QOSF MENTORSHIP PROGRAM SCREENING TASK

Table of Contents

Instructions

> INSTRUCTIONS TO USE THE PACKAGE
>
> - The requirements needed to run this project are given the environment.yml file, It has few commented pip install packages which neds to be run seperately on command line **after installing conda packages** since their are some

> campatibilty issues.
> - Go inside the task4 folder
> - run command for veiwing all addional options ->> python main.py
> - The optional parameters to customize :
>   - -h, --help show this help message and exit
>   - -s SHOTS, --shots SHOTS Set the number of shots
>   - -d DEPTH, --depth DEPTH Set the depth of the Quantum circuit
>   - -n NUM, --num NUM Set the number of qubits
>   - -i ITER, --iter ITER Set the number of iterations for optimal gamma and beta
>   - -g GRAPH, --graph GRAPH Select the type of graph

> IN CASE YOU CAN'T RUN THE PACKAGE, PLEASE CHECK THE SAMPLE OUTPUT IN sample_output.ipynb

0. Introduction

This repository contains the solution of the task4 of QOSF mentorship program.

For this task we had to implement the QAOA algorithm for MaxCut problem the any weighted graph i.e. generalize/extend the idea of unwighted graph to weighted graphs.

The real challenge about the MaxCut probelm is that it comes under the class of problem which are of combinational complexity in nature when solved classically by the use of any turing machine.
Now according to the original paper the objetcive function for this set of classes- defined on bit string of size n- is :

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

Where $z = z1, z2, z3, ..$ is the bit string and $C_\alpha(z) = 1$ if z satisfies clause α and 0 if it doesn't.

Since the Qauntum computers operate on a completely different paradigm, they can leverage the nature of physics at the fundamental levels to by pass this combinational limit and could potentially solve the problems under this category more efficiently.Here the authors have tried to demonstrate that very possiblity of solving the MaxCut problem by QAOA algorithm (original paper link)

The background is to consider the graphs to be weighted and each vertex is a part of connected graph; The aim is to find a optimal cut or rather an arrangement of seperation in which the sum of weights connecting the opposite groups is maximal. Thus transforming the given general clause condition to suite our MaxCut problem, the unitary operator has been defined as

$$U(C, \gamma) = e^{-i\gamma C} = \prod_{\alpha=1}^{m} e^{-i\gamma C_\alpha}$$

The next operator b has been defined as :

$$U(C, \beta) = e^{-i\beta B} = \prod_{j=1}^{m} e^{-i\beta \sigma_j^x}$$

The idea behind using this is that, suppose that we use only $U(C, \gamma)$ then we might come across a state which is the eigen state after which we wouldn't be able to cross it, if the maximum state isn't this. Therefore we need a function which can help us gain momentum when trapped in such state (local maxima) if it hadn't been there our momentum would have been reduced to near zero value which would in fact prevent any more change in the value; this is analogous to having genetic mutations in genetic algorithms which is also used for the same purpose. For this reason we prefer a unitary operator U_B to commute with a U_C.

The initial state is usually prefered to be in superposition of all states, therefore the initial state can be given by:

$$|s\rangle = 1/\sqrt{2^n} \sum_z |z\rangle$$

We can now notice what we have actually achieved by expressing the MaxCut problem using Qubits. what we have essentialy done is reduce the problem from finding the optimal grapgh arrangement to finding the optimal values of β and γ which we can easily do using classical techniques such as classical optimizers or even grid search. Below is the same thing written mathematically (here p is the depth of the circuit or rather nmumber totterized states),

$$|\gamma, \beta\rangle = U(B, \beta_p)U(C, \gamma_p)\ldots U(B, \beta_1)U(C, \gamma_1)\,|s\rangle$$

Let the expectation value of clause C over γ and β be defnied by $F_p(\gamma, \beta)$:

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

Then the maximum of $F_p(\gamma, \beta)$ is M_p:

$$M_p = max_{\gamma,\beta} F_p(\gamma, \beta)$$

They also show that the:

$$lim_{p->\infty} M_p = max_z C(z)$$

The above equation can be interpreted as if we were to use this approximate technique and do it infinite times then we would eventually reach the maximum state, This can be taken as an omen - We could say that this approximation is good enough for practical purposes. Also One observation to be noted here is that if p doesn't grow with n is then complexity is given by $O(m^2 + mn)$ which means the complexity doesn't grow combinationally anymore. A simple grid search on $[0, 2\pi]^p x[0, \pi]^p$ would be enough.

