# 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

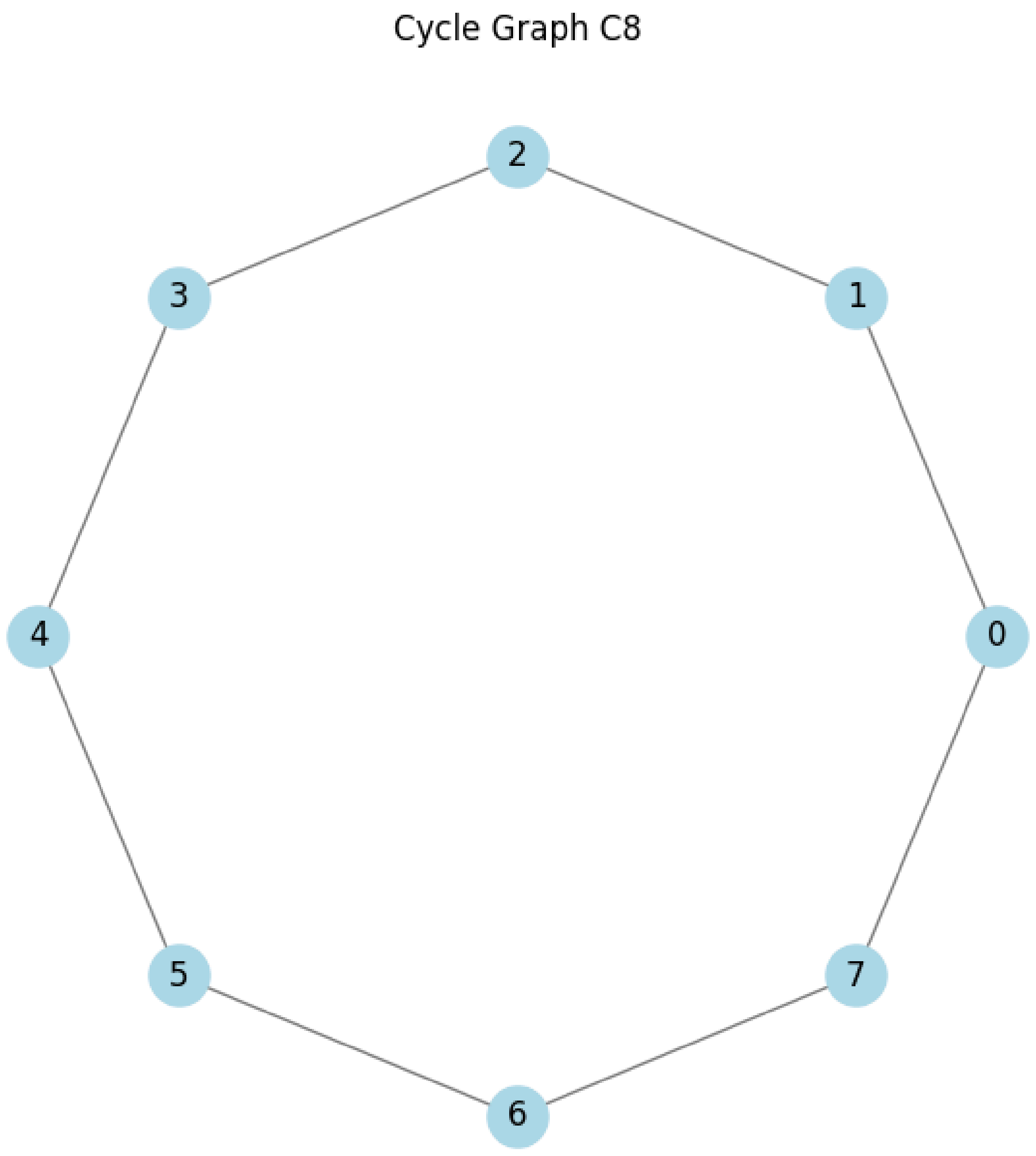
A simple graph, defined using the Python NetworkX library, will serve to illustrate the problem and you can explore further complexities.

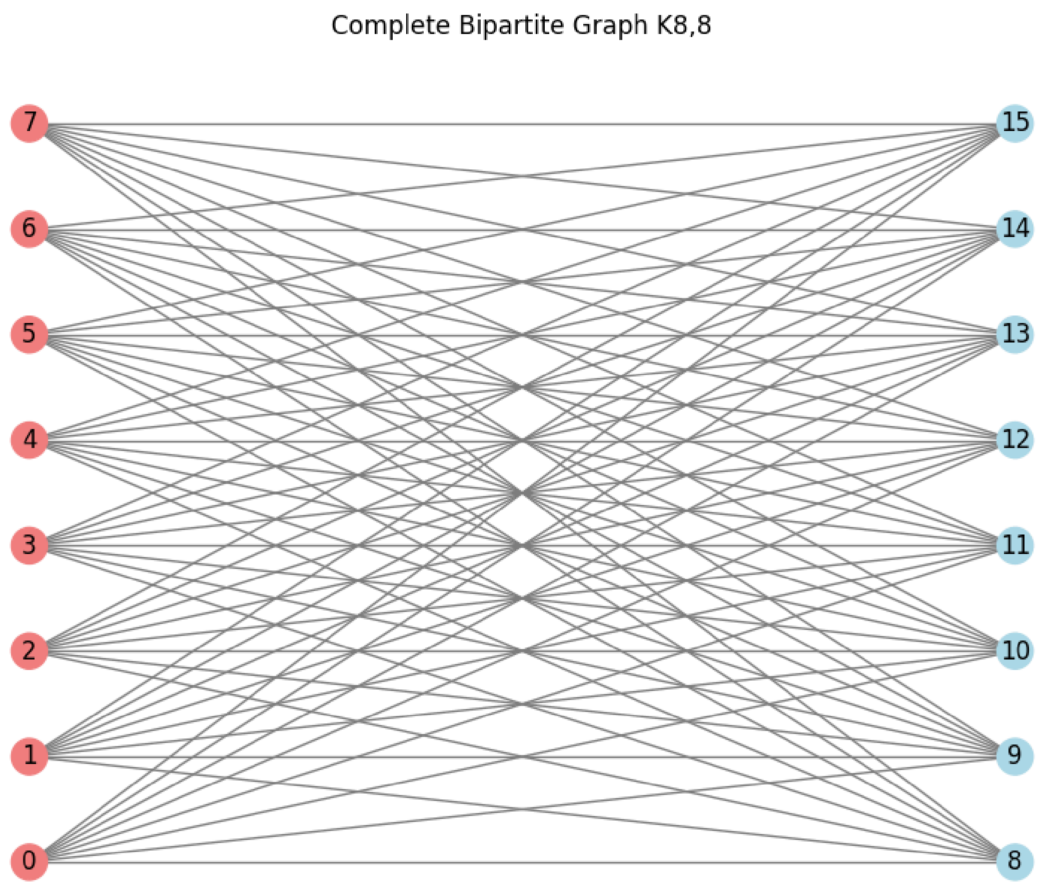
In [3]:
```
# other graphs candidates to check

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
```

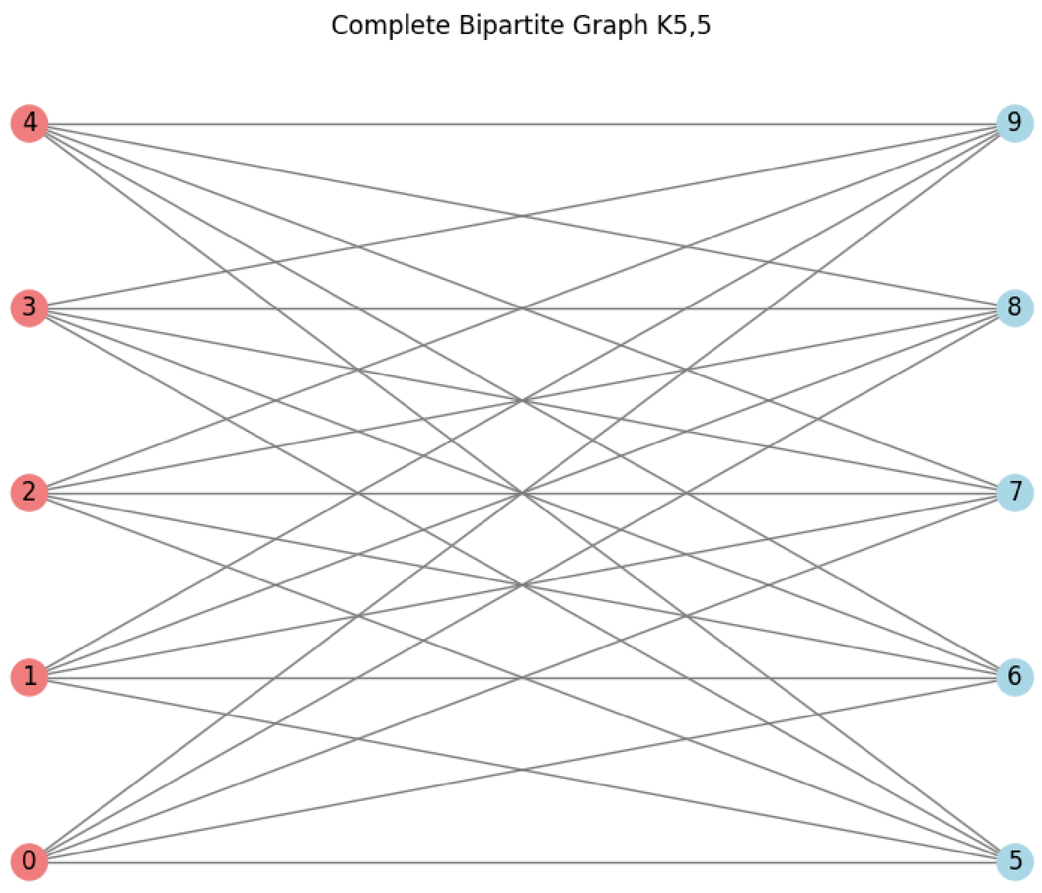
We've suggested a few graph ideas above.

