IonQ iQuHack2025 Challenge

Before proceeding, register once. Then, comment out the registration function call to avoid re-registering when running the notebook again.

In [7]:
```
import requests

def register():
    team_name = input("Enter your team name: ")
    in_person = input("Enter participation type - in-person or remote: ")
    data = []

    while email := input("Enter email separated by commas (enter to cancel): "):
        firstname = input("Enter your first name: ")
        lastname = input("Enter your last name: ")
        github = input("Enter your github handle: ")
        data.append((firstname, lastname, email, github))

    for firstname, lastname, email, github in data:
        url = f"https://ionquhack2025.azurewebsites.net/api/registration?TeamName={
        req = requests.post(url)
        print("\n\n\n"+req.text)


# Comment this out after registration:
# register()
```

Enter the `key` you've got on the previous step here:

In [27]:
```
key = "61f7aaaf50e899e65f777e0e159408b5335faf68fd58ee21d8941de1907d3cfb"
```

Did you comment out the `register()` function call?

Please share your research process on GitHub: https://github.com/iQuHACK/2025-IonQ/discussions

Maximum Cut problem (max-cut) with variational Quantum Imaginary Time Evolution (varQITE) method

In this challenge, we demonstrate how to leverage IonQ Forte's industry-leading capabilities to solve instances of the NP-hard combinatorial optimization problem known as Maximum Cut (MaxCut) using a novel variational Quantum Imaginary Time Evolution (varQITE) algorithm developed by IonQ in conjunction with researchers at Oak Ridge National Labs (ORNL).

What's the problem?

MaxCut 101

The Maximum Cut Problem (MaxCut) is a classic combinatorial optimization problem commonly used as an algorithm benchmark by scientific computing researchers. It has numerous applications in a variety of fields: for example, it is used in circuit desing as part of Very Large Scale Integration (VLSI) to find the optimal layout of circuit components; it is used in the study of social networks to identify communities; it is used in computer vision for image segmentation, etc.

MaxCut is a graph problem: given a graph $G = (V, E)$ with vertex set V and edge set E, it asks for a partition of V into sets S and T maximizing the number of edges crossing between S and T.

Think of it like this: you're trying to cut the graph into two pieces, and you want to make the cut so that it slices through as many edges as possible.

Prepare the code environment

First, we'll set up the coding environment and install necessary dependencies.

In [1]:
```
pip install qiskit qiskit-aer networkx numpy pandas -q
```

```
Note: you may need to restart the kernel to use updated packages.
```

To ensure your code runs correctly everywhere, please only use the imported dependencies. Using external libraries may cause your submission to fail due to missing dependencies on our servers.

In [2]:
```
## IonQ, Inc., Copyright (c) 2025,
# All rights reserved.
# Use in source and binary forms of this software, without modification,
# is permitted solely for the purpose of activities associated with the IonQ
# Hackathon at iQuHack2025 hosted by MIT and only during the Feb 1-2, 2025
# duration of such event.

import matplotlib.pyplot as plt
from IPython import display

import networkx as nx
import numpy as np
import pandas as pd
import time

from typing import List
from qiskit import QuantumCircuit, transpile
from qiskit.circuit import ParameterVector
from qiskit.quantum_info import SparsePauliOp
from qiskit_aer import AerSimulator
```

Graph definition

A simple graph, defined using the Python NetworkX library, will serve to illustrate the problem and you can explore further complexities.

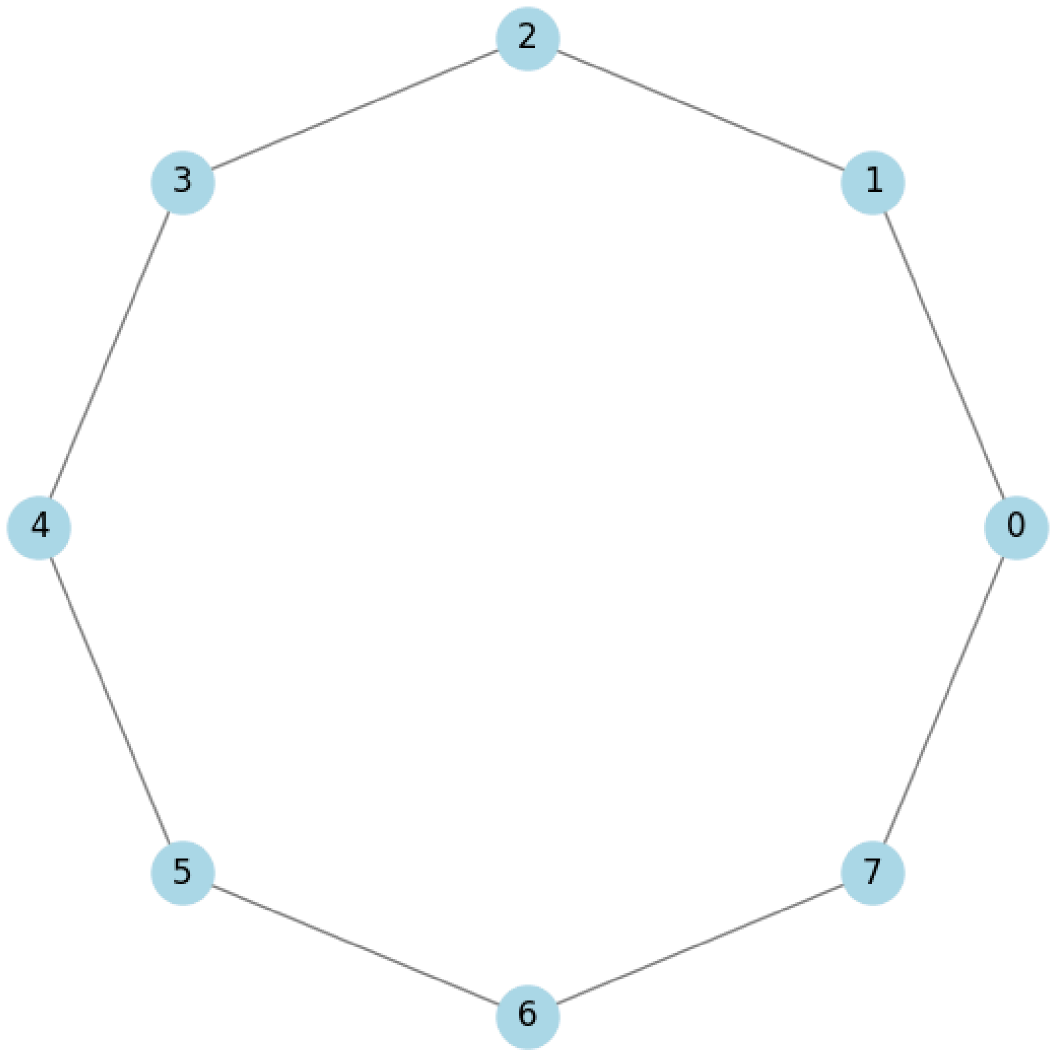
In [3]:
```
# other graphs candidates to check

import networkx as nx
import matplotlib.pyplot as plt
import random

#-> Cycle Graph C8
def cycle_graph_c8():
    G = nx.cycle_graph(8)
    plt.figure(figsize=(6, 6))
    pos = nx.circular_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray', no
    plt.title("Cycle Graph C8")
    plt.show()
    return G

# Path Graph P16
def path_graph_p16():
    G = nx.path_graph(16)
    plt.figure(figsize=(12, 2))
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=True, node_color='lightgreen', edge_color='gray', n
    plt.title("Path Graph P16")
    plt.show()
    return G

#-> Complete Bipartite Graph K8,8
def complete_bipartite_graph_k88():
    G = nx.complete_bipartite_graph(8, 8)
    plt.figure(figsize=(8, 6))
    pos = nx.bipartite_layout(G, nodes=range(8))
    nx.draw(G, pos, with_labels=True, node_color=['lightcoral'] * 8 + ['lightblue']
            edge_color='gray', node_size=300)
    plt.title("Complete Bipartite Graph K8,8")
    plt.show()
    return G

#-> Complete Bipartite Graph K8,8
def complete_bipartite_graph_k_nn(n):
    G = nx.complete_bipartite_graph(n, n)
    plt.figure(figsize=(8, 6))
    pos = nx.bipartite_layout(G, nodes=range(n))
    nx.draw(G, pos, with_labels=True, node_color=['lightcoral'] * n + ['lightblue']
            edge_color='gray', node_size=300)
    plt.title("Complete Bipartite Graph K{},{}".format(n,n))
    plt.show()
    return G

# Star Graph S16
def star_graph_s16():
    G = nx.star_graph(16)
    plt.figure(figsize=(8, 8))
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=True, node_color='gold', edge_color='gray', node_si
    plt.title("Star Graph S16")
    plt.show()
    return G

# Grid Graph 8x4
def grid_graph_8x4():
    G = nx.grid_graph(dim=[8, 4])
    plt.figure(figsize=(12, 6))
    pos = {node: node for node in G.nodes()}
    nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray', no
    plt.title("Grid Graph 8x4")
    plt.show()
    return G

# Grid Graph 8x4
def grid_graph_nxm(n,m):
    G = nx.grid_graph(dim=[n, m])
    plt.figure(figsize=(12, 6))
    pos = {node: node for node in G.nodes()}
    nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray', no
    plt.title("Grid Graph {}x{}".format(n,m))
    plt.show()
    return G


#-> 4-Regular Graph with 8 Vertices
def regular_graph_4_8():
    G = nx.random_regular_graph(d=4, n=8, seed=42)
    plt.figure(figsize=(6, 6))
    pos = nx.circular_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='lightgreen', edge_color='gray', n
    plt.title("4-Regular Graph with 8 Vertices")
    plt.show()
    return G

#-> Cubic (3-Regular) Graph with 16 Vertices
def cubic_graph_3_16():
    G = nx.random_regular_graph(d=3, n=16, seed=42)
    plt.figure(figsize=(8, 6))
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=True, node_color='lightcoral', edge_color='gray', n
    plt.title("Cubic (3-Regular) Graph with 16 Vertices")
    plt.show()
    return G

# Disjoint Union of Four C4 Cycles
def disjoint_union_c4():
    cycles = [nx.cycle_graph(4) for _ in range(4)]
    G = nx.disjoint_union_all(cycles)
    plt.figure(figsize=(12, 6))
    pos = {}
    shift_x = 0
    for component in nx.connected_components(G):
        subgraph = G.subgraph(component)
        pos_sub = nx.circular_layout(subgraph, scale=1, center=(shift_x, 0))
        pos.update(pos_sub)
        shift_x += 3
    nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray', no
    plt.title("Disjoint Union of Four C4 Cycles")
    plt.show()
    return G

# Complete Bipartite Graph K16,16
def complete_bipartite_graph_k1616():
    G = nx.complete_bipartite_graph(16, 16)
    plt.figure(figsize=(12, 6))
    pos = nx.bipartite_layout(G, nodes=range(16))
    nx.draw(G, pos, with_labels=False, node_color=['lightcoral'] * 16 + ['lightblue
            edge_color='gray', node_size=100)
    plt.title("Complete Bipartite Graph K16,16")
    plt.show()
    return G

# 5-Dimensional Hypercube Graph Q5
def hypercube_graph_q5():
    G = nx.hypercube_graph(5)
    plt.figure(figsize=(10, 8))
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=False, node_color='lightgreen', edge_color='gray',
    plt.title("5-Dimensional Hypercube Graph Q5")
    plt.show()
    return G

# Tree Graph with 8 Vertices
def tree_graph_8():
    G = nx.balanced_tree(r=2, h=2)
    G.add_edge(6, 7)
    plt.figure(figsize=(8, 6))
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray', no
    plt.title("Tree Graph with 8 Vertices")
    plt.show()
    return G

# Wheel Graph W16
def wheel_graph_w16():
    G = nx.wheel_graph(16)
    plt.figure(figsize=(8, 8))
    pos = nx.circular_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='lightcoral', edge_color='gray', n
    plt.title("Wheel Graph W16")
    plt.show()
    return G

#-> Random Connected Graph with 16 Vertices
def random_connected_graph_16(p=0.15):
    #n, p = 16, 0.25
    n=16
    while True:
        G = nx.erdos_renyi_graph(n, p, seed=random.randint(1, 10000))
        if nx.is_connected(G):
            break
    plt.figure(figsize=(10, 8))
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=False, node_color='lightgreen', edge_color='gray',
    plt.title("Random Connected Graph with 16 Vertices")
    plt.show()
    return G

# Expander Graph with 32 Vertices
def expander_graph_32():
    G = nx.random_regular_graph(4, 32, seed=42)
    plt.figure(figsize=(10, 8))
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=False, node_color='lightblue', edge_color='gray', n
    plt.title("Expander Graph with 32 Vertices")
    plt.show()
    return G

#-> Expander Graph with n Vertices
def expander_graph_n(n):
    G = nx.random_regular_graph(4, n, seed=42)
    plt.figure(figsize=(10, 8))
    pos = nx.spring_layout(G, seed=42)
    nx.draw(G, pos, with_labels=False, node_color='lightblue', edge_color='gray', n
    plt.title("Expander Graph with {} Vertices".format(n))
    plt.show()
    return G

# Planar Connected Graph with 16 Vertices
def planar_connected_graph_16():
    G = nx.grid_graph(dim=[8, 2])
    G = nx.convert_node_labels_to_integers(G)
    additional_edges = [(0, 9), (1, 10), (2, 11), (3, 12), (4, 13), (5, 14), (6, 15
                        (7, 15), (8, 7)]#, (6, 15), (14, 1), (1, 13), (10, 9), (0,
    G.add_edges_from([e for e in additional_edges if e[0] < 16 and e[1] < 16])
    assert nx.check_planarity(G)[0], "Graph is not planar."
    pos = {node: (node // 2, node % 2) for node in G.nodes()}
    plt.figure(figsize=(16, 8))
    nx.draw(G, pos, with_labels=False, node_color='lightcoral', edge_color='gray',
    plt.title("Planar Connected Graph with 16 Vertices")
    plt.axis('equal')
    plt.show()
    return G
```

We've suggested a few graph ideas above.

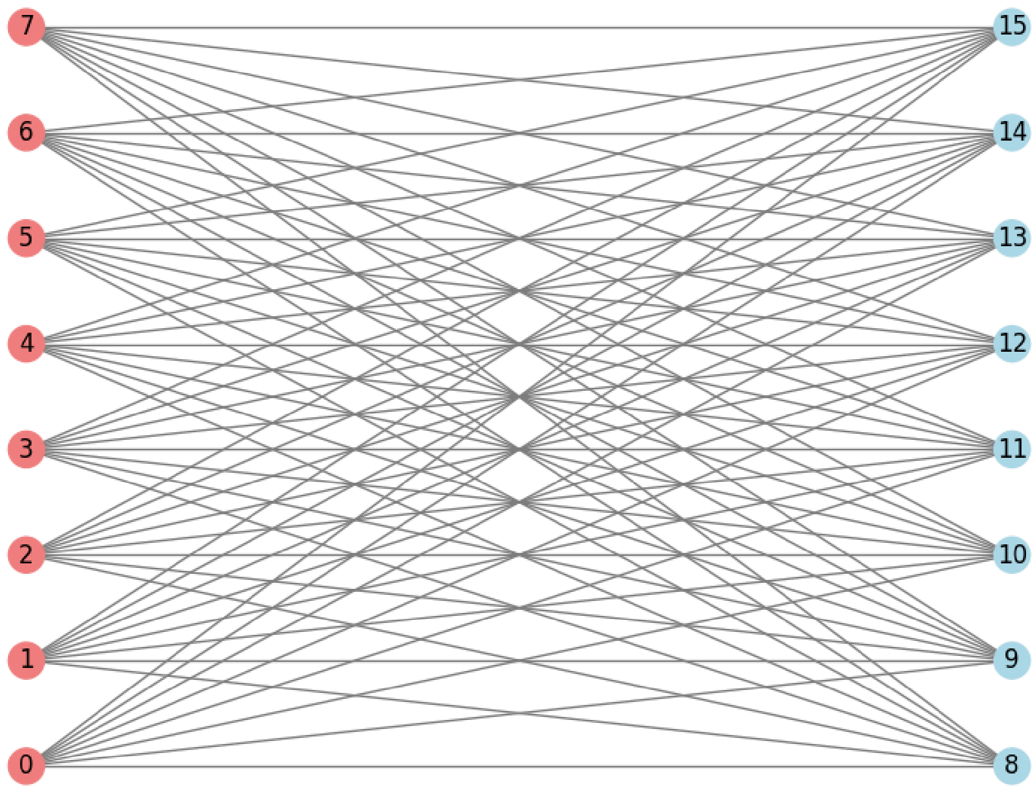
The list below highlights those achievable with minimal computational resources, whether on a local laptop or a cloud instance. Some are simply too large.

