# Date:

**TASK 10: Implement the QAOA algorithm**

**Aim:** To implement the Quantum Approximate Optimization Algorithm (QAOA) using Qiskit and PyTorch to solve the Max-Cut problem, a classical NP-hard problem.

# Mathematical Model of the QAOA Algorithm

* 1. **Max-Cut Problem Formulation**

Given a graph 𝐺 = (𝑉, 𝐸) with weighted adjacency matrix 𝑤ij , the Max-Cut objective is

𝐶(𝑥) = ∑(i,j)𝜖E 𝑤ij . (𝑥i 𝑥j)

where 𝑥i*ϵ*{0,1} indicates the partition of node 𝑖.

The goal is max 𝐶(𝑥)

x*ϵ*{0,1}n

# QAOA Ansatz

The QAOA prepares a quantum state parameterized by angles γ→, 𝛽→.

p

|ψ(γ→, 𝛽→)⟩ = G 𝑒–iβl ∑i Xi 𝑒–iγlHC|+⟩n

l=1

* + - 𝐻C = ∑(i,j)𝜖E 𝑤ij𝑍i𝑍j (cost Hamiltonian)
    - 𝐻B = ∑i 𝑋i (mixer Hamiltonian)

# Expectation Value

The objective is to maximize𝐹(γ→, 𝛽→) = (ψ(γ→, 𝛽→) |HC|ψ(γ→, 𝛽→)⟩, the optimization is performed using a classical optimizer (Adam with finite-difference gradients).

# Algorithm - QAOA Algorithm

1. **Graph Construction**
   * Define adjacency matrix W.
   * Build a NetworkX graph for visualization.

# Classical Baseline

* + Use brute-force enumeration to compute the optimal Max-Cut value (ground truth).

# QAOA Circuit Construction

* + Initialize qubits in |+⟩ .
  + Apply alternating cost and mixer unitaries for depth 𝑝.
  + Use controlled-𝑍 rotation gates to implement 𝑍i𝑍j interactions.

# Expectation Calculation

* + Simulate circuit using **Qiskit Aer statevector simulator**.
  + Compute expected cut value from measurement probabilities.

# Hybrid Optimization

* + Parameters (γ→, 𝛽→) initialized randomly.
  + Compute finite-difference gradients of expectation.
  + Update parameters using PyTorch **Adam optimizer**.

# Circuit Visualization

* + Draw initial and optimized QAOA circuits using qiskit.visualization.

# Program

#!pip install qiskit qiskit-optimization torch networkx numpy

#!pip install qiskit-aer

#!pip install pylatexenc

import os

import numpy as np

import networkx as nx

import torch

from qiskit import QuantumCircuit

from qiskit\_aer import Aer

from qiskit.quantum\_info import Statevector

from qiskit\_optimization.applications import Maxcut

from qiskit\_optimization.problems import QuadraticProgram

# Visualization imports

import matplotlib

# Use Agg backend in headless environments so saving works even

# without GUI

matplotlib.use(os.environ.get("MPLBACKEND", "Agg"))

import matplotlib.pyplot as plt

# -------------------------

# Problem definition

# -------------------------

def make\_graph():

    # Example: 4-node graph (same as Qiskit tutorial)

    w = np.array([

        [0.0, 1.0, 1.0, 0.0],

        [1.0, 0.0, 1.0, 1.0],

        [1.0, 1.0, 0.0, 1.0],

        [0.0, 1.0, 1.0, 0.0]

    ])

    G = nx.from\_numpy\_array(w)

    return G, w

# computes classical objective (cut value) for bitstring x

# (array of 0/1)

def objective\_value(x, w):

    X = np.outer(x, (1 - x))

    w\_01 = np.where(w != 0, 1, 0)

    return np.sum(w\_01 \* X)

# brute-force best solution (for comparison)

def brute\_force\_maxcut(w):

    n = w.shape[0]

    best = -1

    best\_x = None

    for i in range(2\*\*n):

        x = np.array(list(map(int, np.binary\_repr(i, width=n))))

        val = objective\_value(x, w)

        if val > best:

            best = val

            best\_x = x

    return best\_x, best

# -------------------------

# Build QAOA circuit (manual)

# -------------------------

def qaoa\_circuit(n\_qubits, edges, gammas, betas):