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

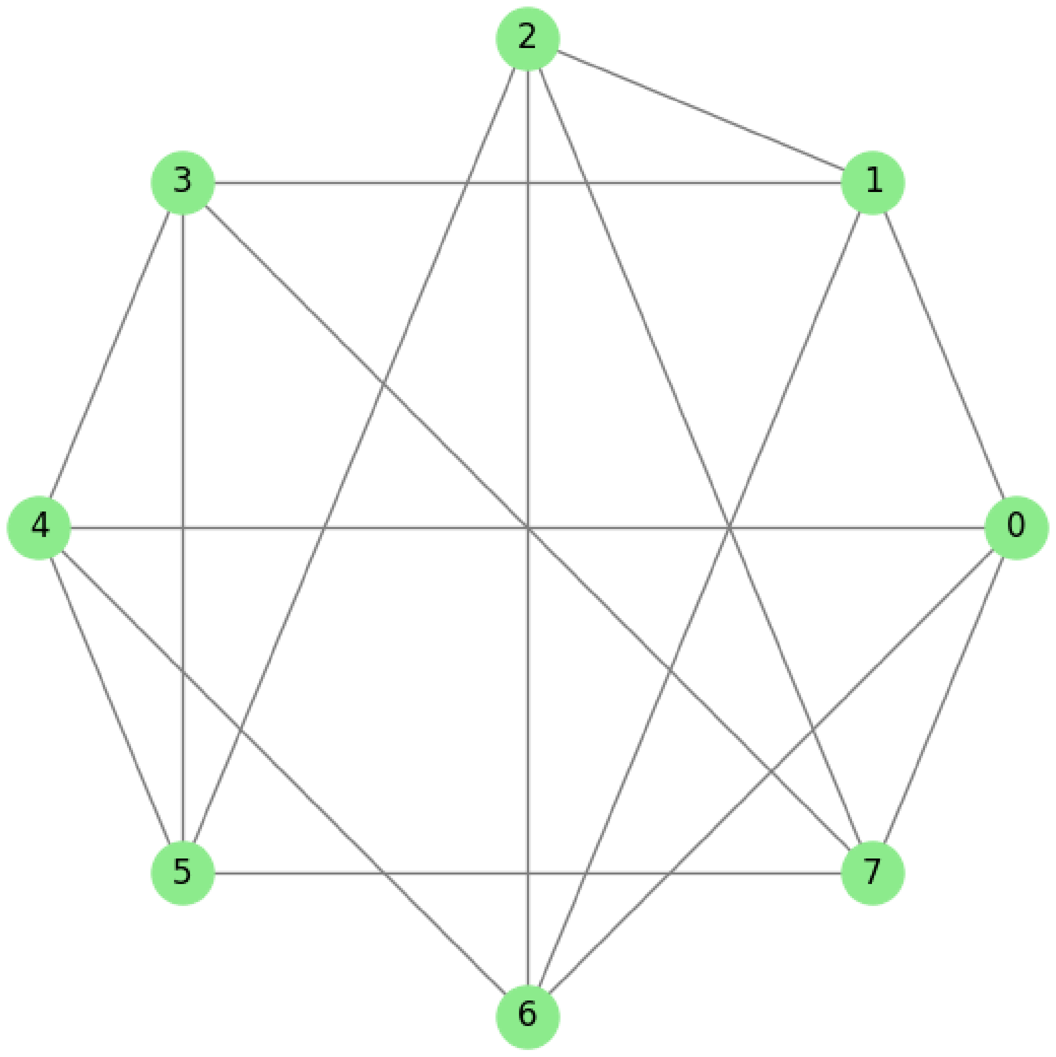
In [4]:
```
# Choose your favorite graph and build your winning ansatz!

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
0
1
2
3
4
5
6
7

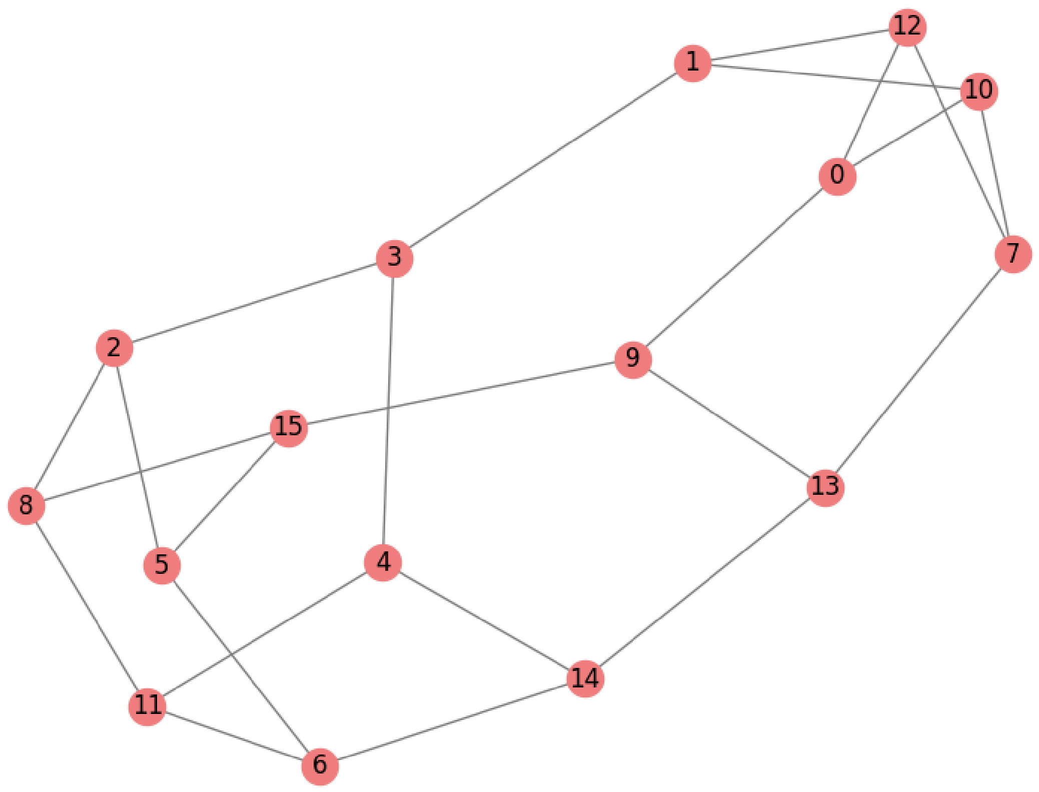

Complete Bipartite Graph K8,8
7
6
5
4
3
2
1
0
15
14
13
12
11
10
9
8


Complete Bipartite Graph K5,5
4
3
2
1
0
9
8
7
6
5

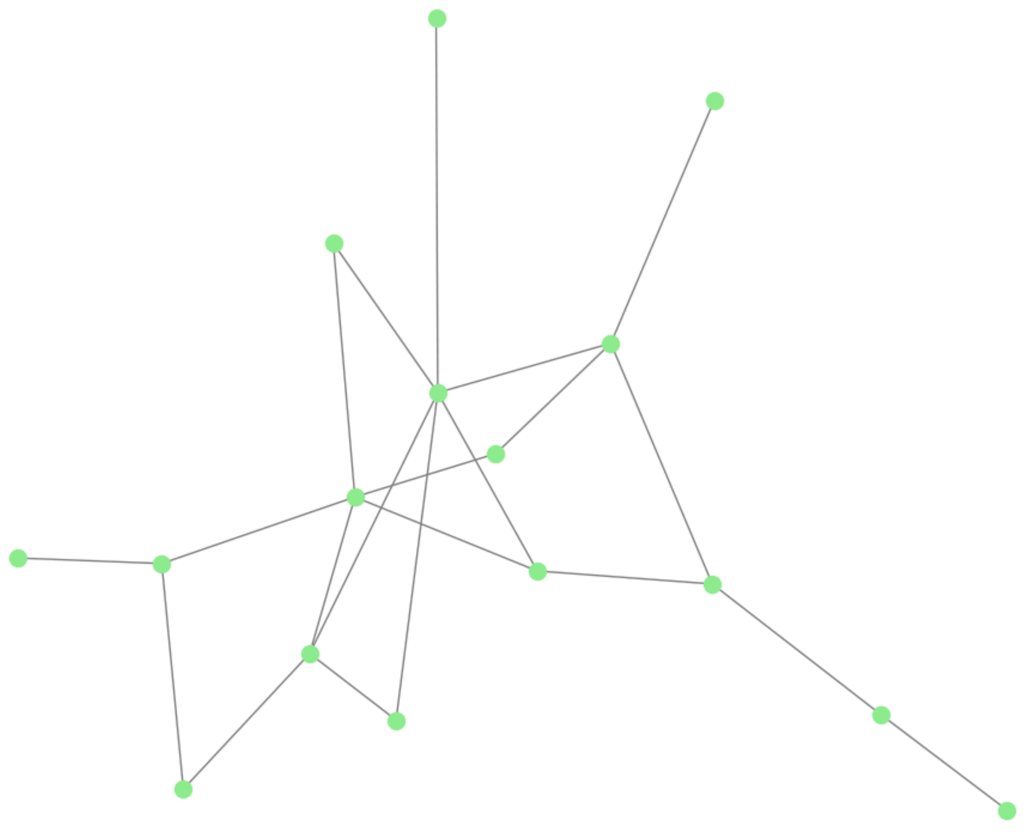
4-Regular Graph with 8 Vertices



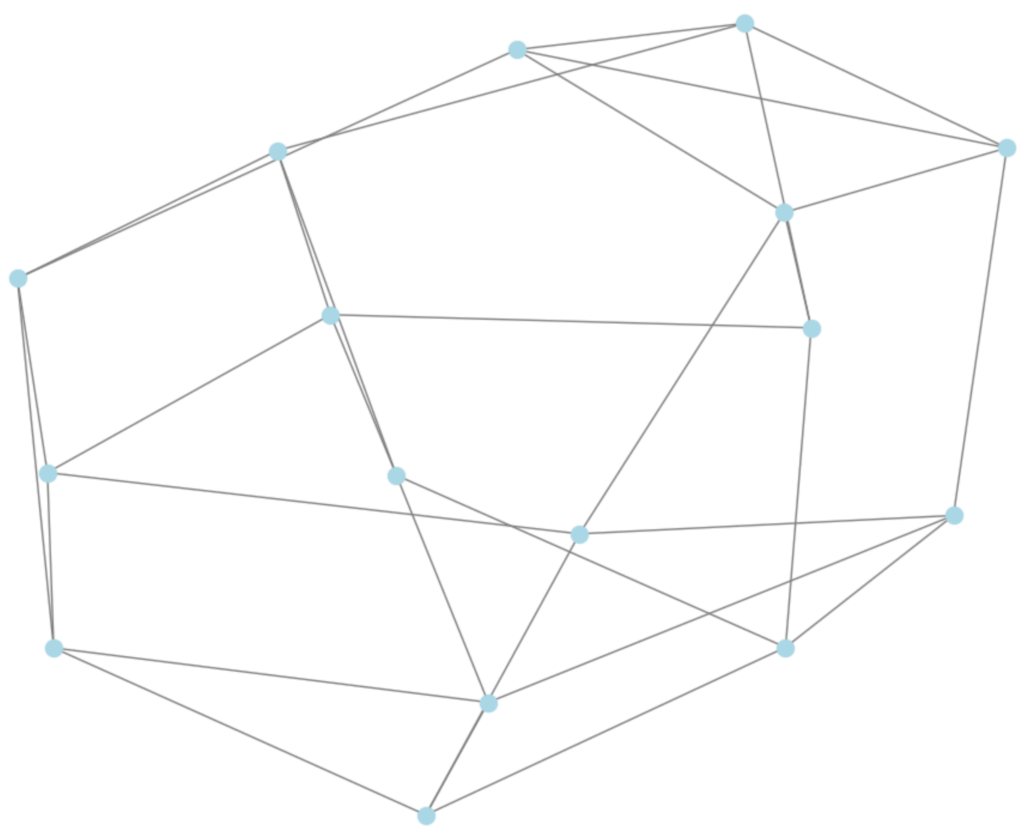
Cubic (3-Regular) Graph with 16 Vertices



Random Connected Graph with 16 Vertices



Expander Graph with 16 Vertices



## Input graph

In [128…