In [4]:
```
# Choose your favorite graph and build your winning ansatz!

graph1 = cycle_graph_c8()
graph2 = complete_bipartite_graph_k88()
graph3 = complete_bipartite_graph_k_nn(5)
graph4 = regular_graph_4_8()
graph5 = cubic_graph_3_16()
graph6 = random_connected_graph_16(p=0.18)
graph7 = expander_graph_n(16)
#graph8 = -> make your own cool graph
```

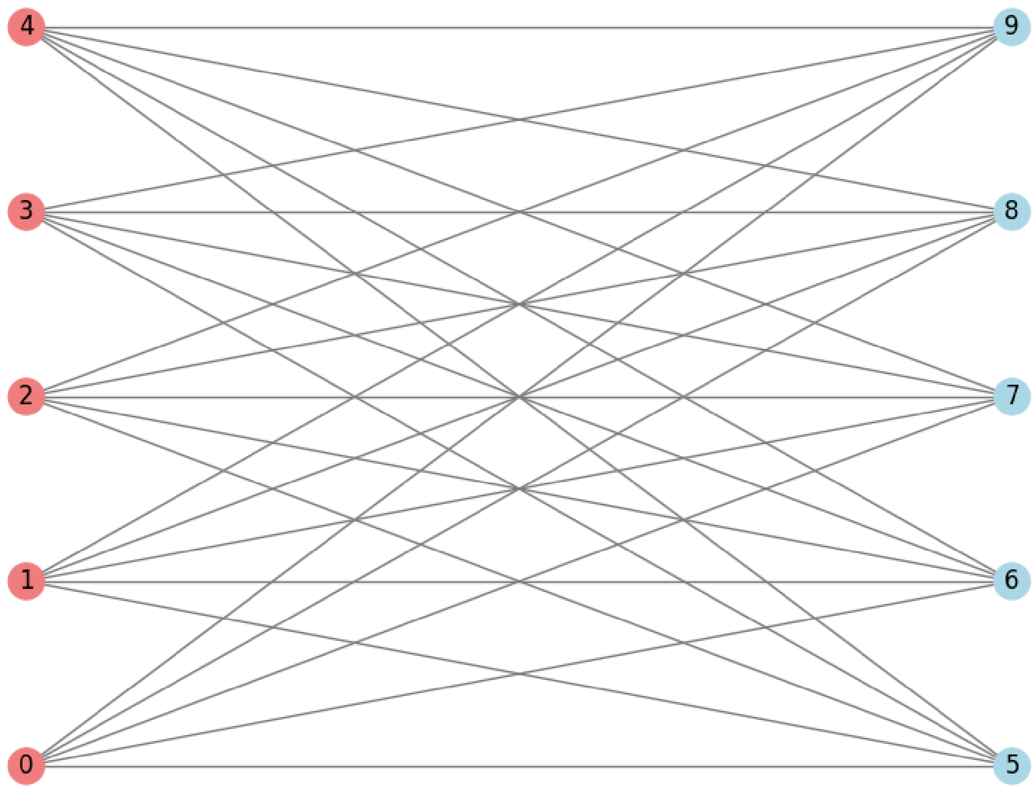
Cycle Graph C8



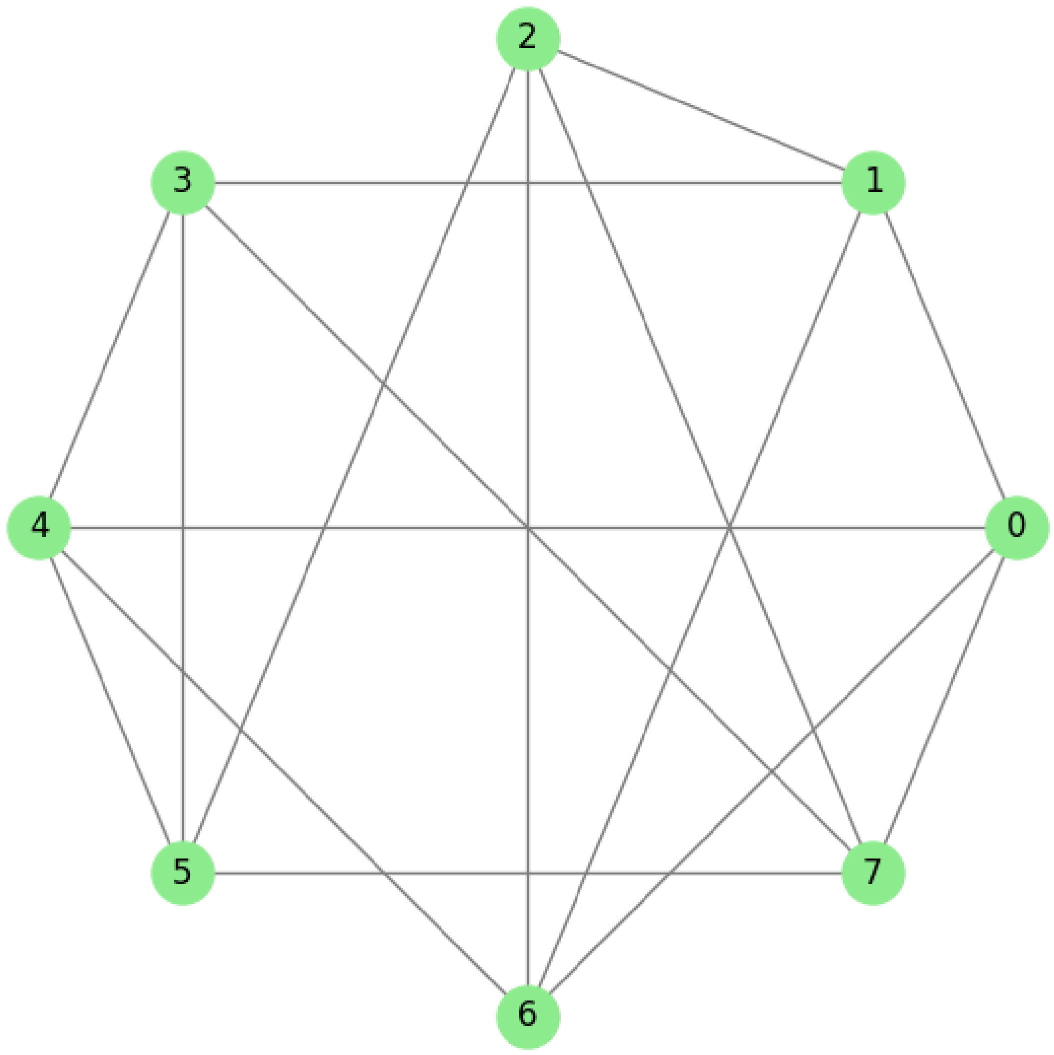
Complete Bipartite Graph K8,8



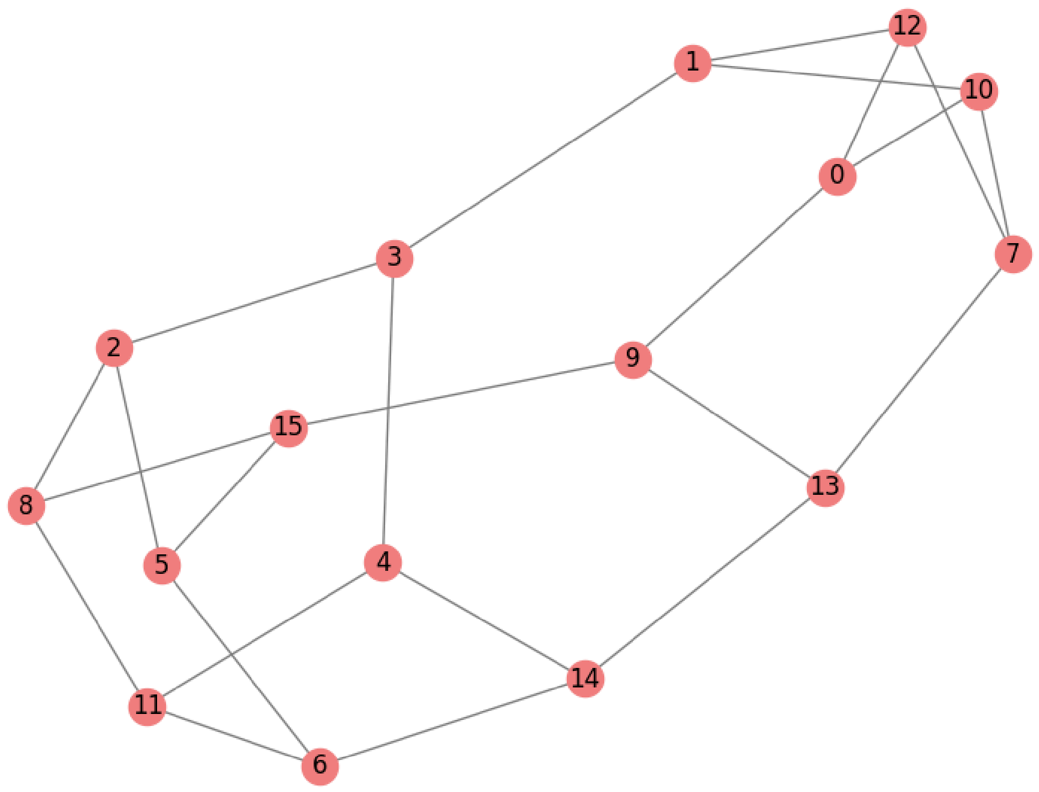
Complete Bipartite Graph K5,5



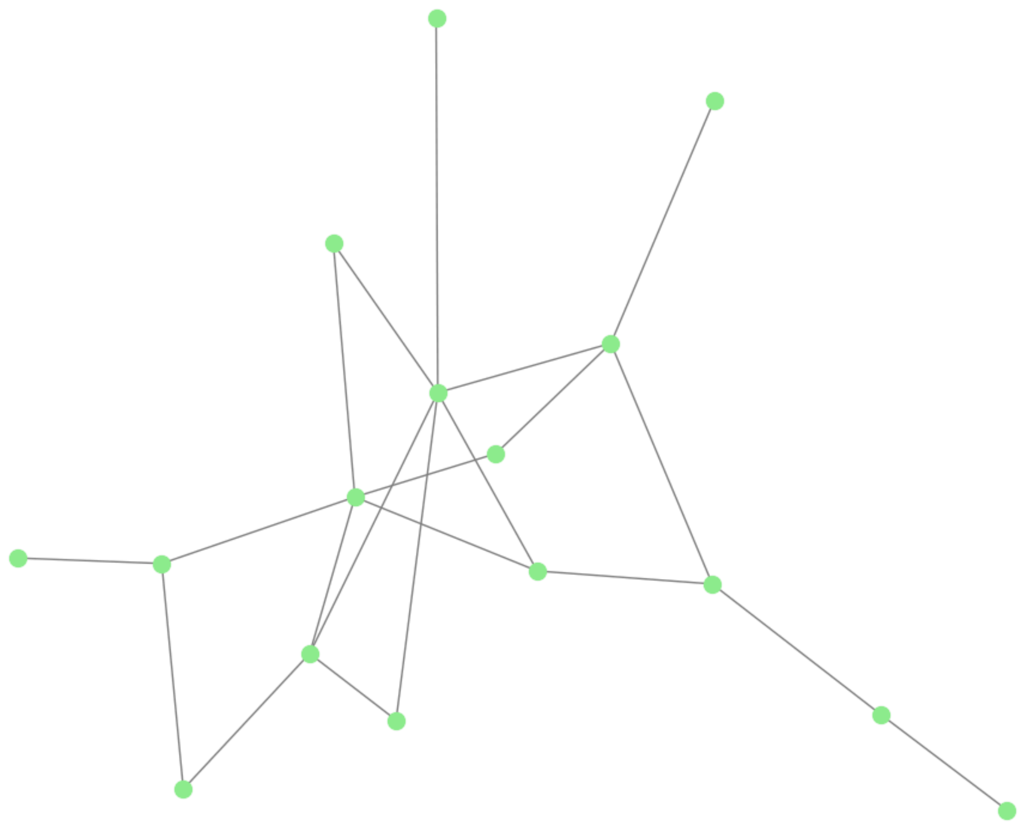
4-Regular Graph with 8 Vertices



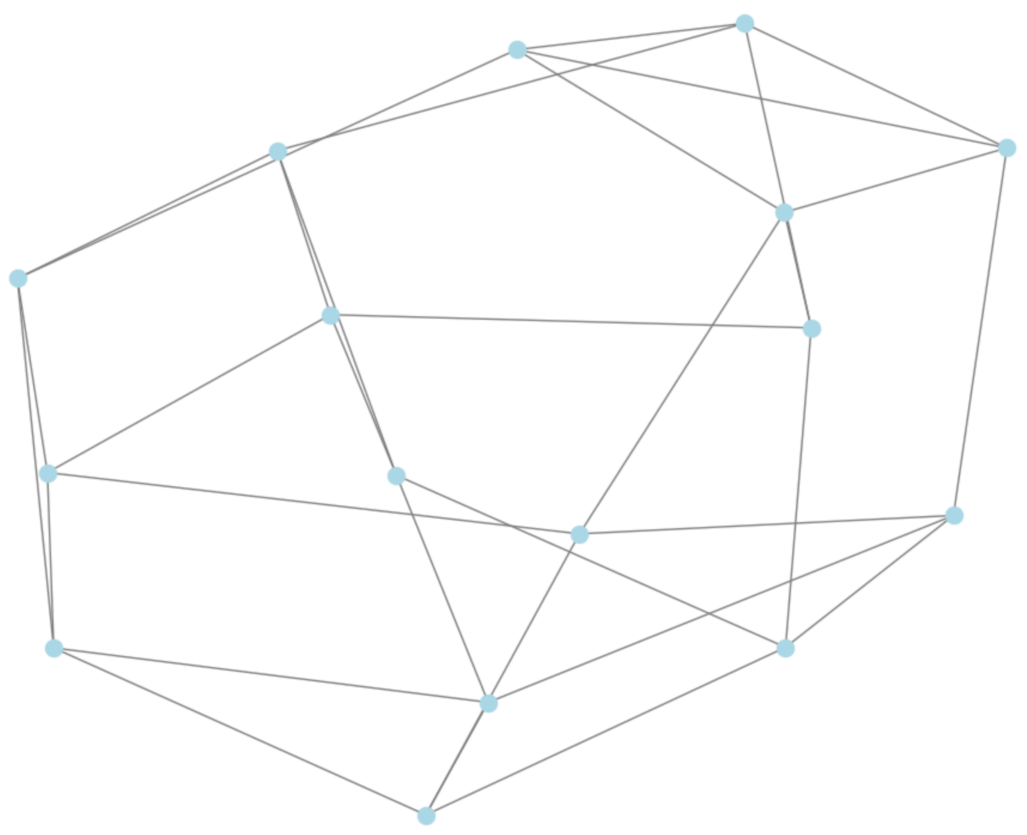
Cubic (3-Regular) Graph with 16 Vertices



Random Connected Graph with 16 Vertices



Expander Graph with 16 Vertices



Input graph

In [215…
```
graph = graph3
graph
```

Out[215…
```
<networkx.classes.graph.Graph at 0x20851340c90>
```

Play around with all of them

Quantum Circuit Generator

We start by generating parameterized circuit based on a given graph.

🤔 Note: Future experiments will focus on optimizing this circuit generator (this function below). Maximizing your score requires minimizing the number of gates in the generated parameterized circuit.

Set of Rules:

Rules: Do not rename the function, import external libraries (except Qiskit), or use classical solvers. Implement any needed algorithms (e.g., sorting) within the function. Minimize gate count. Run locally for all the graphs before submitting the results.

Let's iterate again.

This challenge requires you to work within the provided function's name and scope. You may only use Qiskit libraries. Do not import additional libraries or call external functions. If your solution requires operations like sorting, implement them directly within the function. Critically, do *not* use a classical algorithm to solve the problem and simply input the result. Optimize your code to minimize the number of quantum gates, while aiming for the correct solutions distribution. Thoroughly get your solution results to report at the end.

Important note: To confirm your results, please save this notebook including all cell outputs and upload it in the form here: https://forms.gle/tAjnUd7b5t3oX3b2A . To avoid exceeding the notebook's 10MB file size limit, please be mindful of the number of print statements you use. Feel free to modify the notebook as needed. However, please ensure your code is readable. Please thoroughly comment your code; we will evaluate your problem-solving approach based on the educational clarity of your explanations.

Good luck, and have fun!

In [216…

```python
# Visualization will be performed in the cells below;

def build_ansatz(graph: nx.Graph) -> QuantumCircuit:

    ansatz = QuantumCircuit(graph.number_of_nodes())
    ansatz.h(range(graph.number_of_nodes()))

    theta = ParameterVector(r"$\theta$", graph.number_of_edges())
    for t, (u, v) in zip(theta, graph.edges):
        ansatz.cx(u, v)
        ansatz.ry(t, v)
        ansatz.cx(u, v)

    return ansatz
```

In [217…

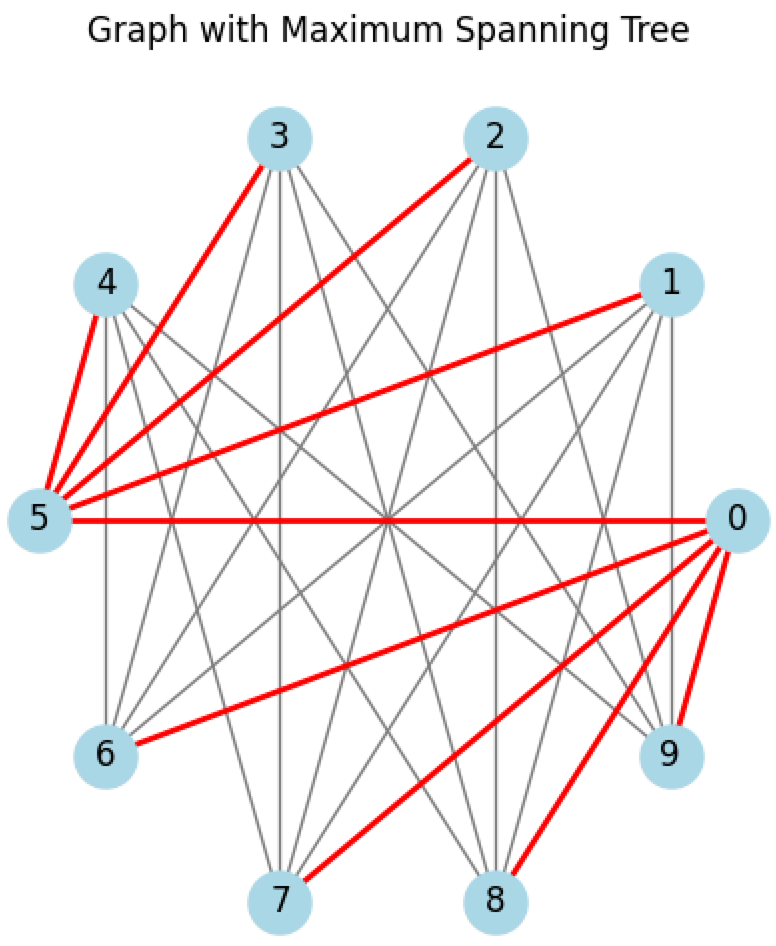
```python
# alternate method to make the ansatz

# function to generate a maximum spanning tree of the graph
def generate_max_spanning_tree(G):
    # for u, v in G.edges:
    #     G[u][v]['weight'] *= -1
    return nx.minimum_spanning_tree(G, algorithm='kruskal')

# function to return the graph with the spanning tree edges removed
def remove_spanning_tree_edges(G, T):
    H = G.copy()
    for edge in T.edges:
        H.remove_edge(*edge)
    return H
```

In [218…

```
# get the maximum spanning tree of the graph by negating all the edge weights and c
T = generate_max_spanning_tree(graph)

# remove the edges of the spanning tree from the graph
H = remove_spanning_tree_edges(graph, T)

# visualize the tree and the graph with the tree edges removed
plt.figure(figsize=(12, 6))
plt.subplot(121)
pos = nx.circular_layout(graph)
nx.draw(graph, pos, with_labels=True, node_color='lightblue', edge_color='gray', no
nx.draw_networkx_edges(T, pos, edge_color='red', width=2)
plt.title("Graph with Maximum Spanning Tree")
```

