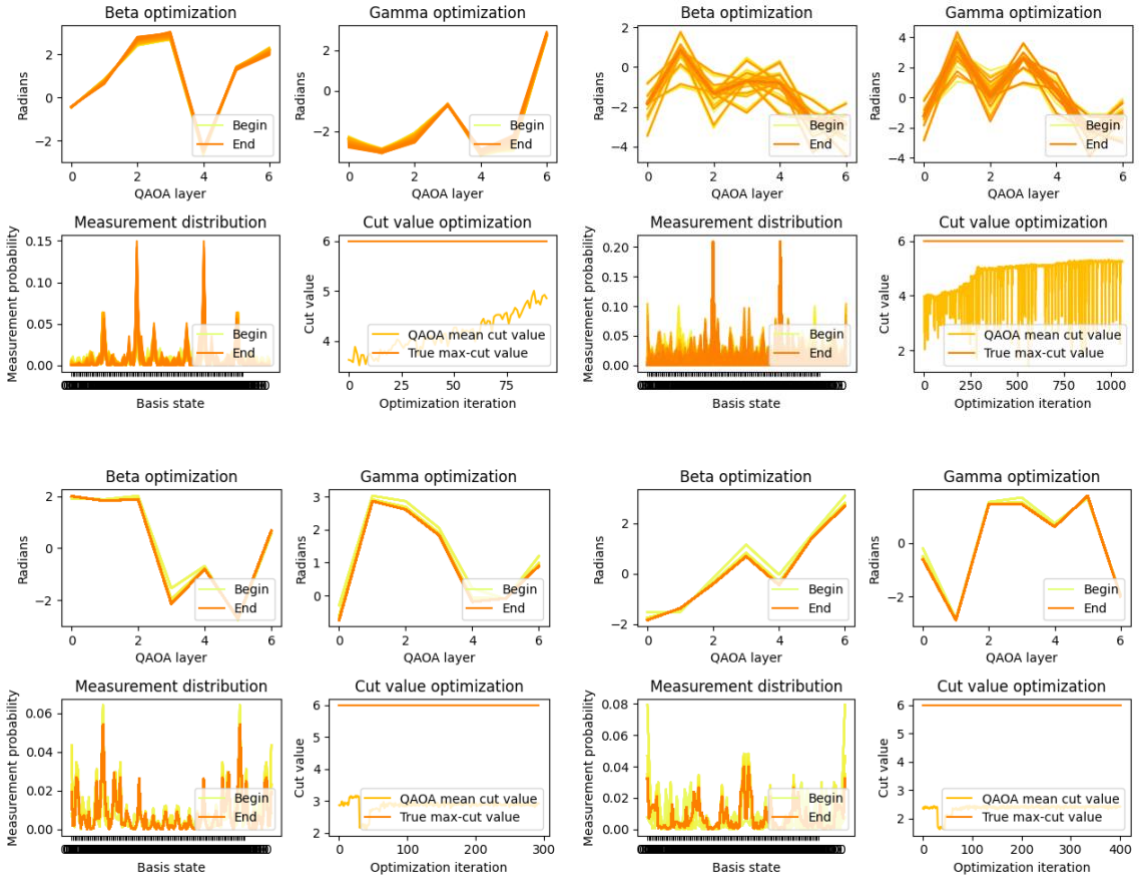
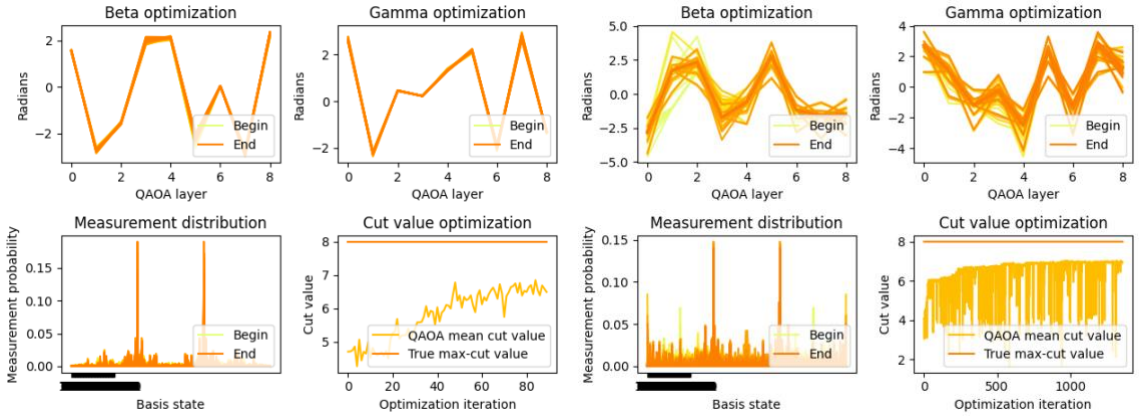


