CS F316 - Assignment I

This report discusses the need for optimizing organ allocation, existing approaches used to tackle it, discusses using Quantum Approximate Optimization Algorithm to solve the problem and demonstrates its feasibility using Qiskit/Pennylane.

Name: Nishal Ahmed Poovatham Kandiyil
ID NO: 2023A7PS0209U
Project Area: Quantum Optimization
Project Title: Optimizing Organ Allocation with Quantum Approximate Optimization Algorithm

1. Project Description

Organ transplantation has saved many lives since its first success in 1954. Organs can be donated from the dead and the living, and one deceased donor can donate up to eight organs [1]. Association of Organ Procurement Organizations, 2024 quantifies that 28% of the organs donated in 2023 where unutilized [2] and demand for organs far exceed the supply and it continues to grow every year [1]. This project tackles the logistics part of the NP-Complete organ assignment problem as done using dynamic labeling algorithm in [1] by selecting the optimal flight(s) from the donor to the recipient hospital which respects the constraint of Cold Ischemia Time (CIT) which is the viability of the organ in question from the time it is detached from the donor [3]. Due to the computational complexity of the problem, even the best classical algorithms will scale exponentially with the number of patients and donors.

Quantum computing has shown great promises in solving such complex combinatorial optimization problems in the form of hybrid classical-quantum techniques namely Quantum Approximate Optimization Algorithm (QAOA) [4]. This project models organ allocation defined in [1] as a Quadratic Unconstrained Binary Optimization (QUBO) problem and finds the most optimal solution leveraging a Quantum Circuit.

2. Methodology

We first express the problem in QUBO form and then reformulate it into a Hamiltonian which can be expressed with qubits, then steer the system into the ground state of the previously obtained Hamiltonian cost function with help of exponential operators with angles.

2.1 Data Synthesis

To initiate the project we synthesize data similar to the one found in [1] using the “Airlines, Airport, and Flight Routes” dataset on Kaggle, which is made with information collected from OpenFlights Airports Database [5]. Airports and flights are represented as nodes and edges respectively. The distance between two airports is calculated using the Haversine distance

formula where the longitudes and latitudes are obtained from and the flight duration is then naively derived from the calculated distance.[5].

$$\mathbf{d} = \mathbf{2r}\,\mathbf{arcsin}\left(\sqrt{\mathbf{sin^2}\left(\frac{\mathrm{lat_2} - \mathrm{lat_1}}{\mathbf{2}}\right) + \mathbf{cos}(\mathrm{lat_1})\,\mathbf{cos}(\mathrm{lat_2})\,\mathbf{sin^2}\left(\frac{\mathrm{lon_2} - \mathrm{lon_1}}{\mathbf{2}}\right)}\right)$$

Equation 1: Haversine Distance Formula

```
def haversine_distance(lat1, lon1, lat2, lon2):
    """Calculate great-circle distance between two points in miles"""
    R = 3959  # Earth radius in miles
    dlat = radians(lat2 - lat1)
    dlon = radians(lon2 - lon1)
    a = sin(dlat/2)**2 + cos(radians(lat1)) * cos(radians(lat2)) *
sin(dlon/2)**2
    c = 2 * atan2(sqrt(a), sqrt(1-a))
    return R * c
```

Listing 1: Haversine Distance function definition

```
def generate_network():
    flights = []
    base_time = datetime(2025,12,3,8,0)
    for _, orig in network_airports.iterrows():
        for _, dest in network_airports.iterrows():
            if orig['IATA'] != dest['IATA']:
                route = (orig['IATA'], dest['IATA'])
                if route in valid_routes:


                    distance = haversine_distance(orig['Latitude'],
orig['Longitude'],
                                                  dest['Latitude'],
dest['Longitude'])
                    duration = timedelta(hours=distance / 500)  # Assume avg
speed 500 mph
                    cost = distance*0.8 + duration.total_seconds()/3600*50
                    for offset in [0, 6]:
                        departure_time = base_time + timedelta(hours=offset)
```

```
                        arrival_time = departure_time +
timedelta(hours=duration.total_seconds()/3600)
                        flights.append({
                            'flight_id': len(flights),
                            'origin': orig['IATA'],
                            'destination': dest['IATA'],
                            'departure_time': departure_time,
                            'distance': distance,
                            'duration_hours': duration.total_seconds()/3600,
                            'cost': cost,
                            'handling_time': 1.5,
                            'arrival_time': arrival_time

                        })
    return pd.DataFrame(flights)
```

Listing 2: Code for data synthesis.

To ensure the feasibility of the project the airports will be limited to a small set of airports located in the United States.