```
graph = graph1
graph
```

Out[128…

```
<networkx.classes.graph.Graph at 0x20809168b90>
```

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 [129…
```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
```

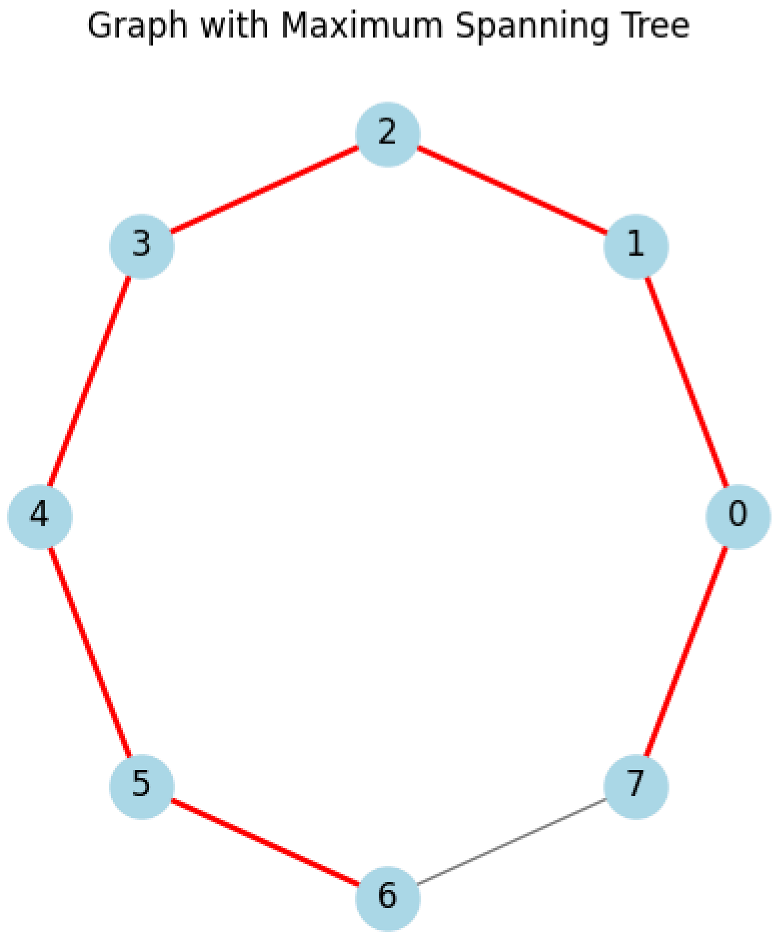
In [130…
```python
# alternate method to make the ansatz

# 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 [131…

```
# 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[131…

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



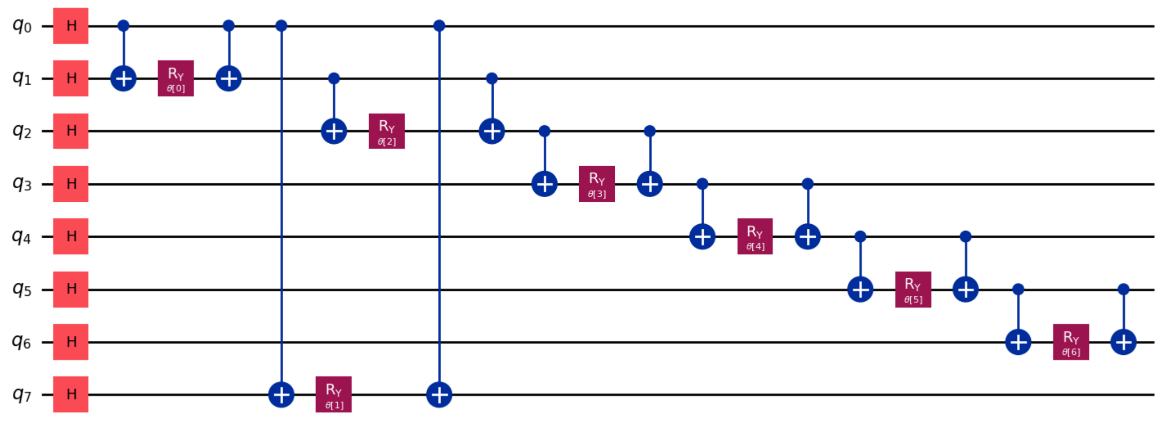
In [132…

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

In [133…

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

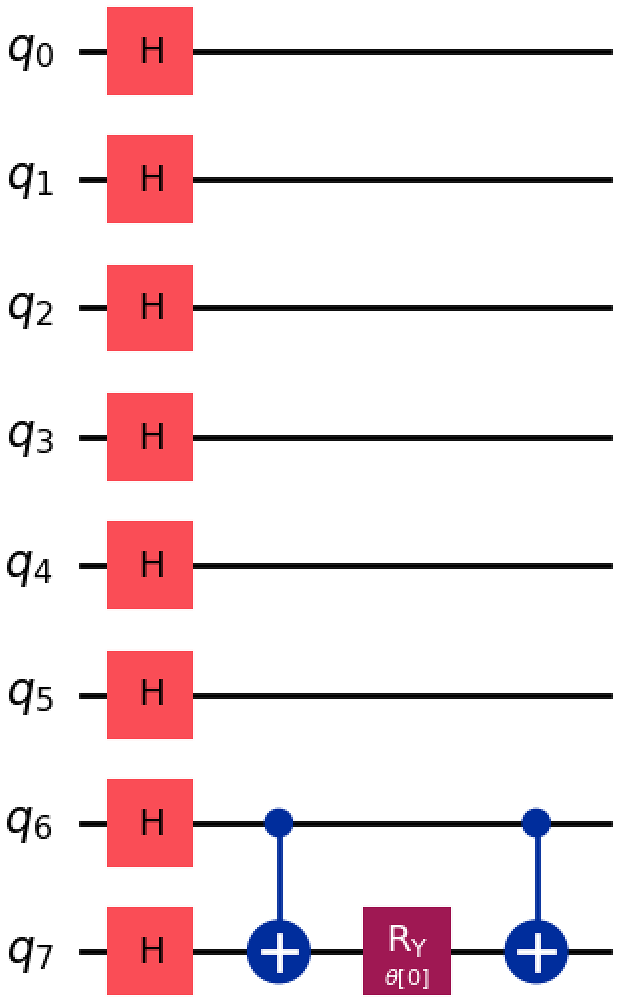
Out[133…



In [134…

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

Out[134…



## 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.

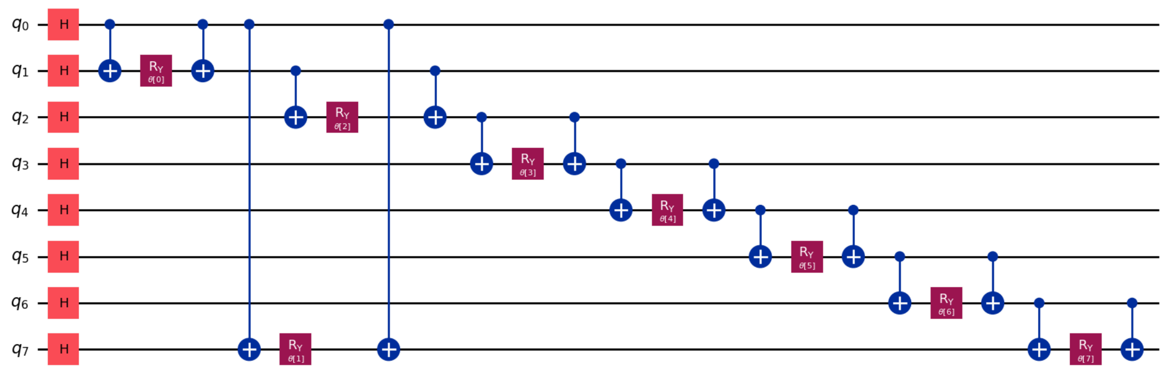
More on that topic: https://arxiv.org/abs/2207.06850

## Back to the problem in hand

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

In [135…
```
ansatz = build_ansatz(graph)
ansatz.draw("mpl", fold=-1)
```

Out[135…



## 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 [136…