    """

    Build QAOA circuit:

      - start in |+>^n

      - for each layer l:

           cost unitary U\_C(gamma\_l) = exp(-i \* gamma\_l \* C)

           mixer U\_B(beta\_l) = product Rx(2\*beta\_l)

    edges: list of tuples (i, j, weight)

    gammas, betas: lists or 1D arrays (length p)

    """

    p = len(gammas)

    qc = QuantumCircuit(n\_qubits)

    # initial layer: Hadamards to create |+>^n

    qc.h(range(n\_qubits))

    for layer in range(p):

        gamma = float(gammas[layer])

        # cost layer: implement exp(-i \* gamma \* w\_ij \* Z\_i Z\_j)

        for (i, j, w) in edges:

            if w == 0:

                continue

            # For ZZ interaction exp(-i \* theta/2 \* Z\_i Z\_j) ->

            # use CNOT-RZ-CNOT with theta = 2\*gamma\*w

            theta = 2.0 \* gamma \* w

            qc.cx(i, j)

            qc.rz(theta, j)

            qc.cx(i, j)

        # mixer layer: RX(2\*beta)

        beta = float(betas[layer])

        for q in range(n\_qubits):

            qc.rx(2.0 \* beta, q)

    return qc

# -------------------------

# Expectation value from statevector

# -------------------------

def expectation\_from\_statevector(statevector, w):

    """Given a statevector and adjacency matrix w, compute

expected MaxCut objective."""

    n = w.shape[0]

    probs = Statevector(statevector).probabilities\_dict()

    exp\_val = 0.0

    for bitstr, p in probs.items():

        # reverse so index 0 => qubit 0

        bits = np.array([int(b) for b in bitstr[::-1]])

        exp\_val += objective\_value(bits, w) \* p

    return exp\_val

# -------------------------

# QAOA + PyTorch classical loop

# -------------------------

def run\_qaoa\_with\_pytorch(w, p=1, init\_std=0.5, maxiter=100,

lr=0.1, finite\_diff\_eps=1e-3,

backend\_name="aer\_simulator\_statevector"):

    n = w.shape[0]

    # edges list with weights (i>j to match earlier convention)

    edges = [(i, j, w[i, j]) for i in range(n) for j in range(i)

if w[i, j] != 0]

    # initial params (gamma\_1..gamma\_p, beta\_1..beta\_p)

    params = torch.randn(2 \* p, dtype=torch.double) \* init\_std

    params.requires\_grad = False  # we will supply grads

    # manually using finite differences

    optimizer = torch.optim.Adam([params], lr=lr)

    backend = Aer.get\_backend(backend\_name)

    best = {"val": -np.inf, "params": None, "bitstring": None}

    for it in range(maxiter):

        # unpack

        gammas = params.detach().numpy()[:p]

        betas = params.detach().numpy()[p:]

        # build circuit, get statevector

        qc = qaoa\_circuit(n, edges, gammas, betas)

        qc.save\_statevector()

        # using Aer simulator

        res = backend.run(qc).result()

        sv = res.get\_statevector(qc)

        # compute expectation (we maximize expected cut)

        exp\_val = expectation\_from\_statevector(sv, w)

        loss = -float(exp\_val)  # minimize negative of

        # expectation

        # keep best

        if exp\_val > best["val"]:

            # extract most likely bitstring

            probs = Statevector(sv).probabilities\_dict()

            most = max(probs.items(), key=lambda kv: kv[1])[0]

            bits = np.array([int(b) for b in most[::-1]])

            best.update({"val": exp\_val, "params":

params.detach().clone(), "bitstring": bits})

        # finite-difference gradient (central difference)

        grads = np.zeros\_like(params.detach().numpy())

        base = params.detach().numpy()

        eps = finite\_diff\_eps

        for k in range(len(base)):

            plus = base.copy()

            minus = base.copy()

            plus[k] += eps

            minus[k] -= eps

            g\_plus = \_qaoa\_expectation\_with\_params(plus, n,

edges, backend, w, p)

            g\_minus = \_qaoa\_expectation\_with\_params(minus, n,

edges, backend, w, p)

            grad\_k = (-(g\_plus - g\_minus) / (2 \* eps))  #

            # derivative of loss = -expectation

            grads[k] = grad\_k

        # set grads into params manually and step optimizer

        params\_grad = \

torch.from\_numpy(grads).to(dtype=torch.double)

        params.grad = params\_grad

        optimizer.step()

        optimizer.zero\_grad()

        if it % 10 == 0 or it == maxiter - 1:

            print(f"Iter {it:03d}: expected cut = {exp\_val:.6f}, loss = {loss:.6f}")

    return best

def \_qaoa\_expectation\_with\_params(flat\_params, n, edges,

backend, w, p):