Out[218…

```
Text(0.5, 1.0, 'Graph with Maximum Spanning Tree')
```



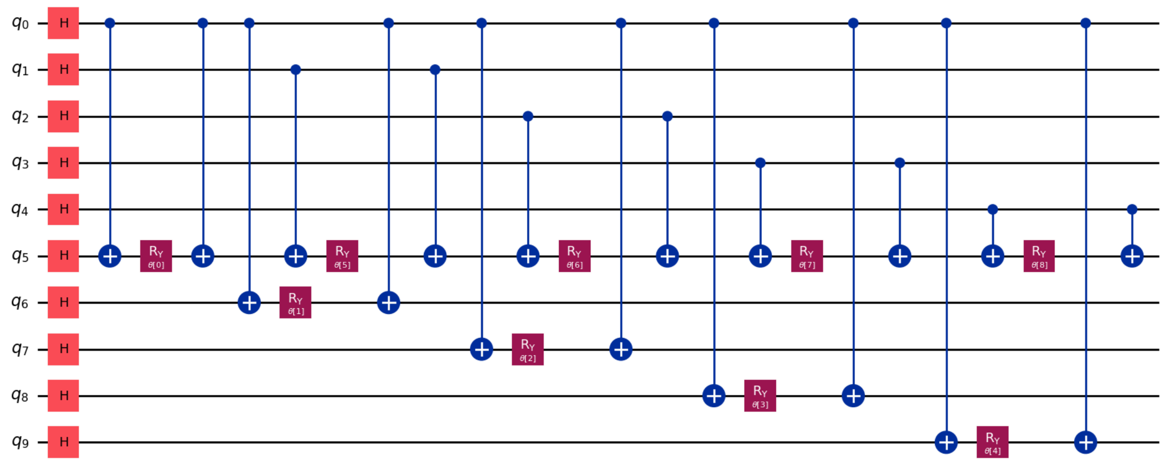
In [219…

```
# build the ansatz circuit for both T and H
ansatz_T = build_ansatz(T)
ansatz_H = build_ansatz(H)
```

In [220…

```
# visualize the ansatz circuit for the tree
ansatz_T.draw("mpl", fold=-1)
```

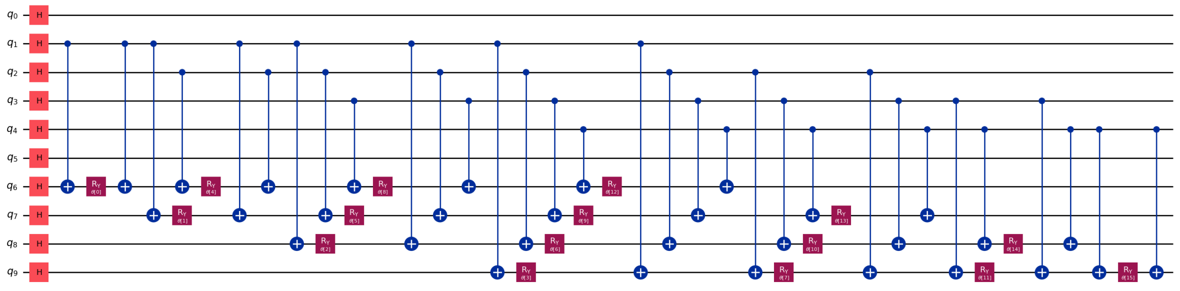
Out[220…



In [221…

```
# visualize the ansatz circuit for the graph with the tree edges removed
ansatz_H.draw("mpl", fold=-1)
```

Out[221…



On Parametrized Quantum Circuits

Parametrized quantum circuits (PQC) recently have gained significant attention as a prominent way to reach quantum advantage in many different fields. They are characterized by their use of varying parameters within quantum gates and can be optimized to solve specific problems, such as machine learning, quantum optimization, or quantum chemistry.

PQCs usually consist of three main parts:

1. Initialization: The qubits are set to a known starting state (usually $|0\rangle$) and then set into superposition with Hadamard gates.
2. Parameterized gates, such as rotation gates (Rx, Ry, Rz), and entanglement gates are adjustable. Their parameters can be optimized to reflect the relationships between the encoded problem data.
3. Measurement: After applying the parameterized gates, the qubits are measured to get results.

These circuits are important elements in Variational Quantum Algorithms (VQA), hybrid quantum-classical systems where classical optimization helps to improve quantum operations. Two most notable examples of VQAs are Variational Quantum Eigensolver (VQE), an algorithm that finds a ground state energy of a given Hamiltonian, and Quantum

Approximate Optimization Algorithm (QAOA), which is designed for solving combinatorial optimization problems. Near-term quantum computers are noisy and limited in size, and PQCs provide a practical way for us to use them more effectively.

More on that topic: https://arxiv.org/abs/2207.06850

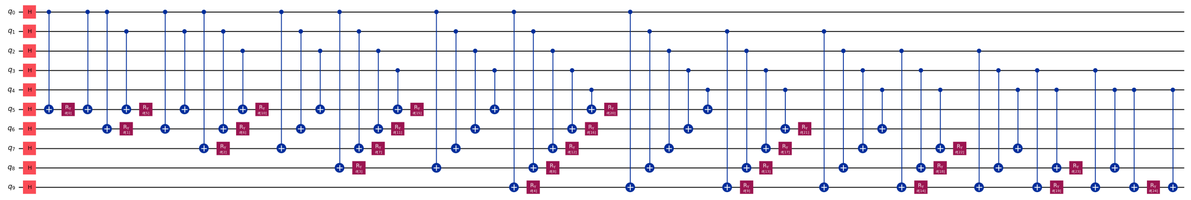
Back to the problem in hand

This function produces parametrized quantum circuits (PQCs), a visualization of which is provided below.

In [222…

```
ansatz = build_ansatz(graph)
ansatz.draw("mpl", fold=-1)
```

Out[222…



Building the MaxCut Hamiltonian

Formally, we write MaxCut as a Quadratic Program (QP) with binary decision variables as follows. For each node $v \in V$, we let x_v denote a binary variable indicating whether v belongs to S or T. The objective is to maximize the number of cut edges:

$$\text{maximize}_x \quad \sum_{(v,w)\in E} (x_v + x_w - 2x_v x_w)$$

Let's break this down. Notice that for each edge $e = (v, w)$ in the graph, the quantity $(x_v + x_w - 2x_v x_w)$ indicates whether e is *cut* by the partition represented by x; that is, the quantity $(x_v + x_w - 2x_v x_w)$ is zero or one, and it equals one only if v and w lie on different sides of the partition specified by x.

The code cell below obtains a symbolic representation of the maximization objective corresponding to the graph above.

In [223…

```
def build_maxcut_hamiltonian(graph: nx.Graph) -> SparsePauliOp:
    """
    Build the MaxCut Hamiltonian for the given graph H = (|E|/2)*I - (1/2)*Σ_{(i,j)
    """
    num_qubits = len(graph.nodes)
    edges = list(graph.edges())
    num_edges = len(edges)

    pauli_terms = ["I"*num_qubits] # start with identity
    coeffs = [-num_edges / 2]
    for (u, v) in edges: # for each edge, add -(1/2)*Z_i Z_j
        z_term = ["I"] * num_qubits
        z_term[u] = "Z"
        z_term[v] = "Z"
        pauli_terms.append("".join(z_term))
        coeffs.append(0.5)

    return SparsePauliOp.from_list(list(zip(pauli_terms, coeffs)))
```

In [224…

```
# get the maxcut hamiltonian for the tree
H_maxcut_T = build_maxcut_hamiltonian(T)

# get the maxcut hamiltonian for the graph with the tree edges removed
H_maxcut_H = build_maxcut_hamiltonian(H)
```

In [225…

```
print(H_maxcut_T)
print(H_maxcut_H)
```

```
SparsePauliOp(['IIIIIIIIII', 'ZIIIIZIIII', 'ZIIIIIZIII', 'ZIIIIIIZII', 'ZIIIIIIIZI',
'ZIIIIIIIIZ', 'IZIIIZIIII', 'IIZIIZIIII', 'IIIZIZIIII', 'IIIIZZIIII'],
              coeffs=[-4.5+0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,
0.5+0.j,
  0.5+0.j,  0.5+0.j,  0.5+0.j])
SparsePauliOp(['IIIIIIIIII', 'IZIIIIZIII', 'IZIIIIIZII', 'IZIIIIIIZI', 'IZIIIIIIIZ',
'IIZIIIZIII', 'IIZIIIIZII', 'IIZIIIIIZI', 'IIZIIIIIIZ', 'IIIZIIZIII', 'IIIZIIIZII',
'IIIZIIIIZI', 'IIIZIIIIIZ', 'IIIIZIZIII', 'IIIIZIIZII', 'IIIIZIIIZI', 'IIIIZIIIIZ'],
              coeffs=[-8. +0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,
0.5+0.j,
  0.5+0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,  0.5+0.j,
  0.5+0.j,  0.5+0.j,  0.5+0.j])
```

Let's keep this function contained within its own cell, as you might need to adapt it for various max-cut problem types later.

In [226…

```
ham = build_maxcut_hamiltonian(graph)
ham
```

Out[226…

```
SparsePauliOp(['IIIIIIIIII', 'ZIIIIZIIII', 'ZIIIIIZIII', 'ZIIIIIIZII', 'ZIIIIIIIZ
I', 'ZIIIIIIIIZ', 'IZIIIZIIII', 'IZIIIIZIII', 'IZIIIIIZII', 'IZIIIIIIZI', 'IZIIIII
IIZ', 'IIZIIZIIII', 'IIZIIIZIII', 'IIZIIIIZII', 'IIZIIIIIZI', 'IIZIIIIIIZ', 'IIIZI
ZIIII', 'IIIZIIZIII', 'IIIZIIIZII', 'IIIZIIIIZI', 'IIIZIIIIIZ', 'IIIIZZIIII', 'III
IZIZIII', 'IIIIZIIZII', 'IIIIZIIIZI', 'IIIIZIIIIZ'],
              coeffs=[-12.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5
+0.j,
   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,
   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,
   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,   0.5+0.j,
   0.5+0.j,   0.5+0.j])
```

As a Hamiltonian energy minimization problem

Quantum MaxCut

Quantum computers are good at finding the ground state of particle systems evolving under the action of a given Hamiltonian. In this section, we'll construct a Hamiltonian whose energies are exactly the values of the MaxCut objective function. This correspondence will effectively translate our classical combinatorial optimization problem into a quantum problem, which we'll approach using our novel heuristic.

Given an objective function $C(x)$, with domain $x \in \{0,1\}^n$, we'll produce a Hamiltonian H_C on n qubits such that

$$H_C \ket x = C(x) \ket x.$$

In the last equation, $\ket x$ denotes the n-qubit computational basis state indexed by the bit-string $x \in \{0,1\}$. Thus the last equation says each of the 2^n computational basis states is an eigenvector of H_C, and the eigenvalue corresponding to $\ket x$ is $C(x)$; that is, H_C is diagonal with respect to the computational basis, and its energies are the values of the objective function C.

We'll obtain the Hamiltonian H_C by replacing each x_j in the expression of $C(x)$ by the operator

$$\hat{X}_j \coloneqq \frac{1}{2}(I - Z_j),$$

where I denotes the identity operator on n qubits and Z_j denotes the Pauli-Z operator acting on the jth qubit. Notice that $\hat{X}_j$ is diagonal with respect to the computational basis, and its eigenvalues are zero and one; in particular,

$$\hat{X}_j \ket x = x_j \ket x.$$

MaxCut Hamiltonian

When we apply the Ising map construction to the MaxCut objective

$$M(x) = \sum_{(v,w)\in E} (x_v + x_w - 2x_v x_w)$$

we obtain the Hamiltonian

$$H_M = \sum_{(v,w)\in E} (X_v + X_w - 2X_v X_w) = \frac{1}{2} \sum_{(v,w)\in E} \big(2I - Z_v - Z_w - (I - Z_v)(I - Z_w)\big) = \frac{1}{2}|$$

In the last equation, $|E|$ denotes the number of edges in the graph.

Energy minimization via QITE

Enter varQITE

With the MaxCut Hamiltonian in hand, we can turn to minimizing its energy using our novel quantum-classical varQITE heuristic. Much like any Variataional Quantum Algorithm (VQA), our novel varQITE method provides a recipe for iteratively updating the parameters in a variational quantum circuit to minimize the expectation value of the MaxCut Hamiltonian, measured with respect to the parametrized state.

A key novelty is that our varQITE algorithm does *not* rely on a classical optimizer to update the circuit parameters; instead, it specifies an explicit update rule based on the solution of a system linear Ordinary Differential Equations (ODEs). The ODEs relate the gradient of the variational circuit parameters to the expected value of certain operators related to the MaxCut Hamiltonian, and they are derived from an Ehrenfest Theorem that applies to imaginary time evolution. For details, see Equation (5) in our varQITE paper.

In any case, setting up the ODE system at each step of the algorithm requires executing a batch of quantum circuits and running some post-processing to evaluate the results.

The code cell below illustrates how to set up the variational ansatz $\ket\Psi(\theta)$ introduced by our varQITE paper in Equation (2). We'll set up the required circuits and the ODEs further down.

Below is simplified version of Quantum Imaginary Time Evolution (QITE). It uses a finite differences approach to estimate gradients, then performs gradient descent updates.

In [227…

```
class QITEvolver:
    """
    A class to evolve a parametrized quantum state under the action of an Ising
    Hamiltonian according to the variational Quantum Imaginary Time Evolution
    (QITE) principle described in IonQ's latest joint paper with ORNL.
    """
    def __init__(self, hamiltonian: SparsePauliOp, ansatz: QuantumCircuit):
        self.hamiltonian = hamiltonian
        self.ansatz = ansatz

        # Define some constants
        self.backend = AerSimulator()
        self.num_shots = 10000
        self.energies, self.param_vals, self.runtime = list(), list(), list()

    def evolve(self, num_steps: int, lr: float = 0.4, verbose: bool = True):
        """
        Evolve the variational quantum state encoded by ``self.ansatz`` under
        the action of ``self.hamiltonian`` according to varQITE.
        """
        curr_params = np.zeros(self.ansatz.num_parameters)
        for k in range(num_steps):
            # Get circuits and measure on backend
            iter_qc = self.get_iteration_circuits(curr_params)
            job = self.backend.run(iter_qc, shots=self.num_shots)
            q0 = time.time()
            measurements = job.result().get_counts()
            quantum_exec_time = time.time() - q0

            # Update parameters-- set up defining ODE and step forward
            Gmat, dvec, curr_energy = self.get_defining_ode(measurements)
            dcurr_params = np.linalg.lstsq(Gmat, dvec, rcond=1e-2)[0]
            curr_params += lr * dcurr_params

            # Progress checkpoint!
            if verbose:
                self.print_status(measurements)
            self.energies.append(curr_energy)
            self.param_vals.append(curr_params.copy())
            self.runtime.append(quantum_exec_time)

    def get_defining_ode(self, measurements: List[dict[str, int]]):
        """
        Construct the dynamics matrix and load vector defining the varQITE
        iteration.
        """
        # Load sampled bitstrings and corresponding frequencies into NumPy arrays
        dtype = np.dtype([("states", int, (self.ansatz.num_qubits,)), ("counts", "f
        measurements = [np.fromiter(map(lambda kv: (list(kv[0]), kv[1]), res.items(

        # Set up the dynamics matrix by computing the gradient of each Pauli word
        # with respect to each parameter in the ansatz using the parameter-shift ru
        pauli_terms = [SparsePauliOp(op) for op, _ in self.hamiltonian.label_iter()
        Gmat = np.zeros((len(pauli_terms), self.ansatz.num_parameters))
        for i, pauli_word in enumerate(pauli_terms):
            for j, jth_pair in enumerate(zip(measurements[1::2], measurements[2::2]
                for pm, pm_shift in enumerate(jth_pair):
                    Gmat[i, j] += (-1)**pm * expected_energy(pauli_word, pm_shift)

        # Set up the load vector
        curr_energy = expected_energy(self.hamiltonian, measurements[0])
        dvec = np.zeros(len(pauli_terms))
        for i, pauli_word in enumerate(pauli_terms):
            rhs_op_energies = get_ising_energies(pauli_word, measurements[0]["state
            rhs_op_energies *= get_ising_energies(self.hamiltonian, measurements[0]
            dvec[i] = -np.dot(rhs_op_energies, measurements[0]["counts"]) / self.nu
        return Gmat, dvec, curr_energy

    def get_iteration_circuits(self, curr_params: np.array):
        """
        Get the bound circuits that need to be evaluated to step forward
        according to QITE.
        """
        # Use this circuit to estimate your Hamiltonian's expected value
        circuits = [self.ansatz.assign_parameters(curr_params)]

        # Use these circuits to compute gradients
        for k in np.arange(curr_params.shape[0]):
            for j in range(2):
                pm_shift = curr_params.copy()
                pm_shift[k] += (-1)**j * np.pi/2
                circuits += [self.ansatz.assign_parameters(pm_shift)]
        # Add measurement gates and return
        [qc.measure_all() for qc in circuits]
        return circuits

    def plot_convergence(self):
        """
        Plot the convergence of the expected value of ``self.hamiltonian`` with
        respect to the (imaginary) time steps.
        """
        plt.plot(self.energies)
        plt.xlabel("(Imaginary) Time step")
        plt.ylabel("Hamiltonian energy")
        plt.title("Convergence of the expected energy")

    def print_status(self, measurements):
        """
        Print summary statistics describing a QITE run.
        """
        stats = pd.DataFrame({
            "curr_energy": self.energies,
            "num_circuits": [len(measurements)] * len(self.energies),
            "quantum_exec_time": self.runtime
        })
        stats.index.name = "step"
        display.clear_output(wait=True)
        display.display(stats)
```

A few utility functions:

In [228…

```
def compute_cut_size(graph, bitstring):
    """
    Get the cut size of the partition of ``graph`` described by the given
    ``bitstring``.
    """
    cut_sz = 0
    for (u, v) in graph.edges:
        if bitstring[u] != bitstring[v]:
            cut_sz += 1
    return cut_sz
```

In [229…
```
def get_ising_energies(
        operator: SparsePauliOp,
        states: np.array
    ):
    """
    Get the energies of the given Ising ``operator`` that correspond to the
    given ``states``.
    """
    # Unroll Hamiltonian data into NumPy arrays
    paulis = np.array([list(ops) for ops, _ in operator.label_iter()]) != "I"
    coeffs = operator.coeffs.real

    # Vectorized energies computation
    energies = (-1) ** (states @ paulis.T) @ coeffs
    return energies
```