Figure 13. Vertices = 7: Nelder-Mead(top-left), Powell(top-right), CG(bottom-left), BFGS(bottom-right)



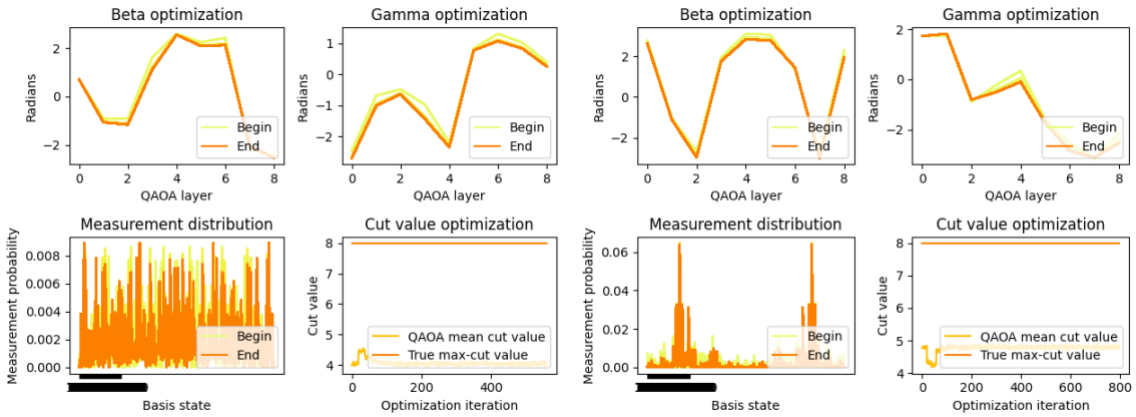


Figure 14. Vertices = 9: Nelder-Mead(top-left), Powell(top-right), CG(bottom-left), BFGS(bottom-right)