Finally to extract the result from the obtained optimal γ and β is easy, we just have to know the corresponding bitstrings with highest probability which we get, it can then be used to get the grouping of vertices and logically the edges connecting opposites groups would be cut.

This particular implementation has been adapted from the tutorial of Jack Ceroni on MaxCut for unwieghted graphs. (link to the tutorial)

1. Importing Packages

```
from qiskit import *
from qiskit.tools.visualization import plot_histogram
from matplotlib import pyplot as plt
%matplotlib inline
import networkx as nx
import random
from scipy.optimize import minimize
print("imports successful")
```

```
print('imports successful')
```

```
imports successful
```

2. Defning the Graph stucture

We implement the graph and edge structure below.

We are going to encode the graph in code by using the "edge list" implementation of graph

```
class Graph:
    def __init__(self, edges_set):
        self.edges_set = edges_set
        self.node_set = []
        for i in edges_set:
            if (i.start_node not in self.node_set):
                self.node_set.append(i.start_node)
            if (i.end_node not in self.node_set):
                self.node_set.append(i.end_node)

class Edge:
    def __init__(self, start_node, end_node, weight = 1):
        self.start_node = start_node
        self.end_node = end_node
        self.weight = weight
```

Here we are making a random graph which will used for our demonstration of algorithm later.

Insturction to use the following edge_list struture:

the Edge struture is given by:

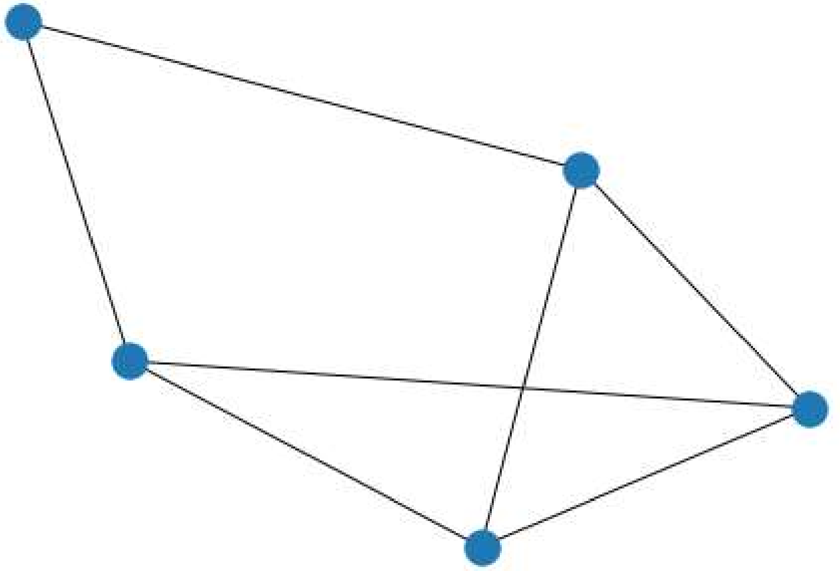
```
Edge(vertex 1, vertex 2, weight)
```

```
Without the weight the default value will be taken to be 1.
```

```
#triangle
#edge_list = [Edge(0,1,2), Edge(1,2,1), Edge(2,0,2)]
#square
#edge_list = [Edge(0,1,2), Edge(1,2,1), Edge(2,3,1), Edge(3,0,2)]
#cool graph
edge_list = [Edge(0, 1, 2), Edge(1, 2, 2), Edge(2, 3, 2), Edge(3,4), Edge(4,0), Edge(1,3), Edge(2,4)]

G = nx.Graph()

for z in edge_list:
    G.add_edge(str(z.start_node), str(z.end_node))

nx.draw(G)
plt.savefig('graph.png')
plt.clf()
```

```
<Figure size 432x288 with 0 Axes>
```

The graph:



3. Implementation of MaxCut using QAOA

3.1 Initialize

Initialzing the qubits to equal superposition(all possible combinations would have equal probabilities)

```
def initialize(qc):
    for q in range(qc.num_qubits):
        qc.h(q)
```

3.2 $U(C,\gamma)$

Here we are enconding the weights of each edge to their corresponding qubits and this is how we model our graph into our circuit.

```
def cost_unitary(qc,gamma):
    for i in edge_list:
        qc.cu1(-2*gamma*i.weight, i.start_node, i.end_node)
        qc.u1(gamma*i.weight, i.start_node)
        qc.u1(gamma*i.weight, i.end_node)
```