```
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 [137…

```
# 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 [138…

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

```
SparsePauliOp(['IIIIIIII', 'ZZIIIIII', 'ZIIIIIIZ', 'IZZIIIII', 'IIZZIIII', 'IIIIZZII
I', 'IIIIIZZII', 'IIIIIIZZI'],
              coeffs=[-3.5+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(['IIIIIIII', 'IIIIIIZZ'],
              coeffs=[-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 [139…

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

Out[139…

```
SparsePauliOp(['IIIIIIII', 'ZZIIIIII', 'ZIIIIIIZ', 'IZZIIIII', 'IIZZIIII', 'IIIZZI
II', 'IIIIZZII', 'IIIIIZZI', 'IIIIIIZZ'],
              coeffs=[-4. +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 $j$th 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 [140…

```
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 [141…

```
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 [142…

```
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 [143…

```
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 [144…

```
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()
```

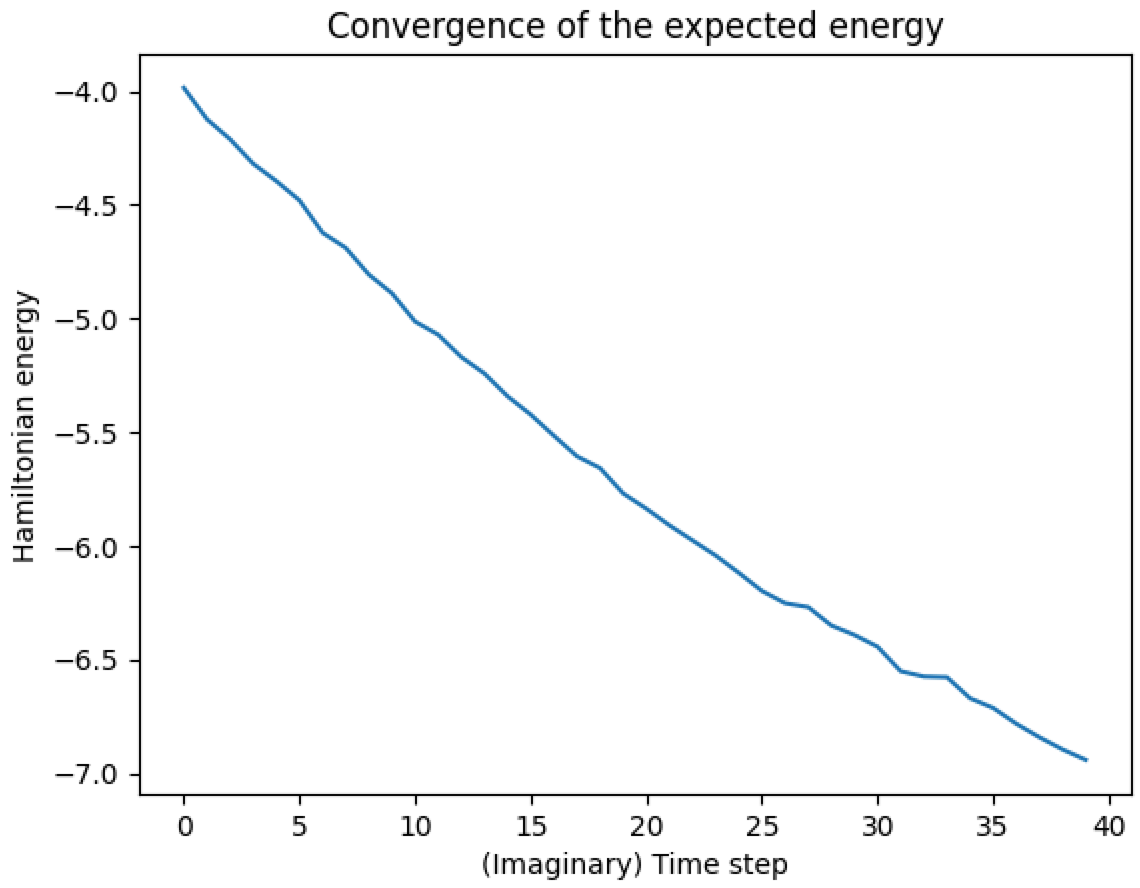
In [145…

```
%%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()
```

| step | curr_energy | num_circuits | quantum_exec_time |
|---|---|---|---|
| 0 | -3.9856 | 17 | 1.116726 |
| 1 | -4.1240 | 17 | 1.258594 |
| 2 | -4.2118 | 17 | 1.207713 |
| 3 | -4.3196 | 17 | 1.167748 |
| 4 | -4.3958 | 17 | 1.176317 |
| 5 | -4.4806 | 17 | 1.239019 |
| 6 | -4.6232 | 17 | 1.167920 |
| 7 | -4.6894 | 17 | 1.181148 |
| 8 | -4.8070 | 17 | 1.204362 |
| 9 | -4.8894 | 17 | 1.219521 |
| 10 | -5.0138 | 17 | 1.215552 |
| 11 | -5.0716 | 17 | 1.208433 |
| 12 | -5.1700 | 17 | 1.217103 |
| 13 | -5.2424 | 17 | 1.226270 |
| 14 | -5.3426 | 17 | 1.172891 |
| 15 | -5.4230 | 17 | 1.221435 |
| 16 | -5.5158 | 17 | 1.164816 |
| 17 | -5.6064 | 17 | 1.176880 |
| 18 | -5.6582 | 17 | 1.196762 |
| 19 | -5.7700 | 17 | 1.239296 |
| 20 | -5.8368 | 17 | 1.172674 |
| 21 | -5.9102 | 17 | 1.245610 |
| 22 | -5.9766 | 17 | 1.212670 |
| 23 | -6.0430 | 17 | 1.230855 |
| 24 | -6.1190 | 17 | 1.150891 |
| 25 | -6.1988 | 17 | 1.189471 |
| 26 | -6.2524 | 17 | 1.180200 |
| 27 | -6.2696 | 17 | 1.233273 |
| 28 | -6.3506 | 17 | 1.177387 |

| step | curr_energy | num_circuits | quantum_exec_time |
|---|---|---|---|
| 29 | -6.3924 | 17 | 1.230455 |
| 30 | -6.4440 | 17 | 1.223171 |
| 31 | -6.5518 | 17 | 1.213212 |
| 32 | -6.5750 | 17 | 1.195563 |
| 33 | -6.5780 | 17 | 1.178016 |
| 34 | -6.6708 | 17 | 1.188533 |
| 35 | -6.7134 | 17 | 1.151463 |
| 36 | -6.7830 | 17 | 1.148260 |
| 37 | -6.8424 | 17 | 1.173934 |
| 38 | -6.8958 | 17 | 1.185962 |