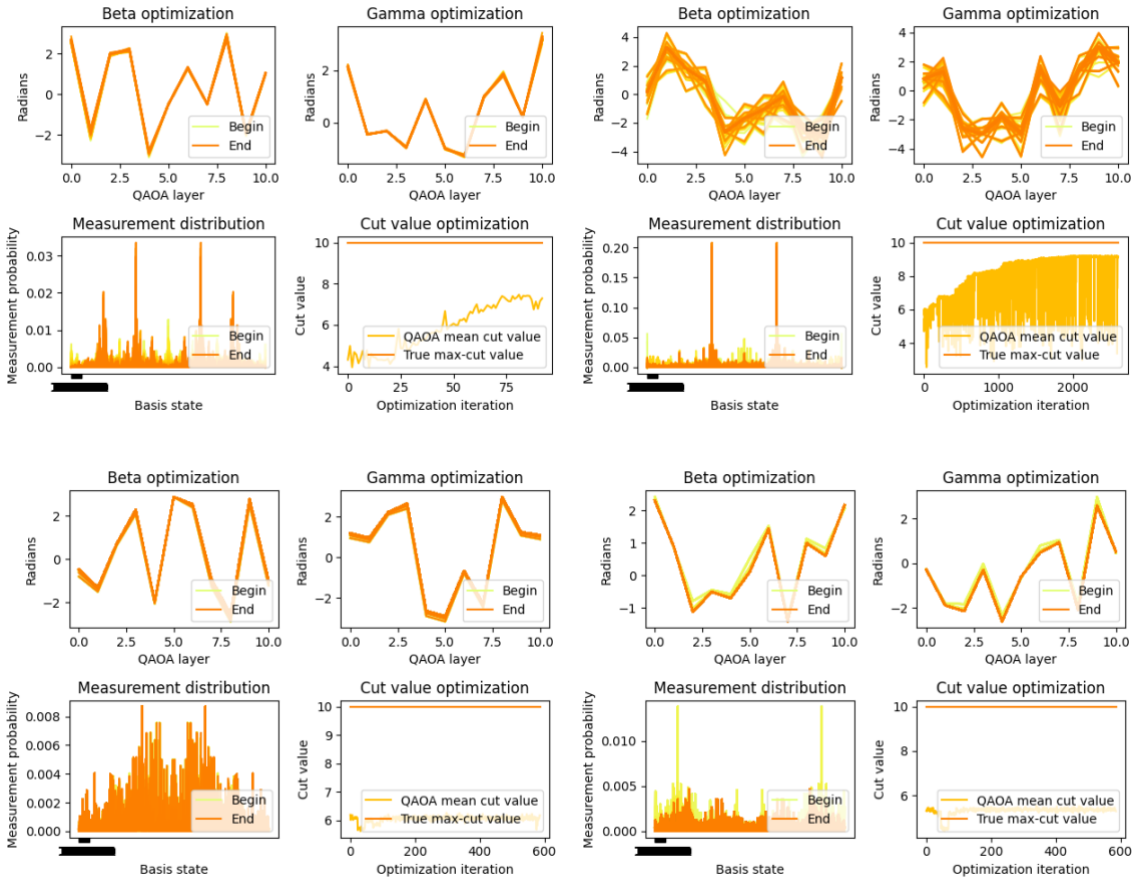


Figure 15. Vertices = 11: Nelder-Mead(top-left), Powell(top-right), CG(bottom-left), BFGS(bottom-right)

We can see that with Powell optimizer, the cut value optimization curve is the best. Nelder-Mead optimizer is the second place. CG optimizer and BFGS optimizer are terrible no matter how many vertices we have.

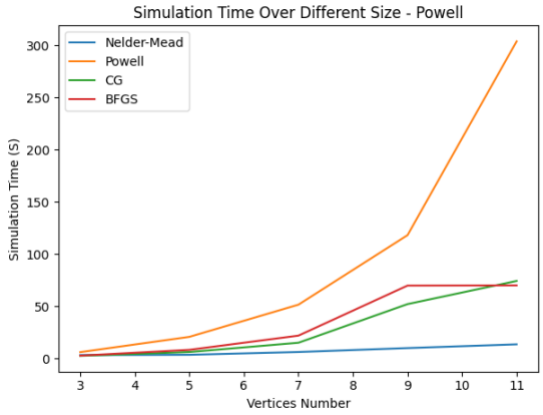


Figure 16. Simulation Time Over Different Vertices Number

We can see that Nelder-Mead optimizer is the best as of time consumption. CG optimizer and BFGS optimizer are similar. Powell optimizer is the most time-consuming one and is significantly larger than others.

Therefore, Nelder-Mead is the best choice considering accuracy and time consumption.

2.3 *Write a paragraph discussing the data plot. As with any discussion about plots, use the handy mnemonic "WALTER" to make sure your readers understand the plot: a) What is the plot and Why it is important? b) Axes. c) Labels. d) Trends. e) Exceptions. f) Recap.*

Explanations are as above.

I tried to expand this experiment by increasing the repetition number from 1000 to 10000. And this is the simulation time plot I got.

