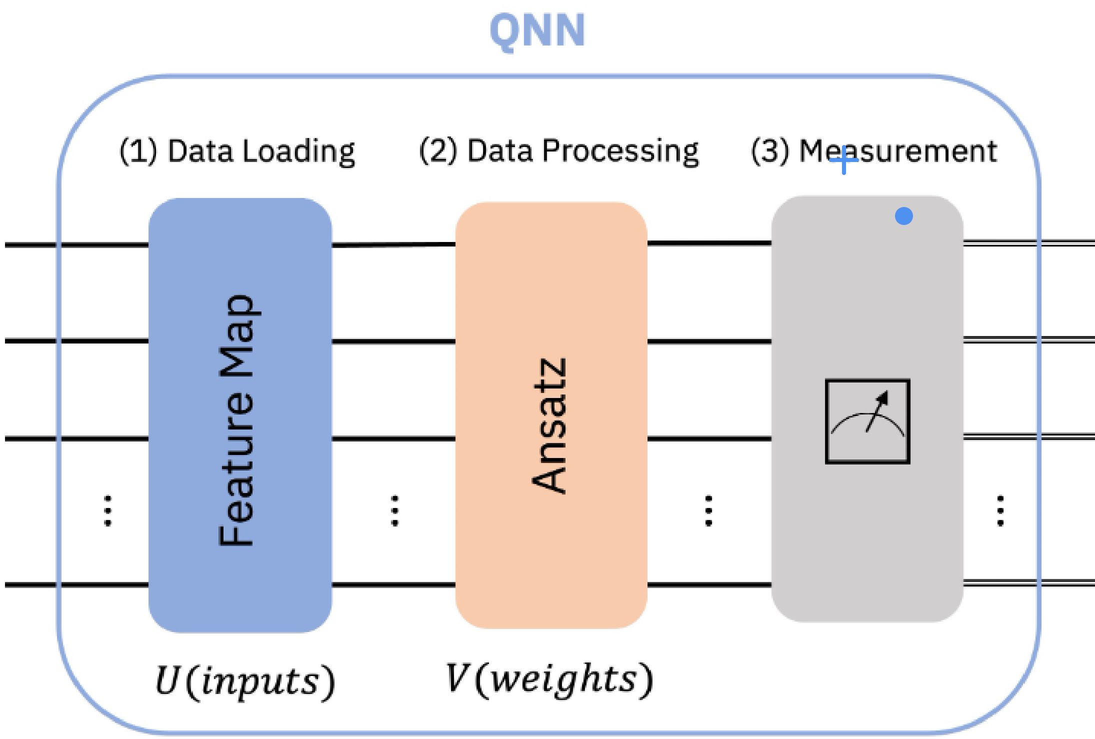
QAOA

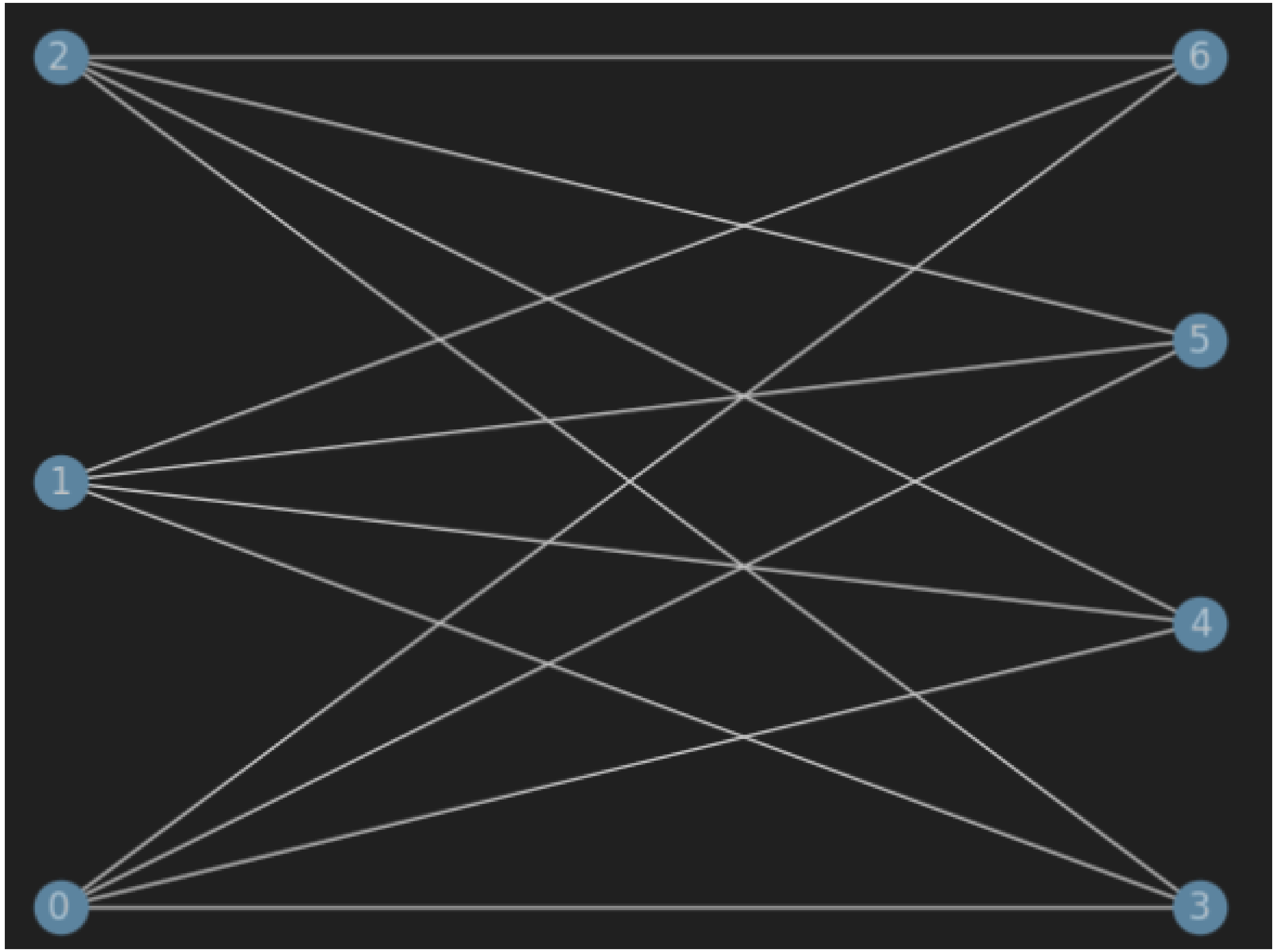
Quantum Approximate Optimization Algorithm (QAOA)

- Quantum variational algorithm
- QAOA searches
  for parameters which minimalize eigenvalues
  of Hamiltonian
- Hybrid algorithm
- Combinatorial optimization problems (e.g., Max-Cut, graph coloring)
- Cost operator based on the optimization problem
- Mixer operator typically consists of X gates (Pauli-X)



Max-cut

```
G = nx.Graph()
edges = []

n = 3
m = 4
for j in range(n,m + n):
    for i in range(n):
        edges.append((i, j))

gList = []

for i in range(n):
    gList.append(i)

print(edges)

G.add_edges_from(edges)
nx.draw(G,pos=nx.bipartite_layout(G, gList), with_labels=True)
```



