

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.

Algorithm - QAOA Algorithm:

1. Graph Construction

- Define adjacency matrix W .
- Build a NetworkX graph for visualization.

2. Classical Baseline

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

3. QAOA Circuit Construction

- Initialize qubits in $|+\rangle$.
- Apply alternating cost and mixer unitaries for depth p .
- Use controlled- Z rotation gates to implement $Z \otimes Z$ interactions.

4. Expectation Calculation

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

5. Hybrid Optimization

- Parameters $\vec{\gamma}, \vec{\beta}$ initialized randomly.
- Compute finite-difference gradients of expectation.
- Update parameters using PyTorch Adam optimizer.

6. Circuit Visualization

- Draw initial and optimized QAOA circuits using qiskit.visualization.

```
#!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
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  $|+\rangle^n$ 
    - for each layer l:
    cost unitary  $U_C(\gamma_l) = \exp(-i * \gamma_l * C)$ 
    mixer  $U_B(\beta_l) = \text{product } R_x(2*\beta_l)$ 
    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  $|+\rangle^n$ 
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  $\exp(-i * \theta/2 * Z_i Z_j) \rightarrow$ 
        # 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)
```

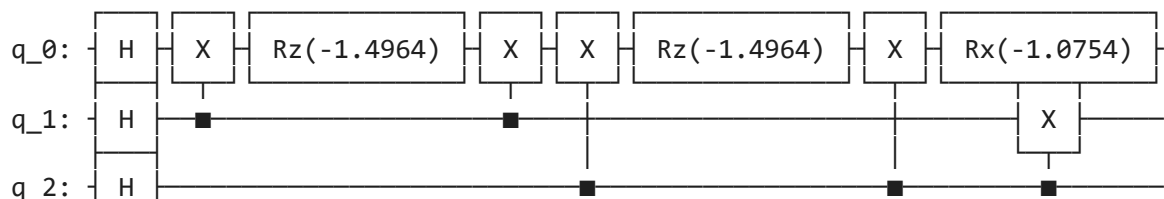
```
show_circuit(qc, filename=filename, style="mpl")
def demo_display_best_circuit(w, best_params, p=1,
filename="qaoa_best_circuit.png"):
    n = w.shape[0]
    if isinstance(best_params, torch.Tensor):
        flat = best_params.detach().cpu().numpy()
    else:
        flat = np.array(best_params)
    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")
```