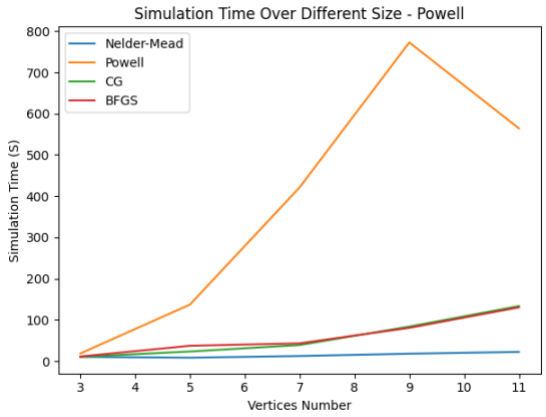


Figure 17. Simulation Time Over Different Vertices Number – 10000 Repetitions

I notice that the simulation time for Powell optimizer went down after certain vertices number. This is strange so I tried to replicate this and for this time, I tried to run the simulation with repetition of 30000.

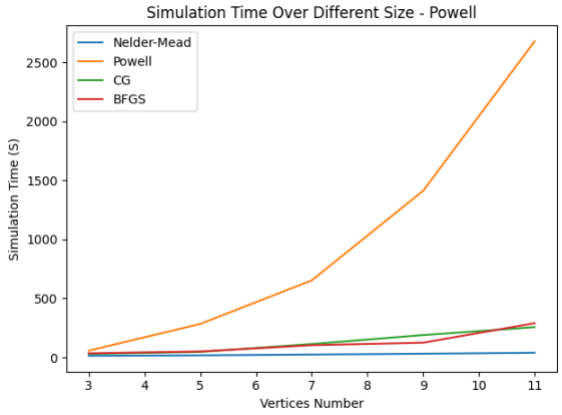


Figure 18. Simulation Time Over Different Vertices Number – 30000 Repetitions

For this time, the decreasing segment disappeared. I assume that the decreasing segment is an occasional data point that cannot represent the big picture.

