Solution of A Variation of the Subset-Sum Problem using Variational Quantum Algorithms

Sayantan Pramanik, M Girish Chandra

TCS Research and Innovation

{sayantan.pramanik, m.gchandra}@tcs.com

August 11, 2020

Given a set of n real numbers, $\xi = \{s_1, s_2, \ldots, s_n\}$, the problem is to find a subset ζ of ξ such that the sum of elements in ζ is as close as possible to a given real number S. The elements of ξ can be thought of as the nodes of a graph that get partitioned into two clusters, namely 0 and 1. The vertices that fall into the cluster 1 are the elements of ζ. To accomplish this using a variational algorithm, such as QAOA, we will be representing the problem as an Ising model, but with a slight change. We will replace the Pauli-Z, σ_z, matrix with the matrix given by:

$$Z = \frac{1}{2}(I - \sigma_z) = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{1}$$

The advantage of using the matrix Z is that it maps the problem from the spin formalism to the $QUBO$ formalism.

For simplification, let us consider an example with just two elements s_1 and s_2 in the set. The Ising Hamiltonian corresponding to such a case can be written as:

$$H = s_1(Z \otimes I) + s_2(I \otimes Z) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_1 & 0 \\ 0 & 0 & 0 & s_1 + s_2 \end{bmatrix} \tag{2}$$

It must be noted that the solution $|01\rangle$ signifies that the second element, s_2, is chosen to be in ζ and the first one, s_1, is rejected. The final cost Hamiltonian H_C can be constructed as:

$$H_c = (H - S)^2 = \begin{bmatrix} S^2 & 0 & 0 & 0 \\ 0 & (s_2 - S)^2 & 0 & 0 \\ 0 & 0 & (s_1 - S)^2 & 0 \\ 0 & 0 & 0 & (s_1 + s_2 - S)^2 \end{bmatrix} \tag{3}$$

It is desirble to find the eigenstate corresponding to the lowest eigenvalue of H_c, and measuring that eigenstate in the computational basis should give us the requisite solution. However, we must remember that squaring the matrix $(H - S)$ might introduce some pseudo-solutions. Using the variational principle, the goal is to find the state $|\Phi\rangle$ that minimises the expression $\langle\Phi|H_c|\Phi\rangle$.

For n elements, Equation (2) can be generalised to:

$$H = \sum_i s_i Z_i \tag{4}$$

and

$$\begin{aligned} H_c = (H-S)^2 = H^2 - HS - SH + S^2 = (\sum_i s_i Z_i)^2 - 2S(\sum_i s_i Z_i) + S^2 \\ = \sum_i s_i^2 Z_i^2 + \sum_{ij, i\neq j} 2s_i s_j (I \otimes \cdots \otimes Z_i \otimes \cdots \otimes I)(I \otimes \cdots \otimes Z_j \otimes \cdots \otimes I) - 2S(\sum_i s_i Z_i) + S^2 \end{aligned} \tag{5}$$

In the above equation, ${Z_i}^2$ is equivalent to applying Z_i twice on a qubit. But from Equation (1), we see that $Z^2 = Z$. Similarly, $(I \otimes \cdots \otimes Z_i \otimes \cdots \otimes I)(I \otimes \cdots \otimes Z_j \otimes \cdots \otimes I)$ is equivalent to applying Z on qubits i and j, which can be truncated to $(I \otimes \cdots \otimes Z_i \otimes \cdots \otimes Z_j \otimes \cdots \otimes I) = Z_i \otimes Z_j$. Using these circuit identities reduces Equation (5) to:

$$H_c = \sum_i s_i^2 Z_i + \sum_{ij, i \neq j} 2 s_i s_j Z_i \otimes Z_j - 2S(\sum_i s_i Z_i) + S^2 \tag{6}$$

In Equation (6), the cost Hamiltonian H_c, has been expressed as an Ising model in Z. The i^{th} node of the graph has a weight given by $s_i(s_i - 2S)$, and the weight of the edge between the i^{th} and j^{th} nodes of the resultant fully-connected graph is $2s_is_j$. Using the transformation in Equation 1, it is easy to rewrite the Ising in terms of σ_z.

$$H_{ising} = H_c = \sum_i s_i(s_i - 2S)Z_i + \sum_{ij} 2s_i s_j Z_i Z_j + S^2 I_{2^n} \tag{7}$$

where I_{2^n} is the $2^n \times 2^n$ identity matrix.

Now that we have acquainted ourselves with the problem, let us start coding it from scratch using QAOA. We begin with importing the requisite packages in block 1.