Feature map

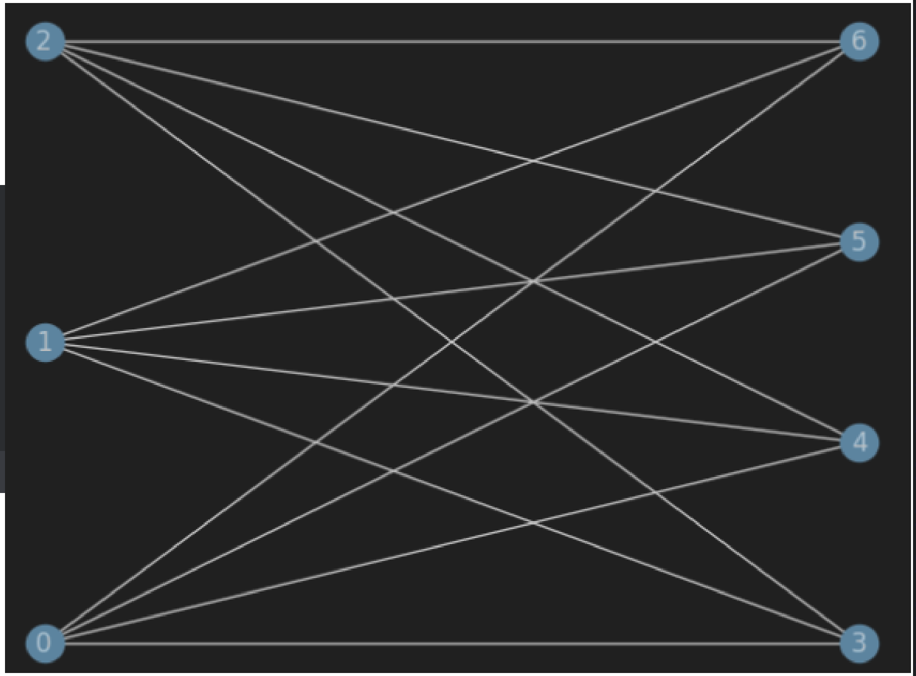
- Cost function operator - use the Estimator to evaluate the expectation value and minimize it using an optimizer. Successful optimization returns the optimal parameter values θ , allowing us to construct the solution state $|\psi(\theta)\rangle$ and compute the observed expectation value $C(\theta)$.
- The mixer operator introduces superposition into the quantum state, allowing exploration of the solution space. In the circuit, it is implemented using $RX(\beta)$ gates on all qubits, where $RX(\beta)$ rotates the qubits around the X-axis, creating a superposition of states.