```
CPU times: total: 11min 16s
Wall time: 50 s
```



In [146…

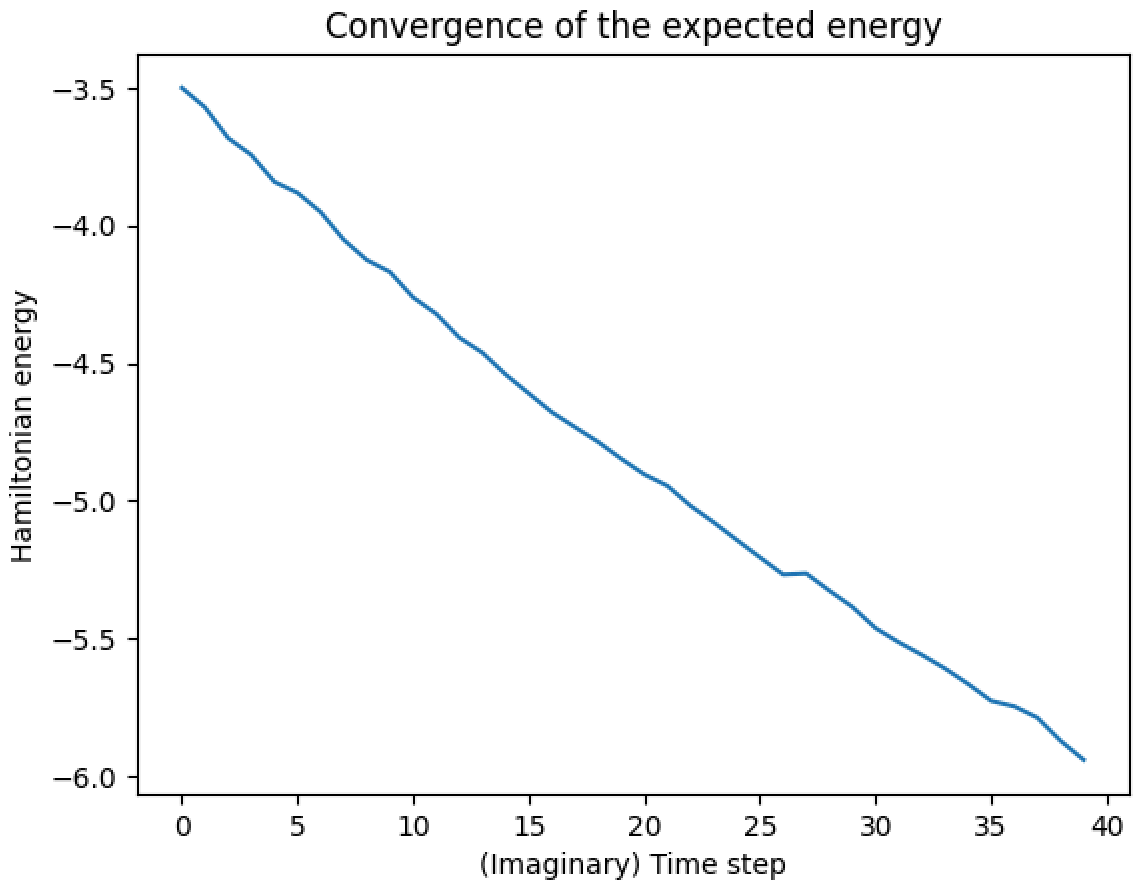
```
%%time

# 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** | -3.4999 | 15 | 0.917646 |
| **1** | -3.5706 | 15 | 1.081908 |
| **2** | -3.6825 | 15 | 1.024604 |
| **3** | -3.7432 | 15 | 1.061090 |
| **4** | -3.8414 | 15 | 1.047837 |
| **5** | -3.8812 | 15 | 1.089578 |
| **6** | -3.9509 | 15 | 1.055255 |
| **7** | -4.0519 | 15 | 1.023758 |
| **8** | -4.1255 | 15 | 1.007456 |
| **9** | -4.1687 | 15 | 1.059985 |
| **10** | -4.2612 | 15 | 1.065635 |
| **11** | -4.3209 | 15 | 1.068485 |
| **12** | -4.4074 | 15 | 1.022863 |
| **13** | -4.4622 | 15 | 1.049885 |
| **14** | -4.5411 | 15 | 1.068680 |
| **15** | -4.6098 | 15 | 1.042266 |
| **16** | -4.6783 | 15 | 1.053987 |
| **17** | -4.7326 | 15 | 1.035232 |
| **18** | -4.7859 | 15 | 1.052152 |
| **19** | -4.8475 | 15 | 1.057591 |
| **20** | -4.9042 | 15 | 1.072317 |
| **21** | -4.9457 | 15 | 1.065712 |
| **22** | -5.0186 | 15 | 1.067645 |
| **23** | -5.0783 | 15 | 1.043957 |
| **24** | -5.1425 | 15 | 1.064821 |
| **25** | -5.2055 | 15 | 1.028622 |
| **26** | -5.2677 | 15 | 1.075222 |
| **27** | -5.2639 | 15 | 1.029957 |
| **28** | -5.3265 | 15 | 1.068135 |

| step | curr_energy | num_circuits | quantum_exec_time |
|---|---|---|---|
| **29** | -5.3856 | 15 | 1.021477 |
| **30** | -5.4627 | 15 | 1.047461 |
| **31** | -5.5140 | 15 | 1.041347 |
| **32** | -5.5596 | 15 | 1.023687 |
| **33** | -5.6091 | 15 | 1.046060 |
| **34** | -5.6652 | 15 | 1.055617 |
| **35** | -5.7268 | 15 | 1.030224 |
| **36** | -5.7468 | 15 | 1.086784 |
| **37** | -5.7879 | 15 | 1.059183 |
| **38** | -5.8718 | 15 | 1.058912 |

```
CPU times: total: 10min 3s
Wall time: 43.9 s
```



In [147…

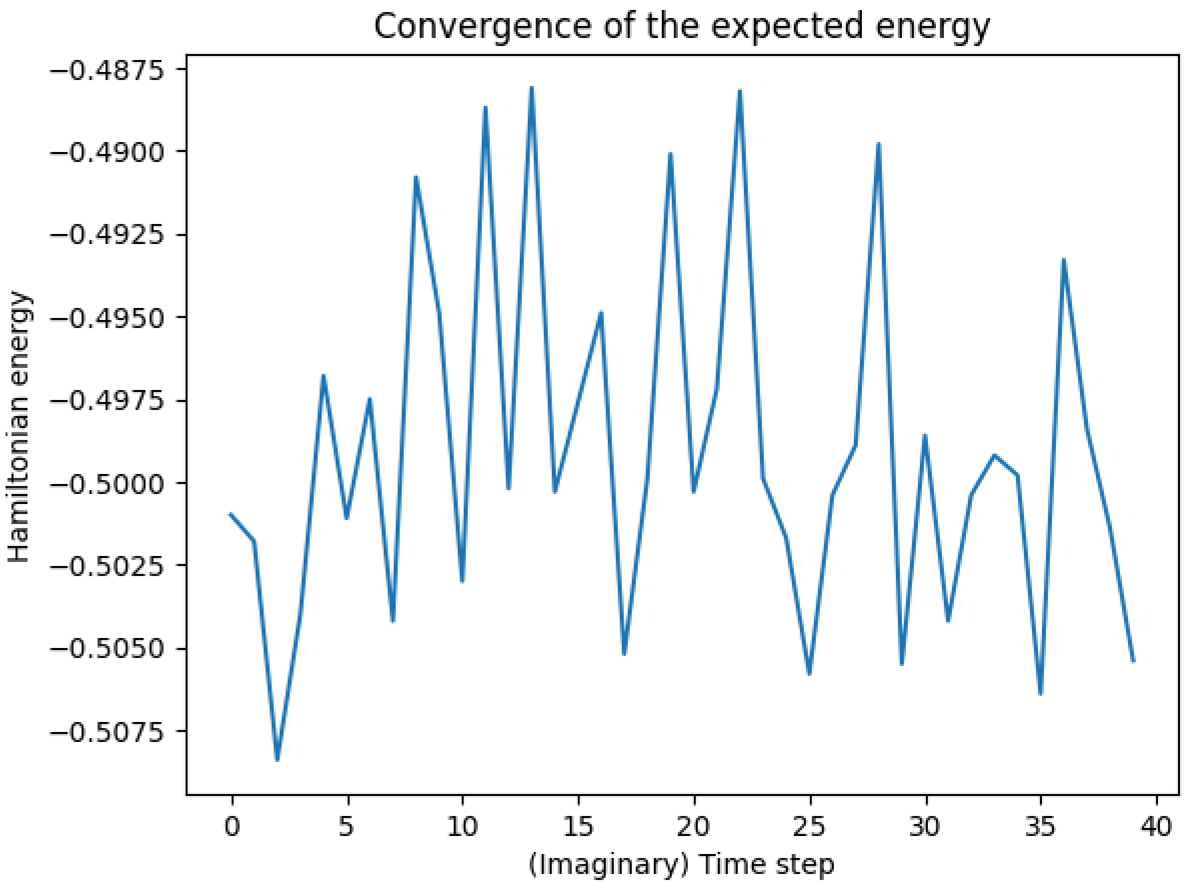
```
%%time

# 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** | -0.5010 | 3 | 0.131467 |
| **1** | -0.5018 | 3 | 0.179009 |
| **2** | -0.5084 | 3 | 0.214893 |
| **3** | -0.5040 | 3 | 0.225893 |
| **4** | -0.4968 | 3 | 0.216763 |
| **5** | -0.5011 | 3 | 0.200153 |
| **6** | -0.4975 | 3 | 0.216691 |
| **7** | -0.5042 | 3 | 0.226025 |
| **8** | -0.4908 | 3 | 0.215778 |
| **9** | -0.4949 | 3 | 0.228082 |
| **10** | -0.5030 | 3 | 0.238853 |
| **11** | -0.4887 | 3 | 0.214051 |
| **12** | -0.5002 | 3 | 0.224296 |
| **13** | -0.4881 | 3 | 0.221195 |
| **14** | -0.5003 | 3 | 0.202111 |
| **15** | -0.4976 | 3 | 0.209109 |
| **16** | -0.4949 | 3 | 0.192441 |
| **17** | -0.5052 | 3 | 0.212367 |
| **18** | -0.4999 | 3 | 0.197809 |
| **19** | -0.4901 | 3 | 0.217182 |
| **20** | -0.5003 | 3 | 0.216813 |
| **21** | -0.4972 | 3 | 0.201377 |
| **22** | -0.4882 | 3 | 0.201429 |
| **23** | -0.4999 | 3 | 0.229549 |
| **24** | -0.5017 | 3 | 0.216919 |
| **25** | -0.5058 | 3 | 0.214272 |
| **26** | -0.5004 | 3 | 0.215825 |
| **27** | -0.4989 | 3 | 0.217781 |
| **28** | -0.4898 | 3 | 0.225631 |

| | curr_energy | num_circuits | quantum_exec_time |
|---|---|---|---|
| **step** | | | |
| **29** | -0.5055 | 3 | 0.211634 |
| **30** | -0.4986 | 3 | 0.223167 |
| **31** | -0.5042 | 3 | 0.230766 |
| **32** | -0.5004 | 3 | 0.214459 |
| **33** | -0.4992 | 3 | 0.219517 |
| **34** | -0.4998 | 3 | 0.233523 |
| **35** | -0.5064 | 3 | 0.220221 |
| **36** | -0.4933 | 3 | 0.226609 |
| **37** | -0.4984 | 3 | 0.223676 |
| **38** | -0.5014 | 3 | 0.215396 |