Appendix: Source Code

2020_672_qaoa.py

```
"""Runs the Quantum Approximate Optimization Algorithm on Max-Cut.

=== EXAMPLE OUTPUT ===

Example QAOA circuit:
0         1         2
│         │         │
H         H         H
│         │         │
ZZ────────ZZ^0.974  │
│         │         │
Rx(0.51π) ZZ────────ZZ^0.974
│         │         │
│         Rx(0.51π) Rx(0.51π)
│         │         │
M('m')────M─────────M
│         │         │
Optimizing objective function ...
The largest cut value found was 2.0.
The largest possible cut has size 2.0.
"""

import itertools

import numpy as np
import networkx
import scipy.optimize
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from matplotlib.lines import Line2D

import time
```

```
import cirq

def main(repetitions=1000, maxiter=64, size=3, optimizor='Nelder-Mead'):

    # Set problem parameters
    n=size
    p=size

    # Generate a random 3-regular graph on n nodes
    # graph = networkx.random_regular_graph(3, n)

    # Recreate the minimal 3-node example from class
    graph = networkx.Graph()
    nodeList = list(range(size))
    graph.add_nodes_from(nodeList)
    edgeList = []
    for i in range(size-1):
        edgeList.append((i, i+1))
    graph.add_edges_from(edgeList)
    networkx.draw(graph)
    plt.show()
    # quit()

    # Each node in n nodes of the MAX-CUT graph corresponds to one of n qubits in the quantum circuit.
    # The state vector across the qubits encodes a node partitioning
    qubits = cirq.LineQubit.range(n)

    hadamard_circuit = cirq.Circuit(
        # Prepare uniform superposition
        cirq.H.on_each(*qubits)
    )
    simulator = cirq.Simulator()
    print ("Put the initial state vector in a superposition of all possible node partitionings:")
    print (simulator.simulate(hadamard_circuit, initial_state=0))
    # quit()
```

```
# Print an example circuit
# Provide classical parameters such that the classical computer can control quantum partitioning
betas = np.random.uniform(-np.pi, np.pi, size=p)
gammas = np.random.uniform(-np.pi, np.pi, size=p)
# betas = [0 for _ in range(p)]
# gammas = [0 for _ in range(p)]
circuit = qaoa_max_cut_circuit(qubits, betas, gammas, graph)
print('Example QAOA circuit:')
print(circuit.to_text_diagram(transpose=True))
# quit()

# Create variables to store the largest cut and cut value found
largest_cut_found = None
largest_cut_value_found = 0

# Initialize simulator
simulator = cirq.Simulator()

# For visualizing the optimization iterations
betas_sched = []
gammas_sched = []
meas_prob_dist_sched = []
cut_mean_sched = []

# Define objective function (we'll use the negative expected cut value)
def f(x):
    # Create circuit
    betas = x[:p]
    gammas = x[p:]
    betas_sched.append(betas)
    gammas_sched.append(gammas)

    # Perform a series of operations parameterized by classical parameters
    # such that the final state vector is a superposition of good partitionings
    circuit = qaoa_max_cut_circuit(qubits, betas, gammas, graph)

    # we also want a simulation where the final quantum state vector is not collapsed via measurement
```

```
    circuit_no_measurement = qaoa_max_cut_circuit_no_measurement(qubits, betas, gammas, graph)
    final_state = simulator.simulate(circuit_no_measurement).final_state
    print("The final quantum state vector without measurement collapse:")
    print(final_state)
    # quit()

    # from amplitudes to measurement probabilities
    meas_prob_dist_sched.append(np.square(np.absolute(final_state)))

    # Sample bitstrings from circuit
    result = simulator.run(circuit, repetitions=repetitions)
    bitstrings = result.measurements['m']
    # Process bitstrings
    nonlocal largest_cut_found
    nonlocal largest_cut_value_found