3.3 $U(C,\beta)$

This helps us to recover from the local maximas that the algorithm might stumble upon

```
def mixer_unitary(qc, beta):
    for i in range(qc.num_qubits):
        qc.rx(2*beta, i)
```

3.4 Assembling the circuit

The previous blocks are now ready to be used to create the circuit and the results are passed to the cost function to calculate the total cost of the current circuit

```
def create_circuit(params, num, depth=2, shots=512):

    gamma = [params[0], params[2], params[4], params[6]]
    beta = [params[1], params[3], params[5], params[7]]

    qc = QuantumCircuit(num)
    initialize(qc)
    qc.barrier()
    for i in range(0, depth):
        cost_unitary(qc, gamma[i])
        qc.barrier()
        mixer_unitary(qc, beta[i])
    qc.measure_all()

    backend = Aer.get_backend('qasm_simulator')
    results = execute(qc,backend=backend,shots=shots).result()
    #print("results :: ", results.get_counts())

    return results.get_counts()
```

3.5 Cost function

The cost function is optimized based on the values of the beta and gamma i.e. it tries to find the optimal values of beta and gamma for which the cost of the circuit is maximal; This cost can be translated to the cost of optimal cut for our problem of MaxCut. The cost function is defined as following:

$$H_c = \sum_{a,b} 1/2(Z_a \bigotimes Z_b - I)$$

where a and b are the different group by each of the Z_i is defined as:

$$f(x) = 1 - 2x$$

where x is each bit of bitstring generated for which we got the probabilities from the quantum circuit (this wqas done to map the bitstring values of 0 and 1 to 1 and -1)

```
def cost_function(params):

    qubit_count = create_circuit(params, num, depth, shots)
    print("qubit count :: ",qubit_count)
    bit_strings = list(qubit_count.keys())

    total_cost = 0

    for bit_string in bit_strings:
        each_bs_cost = 0
        bit_string_encoding = bit_string[::-1]
        for j in edge_list:
            #multiplying the whole equation by -1 so that later minize function from scipy can be used to optimize
            each_bs_cost += -1*0.5* j.weight *( 1 -( (1 - 2*int(bit_string_encoding[j.start_node])) * (1 - 2*int(bit_string_encodi
        #print("bit string freq :: ", qubit_count.get(bit_string))
        total_cost += each_bs_cost*qubit_count.get(bit_string)

    print("Cost: "+str(-1*total_cost/shots))

    return total_cost
```

3.6 Visualize

Visualizing the output using the matplotlib package

```
def visualize(f):
    # Creates visualization of the optimal state
    #curently here
```

```
#curently here
nums = []
freq = []

for k,v in f.items():
    number = 0
    #print("key :: ",k, " values :: ",v)
    for j in range(0, len(k)):
        number += 2**(len(k)-j-1)*int(k[j])
    if (number in nums):
        freq[nums.index(number)] = freq[nums.index(number)] + v
    else:
        nums.append(number)
        freq.append(v)

freq = [s/sum(freq) for s in freq]

print(nums)
print(freq)

x = range(0, 2**num)
y = []
for i in range(0, len(x)):
    if (i in nums):
        y.append(freq[nums.index(i)])
    else:
        y.append(0)

plt.bar(x, y)
plt.show()
```

3.7 Optimizizng the circuit params

Till now we have successfully transformed the given MaxCut problem into an optimization problem over β and γ.

We use a classical optimizer COBYLA to go over the possible combination for them

Ultimately we achieve a list of bitstrings with their probabilities, among them the two bitstring with highest probabilities will constitute our solution. These bitstring are actually complementary of each other meaning if you reverse one of their bit mappings i.e. 0->1 and 1->0 you'll get the other string. They canm be directly translated into our solution by using the following key:

$$\text{for every } i\text{th vertex, it belongs to the group which is at the } i\text{th place in the bitstring}$$

```
def do_max_cut():

    # Defines the optimization method
    init =[float(random.randint(-314, 314))/float(100) for i in range(0, 8)]
    out = minimize(cost_function, x0=init, method="COBYLA", options={'maxiter':2000})
    print(out)

    optimal_params = out['x']
    f = create_circuit(optimal_params, num, depth)
    #print(f)
    visualize(f)

    return


shots = 1024
depth = 2
num = 5
do_max_cut()
```

```
qubit count ::  {'00000': 22, '00001': 23, '10000': 3, '10001': 12, '10010': 46, '10011': 78, '10100': 24, '10101': 8, '10110
Cost: 5.3369140625
qubit count ::  {'00000': 20, '10000': 3, '10001': 21, '10010': 55, '10011': 4, '10100': 38, '10101': 13, '10110': 22, '10111
Cost: 5.9482421875
qubit count ::  {'00000': 22, '00001': 18, '10000': 14, '10001': 9, '10010': 73, '10011': 20, '10100': 43, '10101': 54, '1011
Cost: 5.98046875
qubit count ::  {'00000': 70, '00001': 8, '10000': 25, '10001': 52, '10010': 21, '10011': 66, '10100': 11, '10101': 17, '1011
Cost: 4.4375
qubit count ::  {'00000': 34, '00001': 7, '10000': 33, '10001': 47, '10010': 30, '10011': 59, '10100': 17, '10101': 15, '1011
Cost: 4.9990234375
qubit count ::  {'00000': 28, '00001': 16, '10000': 15, '10001': 10, '10010': 66, '10011': 16, '10100': 46, '10101': 49, '101
Cost: 5.8916015625
qubit count ::  {'00000': 38, '00001': 24, '10000': 8, '10001': 7, '10010': 58, '10011': 19, '10100': 42, '10101': 57, '10110
Cost: 5.83984375
qubit count ::  {'00000': 22, '00001': 23, '10000': 11, '10001': 6, '10010': 74, '10011': 15, '10100': 42, '10101': 40, '1011
Cost: 5.76171875
qubit count ::  {'00000': 27, '00001': 21, '10000': 6, '10001': 12, '10010': 67, '10011': 21, '10100': 46, '10101': 40, '1011
Cost: 5.8857421875
qubit count ::  {'00000': 12, '10000': 12, '10001': 32, '10010': 43, '10011': 101, '10100': 16, '10101': 14, '10110': 57, '10
Cost: 5.6611328125
qubit count ::  {'00000': 4, '00001': 41, '10000': 38, '10001': 11, '10010': 80, '10011': 11, '10100': 29, '10101': 34, '1011
Cost: 6.0693359375
qubit count ::  {'00000': 5, '00001': 13, '10000': 6, '10001': 10, '10010': 169, '10011': 5, '10100': 43, '10101': 39, '10110
Cost: 6.9541015625
qubit count ::  {'00001': 7, '10000': 11, '10001': 27, '10010': 148, '10011': 19, '10100': 29, '10101': 53, '10110': 34, '110
Cost: 6.7353515625
qubit count ::  {'00000': 4, '00001': 14, '10000': 5, '10001': 4, '10010': 151, '10100': 26, '10101': 44, '10110': 19, '10111
Cost: 7.021484375
qubit count ::  {'00000': 9, '00001': 55, '10000': 20, '10001': 6, '10010': 80, '10011': 23, '10100': 19, '10101': 32, '10110
Cost: 5.947265625
qubit count ::  {'00000': 8, '00001': 12, '10000': 6, '10001': 6, '10010': 161, '10100': 31, '10101': 43, '10110': 23, '10111
Cost: 7.0087890625
qubit count ::  {'00000': 15, '00001': 3, '10000': 1, '10001': 11, '10010': 185, '10011': 10, '10100': 25, '10101': 24, '1011
Cost: 6.7275390625
qubit count ::  {'00000': 8, '00001': 11, '10000': 7, '10001': 3, '10010': 162, '10011': 1, '10100': 30, '10101': 40, '10110'
Cost: 6.943359375
qubit count ::  {'00000': 6, '00001': 2, '10000': 1, '10001': 9, '10010': 189, '10011': 2, '10100': 37, '10101': 32, '10110':
Cost: 6.95703125
qubit count ::  {'00000': 9, '00001': 5, '10000': 4, '10001': 14, '10010': 166, '10011': 4, '10100': 33, '10101': 42, '10110'