```
CPU times: total: 2min 7s
Wall time: 9.26 s
```



## 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...

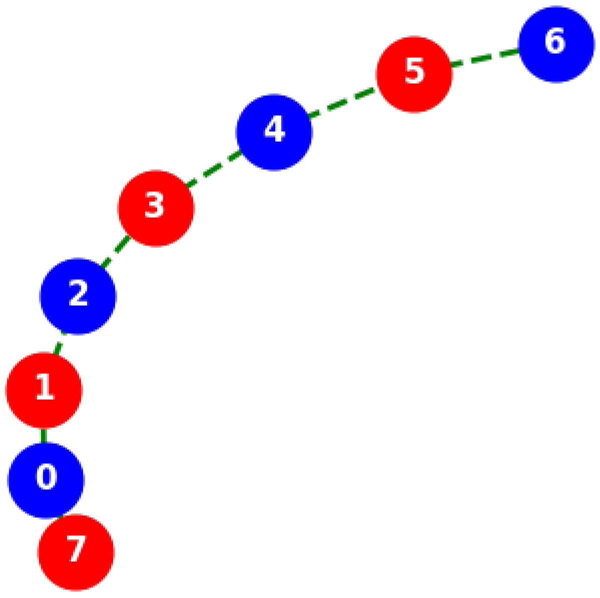
In [148...
```
from qiskit_aer import AerSimulator

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

In [159...
```
# 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), key=lambda t:
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)
```

```
{'11011101': 43, '00110101': 2855, '00101011': 1274, '01010101': 23976, '11010110':
2938, '00000001': 2, '10101000': 167, '00110110': 44, '00101010': 2904, '00101001':
2932, '11011010': 2632, '10001010': 370, '10111100': 2, '01010110': 371, '00111101':
37, '00100011': 17, '01001110': 6, '10010010': 408, '11010101': 2959, '00101000': 1
8, '10101010': 24277, '10110101': 326, '01101001': 358, '11001010': 2809, '0111011
1': 3, '01001011': 143, '10010101': 341, '11010010': 2994, '11101010': 3030, '111001
01': 44, '10011010': 323, '01100101': 330, '00111010': 27, '00100110': 42, '0111010
1': 380, '01011001': 337, '00010101': 3142, '00001010': 39, '10010100': 181, '001110
11': 10, '00100101': 2528, '00110011': 19, '00101101': 2891, '01010011': 138, '10110
010': 384, '10100110': 338, '01001001': 334, '10110110': 333, '11010001': 38, '01110
100': 4, '01001010': 327, '10010110': 420, '10101011': 135, '10110001': 2, '0101110
1': 346, '00100010': 42, '01110001': 5, '01001101': 356, '11010100': 1226, '0110110
1': 384, '10100100': 127, '01010010': 319, '00110100': 15, '00101100': 25, '0100010
1': 300, '11101001': 41, '01101010': 358, '00110010': 40, '00101110': 34, '0000010
1': 41, '10110111': 1, '01011011': 135, '10111101': 5, '01010001': 357, '10101100':
135, '10100010': 295, '01010100': 132, '10111010': 339, '10101101': 353, '10110100':
150, '01011010': 321, '00011011': 19, '00100111': 18, '10100101': 310, '10101001': 3
80, '11100100': 21, '00011101': 32, '10111011': 3, '01010111': 141, '11100010': 29,
'01101011': 161, '00010010': 41, '01101100': 2, '10100000': 1, '00001001': 42, '1010
1110': 394, '01101000': 2, '10010001': 5, '10100001': 7, '00001011': 17, '11110010':
58, '11001101': 51, '11100110': 51, '11000101': 57, '00010110': 42, '11001000': 17,
'00010001': 41, '11100001': 1, '11001100': 17, '11011110': 39, '01000001': 4, '01111
101': 7, '10110000': 1, '01011110': 5, '11011001': 34, '00110111': 15, '11101101': 4
9, '11000110': 35, '11100111': 1, '11001110': 29, '11101100': 21, '11101110': 31, '1
0011110': 4, '00001101': 42, '11111010': 48, '11001001': 49, '11000100': 14, '000101
00': 13, '10100011': 4, '11010000': 12, '11110101': 52, '00100100': 25, '11000000':
1, '11101011': 21, '00010111': 9, '11101000': 24, '10000101': 3, '11010111': 17, '00
011010': 36, '11110001': 1, '11110110': 49, '11011011': 16, '01010000': 1, '1011111
0': 5, '10011101': 3, '10001001': 5, '00111001': 48, '11001011': 16, '11111101': 2,
'11011100': 14, '00100001': 30, '00101111': 15, '00110001': 40, '00011001': 28, '110
00010': 35, '11010011': 27, '00010011': 15, '01001100': 1, '01110110': 6, '0110011
1': 2, '00001110': 1, '10011011': 2, '11110100': 20, '01011111': 1, '10110011': 3,
'10011001': 5, '01101110': 8, '10010111': 3, '10001000': 2, '10000100': 2, '1001000
0': 6, '01100010': 3, '10101111': 4, '01000110': 4, '01000010': 6, '10000010': 9, '1
1111110': 1, '01110010': 7, '10001101': 2, '01101111': 1, '01000100': 2, '01111010':
3, '10000110': 6, '11011000': 14, '01100110': 5, '01100001': 4, '00000110': 1, '0101
1000': 1, '10111001': 6, '01110011': 1, '01001111': 3, '01111001': 3, '10001011': 4,
'01000111': 2, '01111011': 2, '11111011': 1, '10011000': 2, '10001110': 5, '0110001
1': 3, '10010011': 2, '01001000': 2, '01100100': 1, '01000011': 3, '00011000': 1, '0
0001100': 1, '10011100': 1, '00000010': 1, '11111000': 1, '10100111': 1}
```

In [160…

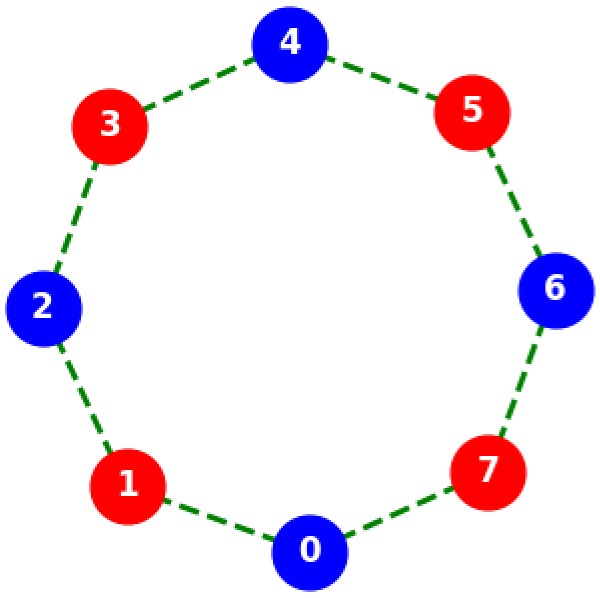
```
interpret_solution(T, best_bs_T)
print("Cut value: "+str(compute_cut_size(T, best_bs_T)))
print(T, best_bs_T)
```



```
Cut value: 7
Graph with 8 nodes and 7 edges 01010101
```

In [161…

```
# 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: 8
Graph with 8 nodes and 8 edges 01010101
```