[1]:
```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute
from qiskit import Aer
from qiskit.tools.visualization import plot_histogram
from scipy.optimize import minimize
import numpy as np
from qiskit.compiler import transpile
backend = Aer.get_backend('qasm_simulator')
```

Next we shall prepare the QAOA ansatz given by:

$$|\beta, \gamma\rangle = e^{-i\beta H_M} e^{-i\gamma H_C} H^{\otimes n} |0\rangle^{\otimes n} \tag{8}$$

where H_M is the standard mixing Hamiltonian, H is the Hadamard gate, and β and γ are the parameters that need to be optimised to minimise the cost $\langle \gamma, \beta | H_C | \beta, \gamma \rangle$. The ansatz can be prepared by following the steps:

1. n qubits are initialised to the state $|0\rangle$, as usual

2. Hadamard gate is applied on each of them
3. The unitary $e^{-i\gamma H_C}$ is implemented as:
   (a) For the i^{th} node in the graph, applying the U_1 gate on the i^{th} qubit with the parameter $s_i(s_i - 2S)\gamma$. The nodes are stored as a list of tuples whose first element is the node-number, and the second element is s_i.
   (b) For the edge between the i^{th} and j^{th} nodes, applying the CRZ (CU_1) gate with i^{th} as the control and j^{th} as the target qubit, and $2s_is_j\gamma$ as the parameter. The edges are stored as a list of triples containing the numbers of the two vertices and the weight of the edge between them.
   (c) The term $e^{-iS^2 I^{\otimes n}}$ is just a global phase, and can safely be ignored.
4. The unitary corresponding to the mixing Hamiltonian is applied as usual using the RX gate with 2β as the parameter on all the qubits.

[2]:
```python
def state_preparation(gammas, betas, p = 1):
    q = QuantumRegister(n)
    circuit = QuantumCircuit(q)

    for i in range(n):
        circuit.h(q[i])

    circuit.barrier()

    for i in range(p):
        gamma = gammas[i]
        beta = betas[i]

        for node in nodes:
            theta = node[1]*(node[1]-2*L)*gamma
            circuit.u1(theta, q[node[0]])

        circuit.barrier()

        for edge in edges:
            theta = edge[2]*gamma
            node1 = edge[0]
            node2 = edge[1]
            circuit.cu1(theta, q[node1], q[node2])

        circuit.barrier()

        for node in nodes:
            circuit.rx(2*beta, q[node[0]])

        circuit.barrier()
    return circuit
```

The following two blocks deal with calculating the value of the cost function. For the sake of convenience, two different functions have been defined to calculate the cost of the nodes and edges. It is easy to see from Equation (9) that the total cost can be found from the summation of the cost of the individual nodes and edges.

$$\begin{aligned}\langle\Psi|H_C|\Psi\rangle &= \langle\Psi|\sum_i s_i(s_i-2S)Z_i + \sum_{ij} 2s_is_jZ_iZ_j + S^2 I^{\otimes n}|\Psi\rangle \\ &= \langle\Psi|\sum_i s_i(s_i-2S)Z_i|\Psi\rangle + \langle\Psi|\sum_{ij} 2s_is_jZ_iZ_j|\Psi\rangle + S^2\langle\Psi|I^{\otimes n}|\Psi\rangle \\ &= \sum_i \langle\Psi_i|s_i(s_i-2S)Z_i|\Psi_i\rangle + \sum_{ij}\langle\Psi_{ij}|2s_is_jZ_iZ_j|\Psi_{ij}\rangle + S^2\end{aligned} \tag{9}$$

where $|\Psi_i\rangle$ is the state of the i^{th} node and $|\Psi_{ij}\rangle = |\Psi_i\rangle \otimes |\Psi_j\rangle$.

If the state $|\Psi_i\rangle$ is given by $\begin{bmatrix}\alpha \\ \beta\end{bmatrix}$, then:

$$\langle\Psi_i|s_i(s_i-2S)Z_i|\Psi_i\rangle = \frac{s_i(s_i-2S)}{2}(1-|\alpha|^2+|\beta|^2) \tag{10}$$

```
[3]: def get_node_cost(gammas, betas, p, node):
         circuit = state_preparation(gammas, betas, p)
         c = ClassicalRegister(1)
         circuit.add_register(c)

         cost = 0
         weight = node[1]*(node[1]-2*L)/2
         cost += weight

         circuit.measure(node[0], 0)
         shots = 1000
         job = execute(circuit, backend, shots = shots)
         result = job.result()
         counts = result.get_counts()

         try:
             a_2 = counts['0']
         except:
             a_2 = 0
         try:
             b_2 = counts['1']
         except:
             b_2 = 0

         expval = (a_2 - b_2)/shots
```

```
    cost += -1*weight*expval

    return cost
```