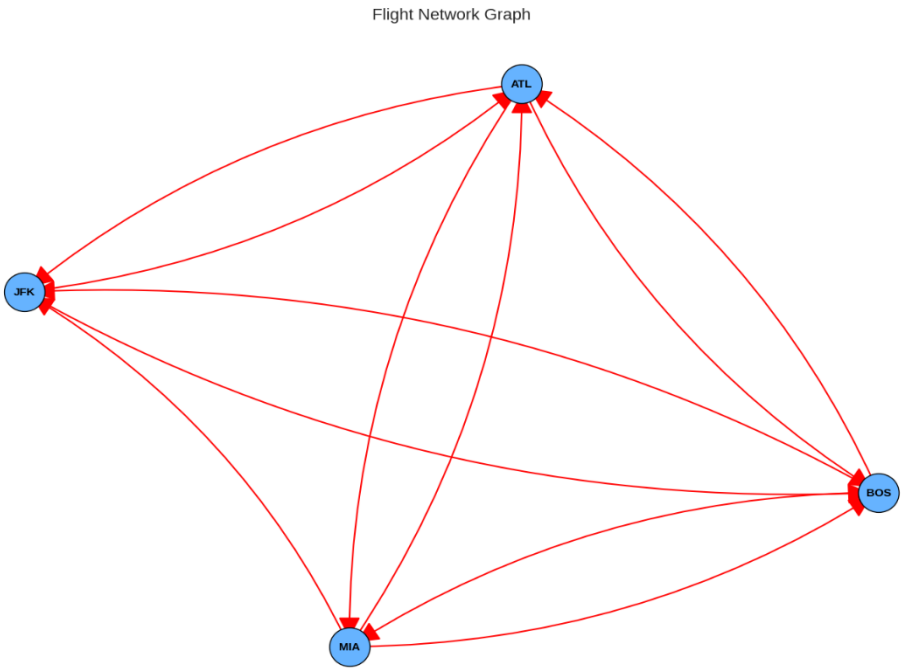

Figure 1: Generated Network Graph

2.2 Labeling Algorithm

The base paper [1] models the problem as a resource constrained shortest path problem and solves it using dynamic programming labeling algorithm. The same algorithm was reimplemented in python and will be used to obtain the ground truth.

2.3 Quantum Approximation Optimization Algorithm

Quantum Approximation Optimization Algorithm (QAOA) is an algorithm which shows promise in solving complex combinatorial optimization problems on Noisy Intermediate-Scale Quantum (NISQ) devices [6]. We begin by defining a cost Hamiltonian and mixer Hamiltonian and constructing circuits $e^{i\gamma H_c}$ and $e^{i\alpha H_c}$ called cost and cost and mixer layers respectively, choosing parameters to build the circuit, preparing and initial state and optimizing the parameters which is proceeded by measurements which yield approximate solution to the problem [6].

2.3.1 Quantum Unconstrained Binary Optimization

QUBO problems are compatible with quantum algorithms and are well studied for combinatorial optimization problems. Assume graph $G = (V, E)$ with V vertices indicating a network of flights and E edges. The general form of QUBO problems is

$$\max_{z\in\{0,1\}^N} \sum_{i,j} A_{ij}\, z_i z_j + \sum_i B_i\, z_i$$

Equation 2: QUBO General Form

The resource constrained optimization problem is not directly compatible with quantum algorithms. We can transform it into QUBO by mapping every vertex as a decision variable $x_i \in 0,1$ for each flight i since QUBO formulation uses binary variables, where $x_i = 1$ indicates a flight taken and $x_i = 0$ implies the inverse. We can now define a part of our objective of the cost as

$$H_{cost} = \max \sum_i c_i x_i$$

where c_i is the cost of flight i. Assume c_i encompasses several factors such as time urgency, cost of the flight which exponentially increases in case of military flights, etc. For this project we restrict ourselves to the former two in which both are calculated arbitrarily. However our objective is subject to constraints but QUBO problems are unconstrained, therefore the original Mixed-Integer Linear Programming (MILP) constraints given in [1] should be added as penalty terms to Equation 3.

The flow constraints as mentioned in [1] can be written as the following

$$\left(1 - \sum_{j\in \mathrm{out}(o)} x_j\right)^2 \quad (1)$$

$$\left(1 - \sum_{i\in \mathrm{in}(d)} x_i\right)^2 \quad (2)$$

$$\sum_{k\neq o,d}\left(\sum_{i\in \mathrm{in}(k)} x_i - \sum_{j\in \mathrm{out}(k)} x_j\right)^2 \quad (3)$$

where (1) and (2) ensures there is only one outgoing and incoming flight respectively, and (3) ensures that the total flow into an airport is equal to the total outflow. All the flow penalties can be multiplied by a carefully tuned penalty parameter A and be defined as H_{flow}.