Cost operator

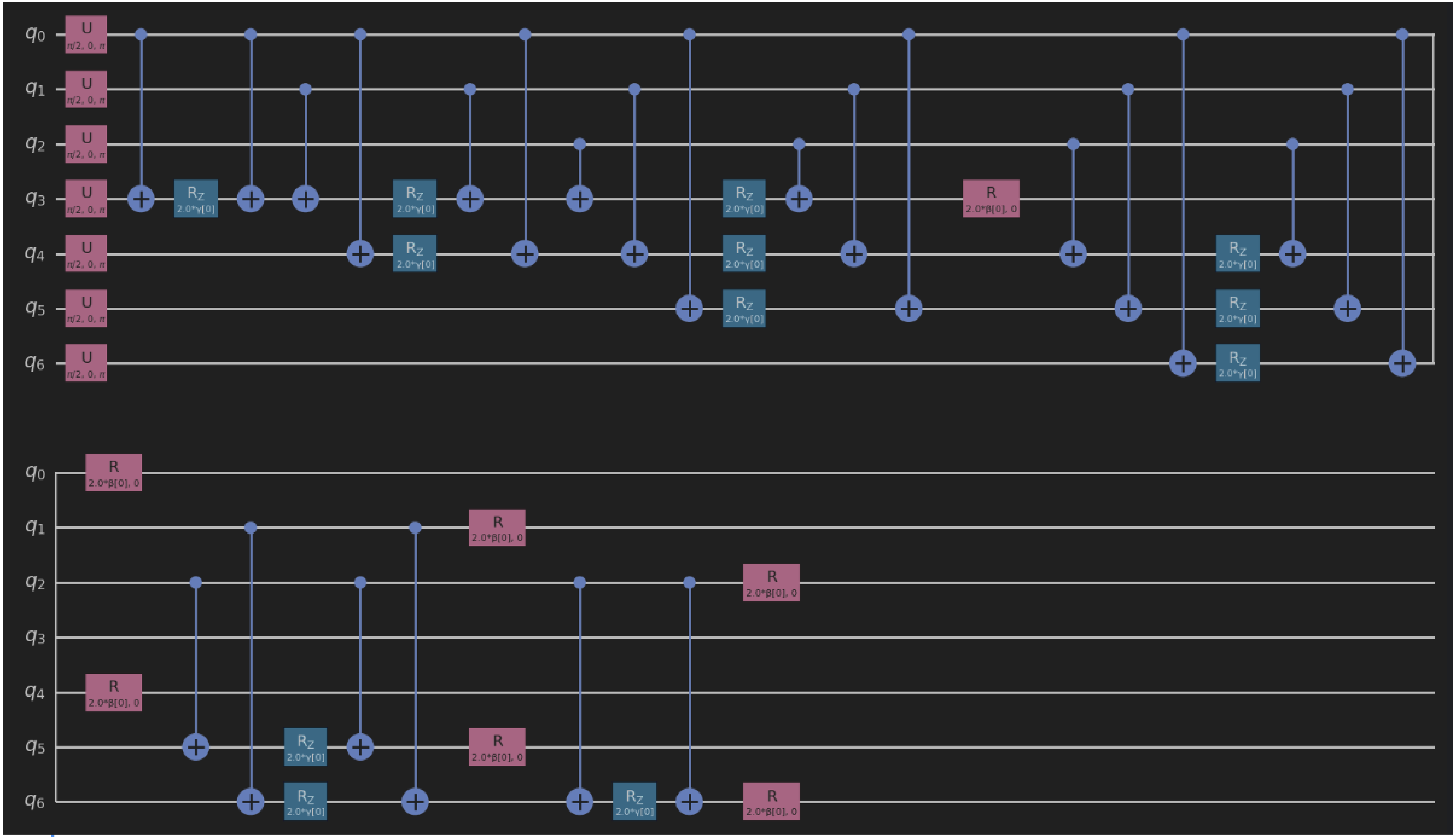
Acting on the i-th and j-th qubit:
C = $ZiZj\ |x0 \cdots xn\rangle = I \otimes \cdots \otimes Zi \otimes Zj \otimes \cdots \otimes I\ |x0 \cdots xn\rangle$
x: $i \in \{ 0 , 1 \} , i = 1 , \ldots , n$

['IIIZIIZ', 'IIIZIZI', 'IIIZZII', 'IIZIIIZ', 'IIZIIZI', 'IIZIZII', 'IZIIIIZ', 'IZIIIZI', 'IZIIZII', 'ZIIIIIZ', 'ZIIIIZI', 'ZIIIZII']

```python
def cost_func(params, ansatz, hamiltonian, estimator):

    pub = (ansatz, [hamiltonian], [params])
    result = estimator.run(pubs=[pub]).result()
    cost = result[0].data.evs[0]

    return cost
```




```python

pauli = []
def createSmp():
    smp = []
    for i in range(n + m):
        smp.append("I")
    return smp

for i in edges:
    smp = createSmp()
    for j in range(len(smp)):
        if j == i[0]:
            smp[j] = 'Z'
        if j == i[1]:
            smp[j] = 'Z'
    smp = smp[::-1]
    word = ''.join(smp)
    pauli.append(word)

print(pauli)
print(edges)

pauliList = []

for i in range(len(pauli)):
    pauliList.append((pauli[i], 1))

print(pauliList)
```



q0
q1
q2
q3
q4
q5
q6
U
π/2, 0, π
Rz
2.0*γ[0]
R
2.0*β[0], 0

Optimization

- Classic algorithm

```python
x0 = 2 * np.pi * np.random.rand(ansatz.num_parameters)
```

```python
res = minimize(cost_func, x0, args=(ansatz, hamiltonian, estimator), method="COBYLA")
res
```

```
 message: Optimization terminated successfully.
 success: True
  status: 1
     fun: -4.227116596992675
       x: [ 3.534e+00  2.858e+00]
    nfev: 39
   maxcv: 0.0
```

