Quantum Computing
Progetto P7

Michele Dalla Chiara - VR464051
Davide Zampieri - VR458470

A.A. 2021 - 2022

1 Quantum Optimisation

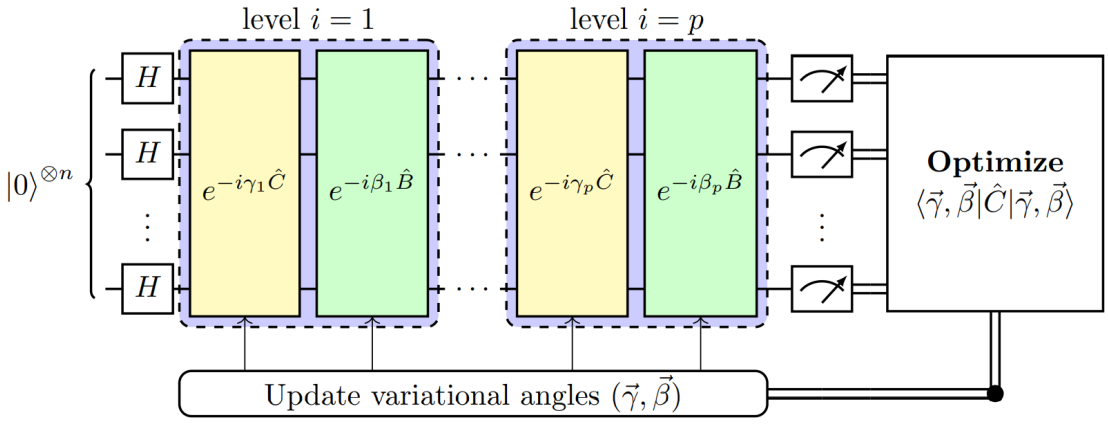
1.1 L'euristica QAOA

QAOA (Quantum Approximate Optimization Algorithm) è un'euristica utilizzabile per risolvere in tempo polinomiale vari problemi di ottimizzazione combinatoria definiti in termini di una matrice Hamiltoniana.

QAOA è un algoritmo ibrido, poiché contiene sia computazione classica che computazione quantistica. La parte quantistica è costituita da un circuito a p livelli, dove all'aumentare di p è possibile ottenere generalmente migliori approssimazioni della soluzione del problema codificato.

Infatti, per $p > 1$, la soluzione fornita da QAOA non può essere calcolata in modo efficiente su un computer classico, poiché la complessità computazionale scala in maniera doppiamente esponenziale in p.

Tutti i problemi NP-completi possono essere formulati in termini di ricerca del ground state di una matrice Hamiltoniana nella formulazione di Ising. QAOA mira a trovare questo stato applicando due Hamiltoniane, $\hat{B}$ e $\hat{C}$, in una sequenza alternata (di lunghezza p) ad una sovrapposizione di n qubit. Dopo la misurazione degli stati dei qubit, viene calcolato un costo che verrà minimizzato da un algoritmo di ottimizzazione classico (variando gli angoli $\vec{\gamma}$ e $\vec{\beta}$).



1.2 Il problema Exact Cover

I problemi di ottimizzazione combinatoria implicano la ricerca di un oggetto ottimale a partire da un insieme finito di oggetti, ad esempio ricercano una stringa di bit ottimale (composta da 0 e 1) in un insieme finito di stringhe di bit. Uno di questi problemi è il cosiddetto problema Exact Cover.

Data una collezione S di sottoinsiemi dell'insieme X, si dice Exact Cover il sottoinsieme S^* di S tale per cui ogni elemento di X è contenuto esattamente in uno dei sottoinsiemi in S^*. S^* deve perciò soddisfare le seguenti condizioni:

- L'intersezione di due sottoinsiemi qualsiasi in S^* deve essere vuota, cioè ogni elemento di X deve essere contenuto al massimo in uno dei sottoinsiemi in S^*.
- L'unione di tutti i sottoinsiemi in S^* deve essere X, cioè S^* deve coprire X.

Questo problema può essere mappato in una matrice Hamiltoniana di Ising avente la seguente forma:

$$\hat{C} = \sum_{i=1}^{n} h_i \hat{\sigma}_i^z + \sum_{i<j} J_{ij} \hat{\sigma}_i^z \hat{\sigma}_j^z$$

dove h_i, J_{ij} sono coefficienti reali e $\hat{\sigma}_i^z$ rappresenta l'operatore di Pauli Z applicato all'i-esimo qubit.