$$H_{\text{flow}} = A\left(1 - \sum_{j\in\text{out}(o)} x_j\right)^2 + A\left(1 - \sum_{i\in\text{in}(d)} x_i\right)^2 + A\sum_{k\neq o,d}\left(\sum_{i\in\text{in}(k)} x_i - \sum_{j\in\text{out}(k)} x_j\right)^2 \quad (4)$$

Time feasibility constraints due to CIT and temporal precedence, if violated can punished by large penalty constant B and can be defined as H_{time} as shown in (4).

$$H_{time} = B\sum_{(i,j)\in\text{incompatible}} x_i x_j \qquad (5)$$

Finally, we can construct the QUBO objective with constraints by adding (4) and (5) to the initial objective H_{cost}.

$$H_{total} = H_{cost} + H_{flow} + H_{time}$$

Equation 3: Full QUBO Objective

```
def build_16qubit_qubo(flights_df, origin, dest, max_time):

    n = len(flights_df); Q = np.zeros((n,n)); A, B = 2000, 5000

    # Costs

    for i in range(n): Q[i,i] += flights_df.iloc[i]['cost']

    # Origin

    outs = [i for i,f in enumerate(flights_df.itertuples()) if f.origin ==
origin]

    for i in outs: Q[i,i] += A

    for i,j in [(outs[a],outs[b]) for a in range(len(outs)) for b in
range(a+1,len(outs))]:

        Q[i,j] -= 2*A; Q[j,i] -= 2*A

    # Destination

    ins = [i for i,f in enumerate(flights_df.itertuples()) if f.destination
== dest]

    for i in ins: Q[i,i] += A

    for i,j in [(ins[a],ins[b]) for a in range(len(ins)) for b in
range(a+1,len(ins))]:

        Q[i,j] -= 2*A; Q[j,i] -= 2*A

    # Time

    for i in range(n):

        fi = flights_16qubit.iloc[i]

        for j in range(n):
```

```
            if i==j: continue
            fj = flights_16qubit.iloc[j]
            if fi.destination == fj.origin:
                gap_h = (fj.departure_time -
fi.arrival_time).total_seconds()/3600
                if gap_h < 1.5: Q[i,j] += B; Q[j,i] += B
    return Q
```

Listing 3: QUBO Formulation Function Definition

2.3.2 Reformulation as Hamiltonian

The QUBO objective defined in *Equation 3* can now be rewritten as a Hamiltonian Ising Model whose ground state corresponds to the ideal solution. This can be done by the substitution $x_i = (1 - \sigma_i)/2$ where $z_i \in \{-1,1\}$

$$H_{\text{Ising}} = \sum_i c_i \left(\frac{1-z_i}{2}\right) + A\left(1 - \sum_{j\in\text{out}(o)} \frac{1-z_j}{2}\right)^2 + A\left(1 - \sum_{i\in\text{in}(d)} \frac{1-z_i}{2}\right)^2 + A\sum_{k\neq o,d}\left(\sum_{i\in\text{in}(k)} \frac{1-z_i}{2} - \sum_{j\in\text{out}(k)} \frac{1-z_j}{2}\right)^2 + B\sum_{(i,j)\in\text{incompatible}} \left(\frac{1-z_i}{2}\right)\left(\frac{1-z_j}{2}\right)$$
$$= -\sum_{i,j} J_{ij} z_i z_j + \sum_i h_i z_i + C$$

Derivation 1: QUBO to Ising

and promoting the z_i variables to a σ_z matrix to obtain a quantum formulation of the problem [4].

$$\sigma_z = Z_i = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$H_{\text{Ising}} = -\sum_{i,j} J_{ij} Z_i Z_j + \sum_i h_i Z_i + C$$

Equation 4: Pauli Z matrix and the quantum formulation of the problem

In *Equation 4* and *Derivation 1* J_{ij} comes from $x_i x_j$ terms in $H_{flow}$and H_{time}, h_i comes from the linear H_{cost} and C is the sum of all constants which plays no part in determining the ground state.

```
def qubo_to_ham(Q):
    coeffs, ops = [], []
    for i in range(n_qubits):
        h = (Q[i,i] + np.sum(Q[i,:]) + np.sum(Q[:,i]) - Q[i,i])/4
        coeffs.append(h); ops.append(qml.PauliZ(i))
    for i in range(n_qubits):
```

```
        for j in range(i+1,n_qubits):
            if abs(Q[i,j])>1e-8:
                coeffs.append(Q[i,j]/4);
ops.append(qml.PauliZ(i)@qml.PauliZ(j))
    return qml.Hamiltonian(coeffs, ops)
```

Listing 4: QUBO to Hamiltonian function definiton.