Result

```python
def convert_to_binary_states(quasi_dists, num_qubits):
    binary_dists = {}

    for index, probability in quasi_dists.items():
        binary_state = format(index, f'0{num_qubits}b')
        binary_dists[binary_state] = probability

    return binary_dists

stany = convert_to_binary_states(samp_dist, n + m)
max_key = max(stany, key=stany.get)
max_prob = stany[max_key]

print(max_prob)
print(max_key[::-1])
```

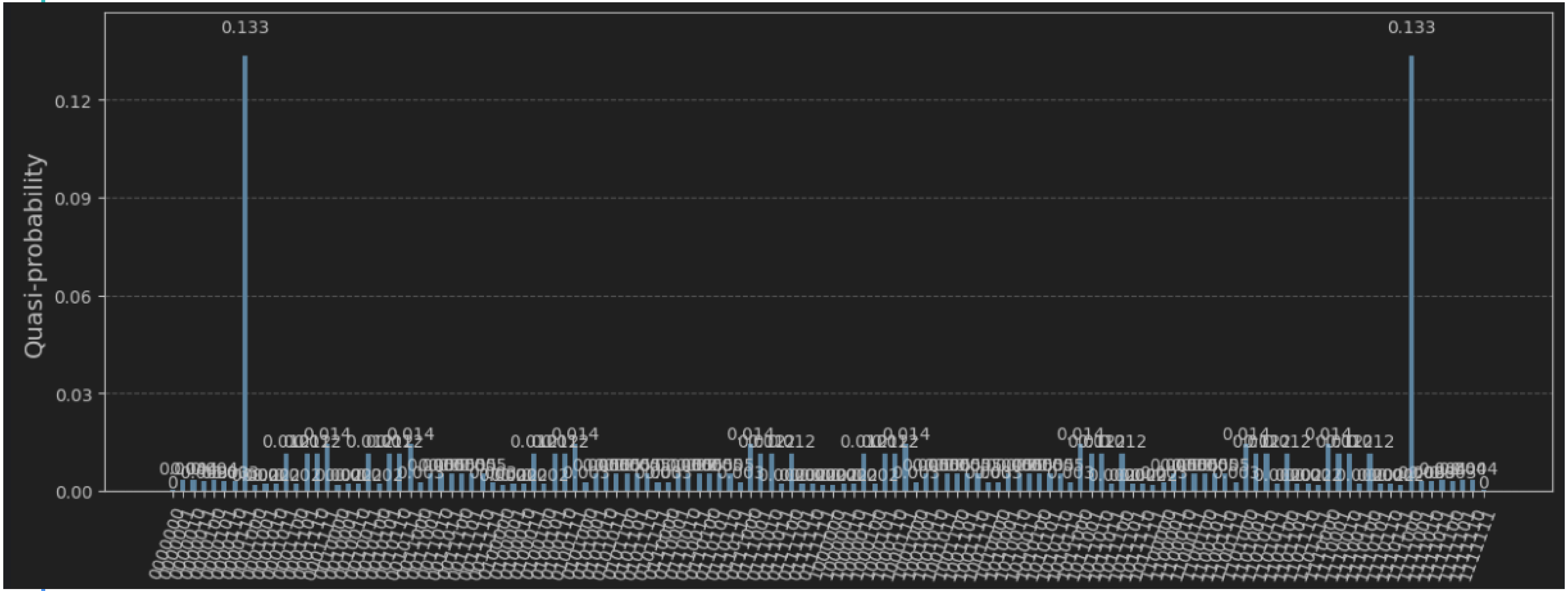
```
0.1334744403087406
1110000
```

Result

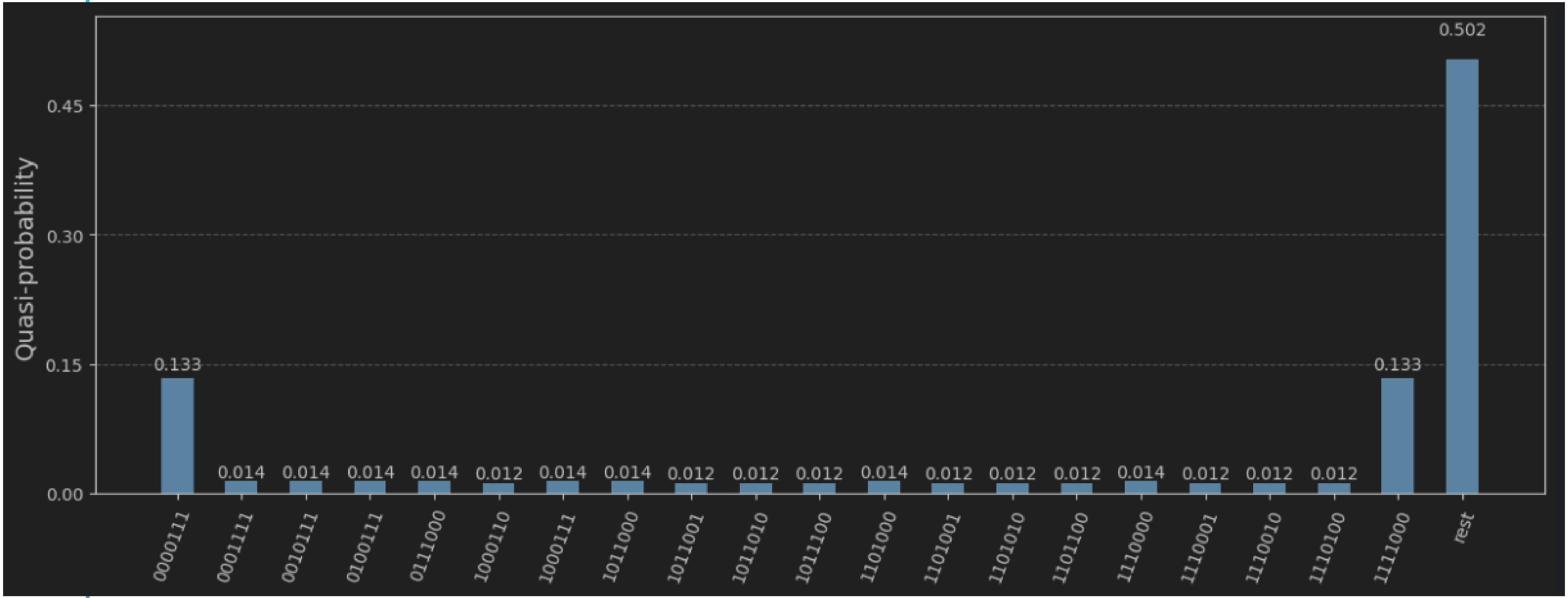
```
def calculate_cut(edges, partition_state):
    num_cuts = 0
    for (i, j) in edges:
        if partition_state[i] != partition_state[j]:
            num_cuts += 1
    return num_cuts

num_cuts = calculate_cut(edges, max_key[::-1])
print(f"Number of cuts: {num_cuts}")
```

```
Numer of cuts: 12
```