Cost: 6.93359375
qubit count ::  {'00000': 8, '00001': 8, '10000': 6, '10001': 4, '10010': 109, '10011': 13, '10100': 29, '10101': 61, '10110'
Cost: 6.619140625
qubit count ::  {'10001': 12, '10010': 182, '10011': 3, '10100': 34, '10101': 50, '10110': 24, '10111': 12, '11000': 1, '1100
```

```
qubit count ::  {'10001': 12, '10010': 182, '10011': 3, '10100': 34, '10101': 50, '10110': 24, '10111': 12, '11000': 1, '110
Cost: 7.12109375
qubit count ::  {'00000': 2, '00001': 2, '10000': 1, '10001': 6, '10010': 177, '10011': 3, '10100': 33, '10101': 48, '10110':
Cost: 7.1552734375
qubit count ::  {'00000': 3, '00001': 6, '10000': 1, '10001': 6, '10010': 153, '10011': 8, '10100': 45, '10101': 51, '10110':
Cost: 7.0556640625
qubit count ::  {'00000': 3, '00001': 6, '10000': 1, '10001': 6, '10010': 154, '10011': 2, '10100': 40, '10101': 52, '10110':
Cost: 7.103515625
qubit count ::  {'00000': 3, '00001': 1, '10001': 13, '10010': 167, '10100': 39, '10101': 37, '10110': 24, '10111': 17, '110
Cost: 7.14453125
qubit count ::  {'00001': 4, '10000': 2, '10001': 8, '10010': 196, '10011': 1, '10100': 30, '10101': 46, '10110': 23, '10111'
Cost: 7.111328125
qubit count ::  {'00000': 1, '00001': 6, '10000': 2, '10001': 4, '10010': 169, '10011': 4, '10100': 37, '10101': 46, '10110':
Cost: 7.1328125
qubit count ::  {'00000': 3, '00001': 8, '10001': 4, '10010': 173, '10011': 5, '10100': 45, '10101': 50, '10110': 21, '10111'
Cost: 7.095703125
qubit count ::  {'00000': 3, '00001': 4, '10000': 3, '10001': 13, '10010': 173, '10011': 8, '10100': 39, '10101': 41, '10110'
Cost: 7.0576171875
qubit count ::  {'00001': 15, '10000': 3, '10001': 5, '10010': 162, '10011': 3, '10100': 31, '10101': 57, '10110': 21, '10111
Cost: 7.017578125
qubit count ::  {'00001': 4, '10000': 2, '10001': 8, '10010': 160, '10011': 1, '10100': 51, '10101': 55, '10110': 33, '10111'
Cost: 7.15234375
qubit count ::  {'00001': 1, '10001': 9, '10010': 172, '10011': 4, '10100': 40, '10101': 29, '10110': 34, '10111': 15, '11001
Cost: 7.1259765625
qubit count ::  {'00000': 1, '00001': 1, '10000': 3, '10001': 4, '10010': 161, '10011': 5, '10100': 37, '10101': 55, '10110':
Cost: 7.12890625
qubit count ::  {'00000': 1, '00001': 4, '10001': 8, '10010': 158, '10011': 3, '10100': 37, '10101': 48, '10110': 25, '10111'
Cost: 7.1611328125
qubit count ::  {'00000': 2, '00001': 1, '10000': 1, '10001': 12, '10010': 191, '10011': 4, '10100': 29, '10101': 37, '10110'
Cost: 7.1806640625
qubit count ::  {'00001': 5, '10000': 2, '10001': 5, '10010': 176, '10011': 1, '10100': 38, '10101': 41, '10110': 25, '10111'
Cost: 7.15234375
qubit count ::  {'00000': 1, '00001': 3, '10001': 7, '10010': 164, '10011': 2, '10100': 29, '10101': 40, '10110': 37, '10111'
Cost: 7.1318359375
qubit count ::  {'00001': 4, '10001': 12, '10010': 162, '10011': 1, '10100': 36, '10101': 53, '10110': 22, '10111': 9, '11001
Cost: 7.123046875
qubit count ::  {'00000': 5, '00001': 3, '10000': 1, '10001': 9, '10010': 181, '10011': 2, '10100': 35, '10101': 52, '10110':
Cost: 7.05859375
qubit count ::  {'00000': 2, '00001': 9, '10000': 1, '10001': 13, '10010': 173, '10100': 32, '10101': 40, '10110': 18, '10111
Cost: 7.09765625
qubit count ::  {'00001': 1, '10001': 7, '10010': 161, '10011': 3, '10100': 34, '10101': 52, '10110': 21, '10111': 7, '11000'
Cost: 7.1689453125