Similarly, if the state $|\Psi_{ij}\rangle$ is given by $\begin{bmatrix}\alpha\\ \beta\\ \gamma\\ \delta\end{bmatrix}$, then:

$$\langle\Psi_{ij}|2s_is_jZ_iZ_j|\Psi_{ij}\rangle = \frac{s_is_j}{2}(1-|\alpha|^2-|\beta|^2-|\gamma|^2+3|\delta|^2) \tag{11}$$

[4]:
```
def get_edge_cost(gammas, betas, p, edge):
    circuit = state_preparation(gammas, betas, p)
    c = ClassicalRegister(2)
    circuit.add_register(c)

    cost = 0
    weight = edge[2]/4
    cost += weight

    circuit.measure(edge[0], 0)
    circuit.measure(edge[1], 1)

    shots = 1000
    job = execute(circuit, backend, shots = shots)
    result = job.result()
    counts = result.get_counts()

    try:
        a_2 = counts['00']
    except:
        a_2 = 0
    try:
        b_2 = counts['01']
    except:
        b_2 = 0
    try:
        c_2 = counts['10']
    except:
        c_2 = 0
    try:
        d_2 = counts['11']
    except:
        d_2 = 0

    expval = (- a_2 - b_2 - c_2 + 3*d_2)/shots
    cost += weight*expval

    return cost
```

Having defined functions to calculate the cost at each iteration, we define a function 'qaoa' that uses the scipy implementation of COBYLA optimiser to optimise the randomly initialised parameters β and γ for each layer of QAOA. The number of layers has been stored in the variable p and the variable L denotes the aforementioned S.

[5]:
```
def qaoa(beta_gamma):
    betas = beta_gamma[:1]
    gammas = beta_gamma[1:]

    optimizer_cost = 0

    for node in nodes:
        optimizer_cost += get_node_cost(gammas, betas, p, node)

    for edge in edges:
        optimizer_cost += get_edge_cost(gammas, betas, p, edge)

    optimizer_cost += L**2

    return optimizer_cost
```

Once the parameters have been optimised to minimise the cost $\langle\gamma,\beta|H_C|\beta,\gamma\rangle$, the final parameters can be used to recreate the circuit from which the final state of the qubits can be measured to get our solution.

[6]:
```
def finalcircuit(gammas, betas, p = 1):
    q = QuantumRegister(n)
    c = ClassicalRegister(n)
    circuit = QuantumCircuit(q, c)

    for i in range(n):
        circuit.h(q[i])

    circuit.barrier()

    for i in range(p):
        gamma = gammas[i]
        beta = betas[i]

        for node in nodes:
            theta = node[1]*(node[1]-2*L)*gamma
            circuit.u1(theta, q[node[0]])

        circuit.barrier()
        for edge in edges:
            theta = edge[2]*gamma
            node1 = edge[0]
            node2 = edge[1]
            circuit.cu1(theta, q[node1], q[node2])

        circuit.barrier()

        for node in nodes:
            circuit.rx(2*beta, q[node[0]])

        circuit.barrier()

    return circuit
```

[7]:
```
p = 3
L = 8.0
nodes = [(0, 1.0), (1, 3.0), (2, 5.0), (3, 7.0)]
n = len(nodes)
edges = []
for i in range(n):
    for j in range(i+1, n):
        weight = 2*nodes[i][1]*nodes[j][1]
        edges.append((i,j,weight))

beta = np.random.uniform(0, np.pi*2, p)
gamma = np.random.uniform(0, np.pi*2, p)
beta_gamma = np.concatenate([beta, gamma])
l = len(beta_gamma)//2

result = minimize(qaoa, beta_gamma, method='cobyla')
result
```

[7]:
```
     fun: 5.030000000000015
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 58
  status: 1
 success: True
       x: array([2.99226177, 2.8977866 , 0.61834471, 5.48768244, 1.48742322,
       6.62370819])
```

The histogram below shows the relative frequency of the obtained measurements, where the least significant bit denotes the first real number of the set ξ, and the most significant bit denotes the last number of the set. For the measurement having the highest frequency, if the reading corresponding to s_i is 1, then it is placed into the new set ζ.