Result

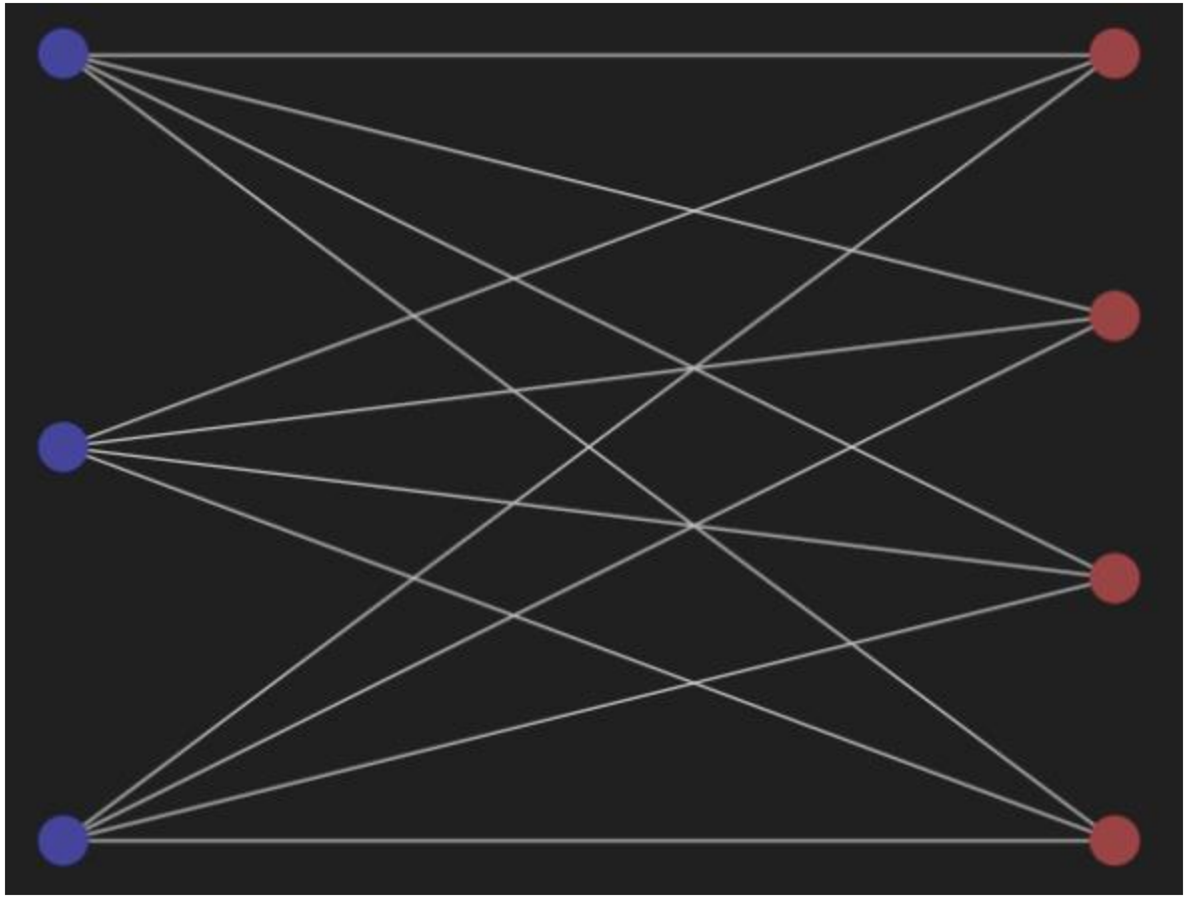


Result



Quasi-probability
0.00
0.15
0.30
0.45
0.133
0.014
0.014
0.014
0.014
0.012
0.014
0.014
0.012
0.012
0.012
0.014
0.012
0.012
0.012
0.014
0.012
0.012
0.012
0.133
0.502
0000111
0001111
0010111
0100111
0111000
1000110
1000111
1011000
1011001
1011010
1011100
1101000
1101001
1101010
1101100
1110000
1110001
1110010
1110100
1111000
rest

Result



More nodes