qubit count ::  {'00001': 3, '10000': 1, '10001': 6, '10010': 181, '10011': 3, '10100': 34, '10101': 42, '10110': 25, '10111'
Cost: 7.251953125
```

```
Cost: 7.251953125
qubit count ::  {'00000': 2, '00001': 3, '10001': 16, '10010': 168, '10011': 2, '10100': 25, '10101': 41, '10110': 36, '10111
Cost: 7.09765625
qubit count ::  {'00000': 2, '00001': 4, '10001': 15, '10010': 168, '10011': 1, '10100': 37, '10101': 46, '10110': 20, '10111
Cost: 7.04296875
qubit count ::  {'00000': 2, '00001': 3, '10000': 1, '10001': 9, '10010': 157, '10011': 4, '10100': 45, '10101': 44, '10110':
Cost: 7.0966796875
qubit count ::  {'00000': 1, '00001': 3, '10001': 11, '10010': 192, '10011': 1, '10100': 28, '10101': 36, '10110': 22, '10111
Cost: 7.1923828125
qubit count ::  {'00000': 1, '00001': 2, '10001': 10, '10010': 153, '10011': 3, '10100': 30, '10101': 51, '10110': 27, '10111
Cost: 7.18359375
qubit count ::  {'00000': 1, '00001': 3, '10001': 3, '10010': 190, '10011': 2, '10100': 36, '10101': 38, '10110': 27, '10111'
Cost: 7.1865234375
qubit count ::  {'00000': 1, '00001': 3, '10001': 8, '10010': 169, '10011': 1, '10100': 28, '10101': 49, '10110': 28, '10111'
Cost: 7.1103515625
qubit count ::  {'00001': 2, '10000': 1, '10001': 9, '10010': 179, '10011': 2, '10100': 35, '10101': 48, '10110': 24, '10111'
Cost: 7.0341796875
qubit count ::  {'00000': 1, '00001': 2, '10001': 7, '10010': 159, '10100': 30, '10101': 56, '10110': 22, '10111': 7, '11001'
Cost: 7.181640625
qubit count ::  {'00000': 1, '00001': 2, '10001': 10, '10010': 158, '10011': 4, '10100': 58, '10101': 42, '10110': 31, '10111
Cost: 7.0859375
qubit count ::  {'00001': 2, '10001': 10, '10010': 184, '10011': 2, '10100': 29, '10101': 50, '10110': 28, '10111': 10, '1100
Cost: 7.205078125
qubit count ::  {'00000': 2, '00001': 1, '10001': 7, '10010': 140, '10011': 4, '10100': 44, '10101': 53, '10110': 31, '10111'
Cost: 7.09375
qubit count ::  {'00000': 2, '00001': 2, '10001': 11, '10010': 169, '10011': 1, '10100': 31, '10101': 46, '10110': 24, '10111
Cost: 7.146484375
qubit count ::  {'10001': 8, '10010': 192, '10100': 31, '10101': 52, '10110': 27, '10111': 7, '11001': 37, '11010': 130, '110
Cost: 7.23046875
qubit count ::  {'00001': 4, '10000': 1, '10001': 5, '10010': 169, '10100': 42, '10101': 37, '10110': 18, '10111': 10, '11001
Cost: 7.1513671875
qubit count ::  {'00000': 2, '00001': 3, '10001': 6, '10010': 169, '10011': 3, '10100': 45, '10101': 36, '10110': 26, '10111'
Cost: 7.109375
qubit count ::  {'00000': 2, '00001': 5, '10001': 8, '10010': 176, '10100': 39, '10101': 58, '10110': 35, '10111': 13, '11000
Cost: 7.1162109375
qubit count ::  {'00000': 1, '00001': 3, '10001': 14, '10010': 168, '10011': 2, '10100': 37, '10101': 53, '10110': 24, '10111
Cost: 7.0791015625
qubit count ::  {'00001': 1, '10001': 8, '10010': 158, '10100': 35, '10101': 38, '10110': 32, '10111': 8, '11000': 2, '11001'
Cost: 7.1328125
qubit count ::  {'00000': 2, '00001': 1, '10001': 3, '10010': 157, '10011': 2, '10100': 30, '10101': 38, '10110': 21, '10111'
Cost: 7.1806640625
qubit count ::  {'00000': 3, '00001': 1, '10000': 2, '10001': 12, '10010': 151, '10011': 2, '10100': 40, '10101': 57, '10110'
Cost: 7.09765625
```

```
qubit count ::  {'00001': 4, '10001': 6, '10010': 158, '10011': 2, '10100': 47, '10101': 54, '10110': 17, '10111': 7, '11000
Cost: 7.162109375