In [162…

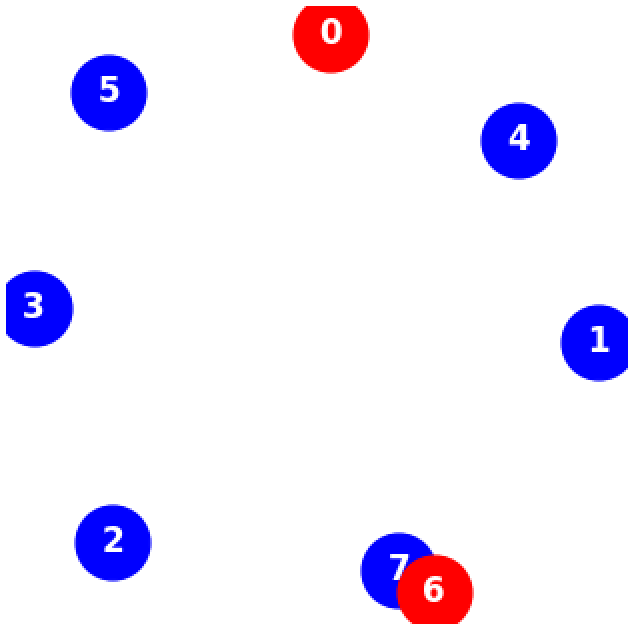
```
# Sample your optimized quantum state using Aer for the tree

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), key=lambda t:
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)
```

```
{'00111010': 502, '00100110': 500, '11000001': 486, '11101100': 527, '11010000': 51
9, '01001010': 264, '01110100': 275, '00100111': 538, '11100010': 520, '01111101': 3
03, '01000001': 266, '11011010': 513, '00110000': 503, '10010000': 267, '00001010':
516, '00111111': 500, '00100001': 445, '00110110': 540, '00101010': 507, '10101000':
257, '00000001': 513, '10001010': 284, '10111100': 228, '01010110': 252, '01100111':
263, '00110111': 548, '00000000': 536, '10101001': 257, '11011100': 544, '00110010':
488, '00101110': 510, '11011001': 512, '11001000': 481, '11000101': 470, '11010100':
528, '10010011': 284, '00011010': 525, '11011011': 507, '11001100': 491, '11100001':
499, '00000100': 514, '00111011': 503, '00100101': 486, '00101011': 526, '00110101':
498, '10011100': 257, '01111010': 307, '01000100': 292, '11011101': 514, '00000010':
502, '10000011': 288, '11100110': 481, '11001101': 515, '00000011': 508, '11011110':
494, '00011000': 511, '11010110': 535, '11010010': 470, '10000101': 275, '11111111':
517, '00010100': 482, '00101111': 512, '00110001': 501, '00100011': 528, '00111101':
494, '11000100': 472, '11101111': 538, '10101100': 295, '11110110': 544, '00010001':
536, '01101101': 258, '00011101': 541, '11110010': 542, '10110011': 274, '01011111':
291, '01011110': 269, '10110000': 279, '11011000': 506, '11111001': 491, '10111111':
275, '01010011': 279, '10110101': 239, '11111110': 490, '10001111': 281, '11000000':
541, '11101011': 464, '10100001': 271, '00001000': 485, '00100100': 509, '00111100':
545, '11111100': 517, '01010101': 271, '01111111': 269, '01000011': 313, '10101111':
282, '11001110': 477, '11100111': 520, '00010111': 528, '11010001': 511, '10100000':
299, '00001001': 541, '11001011': 496, '11100000': 505, '01100101': 276, '00010110':
525, '10110111': 244, '01011011': 285, '11110111': 496, '11001001': 494, '11110001':
513, '01100100': 270, '11000111': 477, '11101110': 516, '01101000': 260, '10000010':
289, '10111011': 270, '01010111': 265, '01011100': 280, '10110110': 282, '01001111':
260, '01110011': 273, '00100010': 482, '00111110': 505, '00110011': 506, '00101101':
532, '11001111': 508, '11100100': 503, '01110111': 263, '01001011': 277, '00001110':
479, '00010000': 532, '01000111': 259, '01111011': 291, '11101001': 544, '00000101':
514, '11011111': 499, '10000110': 279, '10011001': 261, '10010101': 258, '10100010':
291, '10010111': 267, '00011001': 540, '11010111': 512, '10101011': 254, '10111000':
267, '01001110': 236, '01110000': 291, '10010100': 267, '01101011': 264, '00011100':
511, '00111000': 533, '10001110': 274, '10011110': 246, '01110001': 258, '01001101':
290, '00001111': 446, '01111100': 277, '01000110': 283, '00010010': 549, '11101000':
521, '11110100': 485, '01101001': 278, '01100001': 276, '10000111': 276, '11110011':
493, '11111000': 536, '11110000': 497, '01011001': 265, '10000001': 274, '10001000':
292, '01011010': 300, '10110100': 268, '00001011': 511, '11010011': 512, '01000101':
257, '00101001': 486, '01110010': 262, '01101110': 273, '10010010': 264, '01100011':
260, '00101000': 541, '11111101': 518, '10101110': 274, '00100000': 525, '01011000':
288, '01111001': 248, '10110010': 286, '11110101': 549, '00011011': 556, '11000110':
523, '11101101': 491, '01111000': 257, '10100101': 257, '10100111': 282, '01010100':
266, '10111010': 287, '01010000': 283, '10111110': 291, '10001101': 271, '10101101':
285, '11010101': 530, '10011011': 269, '10100110': 289, '00000110': 517, '10011000':
263, '10001011': 277, '11111010': 465, '00001100': 504, '00010011': 560, '11111011':
505, '10111101': 278, '01010001': 281, '11000010': 499, '11100101': 503, '01111110':
271, '01000000': 260, '10101010': 252, '10110001': 262, '01011101': 265, '00000111':
537, '10000000': 277, '00011110': 462, '01101111': 280, '10111001': 269, '01001100':
242, '01110110': 285, '01100110': 282, '00001101': 488, '10001001': 296, '00011111':
489, '00101100': 526, '00110100': 472, '00010101': 498, '11100011': 495, '11001010':
520, '01101100': 277, '10011010': 250, '10011101': 289, '01010010': 303, '01100000':
278, '10010001': 272, '11000011': 549, '11101010': 474, '10100100': 247, '01000010':
227, '00111001': 494, '10000100': 259, '10010110': 277, '10100011': 268, '01001001':
284, '01110101': 241, '10001100': 263, '10011111': 264, '01101010': 276, '01001000':
272, '01100010': 259}
```

In [163…

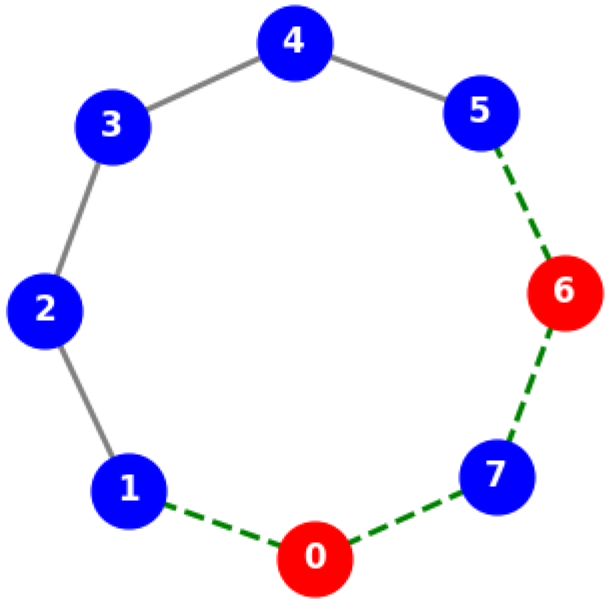
```python
interpret_solution(H, best_bs_H)
print("Cut value: "+str(compute_cut_size(H, best_bs_H)))
print(H, best_bs_H)
```



```
Cut value: 1
Graph with 8 nodes and 1 edges 10000010
```

In [164…

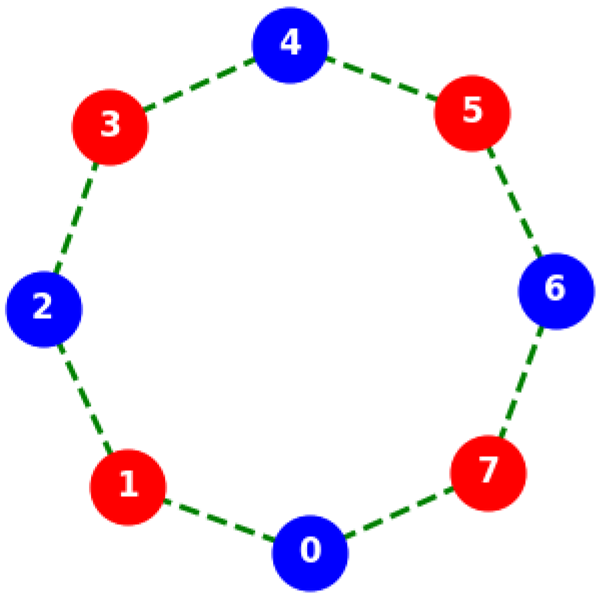
```python
interpret_solution(graph, best_bs_H)
print("Cut value: "+str(compute_cut_size(graph, best_bs_H)))
print(graph, best_bs_H)
```



```
Cut value: 4
Graph with 8 nodes and 8 edges 10000010
```

In [165…

```python
# choose the better result from best_bs_T and best_bs_H
best_bs_final = best_bs_H if compute_cut_size(H, best_bs_H) > compute_cut_size(T, b
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(H, best_bs_H) > compute_cut_size(T, bes
interpret_solution(graph, best_bs_final)
print("Cut value: "+str(final_cut_value))
print(graph, best_bs_final)
```



```
Cut value: 8
Graph with 8 nodes and 8 edges 01010101
```