1.3 L'implementazione di QAOA

QAOA è un algoritmo variazionale che utilizza una matrice unitaria $U(\beta, \gamma)$ in cui i parametri β e γ, che rappresentano un'informazione classica, devono essere ottimizzati per fare in modo che lo stato quantistico in output dal circuito codifichi la soluzione del problema da risolvere. L'unitaria $U(\beta, \gamma)$ è composta da due matrici unitarie: $U(\beta) = e^{-i\beta\hat{B}}$ e $U(\gamma) = e^{-i\gamma\hat{C}}$, dove $\hat{B}$ è chiamata Mixing Hamiltonian e $\hat{C}$ è la matrice Hamiltoniana che rappresenta il problema.

La Mixing Hamiltonian ha generalmente la seguente forma:

$$\hat{B} = \sum_{i=1}^{n} \hat{\sigma}_i^x$$

in cui ciascun termine rappresenta una rotazione X applicata al corrispondente qubit.

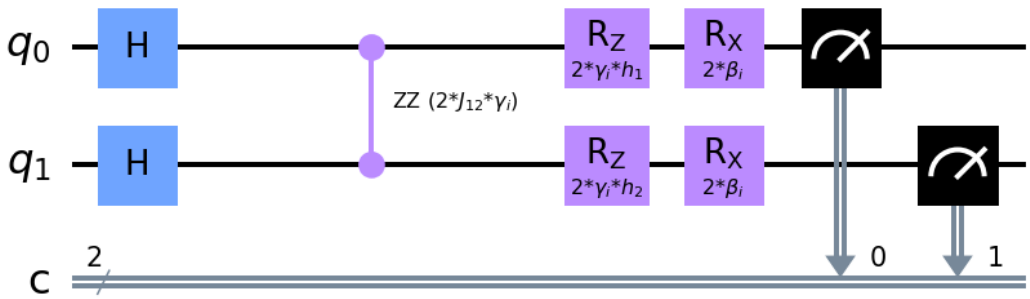
Esaminiamo ora come si presenta il circuito che implementa uno dei livelli usati da QAOA per risolvere il problema Exact Cover. Tale circuito sarà composto dai seguenti tre elementi:

- Preparazione dello stato iniziale, ossia di una sovrapposizione di tutti gli stati della base computazionale ottenuta applicando i gate Hadamard a partire da uno stato con tutti i qubit a zero.
- Applicazione dell'unitaria $U(\gamma)$ specifica per il problema; considerando $n = 2$, si ottiene la matrice Hamiltoniana di Ising $\hat{C} = h_1\hat{\sigma}_1^z + h_2\hat{\sigma}_2^z + J_{12}\hat{\sigma}_1^z\hat{\sigma}_2^z$.
- Applicazione dell'unitaria di Mixing $U(\beta)$.

1.3.1 Codifica del circuito in qiskit

[5]:
```
from qiskit import *
from qiskit.circuit import Parameter

beta = Parameter("$\\beta_i$")
gamma = Parameter("$\\gamma_i$")
J = {(0,1): Parameter("$J_{12}$")}
h = [Parameter("$h_1$"), Parameter("$h_2$")]

qc = QuantumCircuit(len(h), len(h))

# initial state
qc.h(list(range(len(h))))

# problem unitary
for i in range(len(h)):
    for j in range(i+1, len(h)):
        qc.rzz(2*gamma*J[(i,j)], i, j)
for i in range(len(h)):
    qc.rz(2*gamma*h[i], i)

# mixer unitary
for i in range(len(h)):
    qc.rx(2*beta, i)

qc.measure(list(range(len(h))), list(range(len(h))))

qc.draw(output='mpl', scale=2)
```

[5]:



Il passaggio successivo consiste nell'utilizzare un algoritmo di ottimizzazione classico allo scopo di trovare i parametri $\vec{\beta}$ e $\vec{\gamma}$ ottimali, ovvero tali da ridurre al minimo il valore atteso:

- Una funzione di costo viene valutata preparando e misurando ripetutamente $\vec{\beta}$ e $\vec{\gamma}$ su un processore quantistico.
- Viene utilizzato un ottimizzatore classico per ridurre al minimo il valore atteso, ossia per trovare gli angoli variazionali ottimali $\vec{\beta}^*$ e $\vec{\gamma}^*$.
- Se p è sufficientemente alto, $\vec{\beta}^*$ e $\vec{\gamma}^*$ corrispondono al ground state di $\hat{C}$ e costituiscono quindi la soluzione del problema di ottimizzazione.