    values = cut_values(bitstrings, graph)
    max_value_index = np.argmax(values)
    max_value = values[max_value_index]
    if max_value > largest_cut_value_found:
        largest_cut_value_found = max_value
        largest_cut_found = bitstrings[max_value_index]
    mean = np.mean(values)
    cut_mean_sched.append(mean)

    return -mean

# Provide classical parameters such that the classical computer can control quantum partitioning
# Pick an initial guess
x0 = np.random.uniform(-np.pi, np.pi, size=2 * p)
# x0 = [0 for _ in range(2 * p)]

# Optimize for a good set of gammas and betas
# Optimize f
print('Optimizing objective function ...')
scipy.optimize.minimize(f,
                        x0,
                        # method='Nelder-Mead',
```

```
                    method=optimizor,
                    options={'maxiter': maxiter})

# Compute best possible cut value via brute force search
all_bitstrings = np.array(list(itertools.product(range(2), repeat=n)))
all_values = cut_values(all_bitstrings, graph)
max_cut_value = np.max(all_values)

# Print the results
print('The largest cut value found was {}.'.format(largest_cut_value_found))
print('The largest possible cut has size {}.'.format(max_cut_value))

# Visualize the data
fig, ((bax,gax),(dax,max)) = plt.subplots(2,2)

# Create a color coding and legend
max_opt_iter = len(meas_prob_dist_sched)
colors = plt.cm.Wistia(np.linspace(0,1,max_opt_iter))
custom_legend = [Line2D([0], [0], color=colors[0]),
                 Line2D([0], [0], color=colors[-1])]

# Plot gammas and betas as function of optimization iteration
qaoa_layers = [i for i in range(p)]
bax.set_title('Beta optimization')
bax.set_ylabel('Radians')
bax.set_xlabel('QAOA layer')
gax.set_title('Gamma optimization')
gax.set_ylabel('Radians')
gax.set_xlabel('QAOA layer')
for opt_iter, (layer_betas,layer_gammas) in enumerate(zip(betas_sched,gammas_sched)):
    bax.plot(qaoa_layers, layer_betas,  color=colors[opt_iter])
    gax.plot(qaoa_layers, layer_gammas, color=colors[opt_iter])
bax.legend(custom_legend, ['Begin', 'End'], loc='lower right')
gax.legend(custom_legend, ['Begin', 'End'], loc='lower right')

# Plot meas_prob_dist_sched as function of optimization iteration
basis_states = [i for i in range(1<<n)]
```

```
    dax.set_title('Measurement distribution')
    dax.set_ylabel('Measurement probability')
    dax.set_xlabel('Basis state')
    dax.xaxis.set_major_formatter(ticker.StrMethodFormatter("{x:03b}"))
    dax.xaxis.set_ticks(np.arange(0, 111, 1))
    for opt_iter, histogram in enumerate(meas_prob_dist_sched):
        dax.plot(basis_states, histogram, color=colors[opt_iter])
    dax.legend(custom_legend, ['Begin', 'End'], loc='lower right')

    # Plot mean cut value vs. optimization iteration
    max.set_title('Cut value optimization')
    max.set_ylabel('Cut value')
    max.set_xlabel('Optimization iteration')
    max.plot(cut_mean_sched,color=colors[max_opt_iter//2],label='QAOA mean cut value')
    max.plot([max_cut_value for i in range(max_opt_iter)],color=colors[-1],label='True max-cut value')
    max.legend(loc='lower right')

    plt.tight_layout()
    # print('About to savefig')
    plt.savefig('./2020_672_'+str(size)+'V_'+optimizor+'rep'+str(repetitions)+'.png')
    # plt.savefig('./2020_672_'+str(size)+'V_0Start.png')
    # plt.show()


def rzz(rads):
    """Returns a gate with the matrix exp(-i Z⊗Z rads)."""
    return cirq.ZZPowGate(exponent=2 * rads / np.pi, global_shift=-0.5)


def qaoa_max_cut_unitary(qubits, betas, gammas,
                         graph):  # Nodes should be integers
    for beta, gamma in zip(betas, gammas):
        # Need an operator (quantum gate) that encodes an edge
        yield (