qubit count ::  {'00000': 1, '10001': 6, '10010': 171, '10011': 1, '10100': 35, '10101': 39, '10110': 28, '10111': 9, '11001'
Cost: 7.115234375
qubit count ::  {'10001': 9, '10010': 165, '10011': 2, '10100': 39, '10101': 60, '10110': 23, '10111': 8, '11000': 2, '11001'
Cost: 7.2080078125
qubit count ::  {'00000': 1, '00001': 1, '10001': 6, '10010': 176, '10011': 4, '10100': 34, '10101': 52, '10110': 22, '10111'
Cost: 7.1630859375
qubit count ::  {'00000': 2, '00001': 2, '10001': 10, '10010': 156, '10011': 3, '10100': 45, '10101': 50, '10110': 23, '10111
Cost: 7.107421875
qubit count ::  {'00000': 1, '00001': 6, '10001': 7, '10010': 188, '10011': 1, '10100': 39, '10101': 43, '10110': 18, '10111'
Cost: 7.173828125
qubit count ::  {'00000': 2, '00001': 2, '10001': 13, '10010': 156, '10100': 36, '10101': 41, '10110': 32, '10111': 15, '1100
Cost: 7.115234375
qubit count ::  {'00000': 2, '00001': 4, '10001': 5, '10010': 183, '10011': 1, '10100': 25, '10101': 54, '10110': 23, '10111'
Cost: 7.072265625
qubit count ::  {'00000': 1, '00001': 4, '10001': 9, '10010': 178, '10011': 3, '10100': 37, '10101': 43, '10110': 32, '10111'
Cost: 7.142578125
qubit count ::  {'00000': 3, '00001': 1, '10001': 5, '10010': 181, '10011': 1, '10100': 24, '10101': 50, '10110': 27, '10111'
Cost: 7.1923828125
qubit count ::  {'00000': 1, '00001': 2, '10001': 10, '10010': 173, '10011': 3, '10100': 37, '10101': 41, '10110': 24, '10111
Cost: 7.1591796875
qubit count ::  {'00000': 1, '00001': 3, '10001': 8, '10010': 159, '10011': 4, '10100': 35, '10101': 50, '10110': 27, '10111'
Cost: 7.15234375
qubit count ::  {'00000': 2, '00001': 5, '10001': 6, '10010': 165, '10100': 37, '10101': 54, '10110': 27, '10111': 11, '1100
Cost: 7.15234375
qubit count ::  {'00000': 2, '00001': 3, '10001': 14, '10010': 177, '10011': 3, '10100': 31, '10101': 51, '10110': 31, '10111
Cost: 7.1630859375
qubit count ::  {'00001': 2, '10001': 7, '10010': 178, '10011': 2, '10100': 24, '10101': 49, '10110': 24, '10111': 11, '1100
Cost: 7.201171875
qubit count ::  {'00000': 2, '00001': 1, '10000': 2, '10001': 9, '10010': 165, '10011': 4, '10100': 34, '10101': 44, '10110':
Cost: 7.115234375
qubit count ::  {'00000': 2, '00001': 2, '10000': 1, '10001': 9, '10010': 159, '10011': 2, '10100': 39, '10101': 48, '10110':
Cost: 7.123046875
qubit count ::  {'00000': 2, '00001': 4, '10001': 13, '10010': 173, '10011': 1, '10100': 36, '10101': 39, '10110': 16, '10111
Cost: 7.14453125
     fun: -7316.0
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 82
  status: 1
 success: True
       x: array([-2.64944237,  1.7675702 ,  2.20112671, -2.81169537, -2.81761305,
```

```
      -0.74187671, -0.9653046 , -1.33371904])
[1, 17, 18, 19, 20, 21, 22, 23, 25, 26, 27, 28, 29, 30, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14]
[0.00390625, 0.00390625, 0.140625, 0.001953125, 0.037109375, 0.046875, 0.0234375, 0.01953125, 0.017578125, 0.130859375, 0.02
```

4. Results

For this particular example we notice that the highest probabilities are achieved for the mapping 20(10100) and 11(01011).

we can see they complement each other; they are, in fact, one solution itself. Here we use the mappings of 13 (01101) :

Vertex	Label
4	1
3	0
2	1
1	0
0	0

The final cost of the cutting this graph into the this arrangement is 8.

8 is the value for which we obtain the MaxCut when done manually as well, thus we can say that the QAOA works as expected.