1.3.2 Codifica di QAOA in qiskit

[1]:
```
from qiskit import *

# Making the quantum circuit
def create_qaoa_circ(J, h, theta):
    '''
    Funzione che crea il circuito QAOA andando a ripetere U(beta, gamma) p volte␣
 ↪con p=dim(beta/gamma)

    J:      dizionario con chiave (i,j) con i < j
    h:      lista di coefficienti
    theta:  lista di parametri
    '''

    # Indica quante volte bisogna ripetere l'alternanza B - C
    p = len(theta)//2

    # len(h) perché h è una lista di coefficienti, 1 per ogni qbit del circuito
    qc = QuantumCircuit(len(h), len(h))

    # parametri beta e gamma delle matrici B e C
    beta = theta[:p]
    gamma = theta[p:]

    # stato iniziale come H (0) con H che è Hadamard e (0) il vettore nullo
    qc.h(list(range(len(h))))

    for irep in range(0, p):
        # definizione del circuito corrispondente alla matrice C
        # che codifica il problema di ottimizzazione (exact cover)
        for i in range(len(h)):
            for j in range(i+1, len(h)):
                qc.rzz(2*gamma[irep]*J[(i,j)], i, j)
        for i in range(len(h)):
            qc.rz(2*gamma[irep]*h[i], i)
```

```
        # definizione del circuito corrispondente alla matrice mixer unitary B
        for i in range(len(h)):
            qc.rx(2*beta[irep], i)

    # misurazione per poter passare dal problema quantistico a quello classico
    qc.measure(list(range(len(h))), list(range(len(h))))

    return qc

# Cost of a solution (given as a bitstring)
def cost_function(solution, J, h):
    '''
    La funzione di costo genera un valore per la soluzione misurata dal circuito
    (prod_{i=1}{p} {e^(-i * beta_i* B) * e^(-i * gamma_i* C)}) (H^(xn) 0^(xn)).

    Il processo di minimizzazione della funzione di costo genera beta e gamma␣
 ↪migliori allo scopo di abbassare
    la funzione di costo al passo successivo con l'obiettivo ultimo di arrivare␣
 ↪ad ottenere il ground state.

    solution:   bitstring contenente la soluzione del problema
    J:          dizionario con chiave (i,j) con i < j
    h:          lista di coefficienti
    '''

    C = 0
    for i in range(len(h)):
        C +=  h[i] * int(solution[i])
    for (i,j) in J.keys():
        C += J[(i,j)] * int(solution[i]) * int(solution[j])
    return C

# Invertion of the counts, since qiskit shows the qubit states from the last one
def invert_counts(counts):
    return {k[::-1]:v for k, v in counts.items()}

# Compute expectation value based on measurement results
def compute_expectation(counts, J, h):
    '''
    La funzione calcola la media rispetto ai risultati delle misurazioni in␣
 ↪'counts'.

    counts: dizionario con chiave la stringa di bit e come valore il numero di␣
 ↪volte che è stata misurata
    J:      dizionario con chiave (i,j) con i < j
    h:      lista di coefficienti
```

```
    '''
    avg = 0
    sum_count = 0

    # per ognuna delle 2^n stringhe si calcola:  somma_di  [costo(bit_str) *␣
 ↪(#volte_misur / #shots_tot)]
    for bitstring, count in counts.items():
        obj = cost_function(bitstring, J, h)
        avg += obj * count
        sum_count += count

    return avg/sum_count

# Execute the circuit on the chosen backend
def get_expectation(J, h, shots=512):
    '''
    get_expectation è un funtore tale da ritornare la funzione che esegue il␣
 ↪circuito a partire dal parametro theta.
    In particolare get_expectation crea e mette a disposizione il backend␣
 ↪qasm_simulator proprio alla funzione che ritorna.

    J:      dizionario con chiave (i,j) con i < j
    h:      lista di coefficienti
    shorts: intero che indica quante sono le misurazione da effettuare
    '''
    backend = Aer.get_backend('qasm_simulator')
    backend.shots = shots

    def execute_circ(theta):
        '''
        Crea il circuito tramite la serie di parametri J, h e theta. Poi esegue␣
 ↪il circuito con qasm_simulator e
        richiama compute_expectation sulla variabile counts così da sapere␣
 ↪qual'è il valore medio rispetto alla
        variabile theta dei parametri.

        thata: lista di parametri
        '''
        qc = create_qaoa_circ(J, h, theta)
        counts = backend.run(qc, seed_simulator=10,
                             nshots=512).result().get_counts()
        counts = invert_counts(counts)
        return compute_expectation(counts, J, h)

    return execute_circ
```