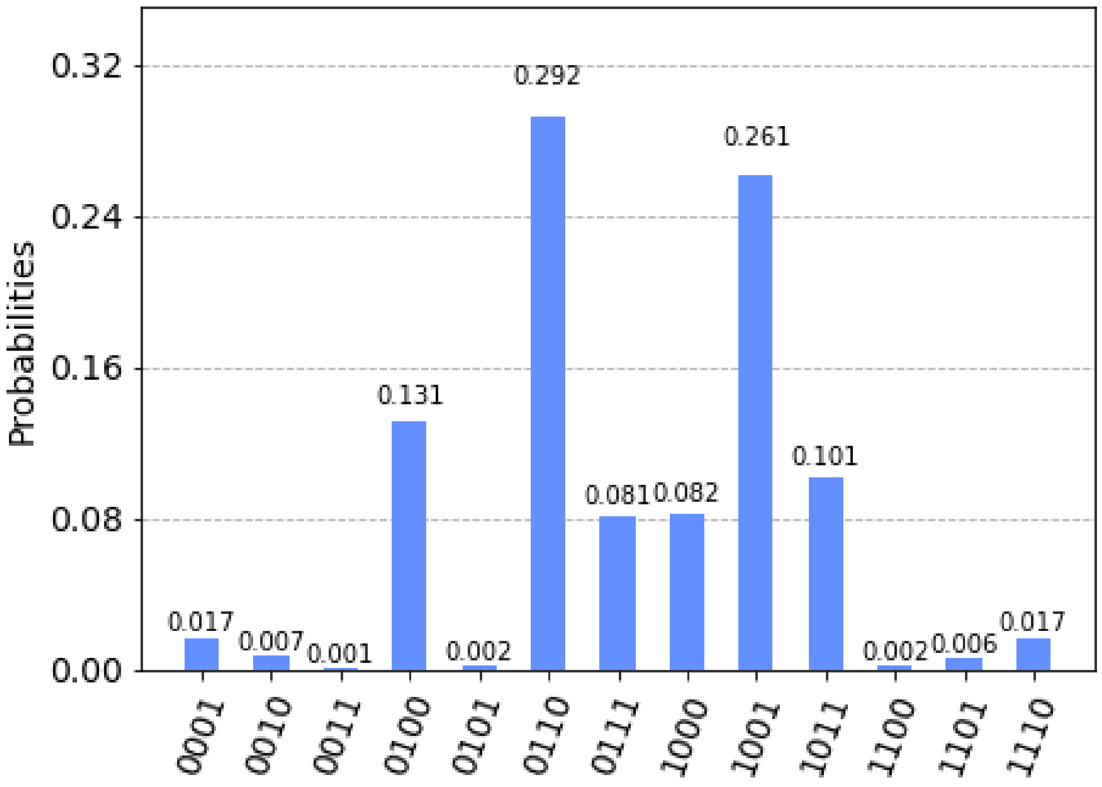
[8]:
```
final_gammas = result.x[1:]
final_betas = result.x[:1]
final_circuit = finalcircuit(final_gammas, final_betas, p)
q_final = final_circuit.qubits
c_final = final_circuit.clbits

final_circuit.measure(q_final, c_final)

final_circuit = transpile(final_circuit)

shots = 1000
job = execute(final_circuit, backend, shots = shots)
plot_histogram(job.result().get_counts(final_circuit))
```

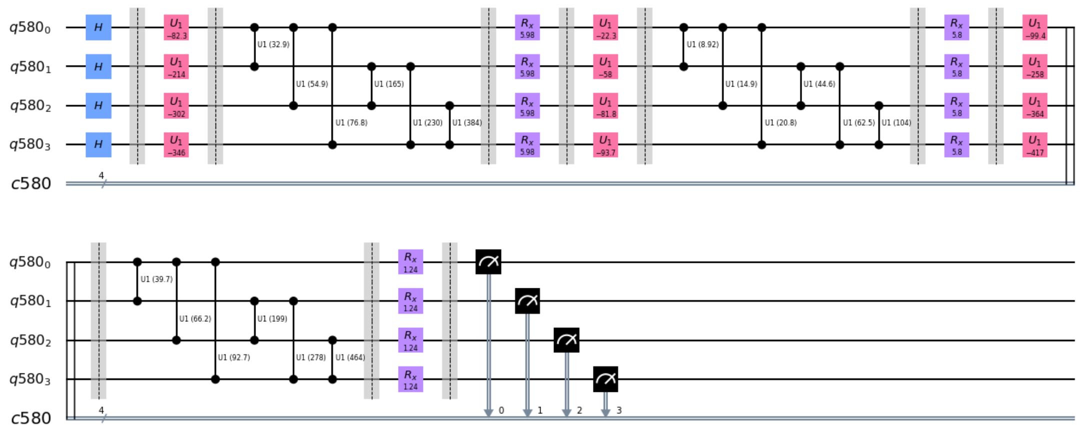
[8]:



The circuit for the algorithm is as shown below:

[9]:
```
final_circuit.draw(output='mpl')
```

[9]:



Scope of improvement:

- The number of times the quantum circuit is created by the algorithm is of the order of n^2, which can be optimised to order n.
- The squaring in Equation (3) leads to the appearance of a false solution along with the actual solution.
- The algorithm seems to be sensitive to the initial values of the parameters. The results shown above were the best out of about ten runs.

```
[ ]:
```