In [230…
```
def expected_energy(
        hamiltonian: SparsePauliOp,
        measurements: np.array
):
    """
    Compute the expected energy of the given ``hamiltonian`` with respect to
    the observed ``measurement``.

    The latter is assumed to by a NumPy records array with fields ``states``
    --describing the observed bit-strings as an integer array-- and ``counts``,
    describing the corresponding observed frequency of each state.
    """
    energies = get_ising_energies(hamiltonian, measurements["states"])
    return np.dot(energies, measurements["counts"]) / measurements["counts"].sum()
```

In [231…
```
def interpret_solution(graph, bitstring):
    """
    Visualize the given ``bitstring`` as a partition of the given ``graph``.
    """
    pos = nx.spring_layout(graph, seed=42)
    set_0 = [i for i, b in enumerate(bitstring) if b == '0']
    set_1 = [i for i, b in enumerate(bitstring) if b == '1']

    plt.figure(figsize=(4, 4))
    nx.draw_networkx_nodes(graph, pos=pos, nodelist=set_0, node_color='blue', node_
    nx.draw_networkx_nodes(graph, pos=pos, nodelist=set_1, node_color='red', node_s

    cut_edges = []
    non_cut_edges = []
    for (u, v) in graph.edges:
        if bitstring[u] != bitstring[v]:
            cut_edges.append((u, v))
        else:
            non_cut_edges.append((u, v))

    nx.draw_networkx_edges(graph, pos=pos, edgelist=non_cut_edges, edge_color='gray
    nx.draw_networkx_edges(graph, pos=pos, edgelist=cut_edges, edge_color='green',

    nx.draw_networkx_labels(graph, pos=pos, font_color='white', font_weight='bold')
    plt.axis('off')
    plt.show()
```

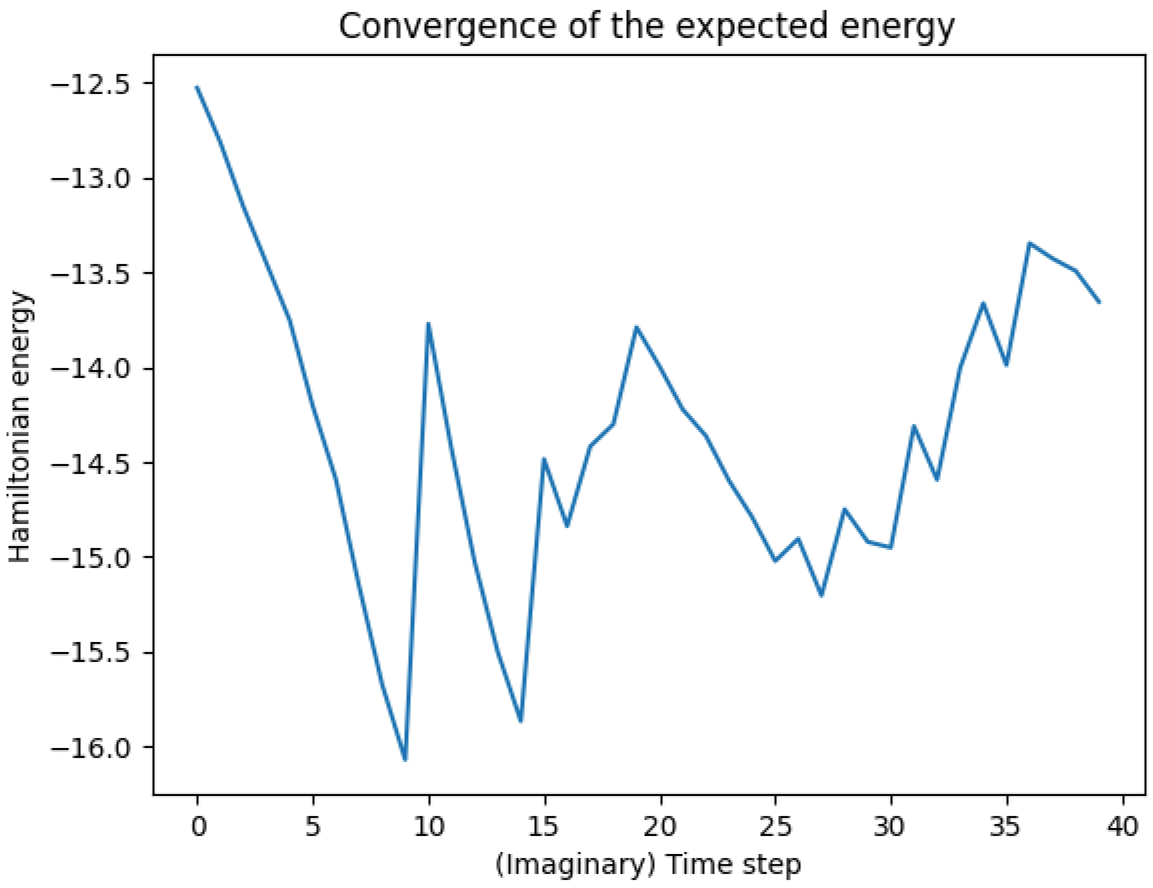
In [232…

```
%%time

# Set up your QITEvolver and evolve!
qit_evolver = QITEvolver(ham, ansatz)
qit_evolver.evolve(num_steps=40, lr=0.1, verbose=True) # lr was 0.5

# Visualize your results!
qit_evolver.plot_convergence()
```

	curr_energy	num_circuits	quantum_exec_time
step			
0	-12.5279	51	2.370668
1	-12.8142	51	2.130529
2	-13.1546	51	2.147762
3	-13.4534	51	2.072397
4	-13.7524	51	2.236612
5	-14.2053	51	2.139423
6	-14.5915	51	2.357646
7	-15.1525	51	2.341868
8	-15.6727	51	2.088616
9	-16.0709	51	2.003760
10	-13.7722	51	1.959255
11	-14.4312	51	2.121217
12	-15.0258	51	2.169194
13	-15.5017	51	2.047987
14	-15.8670	51	2.085690
15	-14.4850	51	2.127400
16	-14.8387	51	2.142435
17	-14.4182	51	2.046287
18	-14.3014	51	2.073431
19	-13.7897	51	2.076020
20	-13.9969	51	2.164500
21	-14.2233	51	2.175838
22	-14.3623	51	2.105482
23	-14.5984	51	2.116762
24	-14.7903	51	2.190129
25	-15.0228	51	2.178893
26	-14.9051	51	2.233080
27	-15.2047	51	2.107476
28	-14.7493	51	2.109163

	curr_energy	num_circuits	quantum_exec_time
step			
29	-14.9213	51	2.083779
30	-14.9519	51	2.210884
31	-14.3112	51	2.181073
32	-14.5943	51	2.270601
33	-14.0034	51	2.266900
34	-13.6636	51	2.105430
35	-13.9899	51	2.108909
36	-13.3472	51	2.104765
37	-13.4281	51	2.156178
38	-13.4933	51	2.233804

```
CPU times: total: 17min 20s
Wall time: 1min 33s
```



In [233…

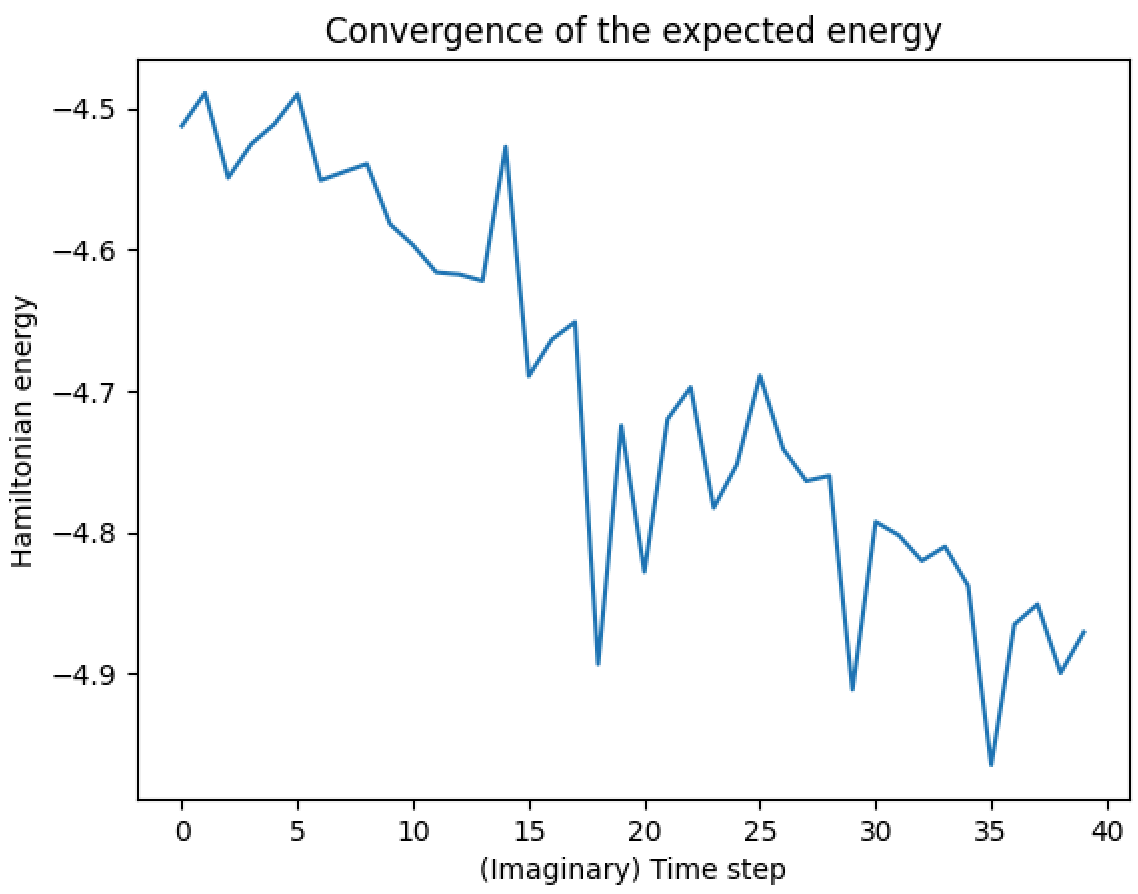
```
%%time

# Set up QITEvolver for the tree
qit_evolver_T = QITEvolver(H_maxcut_T, ansatz_T)
```

```
qit_evolver_T.evolve(num_steps=40, lr=0.1, verbose=True)

# Visualize your results!
qit_evolver_T.plot_convergence()
```

	curr_energy	num_circuits	quantum_exec_time
step			
0	-4.5123	19	0.933317
1	-4.4889	19	0.843050
2	-4.5490	19	0.902101
3	-4.5251	19	0.895521
4	-4.5110	19	0.846815
5	-4.4897	19	0.868420
6	-4.5508	19	0.854072
7	-4.5449	19	0.824460
8	-4.5392	19	0.820140
9	-4.5818	19	0.846750
10	-4.5967	19	0.834790
11	-4.6159	19	0.840648
12	-4.6174	19	0.811155
13	-4.6220	19	0.782974
14	-4.5270	19	0.823226
15	-4.6894	19	0.817499
16	-4.6634	19	0.819090
17	-4.6511	19	0.856570
18	-4.8932	19	0.830004
19	-4.7244	19	0.844155
20	-4.8279	19	0.839158
21	-4.7199	19	0.811243
22	-4.6973	19	0.859713
23	-4.7826	19	0.819254
24	-4.7519	19	0.836474
25	-4.6890	19	0.839059
26	-4.7409	19	0.818591
27	-4.7636	19	0.863229
28	-4.7599	19	0.726387

step	curr_energy	num_circuits	quantum_exec_time
29	-4.9112	19	0.839952
30	-4.7925	19	0.812069
31	-4.8020	19	0.859760
32	-4.8201	19	0.842711
33	-4.8100	19	0.849422
34	-4.8378	19	0.823677
35	-4.9645	19	0.866946
36	-4.8651	19	0.841196
37	-4.8509	19	0.834579
38	-4.8995	19	0.843396

```
CPU times: total: 7min 43s
Wall time: 35.6 s
```



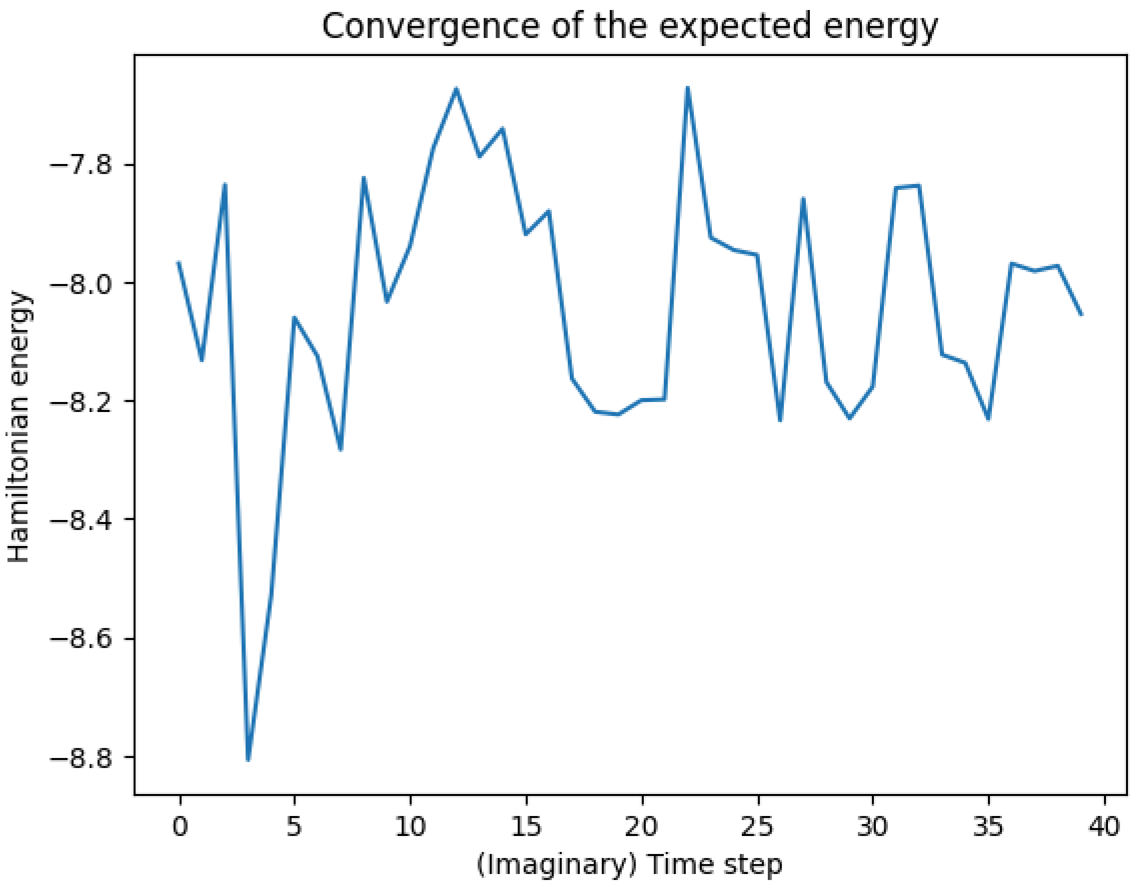
```
In [234…
%%time

# Set up your QITEvolver for the graph with the tree edges removed
qit_evolver_H = QITEvolver(H_maxcut_H, ansatz_H)
```

```
qit_evolver_H.evolve(num_steps=40, lr=0.1, verbose=True)

# Visualize your results!
qit_evolver_H.plot_convergence()
```

	curr_energy	num_circuits	quantum_exec_time
step			
0	-7.9686	33	1.552252
1	-8.1324	33	1.605831
2	-7.8358	33	1.432330
3	-8.8072	33	1.403104
4	-8.5292	33	1.404304
5	-8.0602	33	1.556564
6	-8.1256	33	1.408144
7	-8.2830	33	1.548157
8	-7.8244	33	1.526215
9	-8.0334	33	1.466460
10	-7.9392	33	1.459147
11	-7.7744	33	1.465500
12	-7.6738	33	1.425559
13	-7.7884	33	1.414921
14	-7.7412	33	1.542623
15	-7.9196	33	1.507906
16	-7.8804	33	1.458564
17	-8.1632	33	1.271164
18	-8.2192	33	1.322973
19	-8.2240	33	1.443580
20	-8.1998	33	1.432512
21	-8.1984	33	1.524279
22	-7.6722	33	1.491683
23	-7.9252	33	1.430027
24	-7.9462	33	1.474846
25	-7.9544	33	1.377127
26	-8.2338	33	1.554870
27	-7.8596	33	1.489802
28	-8.1692	33	1.453870

step	curr_energy	num_circuits	quantum_exec_time
29	-8.2306	33	1.489505
30	-8.1768	33	1.497345
31	-7.8414	33	1.339180
32	-7.8372	33	1.326096
33	-8.1228	33	1.419778
34	-8.1368	33	1.460220
35	-8.2316	33	1.448233
36	-7.9690	33	1.474342
37	-7.9816	33	1.469937
38	-7.9728	33	1.368272

```
CPU times: total: 10min 17s
Wall time: 1min 1s
```



Check out your best / most frequent cut!

Following the variational quantum imaginary time evolution (vQITE) loop, we sample the quantum circuit to obtain classical bitstrings. The most frequent bitstring represents our solution, the final score though will take into account all of the right solutions...

In [235...

```
from qiskit_aer import AerSimulator

shots = 100_000
backend = AerSimulator()
```

In [236...