            rzz(-0.5 * gamma).on(qubits[i], qubits[j]) for i, j in graph.edges)
        yield cirq.rx(2 * beta).on_each(*qubits)
```

```
def qaoa_max_cut_circuit(qubits, betas, gammas,
                         graph):  # Nodes should be integers
    return cirq.Circuit(
        # Prepare uniform superposition
        cirq.H.on_each(*qubits),
        # Apply QAOA unitary
        qaoa_max_cut_unitary(qubits, betas, gammas, graph),
        # Measure
        cirq.measure(*qubits, key='m'))


def qaoa_max_cut_circuit_no_measurement(qubits, betas, gammas,
                         graph):  # Nodes should be integers
    return cirq.Circuit(
        # Prepare uniform superposition
        cirq.H.on_each(*qubits),
        # Apply QAOA unitary
        qaoa_max_cut_unitary(qubits, betas, gammas, graph))


def cut_values(bitstrings, graph):
    mat = networkx.adjacency_matrix(graph, nodelist=sorted(graph.nodes))
    vecs = (-1)**bitstrings
    vals = 0.5 * np.sum(vecs * (mat @ vecs.T).T, axis=-1)
    vals = 0.5 * (graph.size() - vals)
    return vals


if __name__ == '__main__':
    # t1 = time.time()
    # main(size=3, optimizor='Nelder-Mead')
    # t2 = time.time()
    # main(size=5, optimizor='Nelder-Mead')
    # t3 = time.time()
    # main(size=7, optimizor='Nelder-Mead')
    # t4 = time.time()
    # main(size=9, optimizor='Nelder-Mead')
    # t5 = time.time()
```

```
# main(size=11, optimizor='Nelder-Mead')
# t6 = time.time()
# timeDiff1 = [t2-t1, t3-t2, t4-t3, t5-t4, t6-t5]
# plt.clf()
# plt.plot([3,5,7,9,11], timeDiff1)
# plt.title('Simulation Time Over Different Size - Zero Started')
# plt.xlabel('Vertices Number')
# plt.ylabel('Simulation Time (S)')
# plt.savefig('SimulationTime_0Start.png')

t1 = time.time()
main(repetitions=30000, size=3, optimizor='Nelder-Mead')
t2 = time.time()
main(repetitions=30000, size=5, optimizor='Nelder-Mead')
t3 = time.time()
main(repetitions=30000, size=7, optimizor='Nelder-Mead')
t4 = time.time()
main(repetitions=30000, size=9, optimizor='Nelder-Mead')
t5 = time.time()
main(repetitions=30000, size=11, optimizor='Nelder-Mead')
t6 = time.time()
timeDiff1 = [t2-t1, t3-t2, t4-t3, t5-t4, t6-t5]

t1 = time.time()
main(repetitions=30000, size=3, optimizor='Powell')
t2 = time.time()
main(repetitions=30000, size=5, optimizor='Powell')
t3 = time.time()
main(repetitions=30000, size=7, optimizor='Powell')
t4 = time.time()
main(repetitions=30000, size=9, optimizor='Powell')
t5 = time.time()
main(repetitions=30000, size=11, optimizor='Powell')
t6 = time.time()
timeDiff2 = [t2-t1, t3-t2, t4-t3, t5-t4, t6-t5]

t1 = time.time()
```

```
main(repetitions=30000, size=3, optimizor='CG')
t2 = time.time()
main(repetitions=30000, size=5, optimizor='CG')
t3 = time.time()
main(repetitions=30000, size=7, optimizor='CG')
t4 = time.time()
main(repetitions=30000, size=9, optimizor='CG')
t5 = time.time()
main(repetitions=30000, size=11, optimizor='CG')
t6 = time.time()
timeDiff3 = [t2-t1, t3-t2, t4-t3, t5-t4, t6-t5]

t1 = time.time()
main(repetitions=30000, size=3, optimizor='BFGS')
t2 = time.time()
main(repetitions=30000, size=5, optimizor='BFGS')
t3 = time.time()
main(repetitions=30000, size=7, optimizor='BFGS')
t4 = time.time()
main(repetitions=30000, size=9, optimizor='BFGS')
t5 = time.time()
main(repetitions=30000, size=11, optimizor='BFGS')
t6 = time.time()
timeDiff4 = [t2-t1, t3-t2, t4-t3, t5-t4, t6-t5]

plt.clf()
plt.plot([3,5,7,9,11], timeDiff1, label='Nelder-Mead')
plt.plot([3,5,7,9,11], timeDiff2, label='Powell')
plt.plot([3,5,7,9,11], timeDiff3, label='CG')
plt.plot([3,5,7,9,11], timeDiff4, label='BFGS')
plt.legend()
plt.title('Simulation Time Over Different Size - Powell')
plt.xlabel('Vertices Number')
plt.ylabel('Simulation Time (S)')
plt.savefig('SimulationTime_Powell_30000.png')
```