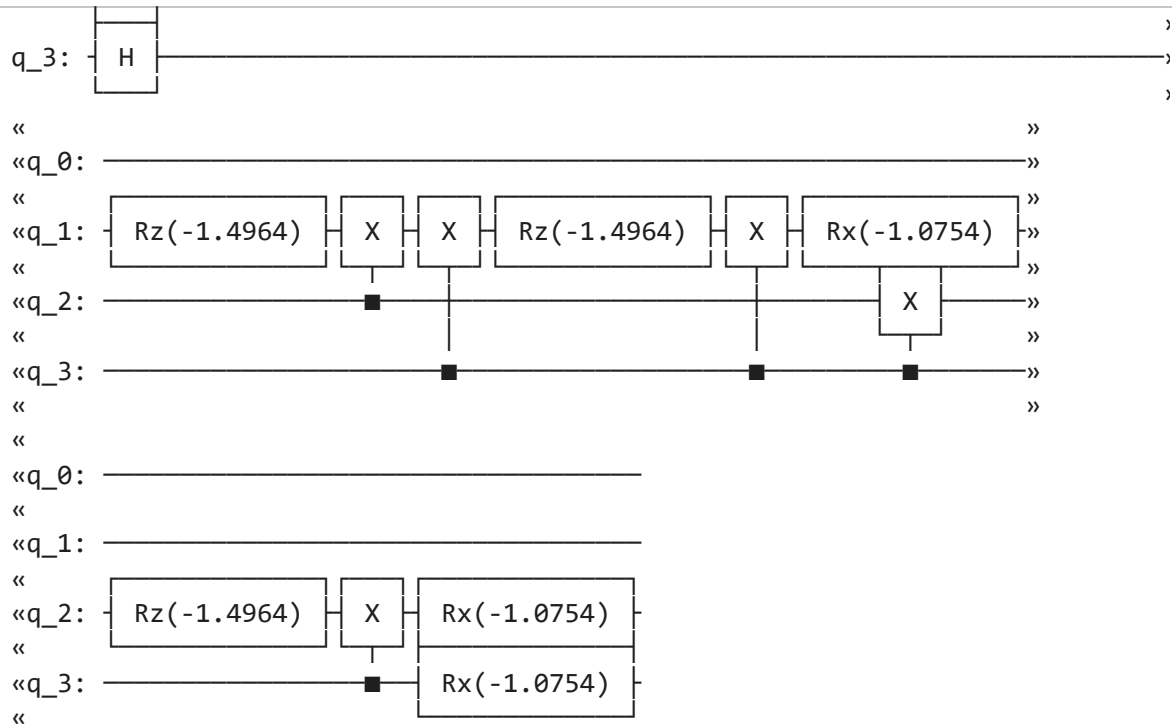
```
# run QAOA p=1 (toy)
best = run_qaoa_with_pytorch(w, p=1, init_std=0.8,
maxiter=80, lr=0.2, finite_diff_eps=1e-3)
print("QAOA best expected value:", best["val"])
print("Most-likely bitstring found:", best["bitstring"])
# evaluate most-likely bitstring exactly
exact_val = objective_value(best["bitstring"], 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.")
```

Graph edges: [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)]

Brute-force best: [0 1 1 0] value: 4

--- Quantum Circuit ---





Matplotlib drawing failed: "The 'pylatexenc' library is required to use 'Matplotlib' to draw the Quantum Circuit diagram above.

Iter 000: expected cut = 2.168545, loss = -2.168545

Iter 010: expected cut = 2.831415, loss = -2.831415

Iter 020: expected cut = 2.987799, loss = -2.987799

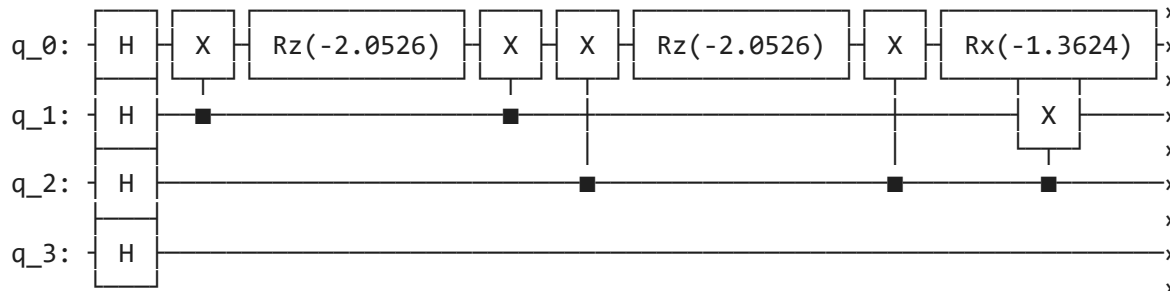
Iter 030: expected cut = 3.045777, loss = -3.045777

```

Iter 040: expected cut = 3.079084, loss = -3.079084
Iter 050: expected cut = 3.085885, loss = -3.085885
Iter 060: expected cut = 3.084443, loss = -3.084443
Iter 070: expected cut = 3.085099, loss = -3.085099
Iter 079: expected cut = 3.085828, loss = -3.085828
QAOA best expected value: 3.0859050639803915
Most-likely bitstring found: [1 0 0 1]
Exact value of that bitstring: 4

```

--- Quantum Circuit ---



Result:

The QAOA implementation successfully demonstrates a hybrid quantum-classical optimization approach to solving the Max-Cut problem.