    """Helper to evaluate expected cut quickly for given params

(no PyTorch)"""

    gammas = flat\_params[:p]

    betas = flat\_params[p:]

    qc = qaoa\_circuit(n, edges, gammas, betas)

    qc.save\_statevector()

    res = backend.run(qc).result()

    sv = res.get\_statevector(qc)

    exp\_val = expectation\_from\_statevector(sv, w)

    return exp\_val

# -------------------------

# Circuit display helpers

# -------------------------

def show\_circuit(qc: QuantumCircuit, filename: str = None,

style: str = "mpl"):

    print("\n--- Quantum Circuit ---")

    try:

        print(qc.draw(output="text"))

    except Exception as e:

        print("Failed to draw Quantum Circuit:", e)

    if style == "mpl":

        try:

            fig = qc.draw(output="mpl", interactive=False)

            fig.tight\_layout()

            if filename:

                fig.savefig(filename, dpi=200,

bbox\_inches="tight")

                print(f"[Saved circuit figure to {filename}]")

            else:

                # if no filename provided, still save to a

                # temporary PNG and show inline if possible

                tempname = "qaoa\_circuit.png"

                fig.savefig(tempname, dpi=200,

bbox\_inches="tight")

                print(f"[Saved circuit figure to {tempname}]")

            plt.close(fig)

        except Exception as e:

            print("Matplotlib drawing failed:", str(e))

            print("Fallback: Quantum Circuit diagram above.")

def demo\_display\_initial\_circuit(w, p=1,

filename="qaoa\_initial\_circuit.png"):

    n = w.shape[0]

    # random params for demo

    gammas = np.random.randn(p) \* 0.8

    betas = np.random.randn(p) \* 0.8

    edges = [(i, j, w[i, j]) for i in range(n) for j in range(i)

if w[i, j] != 0]

    qc = qaoa\_circuit(n, edges, gammas, betas)

    show\_circuit(qc, filename=filename, style="mpl")

def demo\_display\_best\_circuit(w, best\_params, p=1,

filename="qaoa\_best\_circuit.png"):

    n = w.shape[0]

    if isinstance(best\_params, torch.Tensor):

        flat = best\_params.detach().cpu().numpy()

    else:

        flat = np.array(best\_params)

    gammas = flat[:p]

    betas = flat[p:]

    edges = [(i, j, w[i, j]) for i in range(n) for j in range(i)

if w[i, j] != 0]

    qc = qaoa\_circuit(n, edges, gammas, betas)

    show\_circuit(qc, filename=filename, style="mpl")

# -------------------------

# Run example

# -------------------------

if \_\_name\_\_ == "\_\_main\_\_":

    G, w = make\_graph()

    print("Graph edges:", list(G.edges()))

    bf\_x, bf\_val = brute\_force\_maxcut(w)

    print("Brute-force best:", bf\_x, "value:", bf\_val)

    # show an initial example circuit (random parameters)

    demo\_display\_initial\_circuit(w, p=1,

filename="qaoa\_initial\_circuit.png")

    # run QAOA p=1 (toy)

    best = run\_qaoa\_with\_pytorch(w, p=1, init\_std=0.8,

maxiter=80, lr=0.2, finite\_diff\_eps=1e-3)

    print("QAOA best expected value:", best["val"])

    print("Most-likely bitstring found:", best["bitstring"])

    # evaluate most-likely bitstring exactly

    exact\_val = objective\_value(best["bitstring"], w)

    print("Exact value of that bitstring:", exact\_val)

    # Display the optimized circuit using the best parameters

    # (and save)

    if best["params"] is not None:

        demo\_display\_best\_circuit(w, best["params"], p=1,

filename="qaoa\_best\_circuit.png")

    else:

        print("No best params found to display.")

**OUTPUTS:**