vary_p_qaoa.py

```
"""Runs the Quantum Approximate Optimization Algorithm on Max-Cut.

=== EXAMPLE OUTPUT ===

Example QAOA circuit:
0        1        2
|        |        |
H        H        H
|        |        |
ZZ———————ZZ^0.974  |
|        |        |
Rx(0.51π) ZZ———————ZZ^0.974
|        |        |
|        Rx(0.51π) Rx(0.51π)
|        |        |
M('m')———M———————M
|        |        |
Optimizing objective function ...
The largest cut value found was 2.0.
The largest possible cut has size 2.0.
The approximation ratio achieved is 1.0.
"""

import itertools

import numpy as np
import networkx
import scipy.optimize
import matplotlib.pyplot as plt
import statistics

import cirq

def main(repetitions=1000, maxiter=64):

    # Set problem parameters
    n=8
```

```
max_p=12

# For plotting approximation ratios as a function of varying p
approx_ratios_dict = {}
for p in range(1,max_p+1):
    # For each key p we will store a set of values from trials
    approx_ratios_dict[p]=set()

# Each trial uses a new randomly generated problem instance
for trial in range(16):

    # Generate a random 3-regular graph on n nodes
    graph = networkx.random_regular_graph(3, n)

    for p in range(1,max_p+1):

        # Each node in n nodes of the MAX-CUT graph corresponds to one of n qubits in the quantum circuit.
        # The state vector across the qubits encodes a node partitioning
        qubits = cirq.LineQubit.range(n)

        # Create variables to store the largest cut and cut value found
        largest_cut_found = None
        largest_cut_value_found = 0
        final_mean = None

        # Initialize simulator
        simulator = cirq.Simulator()

        # Define objective function (we'll use the negative expected cut value)
        def f(x):
            # Create circuit
            betas = x[:p]
            gammas = x[p:]

            # Perform a series of operations parameterized by classical parameters
            # such that the final state vector is a superposition of good partitionings
            circuit = qaoa_max_cut_circuit(qubits, betas, gammas, graph)
```

```
    # Sample bitstrings from circuit
    result = simulator.run(circuit, repetitions=repetitions)
    bitstrings = result.measurements['m']
    # Process bitstrings
    nonlocal largest_cut_found
    nonlocal largest_cut_value_found
    values = cut_values(bitstrings, graph)
    max_value_index = np.argmax(values)
    max_value = values[max_value_index]
    if max_value > largest_cut_value_found:
        largest_cut_value_found = max_value
        largest_cut_found = bitstrings[max_value_index]
    mean = np.mean(values)
    nonlocal final_mean
    final_mean = mean

    return -mean

# Provide classical parameters such that the classical computer can control quantum partitioning
# Pick an initial guess
x0 = np.random.uniform(-0.5*np.pi, 0.5*np.pi, size=2 * p)

# Optimize for a good set of gammas and betas
# Optimize f
print('Optimizing objective function ...')
scipy.optimize.minimize(f,
                        x0,
                        method='Nelder-Mead',
                        options={'maxiter': maxiter})

# Compute best possible cut value via brute force search
all_bitstrings = np.array(list(itertools.product(range(2), repeat=n)))
all_values = cut_values(all_bitstrings, graph)
max_cut_value = np.max(all_values)

approx_ratios_dict[p].add( final_mean / max_cut_value )
```

```
        # Print the results
        print('trial={},p={}'.format(trial,p))
        print('The largest cut value found was {}.'.format(largest_cut_value_found))
        print('The largest possible cut has size {}.'.format(max_cut_value))

    fig, ax = plt.subplots()
    ax.set_title('Approximation ratio vs. p')
    ax.set_ylabel('Approximation ratio')
    ax.set_xlabel('p')
    ps, approx_ratios = zip(*sorted(approx_ratios_dict.items())) # unpack a list of pairs into two tuples
    ax.errorbar(ps, [statistics.mean(set) for set in approx_ratios],
                yerr=[statistics.stdev(set) for set in approx_ratios])
    plt.tight_layout()
    plt.savefig('./vary_p_qaoa_p12.png')
    # plt.show()


def rzz(rads):
    """Returns a gate with the matrix exp(-i Z⊗Z rads)."""
    return cirq.ZZPowGate(exponent=2 * rads / np.pi, global_shift=-0.5)


def qaoa_max_cut_unitary(qubits, betas, gammas,
                         graph):  # Nodes should be integers
    for beta, gamma in zip(betas, gammas):
        # Need an operator (quantum gate) that encodes an edge
        yield (
            rzz(-0.5 * gamma).on(qubits[i], qubits[j]) for i, j in graph.edges)
        yield cirq.rx(2 * beta).on_each(*qubits)


def qaoa_max_cut_circuit(qubits, betas, gammas,
                         graph):  # Nodes should be integers
    return cirq.Circuit(
        # Prepare uniform superposition
        cirq.H.on_each(*qubits),
        # Apply QAOA unitary
```

```
        qaoa_max_cut_unitary(qubits, betas, gammas, graph),
        # Measure
        cirq.measure(*qubits, key='m'))


def qaoa_max_cut_circuit_no_measurement(qubits, betas, gammas,
                        graph):  # Nodes should be integers
    return cirq.Circuit(
        # Prepare uniform superposition
        cirq.H.on_each(*qubits),
        # Apply QAOA unitary
        qaoa_max_cut_unitary(qubits, betas, gammas, graph))


def cut_values(bitstrings, graph):
    mat = networkx.adjacency_matrix(graph, nodelist=sorted(graph.nodes))
    vecs = (-1)**bitstrings
    vals = 0.5 * np.sum(vecs * (mat @ vecs.T).T, axis=-1)
    vals = 0.5 * (graph.size() - vals)
    return vals


if __name__ == '__main__':
    main()
```