2.3.4 Pennylane Implementation

QAOA is implemented on Pennylane exactly how it was shown to do in [6].

```
def qaoa_circuit(params, hamiltonian, n_qubits, p=2):
    for i in range(n_qubits):
        qml.Hadamard(wires=i)

    for layer in range(p):
        gamma = params[2*layer]      # Cost angle
        beta = params[2*layer + 1]   # Mixer angle

        qml.ApproxTimeEvolution(hamiltonian, gamma, n=1)
        for i in range(n_qubits):
            qml.RX(2*beta, wires=i)
```

Listing 5: Code for QAOA as given in [6].

The default.qubit simulator was used which allows for simulating 64 qubits at most. The initial modeling of this problem has QUBO matrix of size (165,165) which had to be scaled down to (22,22) for simulation.

2.3.5 Optimization

Parameters passed in QAOA are optimized classically using an Adam's Optimizer with an initial learning rate of 0.1. Parameters are initialized at random and the optimization is run for 50 iterations.

```
@qml.qnode(dev)
def cost_function(params):
    qaoa_circuit(params, hamiltonian, n_qubits)
    return qml.expval(hamiltonian)


optimizer = qml.AdamOptimizer(stepsize=0.1)
```

```
params = np.random.uniform(0, np.pi, 4)


for iteration in range(50):
    params, cost = optimizer.step_and_cost(cost_function, params)
```

Listing 6: Code for Optimization

Other techniques such as warm start will be explored in future studies.

2.3.6 Measurement

The optimized quantum state is now sample to extract meaningful solutions, in this case optimal flight paths.

```
@qml.qnode(dev, shots=1024)
def sample_16(params): qaoa_16(params); return qml.sample()
```

Listing 7: Function definiton for measurement

The bitstring obtained is converted to flight indices, the obtained set of indices are cleaned to remove unfeasible paths and reconstruct valid paths from origin.

3. Results

For the small, synthesized dataset the optimized set of flights according to dynamic programming labeling algorithm was

Cell Output:

Classical: JFK → BOS → MIA

The output after measuring QAOA run are as follows

Best state: 0100001011001110

Selects flights: [1, 2, 3, 6, 7, 9, 14]

After arranging the selected flights in chronological order the unfeasible flight are removed, which are all flights except '1' and '3' and the flight details of those flights are given in Figure 2.

```
flights_16qubit[flights_16qubit['flight_id'].isin([1,3])]
```

✓ 0.0s

	flight_id	origin	destination	departure_time	arrival_time	duration_hours	cost	handling_time
1	1	JFK	BOS	2025-12-03 14:00:00	2025-12-03 15:16:48.613020	1.280170	376.076628	1.5
3	3	JFK	ATL	2025-12-03 14:00:00	2025-12-03 15:53:32.644179	1.892401	651.580522	1.5

Figure 2: Selected Flights after Data Cleaning

The resulting path is equivalent to path obtained using classical methods, however due to the inaccessibility of quantum hardware able to manipulate a large number of qubits, no foreseeable

advantage can be guaranteed. However this report was able to reproduce similar results to classical methods on a simulated system which showcases the potential and impact quantum computers might have in the future, especially when it comes NP-complete problems such as organ assignment.

References

[1] I. Balster, J. A. Caetano, G. M. Ribeiro, and L. Bahiense, “Optimizing the Transport of Organs for Transplantation,” *Comput. Oper. Res.*, vol. 176, p. 106934, Apr. 2025, doi: 10.1016/j.cor.2024.106934.

[2] “Artificial intelligence in healthcare and medicine technology development review,” *Eng. Appl. Artif. Intell.*, vol. 143, p. 109801, Mar. 2025, doi: 10.1016/j.engappai.2024.109801.

[3] O. Alagoz, A. J. Schaefer, and M. S. Roberts, “Optimizing Organ Allocation and Acceptance,” in *Handbook of Optimization in Medicine*, H. E. Romeijn and P. M. Pardalos, Eds., New York, NY: Springer US, 2009, pp. 1–24. doi: 10.1007/978-0-387-09770-1_1.

[4] “Quantum approximate optimization algorithm,” IBM Quantum Documentation. Accessed: Oct. 03, 2025. [Online]. Available: https://quantum.cloud.ibm.com/docs/en/tutorials/quantum.cloud.ibm.com/docs/en/tutorials/quantum-approximate-optimization-algorithm

[5] “Airlines, Airport, and Flight Routes.” Accessed: Dec. 02, 2025. [Online]. Available: https://www.kaggle.com/datasets/elmoallistair/airlines-airport-and-routes

[6] J. Ceroni, “Intro to QAOA,” *PennyLane Demos*, Nov. 2020, Accessed: Dec. 02, 2025. [Online]. Available: https://pennylane.ai/qml/demos/tutorial_qaoa_intro