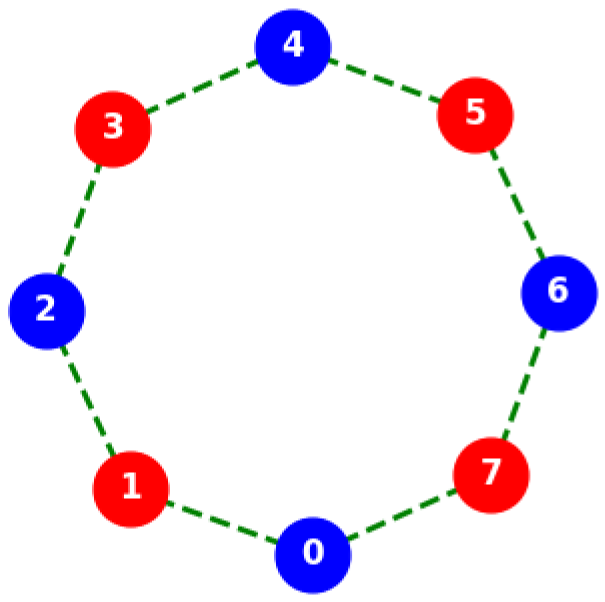
In [166…

```
# 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)
```

```
{'01011100': 19, '10110110': 337, '10011001': 25, '10101010': 25299, '01010101': 255
53, '11010010': 1549, '10010101': 1515, '11101010': 1455, '01110111': 5, '01001011':
384, '00010101': 1563, '01010100': 1521, '10111010': 320, '00101001': 1458, '1100111
0': 23, '00101011': 1490, '00110101': 1475, '10111100': 12, '01010110': 1516, '01010
010': 1517, '00101101': 1580, '00110011': 26, '01011010': 1567, '10110100': 400, '01
101010': 1450, '10101101': 1553, '01110001': 8, '01001101': 345, '11010100': 1572,
'10101100': 371, '10100100': 377, '11010110': 1510, '11001000': 20, '10101111': 18,
'10100010': 359, '01101101': 370, '10101011': 1568, '11011010': 1495, '10010100': 34
7, '00111011': 26, '00100101': 1485, '01010001': 371, '10111101': 15, '10010110': 35
3, '01110100': 23, '01001010': 1516, '10100001': 20, '01101011': 379, '01001001': 34
0, '10100110': 389, '11001010': 1525, '10011010': 383, '10101001': 1507, '01101001':
363, '10110101': 1515, '10101110': 353, '10100101': 1484, '11010000': 27, '0111010
1': 398, '10111111': 1, '01010011': 369, '10110001': 21, '01011101': 350, '1011011
1': 20, '01011011': 365, '10110010': 353, '01100101': 343, '10101000': 372, '0101011
1': 343, '10111011': 22, '10010010': 374, '01000101': 345, '10011011': 19, '0101100
1': 355, '11101100': 21, '10001010': 360, '01110110': 24, '01001100': 26, '0010101
0': 20, '11101110': 21, '00001011': 16, '00010111': 23, '10100011': 20, '10010111':
27, '10001011': 21, '10100111': 21, '00000101': 20, '00001101': 25, '11111010': 22,
'01111001': 3, '00100111': 24, '10110011': 18, '01011111': 5, '00111001': 17, '01001
000': 22, '00111101': 16, '00100011': 19, '10110000': 3, '01011110': 22, '11100100':
16, '10010001': 31, '10011100': 7, '01101100': 17, '01111011': 2, '01000111': 8, '01
101111': 6, '10001001': 22, '01100011': 5, '00010011': 36, '11010101': 20, '0100111
0': 24, '01100010': 23, '11001100': 13, '11110010': 21, '11000100': 18, '01100110':
16, '00110111': 17, '01000110': 17, '01111100': 1, '11101101': 2, '11000110': 21, '1
1110110': 21, '01101110': 26, '10011000': 6, '01001111': 5, '01110011': 4, '1011100
1': 23, '10001110': 8, '10011101': 17, '01100001': 7, '11011000': 21, '10010011': 2
9, '11101000': 18, '01010000': 18, '10111110': 5, '00111111': 1, '00100001': 27, '00
011001': 17, '11000010': 20, '10111000': 9, '11011110': 19, '11100010': 14, '1000110
0': 2, '10001101': 26, '01000010': 30, '01111010': 16, '01000100': 24, '11110100': 2
1, '00110001': 21, '00101111': 14, '00100100': 1, '01011000': 24, '00011101': 15, '0
1000001': 7, '01111101': 3, '10100000': 5, '00001001': 28, '01110010': 25, '1000010
1': 16, '01000011': 3, '11011100': 24, '01101000': 18, '00010001': 21, '11111100':
2, '01100100': 14, '10011110': 5, '10000110': 3, '01111110': 1, '11001101': 1, '1110
0110': 12, '10000010': 5, '00011011': 21, '01100111': 3, '10001000': 2, '10010000':
3, '00010100': 1, '11010111': 1, '01100000': 1, '11100101': 1, '10000100': 2, '11101
011': 1, '11001011': 1, '00111010': 1, '11111000': 2, '11010011': 1, '10011111': 1,
'00010010': 1}
```

In [167…

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



```
Cut value: 8
Graph with 8 nodes and 8 edges 01010101
```

## Drumroll please... the scores!

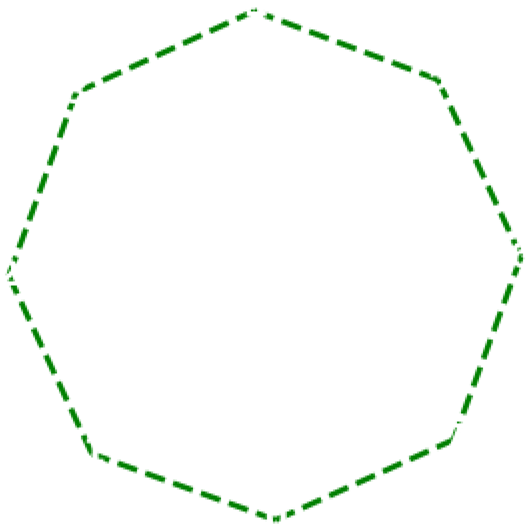
In [168…

```
%%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)
```

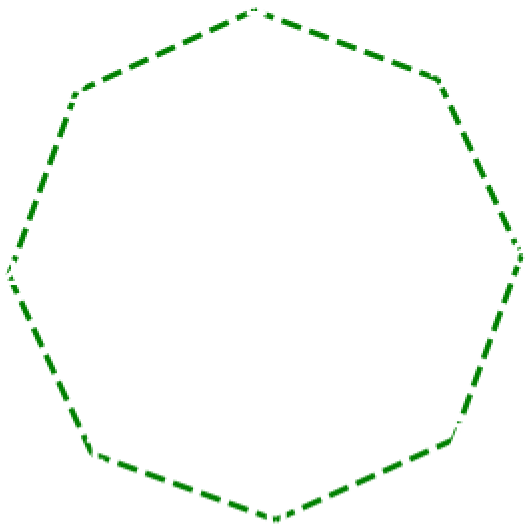
```
CPU times: total: 15.6 ms
Wall time: 59.7 ms
```

In [169…

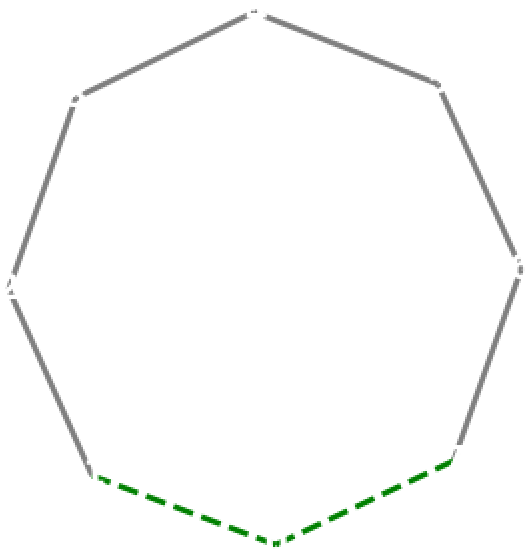
```
# This is classical brute force solver results:
interpret_solution(graph, xbest_brute)
print(graph, xbest_brute)
print("\nBest solution = " + str(xbest_brute) + " cost = " + str(best_cost_brute))
print(XS_brut)

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 with 8 nodes and 8 edges [1, 0, 1, 0, 1, 0, 1, 0]

Best solution = [1, 0, 1, 0, 1, 0, 1, 0] cost = 8.0
['10101010', '01010101']
```



```
Graph with 8 nodes and 8 edges [1, 0, 1, 0, 1, 0, 1, 0]

Best balanced = [1, 0, 1, 0, 1, 0, 1, 0] cost = 8.0
['10101010', '01010101']
```



```
Graph with 8 nodes and 8 edges [1, 0, 0, 0, 0, 0, 0, 0]

Best connected = [1, 0, 0, 0, 0, 0, 0, 0] cost = 2.0
['10000000', '01000000', '11000000', '00100000', '01100000', '11100000', '00010000',
'00110000', '01110000', '11110000', '00001000', '00011000', '00111000', '01111000',
'11111000', '00000100', '00001100', '00011100', '00111100', '01111100', '11111100',
'00000010', '00000110', '00001110', '00011110', '00111110', '01111110', '11111110',
'00000001', '10000001', '11000001', '11100001', '11110001', '11111001', '11111101',
'00000011', '10000011', '11000011', '11100011', '11110011', '11111011', '00000111',
'10000111', '11000111', '11100111', '11110111', '00001111', '10001111', '11001111',
'11101111', '00011111', '10011111', '11011111', '00111111', '10111111', '01111111']
```

In [170…

```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: 50852 out of 100000
Balanced max-cut: 50852 out of 100000
Connected max-cut: 10 out of 100000
```

In [171…

```
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 [172…

```
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: 0.40682
Balanced score: 0.40682
Connected score: 8e-05
```

In [173…

```
# 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: 0.49311
Balanced score: 0.49311
Connected score: 0.0001
```

In [174…

```
# 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: 0.41725
Balanced score: 0.41725
Connected score: 8e-05
```

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 [ ]:
```