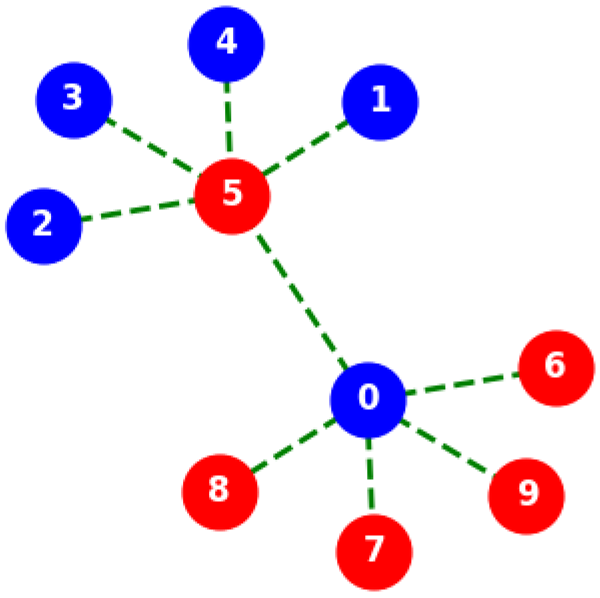
```
# Sample your optimized quantum state using Aer for the tree

optimized_state_T = ansatz_T.assign_parameters(qit_evolver_T.param_vals[-1])
optimized_state_T.measure_all()
counts_T = backend.run(optimized_state_T, shots=shots).result().get_counts()

# Find the sampled bitstring with the largest cut value
cut_vals_T = sorted(((bs, compute_cut_size(T, bs)) for bs in counts_T), key=lambda
best_bs_T = cut_vals_T[-1][0]

# Now find the most likely MaxCut solution as sampled from your optimized state
# We'll leave this part up to you!!!
most_likely_soln = ""

print(counts_T)
```

```
{'1001001000': 529, '0110110001': 1506, '1011010001': 3, '0010011111': 454, '1001010
010': 600, '0110100101': 731, '1001000110': 1402, '1001011110': 480, '0110101011': 1
743, '1111010111': 26, '0111010110': 1, '0110111101': 1542, '0110011001': 315, '0110
101111': 1585, '1001011010': 740, '1111010011': 12, '1101001100': 310, '0000001101':
3, '1101010100': 661, '1011001100': 4, '1001000010': 1572, '1101001110': 600, '10101
10010': 1, '1011001110': 9, '1101010010': 239, '0110011111': 1077, '1101101110': 74,
'0100100011': 7, '0001100000': 157, '1100010000': 85, '0110000101': 1013, '100111001
0': 1177, '0111001001': 183, '0010110001': 613, '0101111110': 66, '1001111110': 132
4, '0100101100': 8, '1110100000': 1, '0111001101': 143, '1001100000': 1047, '1011001
001': 7, '0010100111': 592, '0101001100': 40, '1110110001': 183, '1000111100': 12,
'0001000010': 195, '0100011111': 11, '0101100110': 15, '1011111010': 2, '011000110
1': 1119, '1100011010': 44, '0110010111': 1201, '1101100100': 342, '1101000010': 62
0, '1010111110': 1, '1011000010': 7, '1101011110': 180, '0110100111': 1457, '1001010
000': 1538, '1111011011': 16, '0010101111': 623, '0101000010': 91, '1000111000': 21,
'1101111110': 492, '1111011000': 3, '0110100011': 1648, '1001010100': 1626, '1111011
111': 24, '1000011000': 185, '0000101010': 3, '0000110110': 25, '1101011010': 333,
'1011000110': 9, '1010111010': 1, '1101000110': 570, '1000011010': 98, '0000101000':
25, '1111111010': 2, '1000000010': 214, '0110110111': 512, '1101011000': 606, '01010
00101': 2, '0101110000': 5, '0010111101': 629, '1110101111': 217, '0111101111': 211,
'0101110001': 2, '0001010000': 201, '0111101101': 78, '0010100011': 640, '011001101
1': 856, '1101101010': 41, '0101101100': 63, '0110000001': 1250, '1100010100': 81,
'1001100100': 864, '0111000111': 28, '1001101100': 933, '0110110101': 1655, '0110111
011': 908, '1101110111': 1, '1001011000': 1507, '1100001100': 34, '1011010110': 5,
'1100110110': 57, '0110111001': 1421, '1001000000': 636, '0110101101': 667, '1001011
100': 1682, '1111010001': 6, '1000110001': 1, '1101010110': 169, '1011001010': 4, '1
010110110': 2, '1101001010': 674, '0110001111': 155, '1100011100': 71, '0101100100':
36, '1000100100': 127, '1100101010': 2, '1110111001': 203, '1101110110': 551, '10111
11001': 6, '0010000111': 95, '0001110010': 160, '1010001101': 54, '1010010111': 68,
'0010101011': 646, '0101000110': 73, '1010011101': 10, '0011111001': 83, '101000011
1': 19, '0110110011': 819, '1001001100': 722, '0001111010': 132, '0001000100': 120,
'1001000100': 847, '0010111001': 553, '1000110010': 135, '1001001110': 1529, '100111
1010': 1005, '1000010100': 227, '1111111111': 16, '1101010000': 617, '1101101000': 4
60, '1101000100': 337, '1011000100': 3, '1101011100': 631, '1100000010': 97, '001011
0101': 714, '0101111010': 52, '0010001101': 451, '1000001100': 106, '1000010000': 19
5, '0111000110': 1, '1110101011': 215, '1111001010': 1, '1001001010': 1668, '1000101
000': 182, '1110011010': 1, '0001110110': 180, '0100001101': 5, '0001001100': 95, '0
100010111': 3, '0101110010': 49, '0010111111': 264, '1110111011': 132, '1000110000':
16, '1111010000': 1, '1010111101': 86, '0000000100': 7, '0001011100': 259, '10001000
00': 127, '1100101110': 7, '0010110111': 210, '0101111100': 6, '0100110001': 8, '000
0010000': 31, '0111010001': 32, '1011110111': 4, '0010001001': 490, '0110011101': 15
1, '0111001011': 5, '1001100110': 318, '0101001010': 85, '0010100101': 292, '1110110
101': 221, '1010111011': 45, '0000000010': 25, '0110010011': 986, '1101100000': 419,
'0011001110': 1, '0011100111': 92, '0010011011': 345, '0111100111': 218, '111001100
1': 46, '0010110011': 303, '1001101000': 1224, '0010000101': 373, '1011110010': 7,
'0101101110': 6, '0110000111': 190, '1100010010': 30, '0111110101': 215, '110010111
1': 1, '1011101111': 10, '1111100011': 31, '1110010100': 1, '0100000001': 8, '000111
1100': 16, '0100011011': 2, '0001000110': 196, '0110101001': 439, '1110000001': 167,
'1100011000': 66, '1101111100': 30, '1010011011': 42, '1010000001': 82, '100111011
0': 1337, '0001010110': 55, '1001010110': 376, '0110100001': 488, '1111011101': 1,
'0110001001': 1364, '0110111111': 604, '1010100011': 104, '1110000101': 129, '111011
0111': 75, '1000111110': 143, '1111011110': 1, '1101111000': 85, '1111100110': 2, '1
011101100': 4, '1000100110': 45, '1100101100': 58, '1000000100': 111, '0101010100':
92, '0000011100': 33, '0001010100': 222, '0010000001': 475, '1101001000': 187, '0010
001111': 56, '0101101000': 63, '0111111101': 221, '0100100111': 7, '0001100100': 10
2, '1000000110': 169, '1111111110': 3, '1000011110': 69, '1000000000': 76, '10000111
00': 200, '0111111011': 117, '0111010011': 118, '0111011011': 97, '1111001001': 31,
'0000100100': 15, '0000000111': 1, '1101000000': 254, '1101111010': 421, '001101011
1': 52, '0001101000': 187, '1011011101': 2, '0010010011': 419, '0111100011': 231, '1
001111100': 84, '0001011000': 240, '0111111111': 82, '1001101010': 78, '0111000101':
122, '0001110000': 16, '0100010001': 2, '0001001110': 200, '0111101011': 223, '00111
01001': 18, '1010101100': 2, '1101101100': 336, '0111000001': 155, '1001101110': 22
5, '1001110000': 144, '1111011010': 1, '1000111010': 143, '1110110011': 94, '0010011
101': 65, '1011010011': 8, '0010111011': 402, '0101110110': 77, '1111110001': 23, '0
111111001': 175, '1110100011': 248, '0111010111': 153, '1010110001': 88, '000000101
0': 22, '1110111101': 215, '1000110110': 182, '0101011100': 102, '0000010100': 35,
'0100110101': 11, '1110100001': 65, '0111001110': 1, '0011110101': 84, '0111111110':
2, '0100010100': 11, '1110011011': 114, '1000010010': 78, '1111110010': 4, '10000010
10': 225, '0011101111': 93, '0010011001': 133, '1101110010': 451, '0111100001': 64,
'0110010101': 99, '1101100110': 120, '1011011010': 1, '1100000100': 46, '110011101
0': 49, '0110001011': 30, '0101100000': 63, '1010101001': 27, '0111110001': 200, '10
00001000': 71, '0000111110': 19, '0100010000': 10, '0100001100': 4, '1110011111': 14
3, '0001010010': 94, '0010010101': 39, '1011011011': 8, '0010101001': 186, '10110001
11': 3, '0000000000': 18, '1111001101': 22, '0011111011': 56, '1110010111': 163, '01
00000010': 10, '0011000101': 48, '0011101100': 2, '0011000001': 73, '0001101010': 1
9, '0100101011': 10, '0001101100': 114, '0000001000': 10, '0010100001': 177, '010100
1110': 79, '1100110100': 3, '1011010100': 10, '1100001010': 78, '0110010001': 256,
'1101100010': 62, '0111010101': 15, '1101010111': 1, '1010110111': 20, '0010101101':
256, '0101000000': 34, '1010001001': 90, '0011110111': 21, '0011011111': 53, '010011
0011': 3, '0000010010': 8, '0101011110': 18, '0011100101': 39, '0010010111': 489, '1
000010110': 51, '0000111000': 3, '1111110110': 2, '1000001110': 182, '1110100101': 1
03, '1011110110': 4, '0101101010': 5, '0110000011': 77, '1100010110': 22, '101010111
1': 104, '1010101011': 110, '0101000100': 46, '1010111001': 83, '1100111110': 64, '1
100000000': 30, '1110101101': 98, '0111000000': 1, '1110100111': 191, '0100110010':
8, '0100000100': 6, '1110010001': 35, '0011100001': 32, '0100111101': 3, '000001111
0': 10, '0101010010': 25, '1010100111': 82, '1011010111': 11, '0111011111': 155, '00
01000000': 83, '0001111110': 169, '0111101001': 51, '0011101011': 86, '0000110010':
13, '0000101110': 4, '0100001001': 7, '1001100010': 151, '1100101000': 54, '10111010
00': 6, '0011010011': 52, '0011011001': 15, '1110001101': 168, '0011110001': 74, '00
01001000': 75, '0101001000': 29, '1010110101': 78, '0000001110': 32, '0001001010': 1
94, '1011000001': 10, '1010111111': 31, '0000000110': 23, '1111101110': 1, '10111000
00': 5, '1000101110': 27, '1100100000': 58, '1111111001': 10, '0000110111': 1, '0111
110111': 55, '0000011000': 16, '0100111001': 9, '0101010000': 82, '1010011001': 17,
'0011111101': 76, '1010010011': 55, '0011011101': 12, '1110001001': 164, '111000001
1': 7, '0111100101': 102, '0101101111': 1, '1011110001': 12, '1111101111': 24, '1011
100011': 14, '1100110010': 55, '1001110100': 44, '1010000101': 56, '1010011111': 52,
'0111110011': 105, '0011001001': 72, '0011111111': 38, '0000100000': 12, '101111101
1': 4, '0000100001': 3, '0100011000': 11, '1100000110': 71, '1011011100': 12, '11110
10100': 3, '1000110100': 4, '1110111111': 92, '0111011101': 22, '0111000011': 6, '10
01111000': 228, '1100011110': 22, '0101100010': 7, '1011111110': 9, '1000000111': 1,
'1111111101': 31, '0001101110': 29, '0100101111': 10, '0000101100': 11, '011101100
1': 34, '1111001011': 4, '1011010000': 7, '1100001110': 80, '0011100110': 1, '001100
1101': 53, '1000101010': 11, '1100100100': 49, '0001011010': 96, '1111000001': 28,
'0011000111': 16, '0011001010': 1, '0011100011': 76, '1000010011': 1, '1111110011':
13, '0101010110': 24, '0000011010': 13, '1111100111': 30, '1011101011': 9, '01010110
00': 82, '1010000011': 7, '1011110011': 4, '0101101101': 1, '1111100101': 17, '11100
10011': 126, '0100000110': 11, '0011101101': 25, '0101111101': 2, '0001011110': 67,
'0000110000': 3, '1010110011': 59, '0000001100': 13, '1011001101': 5, '1111110101':
26, '1101110000': 51, '1100001000': 23, '0011110011': 51, '0100100010': 1, '10101001
01': 43, '1011011111': 10, '0010010001': 93, '0010001011': 15, '1010010001': 17, '00
00111010': 18, '0000100110': 10, '1000101100': 128, '1100100110': 17, '0100101000':
14, '1110000111': 37, '0101111000': 12, '1111111011': 17, '0011010001': 11, '0011010
101': 4, '1110011101': 17, '0100001010': 12, '1010100001': 22, '0011011011': 42, '11
```

```
11000101': 16, '0001100110': 44, '0010000011': 27, '0000010110': 8, '0101011010': 2
9, '1110001111': 18, '0001110111': 1, '0100001110': 7, '0001000111': 1, '010001110
0': 13, '1110010101': 8, '0100000000': 4, '1010101101': 43, '1110001011': 7, '100010
0010': 29, '0100100100': 10, '1100100111': 1, '1011100111': 13, '1111101011': 19, '1
110101001': 43, '0011110110': 1, '1010001000': 1, '0000110101': 4, '1100101001': 2,
'0001111000': 23, '0100111000': 2, '0000100011': 3, '0000111101': 1, '1000001001':
1, '0000111001': 2, '1111110111': 11, '1010010101': 9, '1010001111': 10, '111110000
1': 3, '1011101101': 7, '0101100011': 1, '1011111101': 8, '1111011001': 7, '11110000
11': 2, '0100100110': 5, '1111000111': 5, '1011000011': 1, '1011011000': 9, '0000010
111': 2, '1101110100': 14, '0100000101': 6, '0100111111': 5, '0100010110': 2, '11001
10000': 5, '1011000000': 4, '1111100010': 1, '0001100010': 25, '1111101001': 6, '000
0100111': 1, '0000001001': 2, '0101101011': 1, '1011110101': 10, '0100111010': 8, '0
100100101': 3, '0100111011': 5, '1111000100': 2, '0100011010': 3, '0100110111': 3,
'1100111000': 6, '1010001011': 4, '1011100101': 3, '0111111010': 2, '0100110000': 1,
'1111001111': 3, '1011111111': 4, '1011100001': 1, '1111101101': 14, '0100001000':
1, '1111010010': 2, '1100111101': 2, '1111101000': 5, '0100111110': 4, '1100111100':
6, '0100110110': 8, '0100011001': 4, '0101010111': 3, '0011000110': 2, '1011101110':
4, '1001101011': 1, '0100010011': 2, '0111001111': 6, '0100011110': 3, '0001000001':
1, '1100100010': 8, '1011000101': 9, '0011001011': 2, '0000101101': 2, '0000110011':
2, '0000110001': 3, '0001110100': 4, '1111000010': 2, '0000101011': 3, '0111010100':
1, '1000111011': 1, '1111101100': 1, '1011100100': 4, '1111101010': 2, '0100010010':
4, '1111100100': 1, '1011101010': 2, '1111011100': 2, '1010000110': 1, '1011001000':
1, '1111000110': 1, '1010100100': 1, '1101100111': 1, '1011010010': 2, '1100101011':
1, '1000100111': 1, '0110111110': 1, '1011001111': 1, '0011010000': 3, '0000111011':
2, '0100100000': 5, '1100111001': 1, '0000011111': 3, '0001010001': 1, '0110000000':
1, '1010010000': 2, '0101010011': 1, '0100100001': 2, '1010001010': 2, '0011001111':
7, '0110110110': 1, '0011100000': 2, '1111001110': 3, '0100101101': 1, '1111010101':
2, '1010101000': 1, '0011110010': 1, '0011000100': 1, '0000100010': 2, '0111111000':
1, '0011010110': 1, '0101100111': 1, '0011101000': 1, '0000011001': 1, '1000111001':
1, '1111010110': 1, '1010000010': 1, '1010010100': 2, '1011111000': 1, '0001001001':
1, '1010000100': 1, '1010011100': 1, '0000010001': 1, '1111000000': 2, '0000101111':
1, '1010011110': 1, '1111001000': 1, '1110101000': 1, '0101110101': 1, '0101110100':
2, '1100100001': 1, '0000111111': 2, '0000010011': 1, '0011010100': 1, '0100101110':
3, '1110110010': 1, '0101011011': 1, '0101110111': 1, '0000110100': 1, '0001101111':
1, '1100110101': 1, '1100011111': 1, '1110001100': 1}
```

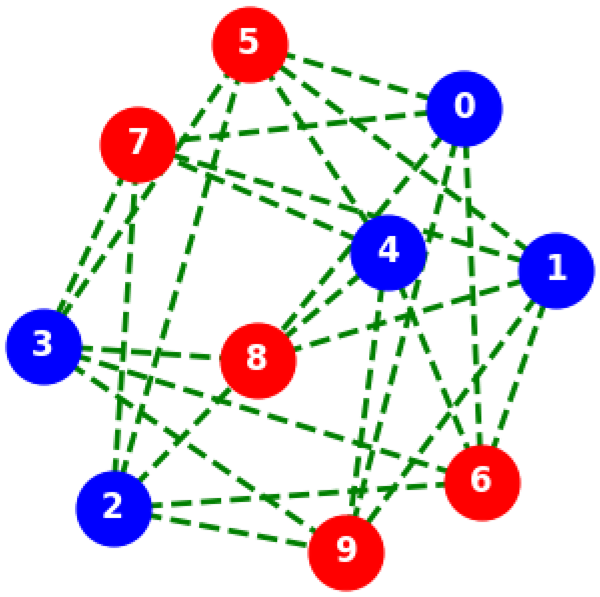
In [237…

```python
interpret_solution(T, best_bs_T)
print("Cut value: "+str(compute_cut_size(T, best_bs_T)))
print(T, best_bs_T)
```



```
Cut value: 9
Graph named 'complete_bipartite_graph(5, 5)' with 10 nodes and 9 edges 0000011111
```

In [238…
```
# interpret the same solution, but with the whole graph instead of the tree
interpret_solution(graph, best_bs_T)
print("Cut value: "+str(compute_cut_size(graph, best_bs_T)))
print(graph, best_bs_T)
```



```
Cut value: 25
Graph named 'complete_bipartite_graph(5, 5)' with 10 nodes and 25 edges 0000011111
```

In [239…
```
# Sample your optimized quantum state using Aer for the tree

optimized_state_H = ansatz_H.assign_parameters(qit_evolver_H.param_vals[-1])
optimized_state_H.measure_all()
counts_H = backend.run(optimized_state_H, shots=shots).result().get_counts()
```

```
# Find the sampled bitstring with the largest cut value
cut_vals_H = sorted(((bs, compute_cut_size(H, bs)) for bs in counts_H), key=lambda
best_bs_H = cut_vals_H[-1][0]

# Now find the most likely MaxCut solution as sampled from your optimized state
# We'll leave this part up to you!!!
most_likely_soln = ""

print(counts_H)
```

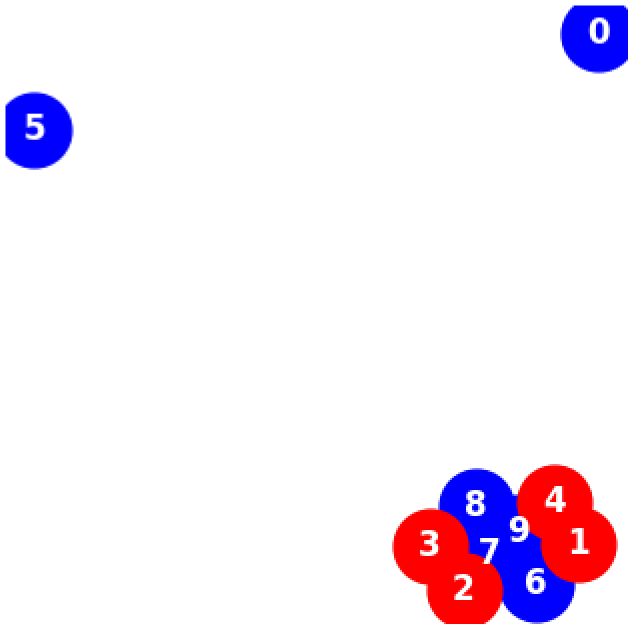
```
{'1111111101': 227, '1000011101': 66, '1010010100': 772, '0100011011': 56, '00010001
10': 53, '0100000001': 118, '0001111100': 47, '1110010100': 115, '0110100010': 83,
'1001010101': 45, '1110110101': 88, '1001011010': 484, '0110101111': 1038, '11110100
11': 92, '0001011100': 39, '0010110110': 2, '0101111011': 82, '0011111110': 272, '10
10011000': 970, '0100100110': 381, '1001111010': 476, '0011011110': 255, '111000100
0': 196, '1001010110': 678, '0110100001': 114, '1111011101': 251, '1110110110': 2,
'1000111101': 61, '1000011010': 304, '0000101000': 67, '1000000010': 107, '101011100
0': 987, '0010011001': 52, '1111110100': 8, '1000010100': 460, '0010100100': 246, '0
110001001': 664, '0010000010': 195, '1011011001': 452, '0010101110': 217, '010100000
1': 146, '1100010111': 3, '0010001001': 176, '0110000010': 82, '1011110111': 299, '0
000111110': 263, '1000001000': 16, '0000100010': 238, '1110011111': 8, '0001110001':
96, '0100001100': 14, '0100010000': 2, '0001001101': 1291, '0000101001': 92, '100000
0011': 127, '1010111001': 1012, '1001110000': 1072, '1000110000': 159, '1111010000':
1, '0110101110': 990, '1001011011': 471, '1110010010': 1250, '0100000111': 425, '000
1111110': 255, '0001000000': 5, '0100011101': 75, '1001010011': 1, '0110100100': 46
0, '1110110011': 1207, '1000111010': 295, '1011111100': 105, '0110001111': 951, '110
0011100': 121, '0010101011': 3, '0101000110': 1006, '0101100111': 950, '0110001110':
1058, '0111000101': 279, '1001010100': 23, '0110100011': 73, '1111011111': 3, '11101
10100': 117, '1000111111': 115, '0010100000': 2, '0101001101': 142, '0000001001': 6
4, '1000001100': 3, '1111110000': 5, '1000010000': 137, '0001110110': 161, '01000011
01': 15, '0001001100': 1248, '1100011010': 136, '0101100110': 980, '0011111101': 43,
'1010011001': 1037, '0100100111': 411, '0110010011': 9, '1101100000': 250, '11101000
10': 35, '0111001111': 151, '1001100010': 74, '0111000100': 306, '1011011000': 421,
'0000111111': 239, '1010000011': 80, '0011101111': 31, '0011000100': 162, '111101010
0': 4, '1001011111': 133, '0110101010': 27, '1000110100': 466, '1100111110': 5, '001
0010010': 1, '1100000000': 261, '1011011110': 152, '1001001011': 3, '1101100010': 3
9, '1010100010': 82, '0111001100': 9, '1001010010': 1, '0110100101': 475, '111011001
0': 1229, '0010011111': 270, '1011010001': 20, '0001000001': 6, '0100011110': 4, '11
10010011': 1243, '0100000110': 413, '0001111101': 31, '0101000010': 79, '001010111
1': 195, '0010101100': 5, '0101000111': 997, '1011111000': 416, '1111100001': 231,
'0110000000': 131, '1100010101': 64, '0101101011': 804, '1011110101': 53, '111010100
0': 181, '1000001001': 17, '0000111101': 27, '0000100011': 237, '0011101001': 2, '01
11101011': 489, '1000011011': 284, '0010111000': 49, '1101010000': 210, '010110001
0': 82, '1011111110': 114, '1011000100': 50, '1101011100': 167, '1101000100': 1, '00
00000011': 235, '1010111100': 77, '0101001010': 820, '0010100101': 241, '001111111
1': 270, '1001111000': 1, '0111000011': 72, '1011001000': 1, '0101001011': 787, '101
0011100': 89, '0101111111': 6, '0110101000': 683, '1101110000': 208, '1101100011': 3
0, '1010100011': 91, '0111000000': 109, '1001110001': 1021, '0001101101': 1221, '001
1001101': 65, '1110000010': 43, '1111000100': 126, '1010000101': 81, '1010011111': 1
42, '0001101100': 1278, '0100100010': 104, '1011010110': 277, '1010010011': 150, '11
00111010': 152, '1011011010': 1, '0010010110': 2, '1111110011': 99, '0011100011': 14
5, '1101000001': 296, '1010111111': 130, '0000000110': 21, '0101001111': 5, '0010100
010': 174, '1111111100': 281, '1000000000': 4, '0001010000': 86, '0101110001': 8, '0
010111110': 270, '0010001000': 161, '0110000011': 79, '1100010110': 7, '0101101010':
761, '1011110110': 266, '0010011110': 273, '1011010010': 14, '1100110010': 81, '0111
001010': 482, '1110100101': 190, '1111011110': 1, '1001010111': 642, '0110100000': 1
13, '1110110111': 1, '1000111110': 139, '0000111010': 115, '0000100110': 18, '101000
1111': 15, '1010010101': 792, '0101010001': 20, '0000011111': 221, '1110111101': 14
1, '1000110110': 17, '1111010110': 68, '1001011101': 86, '0111111101': 127, '1111100
100': 112, '0101111101': 84, '0001011110': 207, '0111001110': 137, '1110100001': 22
7, '1000111011': 267, '0110100111': 1, '1001010000': 995, '1011011100': 93, '1100000
110': 28, '0010010100': 17, '1100111100': 116, '0000001111': 5, '1101001010': 16, '1
010110110': 8, '1101010110': 178, '0100110101': 1, '0101011100': 97, '0000010100':
4, '0100100001': 133, '0001100010': 186, '0010101000': 150, '1101011010': 227, '0000
000001': 3, '1101000110': 49, '0001110111': 170, '0100001110': 15, '0100010110': 1,
'0001001011': 95, '1101011110': 2, '1011000010': 46, '1010111110': 116, '110100001
```

```
0': 35, '0110101001': 695, '1101110001': 191, '0100111101': 66, '0000011110': 259,
'0111101010': 490, '1101111100': 189, '0110001000': 696, '0010000011': 191, '1100001
001': 79, '1101011011': 249, '1011000101': 57, '0000000010': 237, '0000110110': 6,
'1111111000': 15, '0100001000': 288, '0110011110': 7, '1010101111': 15, '011100110
1': 3, '1110100000': 284, '1110001001': 184, '0011011101': 49, '1100001000': 60, '10
00111100': 62, '1001010001': 1061, '1111011100': 245, '0110111000': 2, '1001000001':
7, '1111000001': 247, '0000101101': 83, '0100011100': 53, '0001000111': 38, '0100000
000': 117, '0001111011': 204, '1110010101': 104, '1111110111': 87, '1000001101': 3,
'1000010111': 21, '1000110101': 485, '1010010010': 142, '0010110111': 1, '010111110
0': 100, '0001011011': 172, '0111100100': 298, '1001110110': 687, '0010000101': 239,
'0010111100': 31, '0101110111': 46, '0001010110': 169, '1111011001': 19, '110111110
1': 188, '0111001011': 491, '0010010000': 2, '1100000010': 58, '1001111011': 476, '0
010001110': 225, '1110111001': 48, '1101110110': 170, '0110000110': 1, '1100010011':
63, '1011110011': 5, '0101101101': 126, '1001111111': 108, '1110101001': 160, '00110
01100': 48, '0000011010': 118, '0101010110': 49, '1010111101': 77, '1101000011': 44,
'1101011101': 165, '1011000011': 59, '1000011111': 130, '0001000011': 160, '11100100
01': 1, '0001111111': 285, '1010100100': 87, '0111111100': 111, '1100011011': 136,
'1110101111': 74, '0111000010': 68, '1110111000': 35, '1001011000': 2, '1101110111':
165, '0111100010': 47, '1001110111': 671, '1110010111': 1, '0100000010': 116, '01000
11010': 48, '1101010100': 7, '0101001001': 15, '0000001101': 109, '1010110100': 802,
'0101001100': 144, '0110010101': 3, '1101100110': 49, '0100001011': 48, '000111000
0': 98, '0100111010': 61, '0101010111': 40, '0000001110': 5, '1010110101': 787, '110
1001011': 14, '1000110001': 140, '1110111100': 153, '1111010001': 4, '1001011100': 8
7, '1100010000': 19, '0110000101': 434, '0111000001': 140, '1110101110': 62, '110011
0111': 3, '1011010111': 303, '1011111001': 420, '0101011011': 102, '0000010101': 9,
'1100010100': 62, '0110000001': 121, '1011110100': 42, '0101101100': 139, '010010111
1': 23, '1110010110': 1, '0001111010': 180, '0100000011': 115, '1100010001': 28, '01
10000100': 494, '1100101000': 63, '1101000000': 262, '0000000111': 22, '0001101011':
104, '0100101010': 48, '1011011101': 108, '1100000001': 298, '1100111101': 145, '111
1001000': 2, '0111011100': 103, '1000101100': 1, '1100100110': 23, '0000110001': 15,
'0100001001': 291, '0010011100': 35, '1011010100': 45, '0110111100': 85, '010111000
0': 17, '0010111101': 53, '0001010001': 100, '1100011101': 148, '0110001010': 33, '0
101100011': 93, '1011111101': 95, '0000001100': 103, '1010110011': 146, '101111111
1': 117, '0101100001': 129, '1111100011': 36, '1011101111': 4, '1100101111': 2, '100
0100011': 122, '0011111000': 34, '1010011110': 118, '1111001001': 7, '1101111011': 2
26, '0011100010': 129, '1010001110': 18, '0111101111': 162, '0100101000': 243, '0111
100101': 255, '0111101001': 19, '0011101011': 71, '1111000101': 106, '1001100011': 8
0, '1110100011': 39, '1111101010': 7, '1011100100': 53, '0010111111': 284, '00101101
01': 12, '0101111010': 86, '1011010101': 56, '1100110101': 53, '1001111101': 73, '11
01100001': 296, '0110010010': 5, '0010000100': 226, '1110000001': 242, '1100100000':
294, '1100100010': 41, '1011100010': 70, '0001011010': 185, '1111100101': 121, '1110
000101': 170, '0011001111': 25, '0011100100': 144, '1101111010': 232, '1000010101':
456, '1111010010': 104, '0111010110': 16, '0110111101': 84, '0011000011': 139, '0011
101010': 55, '0000101100': 100, '0100001010': 55, '1110011101': 153, '0010010101': 1
1, '1100000111': 24, '1100111011': 129, '0010101001': 171, '0000000000': 4, '1101000
111': 42, '1011101110': 3, '1111100000': 280, '1111000000': 222, '1000011110': 110,
'0011000101': 132, '1010001001': 46, '0011110111': 65, '0001000010': 159, '010001111
1': 8, '0111111000': 2, '0001011111': 252, '0100101001': 274, '0100100000': 128, '00
01100011': 154, '1001111110': 112, '1100100001': 280, '0010011000': 49, '100001000
1': 145, '0010010001': 1, '1100000011': 53, '1011011111': 135, '0110001011': 32, '01
01100000': 119, '1110100100': 195, '0000011011': 100, '0100111100': 62, '011110001
1': 55, '0001001010': 112, '0100001111': 23, '1110011100': 160, '1101010010': 5, '10
11001110': 2, '0000001011': 6, '1010110010': 142, '1110000000': 267, '0011011111': 2
78, '0011010111': 60, '0000010110': 2, '0101011010': 104, '0111100001': 117, '001000
1111': 246, '0111011101': 129, '1111100010': 29, '1000100010': 123, '1010000100': 9
4, '0010111001': 49, '1010100101': 84, '1110001111': 93, '0111100000': 121, '1001000
```

```
010': 78, '1111000010': 35, '1110011000': 49, '0100101110': 20, '0101010100': 1, '00
00011100': 22, '0100111011': 53, '0000111011': 110, '0011001110': 29, '0110011101':
89, '1010001000': 49, '0011110110': 64, '0011111100': 55, '1010110001': 1, '00000010
10': 8, '1011001111': 1, '1101010001': 191, '0000111100': 32, '0010101101': 2, '0101
000000': 144, '0011000010': 113, '0011011000': 18, '1110001110': 71, '1110000100': 1
88, '0101011111': 6, '0000010001': 13, '0101110110': 48, '0001010111': 194, '0001100
111': 46, '0001100110': 52, '1111011000': 23, '1101111110': 8, '1100101001': 80, '10
01110100': 31, '1111000011': 38, '1001000011': 107, '1000010110': 20, '0111011110':
7, '0111110110': 15, '1110000011': 41, '1010000010': 88, '0011101110': 26, '00110111
00': 50, '1011100101': 59, '0010100011': 161, '1110111110': 6, '1000110111': 9, '011
0101011': 21, '1001011110': 128, '1111010111': 70, '0110000111': 1, '1100010010': 5
9, '0011101101': 84, '1111101111': 15, '1011100011': 53, '1101010111': 173, '0101000
011': 81, '0011001011': 74, '0110010100': 3, '1101100111': 69, '0011111001': 21, '10
10011101': 66, '1100001111': 1, '1011010011': 6, '0010011101': 35, '1100110011': 60,
'1010101001': 50, '0000001000': 65, '0101011101': 93, '0111001001': 12, '101000000
1': 5, '0000101011': 6, '0111101110': 157, '0100100011': 95, '0000011101': 31, '1100
100111': 24, '0100101101': 17, '0100101100': 13, '0011011001': 19, '0100111111': 5,
'0001101010': 101, '1010101000': 42, '0110011100': 84, '1111001010': 11, '001110010
1': 128, '0000110101': 10, '1100110100': 62, '0010000001': 5, '0000010000': 12, '011
0111001': 2, '1001000000': 7, '1110011001': 37, '1001111100': 83, '1000101000': 21,
'1000111000': 5, '0100110001': 5, '1111110010': 83, '1010100001': 8, '0000111000':
1, '1111110110': 53, '0011010110': 92, '0111011000': 2, '1111001111': 16, '011011111
1': 8, '1000000001': 9, '1100110001': 32, '0010100001': 4, '0101001110': 4, '0000100
000': 4, '1111101110': 8, '1011100000': 8, '1100011000': 1, '0011010000': 3, '001000
1101': 4, '1111101000': 5, '1001110011': 1, '0111001000': 13, '0011001010': 51, '100
0011100': 52, '0111010010': 1, '1000101001': 15, '1010101110': 20, '0110011111': 1,
'0011001001': 7, '0011101000': 5, '0111101100': 8, '1001110101': 25, '1000101111':
1, '1100100011': 62, '0110111110': 9, '0110110010': 5, '0001011101': 45, '011110100
0': 15, '0011101100': 59, '0010000000': 4, '0101101110': 2, '1011110010': 14, '10100
00000': 5, '0111110111': 15, '0101010000': 21, '1101101010': 17, '1011101001': 1, '0
000110111': 2, '1111111001': 21, '0101011110': 5, '1001000111': 1, '1011010000': 22,
'1100001110': 2, '0000100111': 14, '0110110011': 7, '0001100000': 4, '0000110000': 1
4, '1001001010': 3, '1011110000': 19, '1100110000': 40, '0101101111': 1, '101111000
1': 18, '1101110010': 5, '1011000001': 10, '1101011111': 7, '0101000101': 1, '001010
1010': 6, '1000110011': 10, '1001100001': 9, '0111010111': 14, '1000110010': 8, '110
0110110': 1, '0101001000': 10, '1101010101': 6, '1101101011': 14, '0101101001': 6,
'0111111001': 1, '0111111110': 4, '0110110100': 1, '0010001100': 6, '0010001011': 2,
'0100110111': 1, '1010110111': 3, '0000100001': 3, '0000101010': 2, '1000011000': 6,
'0101111110': 4, '1001001100': 3, '0111010011': 1, '1111101011': 8, '1111110101': 4,
'1100101110': 4, '1000100000': 7, '1001100000': 11, '0111011111': 5, '1001001101':
7, '0000110100': 7, '0100101011': 47, '0001101000': 2, '0111111111': 9, '100110110
0': 5, '1101111111': 4, '1111001110': 10, '0111100111': 2, '1100111111': 6, '1111010
101': 7, '1011100001': 2, '1100011110': 3, '1000010011': 6, '0100110000': 3, '000010
1110': 8, '0011110100': 2, '1101110101': 5, '0010110100': 11, '1010110000': 2, '0011
100001': 5, '0001100001': 3, '1000111001': 1, '0001001111': 1, '0100010010': 1, '110
1101000': 1, '1110010000': 1, '1001101011': 3, '0001010101': 1, '0111011001': 1, '11
11001011': 6, '0000101111': 9, '1010100000': 5, '1111111111': 3, '1011000000': 2, '0
011000001': 3, '1000100001': 5, '0111000111': 1, '1011001101': 1, '1101010011': 5,
'1001000110': 4, '0010001010': 5, '1000011001': 5, '1010010110': 8, '0111000110': 3,
'1110101011': 1, '1101110100': 4, '0011001000': 4, '1111101001': 4, '0011000000': 4,
'0101101000': 7, '1000010010': 4, '0100111110': 2, '1001101000': 1, '1010010111': 6,
'0011100110': 1, '0011110001': 1, '1100101010': 1, '0111101101': 8, '1010010001': 5,
'1001101010': 1, '0101000100': 1, '0110011001': 2, '0110011000': 2, '0000010111': 5,
'1011101000': 1, '0110001100': 2, '1100011111': 2, '1001100111': 2, '1001101101': 1,
'0011110000': 1, '1110011110': 1, '0110110101': 1, '1010011011': 2, '1101110011': 3,
'0001101001': 2, '1101101101': 2, '0001010100': 1, '1000000110': 1, '1111111110': 1,
'1001100110': 2, '0011110010': 1, '0001001000': 1, '1001101111': 1, '0010110011': 1,
'0100010001': 1, '0101100101': 1, '1101101001': 1, '1000100110': 1, '0110110111': 1,
'1011101011': 1}
```

In [240…

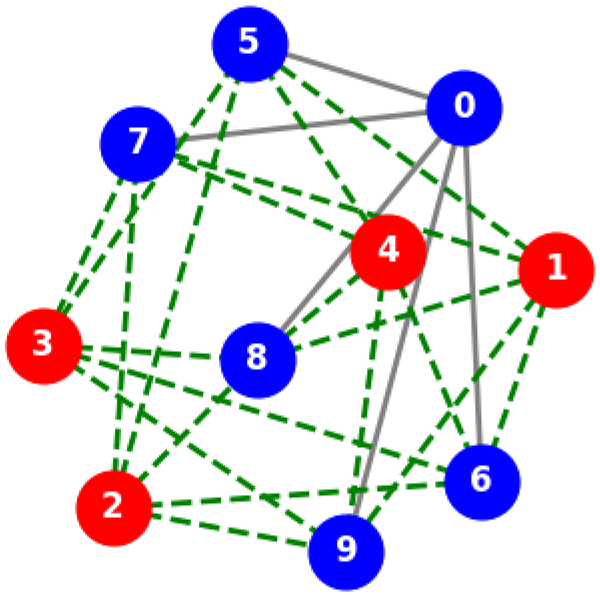
```
interpret_solution(H, best_bs_H)
print("Cut value: "+str(compute_cut_size(H, best_bs_H)))
print(H, best_bs_H)
```



```
Cut value: 16
Graph named 'complete_bipartite_graph(5, 5)' with 10 nodes and 16 edges 0111100000
```

In [241…

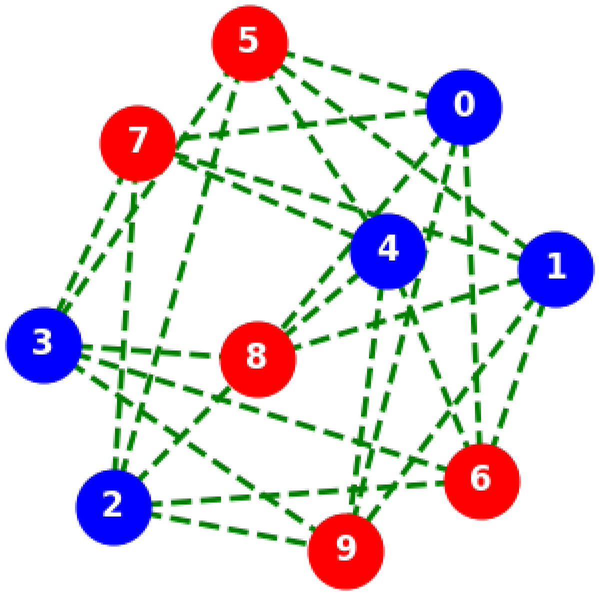
```
interpret_solution(graph, best_bs_H)
print("Cut value: "+str(compute_cut_size(graph, best_bs_H)))
print(graph, best_bs_H)
```



```
Cut value: 20
Graph named 'complete_bipartite_graph(5, 5)' with 10 nodes and 25 edges 0111100000
```

In [243…

```
# choose the better result from best_bs_T and best_bs_H
best_bs_final = best_bs_H if compute_cut_size(graph, best_bs_H) > compute_cut_size(
final_cut_value = compute_cut_size(graph, best_bs_final)

# choose the better result ansatz from ansatz_T and ansatz_H
ansatz_final = ansatz_H if compute_cut_size(graph, best_bs_H) > compute_cut_size(gr

interpret_solution(graph, best_bs_final)
print("Cut value: "+str(final_cut_value))
print(graph, best_bs_final)
```



```
Cut value: 25
Graph named 'complete_bipartite_graph(5, 5)' with 10 nodes and 25 edges 0000011111
```

In [244…

```
# from qiskit_aer import AerSimulator

# shots = 100_000

# Sample your optimized quantum state using Aer
# backend = AerSimulator()
optimized_state = ansatz.assign_parameters(qit_evolver.param_vals[-1])
optimized_state.measure_all()
counts = backend.run(optimized_state, shots=shots).result().get_counts()

# Find the sampled bitstring with the largest cut value
cut_vals = sorted(((bs, compute_cut_size(graph, bs)) for bs in counts), key=lambda
best_bs = cut_vals[-1][0]

# Now find the most likely MaxCut solution as sampled from your optimized state
# We'll leave this part up to you!!!
most_likely_soln = ""

print(counts)
```

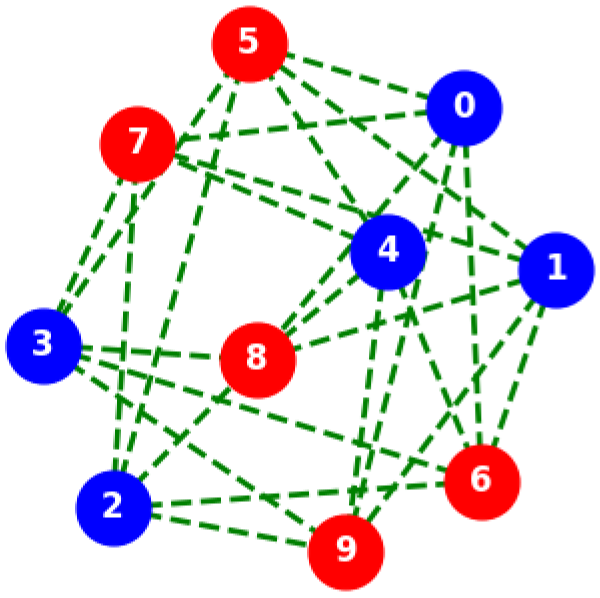
```
{'1000100111': 3, '1100101011': 365, '1010010110': 8, '1010001110': 39, '101000010
1': 57, '1001111101': 1764, '0010001110': 96, '0000001110': 147, '1010110101': 48,
'1101010101': 179, '0101001000': 24, '0000111011': 865, '0000100101': 94, '000100110
0': 7, '0100010111': 150, '0001110110': 129, '0100001101': 378, '1110011010': 75, '1
111101010': 229, '1011100100': 23, '1001111000': 2, '0111000011': 202, '0000011011':
190, '0101010101': 104, '0100111100': 579, '1100010111': 57, '0110000010': 1775, '00
10001001': 374, '1000000101': 26, '1000011111': 131, '1100010001': 190, '011000010
0': 248, '0101101111': 16, '1011110001': 76, '1111000000': 635, '0000100110': 87, '0
000111010': 85, '1000110001': 93, '1110111100': 103, '1111010001': 132, '011010110
1': 136, '0011101100': 114, '0111101000': 402, '1101110000': 4, '0110101000': 43, '1
110010000': 152, '0100000101': 83, '0100011111': 7, '1101011101': 46, '1011000011':
632, '1010111101': 3, '0000000100': 1, '1101000011': 563, '1000111100': 163, '111011
0001': 246, '0110100110': 493, '1001010001': 60, '1100110010': 31, '0010011110': 60,
'1011010010': 302, '0010100101': 140, '0101001010': 63, '0110111100': 487, '01110100
01': 56, '1101110110': 362, '1001011001': 485, '1000100110': 22, '1100101100': 40,
'1011101100': 539, '0010001101': 233, '1000101111': 1179, '0110010100': 453, '110110
0111': 22, '0000010111': 956, '1011111011': 9, '0101100101': 147, '0111111000': 19,
'0001011111': 631, '1101000111': 432, '0010101001': 71, '1011000111': 551, '00001101
11': 589, '1111111001': 131, '0110000111': 4, '1100010010': 32, '1011110010': 332,
'0101101110': 134, '0100100001': 149, '0111100010': 13, '1110011110': 35, '000111001
0': 21, '0100010011': 489, '0110101100': 29, '1001011101': 265, '1111010110': 83, '1
110111101': 1, '1000110110': 62, '0111010000': 1116, '0111101100': 279, '001110100
0': 63, '1110110000': 19, '1000111011': 298, '1111000100': 832, '1000110101': 74, '1
111010101': 88, '1100111101': 13, '1100000001': 10, '1011011101': 25, '0010010011':
355, '0111001110': 91, '1110100001': 296, '0011001110': 205, '0110111000': 940, '111
1010100': 306, '1001011111': 175, '0110101010': 246, '0000101111': 310, '010111011
1': 9, '0010111100': 579, '1110010011': 56, '0100000110': 172, '0001111101': 179, '0
101011011': 28, '0000010101': 217, '0011110101': 15, '0111100100': 75, '1101000110':
1, '0000000001': 3, '1010111010': 66, '0010101000': 3, '1011000110': 3, '110101101
0': 165, '0011111001': 40, '1010000111': 11, '0101100001': 155, '0111100000': 140,
'0011001001': 39, '1101001111': 6, '0000001010': 24, '1010110001': 139, '101100111
1': 7, '1101010001': 110, '0111001001': 67, '1001110010': 160, '0011000011': 157, '0
011101010': 251, '1000011110': 669, '1001111001': 693, '0111000010': 70, '111010111
1': 19, '1001101111': 644, '0100100101': 145, '1100011000': 11, '0010000110': 424,
'1000001110': 67, '1111110110': 353, '0000111000': 228, '0111101110': 102, '00001111
00': 757, '1011010111': 5, '1100110111': 35, '0001111011': 747, '1110010101': 130,
'0100011100': 7, '0001000111': 312, '0011000111': 216, '0011101110': 213, '111100011
1': 246, '1001000111': 876, '1101100011': 5, '0110010000': 663, '1110111010': 31, '1
000110011': 27, '1111010011': 5, '0110101111': 6, '1001011010': 304, '0010111110':
7, '0101110001': 61, '0001010000': 18, '0001010101': 139, '0010010111': 144, '110000
0101': 58, '1111010010': 487, '1001110110': 607, '0101000100': 12, '1101011001': 34
5, '0011010100': 314, '1101001000': 87, '1011001000': 84, '0010100110': 342, '010000
0100': 11, '0001111111': 687, '1110010001': 243, '0001000011': 108, '1100010011': 14
3, '0110000110': 690, '0100111000': 560, '0111110001': 78, '1011000010': 2, '1101011
110': 125, '1101000010': 1, '1010111110': 14, '0000000101': 46, '1000001100': 3, '11
11110000': 40, '1101111101': 410, '1111011001': 84, '0000011010': 142, '0101010110':
19, '1101110101': 37, '0110010101': 84, '1010100110': 57, '0000110101': 70, '0000101
011': 281, '1001011011': 43, '1111010000': 311, '0110101110': 66, '0100001001': 589,
'1100100110': 28, '1000101100': 20, '1111101100': 960, '1011100110': 2, '011101101
0': 48, '1000001111': 127, '1111110101': 17, '1000010101': 358, '1111011101': 9, '01
10100001': 678, '1101111001': 458, '1001010110': 114, '1110110110': 41, '100011110
1': 71, '0010100000': 2, '0101001101': 45, '0000001001': 361, '1011000100': 131, '00
00000011': 1, '1101000100': 40, '1010111100': 132, '1101101011': 187, '0110011010':
51, '1010101011': 222, '1010010101': 203, '1011110110': 597, '0101101010': 195, '011
0001001': 596, '0010000010': 362, '1110100110': 89, '1001000011': 464, '1111000011':
697, '1110110101': 37, '1001010101': 221, '0110100010': 265, '0101011101': 12, '0000
```

```
010011': 926, '0000011100': 8, '0101010100': 217, '0100111011': 115, '0101100110':
1, '1011111010': 99, '1100011010': 124, '0110001101': 163, '0101001001': 47, '101100
1100': 64, '1101010100': 90, '1101001100': 59, '0000001101': 548, '0001011011': 147,
'0101111100': 13, '0010110111': 77, '0001011110': 272, '1000010011': 263, '011111110
0': 9, '0100100010': 24, '0011110010': 49, '1101110010': 284, '0011010000': 135, '11
00001101': 49, '1011010001': 81, '1100110001': 201, '0000111001': 2, '1000001101': 8
0, '1000010111': 386, '1111101000': 944, '1001110001': 44, '1001111011': 272, '10000
10010': 14, '1000001010': 11, '1111110010': 528, '0100101110': 84, '0001101111': 17
0, '0011000010': 19, '1001001000': 15, '0110110001': 118, '1111001000': 606, '100001
1101': 11, '1000000111': 13, '1111111101': 35, '1100010101': 246, '0110000000': 162,
'0101101011': 70, '1011110101': 33, '1110000100': 740, '1110111110': 5, '100011011
1': 257, '1111010111': 2, '0110101011': 38, '1001011110': 715, '0000100001': 22, '00
00111111': 647, '0001111001': 144, '0111010100': 699, '0000011110': 286, '010101001
0': 30, '0100010001': 84, '0001001110': 241, '0010101010': 171, '0101000101': 65, '1
010111000': 151, '1111100101': 138, '1000100101': 39, '0100101101': 312, '010110001
1': 2, '1011111101': 146, '0110001010': 59, '1010101101': 31, '1011101101': 7, '1111
100001': 265, '0010011100': 7, '1100001010': 22, '1011010100': 123, '1001110011': 1,
'0111001000': 240, '0010110110': 11, '0101111011': 102, '1010010100': 81, '100110111
0': 116, '1110101110': 32, '0111000001': 85, '0011010101': 75, '1110000000': 691, '1
110100101': 157, '0111001010': 85, '1001100101': 47, '0101000011': 154, '010011010
1': 38, '0000010100': 59, '0001000110': 5, '0100011011': 37, '0100000001': 14, '0001
111100': 30, '1110010100': 184, '1011011001': 159, '1100111001': 6, '0000010110': 1,
'0101011010': 97, '0100110111': 103, '1111100000': 615, '1011101110': 86, '110010111
0': 25, '0111101001': 14, '0011101011': 58, '1000111111': 118, '1111100100': 182, '1
011101010': 129, '1000100100': 1, '1100101010': 85, '1011101000': 156, '0111101010':
305, '0101000001': 14, '0010101110': 121, '1001111110': 76, '0011001000': 54, '01111
00001': 706, '0001010001': 25, '0000011111': 578, '0101010001': 65, '0111001101': 1
5, '1110100000': 613, '0111000100': 291, '1010101100': 60, '1101101100': 374, '01111
10101': 18, '1001010010': 129, '0110100101': 261, '1101001101': 6, '1010110011': 11,
'1101010110': 84, '1011001010': 40, '0000001111': 22, '1010110110': 55, '110100101
0': 24, '1111101011': 54, '1011100111': 23, '1000101011': 744, '1011111001': 180, '1
110101100': 106, '0111000111': 95, '1001101100': 170, '0001010011': 114, '111110110
1': 4, '1011100001': 47, '1000101101': 61, '1100100001': 50, '1101111010': 80, '0101
100010': 1, '1011111110': 11, '1100011110': 136, '1110001001': 143, '1111001010': 6
4, '1100111000': 178, '0010101101': 209, '1010000100': 110, '0001101110': 261, '1101
110001': 111, '0110101001': 113, '1010001000': 8, '0011110110': 139, '0000010001': 5
9, '0101011111': 13, '1011011110': 173, '0010010010': 7, '0001101011': 167, '0100101
010': 143, '0111100101': 218, '0001000101': 48, '0100011010': 74, '1110010111': 7,
'0100000010': 143, '0100101100': 17, '1000011011': 75, '0010101100': 11, '010100011
1': 153, '1100101111': 163, '1111100011': 8, '1011101111': 67, '1010100001': 48, '11
00010000': 21, '0110000101': 172, '1011011011': 3, '1100000111': 8, '0010010101': 12
7, '1001010011': 34, '0110100100': 51, '1111011010': 84, '1110110011': 7, '100011101
0': 152, '0111000101': 127, '1110101010': 185, '1001101010': 83, '1000010100': 23,
'0010011011': 6, '1011010101': 118, '1100110101': 56, '1111001110': 43, '100100111
0': 120, '0001011010': 168, '0011100010': 2, '1101001110': 80, '1010110010': 41, '10
11001110': 88, '1101010010': 216, '0101110110': 76, '0010111011': 35, '0001010111':
83, '0101000110': 7, '0010101011': 79, '0000101110': 160, '0111010010': 56, '1000101
001': 21, '1100100101': 97, '1011100101': 92, '0110010001': 102, '0100110011': 61,
'0000010010': 2, '0101011110': 62, '0110011011': 2, '1101101010': 129, '1010101010':
114, '0011111000': 11, '1010000110': 99, '1010011110': 140, '0011011010': 83, '00111
11010': 52, '0100101001': 128, '0010010110': 3, '1100000100': 32, '1011011010': 153,
'1100111010': 83, '0001111000': 9, '1110111000': 313, '0000110011': 116, '000010110
1': 518, '1100110110': 34, '1100001100': 4, '0010011010': 62, '1011010110': 110, '00
10000101': 93, '1110101000': 83, '1001101000': 24, '0111000110': 16, '1110101011': 8
9, '1001101011': 465, '1010000010': 47, '1010011010': 113, '1000101110': 78, '001110
0001': 123, '0111110000': 109, '0111000000': 145, '1110101101': 13, '1101000001': 5,
'0000000110': 138, '0100101011': 140, '0001101010': 141, '0110010011': 178, '0011011
110': 44, '1110001000': 52, '0101111110': 5, '0010110001': 50, '0111011011': 2, '111
1001001': 13, '1001001001': 6, '0110110000': 53, '0100010100': 124, '1110000010': 23
5, '0010010100': 194, '1100000110': 36, '1100111100': 160, '0101001110': 125, '00101
00001': 125, '1100001001': 114, '1110000110': 147, '0101111001': 84, '0000101010': 8
8, '0100000111': 3, '0001111110': 34, '1110010010': 63, '1110100100': 147, '00001010
00': 1, '1000011010': 202, '1010010010': 31, '1010001010': 18, '1111011110': 29, '11
01111000': 8, '0110100000': 151, '1001010111': 31, '1110110111': 8, '1000111110': 10
4, '0011110111': 4, '1010001001': 86, '1001001111': 52, '0110000001': 73, '110001010
0': 53, '1011110100': 1, '0101101100': 95, '0100111111': 13, '0101010000': 125, '111
1110001': 167, '1000010001': 116, '1110101001': 24, '1001111111': 173, '1101111110':
13, '1000111000': 93, '0010111000': 459, '0011010110': 23, '0111011001': 17, '011110
1101': 21, '0111110110': 97, '1010010001': 118, '0100100110': 140, '0001101100': 56,
'0001100111': 14, '0010110101': 26, '0101111010': 56, '1110000101': 96, '001000001
1': 8, '0110001000': 16, '0000011101': 2, '0101010011': 52, '0101000010': 6, '001010
1111': 35, '1010101111': 129, '1101101110': 60, '1110100010': 35, '1100101001': 20,
'0110001110': 62, '1110011101': 1, '0100001010': 30, '0001110011': 9, '1010010111':
56, '0111011000': 1, '1111001100': 114, '0101011001': 73, '0101101000': 53, '0010001
010': 40, '0011110011': 10, '0100000011': 2, '1110010110': 5, '0001111010': 95, '011
1010011': 25, '0110111010': 30, '0111010101': 56, '1010100100': 31, '1010010011': 7
5, '1101101000': 122, '0100110001': 90, '0000010000': 13, '0000101001': 76, '0001001
010': 28, '0100010101': 169, '1010111011': 8, '0000000010': 44, '1101000101': 37, '1
101011011': 7, '1011000101': 70, '1001111010': 207, '1100000010': 22, '0010010000':
76, '1110001110': 16, '0001110001': 37, '0100010000': 63, '0001001101': 77, '0010000
100': 31, '1001100111': 40, '0100111010': 36, '1111000101': 85, '0001101101': 60, '0
111010110': 14, '0000111101': 11, '1000001001': 76, '0011010001': 38, '0011111101':
13, '0110011000': 46, '0100001110': 89, '0001110111': 70, '0011111011': 33, '1001110
101': 65, '0101001111': 1, '0010100010': 69, '1011001001': 12, '0101001100': 13, '00
11000100': 51, '1100001110': 36, '1011010000': 34, '1111000001': 33, '0000110001': 4
5, '0110010110': 3, '1101100101': 65, '1010110111': 40, '0101110010': 26, '000101010
0': 96, '0011101101': 25, '1011100000': 8, '1111101110': 51, '0000101100': 4, '11000
00011': 19, '0010010001': 63, '1110011000': 18, '1001100011': 8, '0111110010': 74,
'1101100100': 5, '0110010111': 27, '1010001111': 13, '0001011001': 97, '0001100101':
56, '1100100100': 5, '1000101010': 59, '1010011001': 4, '1111100111': 9, '101110101
1': 126, '1010101110': 65, '0010001000': 3, '1100010110': 4, '0110000011': 4, '00111
00101': 130, '1110110010': 44, '1110001010': 40, '0001001111': 19, '0100010010': 5,
'0011011011': 10, '1100011001': 9, '1010011000': 11, '0011000101': 72, '1011001101':
3, '1101010011': 16, '0011110001': 33, '1000100001': 12, '1100101101': 43, '00111001
10': 7, '0011010010': 46, '1101010000': 27, '0011001010': 66, '1100101000': 10, '011
1111011': 5, '1010100101': 104, '0011010011': 44, '0001000001': 3, '0100011110': 44,
'1100110011': 19, '0011101111': 11, '0110110101': 33, '1110001101': 21, '011001111
0': 10, '1101101111': 94, '0010111101': 1, '0101110000': 10, '1000110010': 20, '1100
110000': 4, '0001100001': 30, '1111111110': 3, '1000000110': 21, '1100011011': 10,
'1001001010': 25, '0010001111': 7, '0110110111': 14, '0010110010': 5, '0101111111':
15, '1110011011': 1, '0001110101': 33, '0010110011': 33, '1001010100': 33, '11111011
11': 5, '1011100011': 4, '0001010110': 31, '0011001101': 27, '0101101101': 50, '1000
000010': 5, '1111111010': 36, '0001010010': 16, '0111011110': 16, '0011011001': 28,
'0100011000': 23, '0110110011': 28, '1010000011': 20, '0010011000': 15, '100000001
1': 17, '1001001100': 26, '1101111011': 33, '0010111010': 40, '0101110101': 34, '101
1000000': 9, '0101110011': 9, '0011100111': 9, '1101100001': 61, '0110010010': 6, '1
011111000': 6, '1110001100': 10, '1010001101': 33, '0010001100': 4, '0001001001': 4
5, '0100101111': 42, '0011000110': 16, '1110000001': 35, '1111000110': 4, '010111110
1': 38, '0010110000': 12, '0100110110': 6, '1110100111': 1, '0111001100': 23, '00111
01001': 6, '0111101011': 19, '0000111110': 32, '1100111011': 41, '1001101101': 10,
'0110111011': 6, '1001111100': 11, '0110110110': 7, '0011111110': 7, '1101011111':
5, '1011000001': 6, '0001011101': 28, '1011111100': 6, '0101100100': 1, '000110100
```

```
1': 6, '1000111001': 18, '1000010110': 13, '0110110010': 9, '0100111110': 6, '011111
1001': 28, '1001001101': 8, '1100111110': 12, '0011001100': 19, '0111111010': 20, '0
101010111': 16, '1110000011': 18, '1001110111': 16, '0111100110': 9, '1011010011': 1
3, '1100001111': 17, '0011100100': 7, '1010101000': 19, '1001000100': 4, '010011001
0': 6, '0011000001': 20, '1111111000': 2, '0000110110': 14, '1101001001': 14, '00000
11000': 10, '1100100010': 1, '0111111101': 4, '1001101001': 1, '0000100010': 5, '100
1100001': 6, '0101100111': 6, '0011110000': 11, '1101111100': 4, '0001100110': 1, '0
000100011': 1, '1011101001': 5, '0100111001': 3, '0100101000': 6, '1001000101': 21,
'0001101000': 10, '0011010111': 9, '1000011001': 12, '1110111001': 7, '0000011001':
3, '1010100010': 2, '0110001100': 4, '1100011111': 4, '1010000000': 11, '001000000
1': 13, '0111110111': 2, '1010001100': 7, '1010010000': 15, '0101101001': 7, '100011
0000': 2, '1110011100': 3, '0100001111': 6, '0011111100': 14, '1010101001': 16, '001
0100100': 3, '1010000001': 6, '0011000000': 4, '0111100111': 7, '0101111000': 9, '00
11001111': 2, '0011011101': 3, '0100011001': 1, '1010111001': 2, '1010100000': 8, '1
010011100': 2, '1000000001': 2, '1111111011': 4, '1101101101': 6, '1000000100': 2,
'1110000111': 4, '0100110000': 7, '0111010111': 3, '1100001000': 5, '1100111111': 3,
'1010011011': 1, '1011110111': 5, '0000000111': 3, '0110011100': 2, '1111111100': 2,
'0000110000': 1, '0000100111': 1, '1001010000': 2, '0001110000': 1, '0010000000': 3,
'1101111111': 5, '0110111001': 2, '1101010111': 4, '0011100000': 3, '0100010110': 3,
'1111000010': 4, '0111110011': 2, '0010111001': 2, '1000011100': 3, '0010011001': 1,
'1010110000': 2, '1001110000': 1, '1000101000': 1, '0001001000': 2, '1111110011': 1,
'0110001111': 1, '0100111101': 1, '1111100010': 1, '1000010000': 1, '1111101001': 2,
'1011110000': 3, '1000100010': 1, '0111100011': 3, '1101101001': 1, '0011100011': 1,
'0111011101': 1, '0001100011': 1, '1010100011': 1, '0111111110': 1, '1110011001': 1}
```

In [245…

```python
interpret_solution(graph, best_bs)
print("Cut value: "+str(compute_cut_size(graph, best_bs)))
print(graph, best_bs)
```



```
Cut value: 25
Graph named 'complete_bipartite_graph(5, 5)' with 10 nodes and 25 edges 0000011111
```

Drumroll please... the scores!

In [246…

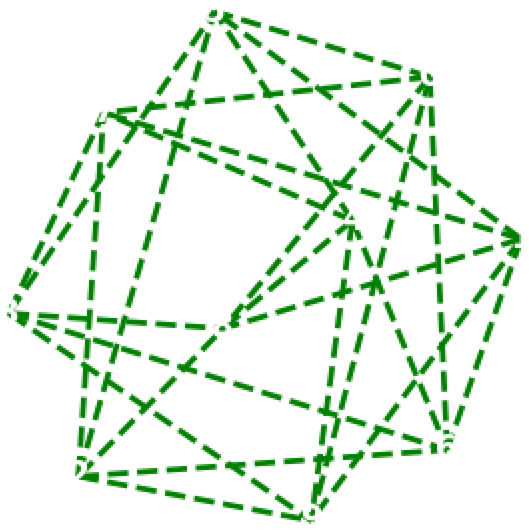
```
%%time
# Brute-force approach with conditional checks

verbose = False

G = graph
n = len(G.nodes())
w = np.zeros([n, n])
for i in range(n):
    for j in range(n):
        temp = G.get_edge_data(i, j, default=0)
        if temp != 0:
            w[i, j] = 1.0
if verbose:
    print(w)

best_cost_brute = 0
best_cost_balanced = 0
best_cost_connected = 0

for b in range(2**n):
    x = [int(t) for t in reversed(list(bin(b)[2:].zfill(n)))]

    # Create subgraphs based on the partition
    subgraph0 = G.subgraph([i for i, val in enumerate(x) if val == 0])
    subgraph1 = G.subgraph([i for i, val in enumerate(x) if val == 1])

    bs = "".join(str(i) for i in x)

    # Check if subgraphs are not empty
    if len(subgraph0.nodes) > 0 and len(subgraph1.nodes) > 0:
        cost = 0
        for i in range(n):
            for j in range(n):
                cost = cost + w[i, j] * x[i] * (1 - x[j])
        if best_cost_brute < cost:
            best_cost_brute = cost
            xbest_brute = x
            XS_brut = []
        if best_cost_brute == cost:
            XS_brut.append(bs)

        outstr = "case = " + str(x) + " cost = " + str(cost)

        if (len(subgraph1.nodes)-len(subgraph0.nodes))**2 <= 1:
            outstr += " balanced"
            if best_cost_balanced < cost:
                best_cost_balanced = cost
                xbest_balanced = x
                XS_balanced = []
            if best_cost_balanced == cost:
                XS_balanced.append(bs)

        if nx.is_connected(subgraph0) and nx.is_connected(subgraph1):
            outstr += " connected"
```

```
                if best_cost_connected < cost:
                    best_cost_connected = cost
                    xbest_connected = x
                    XS_connected = []
                if best_cost_connected == cost:
                    XS_connected.append(bs)
            if verbose:
                print(outstr)
```

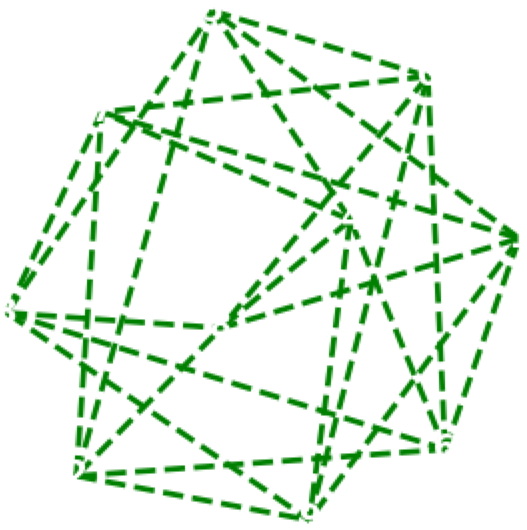
```
CPU times: total: 0 ns
Wall time: 92.4 ms
```

In [247…

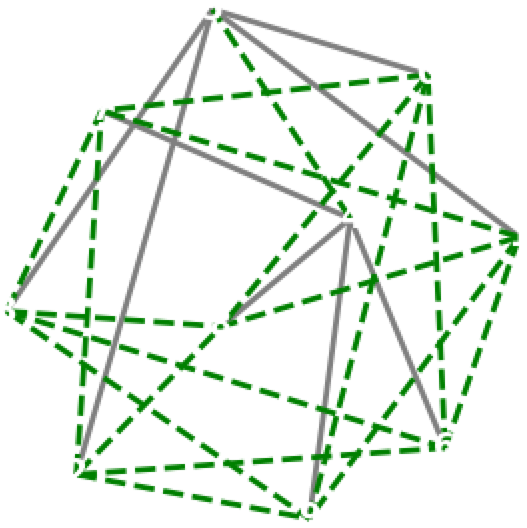
```
# This is classical brute force solver results:
interpret_solution(graph, xbest_brute)
print(graph, xbest_brute)
print("\nBest solution = " + str(xbest_brute) + " cost = " + str(best_cost_brute))
print(XS_brut)

interpret_solution(graph, xbest_balanced)
print(graph, xbest_balanced)
print("\nBest balanced = " + str(xbest_balanced) + " cost = " + str(best_cost_balan
print(XS_balanced)

interpret_solution(graph, xbest_connected)
print(graph, xbest_connected)
print("\nBest connected = " + str(xbest_connected) + " cost = " + str(best_cost_con
print(XS_connected)
plt.show()
```



```
Graph named 'complete_bipartite_graph(5, 5)' with 10 nodes and 25 edges [1, 1, 1, 1,
1, 0, 0, 0, 0, 0]

Best solution = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0] cost = 25.0
['1111100000', '0000011111']
```



```
Graph named 'complete_bipartite_graph(5, 5)' with 10 nodes and 25 edges [1, 1, 1, 1,
1, 0, 0, 0, 0, 0]

Best balanced = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0] cost = 25.0
['1111100000', '0000011111']
```



```
Graph named 'complete_bipartite_graph(5, 5)' with 10 nodes and 25 edges [1, 1, 1, 1,
0, 1, 0, 0, 0, 0]

Best connected = [1, 1, 1, 1, 0, 1, 0, 0, 0, 0] cost = 17.0
['1111010000', '1110110000', '1101110000', '1011110000', '0111110000', '1111001000',
'1110101000', '1101101000', '1011101000', '0111101000', '1111000100', '1110100100',
'1101100100', '1011100100', '0111100100', '1111000010', '1110100010', '1101100010',
'1011100010', '0111100010', '1000011110', '0100011110', '0010011110', '0001011110',
'0000111110', '1111000001', '1110100001', '1101100001', '1011100001', '0111100001',
'1000011101', '0100011101', '0010011101', '0001011101', '0000111101', '1000011011',
'0100011011', '0010011011', '0001011011', '0000111011', '1000010111', '0100010111',
'0010010111', '0001010111', '0000110111', '1000001111', '0100001111', '0010001111',
'0001001111', '0000101111']
```

In [248…

```python
# And this is how we calculate the shots counted toward scores for each class of th

sum_counts = 0
for bs in counts:
    if bs in XS_brut:
        sum_counts += counts[bs]

print(f"Pure max-cut: {sum_counts} out of {shots}")

sum_balanced_counts = 0
for bs in counts:
    if bs in XS_balanced:
        sum_balanced_counts += counts[bs]

print(f"Balanced max-cut: {sum_balanced_counts} out of {shots}")

sum_connected_counts = 0
for bs in counts:
    if bs in XS_connected:
        sum_connected_counts += counts[bs]

print(f"Connected max-cut: {sum_connected_counts} out of {shots}")
```

```
Pure max-cut: 1193 out of 100000
Balanced max-cut: 1193 out of 100000
Connected max-cut: 8170 out of 100000
```

In [249…

```python
def final_score(graph, XS_brut,counts,shots,ansatz,challenge):

    if(challenge=='base'):
        sum_counts = 0
        for bs in counts:
            if bs in XS_brut:
                sum_counts += counts[bs]
    elif(challenge=='balanced'):
        sum_balanced_counts = 0
        for bs in counts:
            if bs in XS_balanced:
                sum_balanced_counts += counts[bs]
        sum_counts = sum_balanced_counts
    elif(challenge=='connected'):
        sum_connected_counts = 0
        for bs in counts:
            if bs in XS_connected:
                sum_connected_counts += counts[bs]
        sum_counts = sum_connected_counts


    transpiled_ansatz = transpile(ansatz, basis_gates = ['cx','rz','sx','x'])
    cx_count = transpiled_ansatz.count_ops()['cx']
    score = (4*2*graph.number_of_edges())/(4*2*graph.number_of_edges() + cx_count)

    return np.round(score,5)
```

In [250…

```
print("Base score: " + str(final_score(graph,XS_brut,counts,shots,ansatz,'base')))
print("Balanced score: " + str(final_score(graph,XS_brut,counts,shots,ansatz,'balan
print("Connected score: " + str(final_score(graph,XS_brut,counts,shots,ansatz,'conn
```

```
Base score: 0.00954
Balanced score: 0.00954
Connected score: 0.06536
```

In [251…

```
# calculate our version of the score for the final solution
print("Base score: " + str(final_score(graph,XS_brut,counts,shots,ansatz_H,'base'))
print("Balanced score: " + str(final_score(graph,XS_brut,counts,shots,ansatz_H,'bal
print("Connected score: " + str(final_score(graph,XS_brut,counts,shots,ansatz_H,'co
```

```
Base score: 0.01028
Balanced score: 0.01028
Connected score: 0.07043
```

In [252…

```
# calculate our version of the score for the final solution
print("Base score: " + str(final_score(graph,XS_brut,counts,shots,ansatz_T,'base'))
print("Balanced score: " + str(final_score(graph,XS_brut,counts,shots,ansatz_T,'bal
print("Connected score: " + str(final_score(graph,XS_brut,counts,shots,ansatz_T,'co
```

```
Base score: 0.01094
Balanced score: 0.01094
Connected score: 0.07495
```

This is the main challenge: design optimal ansatz generators and Hamiltonians for the max-cut problem, subject to three specific conditions. Solutions for each condition can be submitted independently; however, a unified approach applicable to all three conditions will receive additional credit.

Submit your results

I guess this is the most important part. You probably want to report your results.

Please, follow the link: https://forms.gle/gkpSCe7HGr7QHZXQ6

Let's revisit the problem statement. **This is important!!!** We have three scoring methods to report:

1. Counts shots achieving the optimal max-cut value. Sorry, no approximation value this time.
2. Adds a balance constraint—subgraphs must have equal cardinality.
3. Requires connected subgraphs—all vertices within a subgraph must be mutually reachable.

For **in-person participants**: Prepare a presentation that explains your research and demonstrates the value of your proposed solution.

Can you solve this? Good luck!

Feedback

If you have any suggestions or recommendations abou the hackathon challange, please, don't hesitatate to leave your comment here: https://iter.ly/u42um

```
In [ ]:
```