1.4 Calcolo dei parametri per Exact Cover

Per due sottoinsiemi esistono quattro diverse istanze del problema Exact Cover. Ognuna di esse può essere codificata attraverso i coefficienti h_i e J_{ij} (di una matrice Hamiltoniana di Ising) riassunti nella tabella sottostante.

Problema	Sottoinsiemi	h_1	h_2	J_{12}	Soluzione
A	$S_1 = \{x_1, x_2\}, S_2 = \{x_1\}$	-1/2	0	1/2	$\|10\rangle$
B	$S_1 = \{x_1, x_2\}, S_2 = \{\}$	-1	0	0	$\|10\rangle$ o $\|11\rangle$
C	$S_1 = \{x_1\}, S_2 = \{x_2\}$	-1/2	-1/2	0	$\|11\rangle$
D	$S_1 = \{x_1, x_2\}, S_2 = \{x_1, x_2\}$	0	0	1	$\|10\rangle$ o $\|01\rangle$

Questi coefficienti sono definiti da

$$h_i = \frac{1}{2} \sum_{j=1}^{u} K_{ji} \left(\sum_{k=1}^{v} K_{jk} - 2 \right)$$

e

$$J_{ij} = \frac{1}{2} \sum_{k=1}^{u} K_{ki} K_{kj}$$

dove u indica la cardinalità di X mentre v indica la cardinalità di S e dove K_{ij} indica l'elemento alla posizione (i, j) di una matrice di incidenza che contiene 1 se $x_i \in S_j$ e 0 altrimenti.

Nella colonna Soluzione, i qubit rappresentano i sottoinsiemi S_i (1 significa che il sottoinsieme corrispondente appartiene alla soluzione).

1.4.1 Codifica dei problemi in qiskit

[8]:
```
def calculate_parameters(S, K, X):
    '''
    Calcola i parametri J_{ij} e h_{i} considerando l'istanza del problema␣
 ↪(Exact Cover). Ogni problema avrà i propri
    coefficienti J_{ij} e h_{i} che caratterizzeranno il problema.

    S:  lista di sottoinsiemi di X
    K:  matrice di incidenza
    X:  lista di oggetti
    '''
    J = {}
    h = []

    # calcolo i J_{ij}
    for i in range(len(S)):
        for j in range(i+1, len(S)):
            J[(i,j)] = 0
            for k in range(len(X)):
                J[(i,j)] += K[k][i]*K[k][j]
            J[(i,j)] *= 0.5
```

```
    # calcolo gli h_{i}
    for i in range(len(S)):
        _sum2 = 0
        for j in range(len(X)):
            _sum = 0
            for k in range(len(S)):
                _sum += K[j][k]
            _sum2 += K[j][i]*(_sum-2)
        hi = 0.5*_sum2

        h.append(hi)

    return J, h

def calculate_incidence_matrix(S, X):
    '''
    Calcola la matrice di incidenza K_{ij}.
    { K_{ij}=1      se c_{i} appartiene a V_{j}
    { K_{ij}=0      se c_{i} non appartiene a V_{j}

    S:  lista di sottoinsiemi di X
    X:  lista di oggetti
    '''
    res = []

    for ci in X:
        row = []
        for Vj in S:
            if ci in Vj:
                row.append(1)
            else:
                row.append(0)
        res.append(row)

    return res

# Optimize and analyze the result
from scipy.optimize import minimize
from qiskit.visualization import plot_histogram

def optimize(J, h, P=3):
    expectation = get_expectation(J, h)

    res = minimize(expectation,
                        [1.0] * 2 * P,
                        method='COBYLA')
```

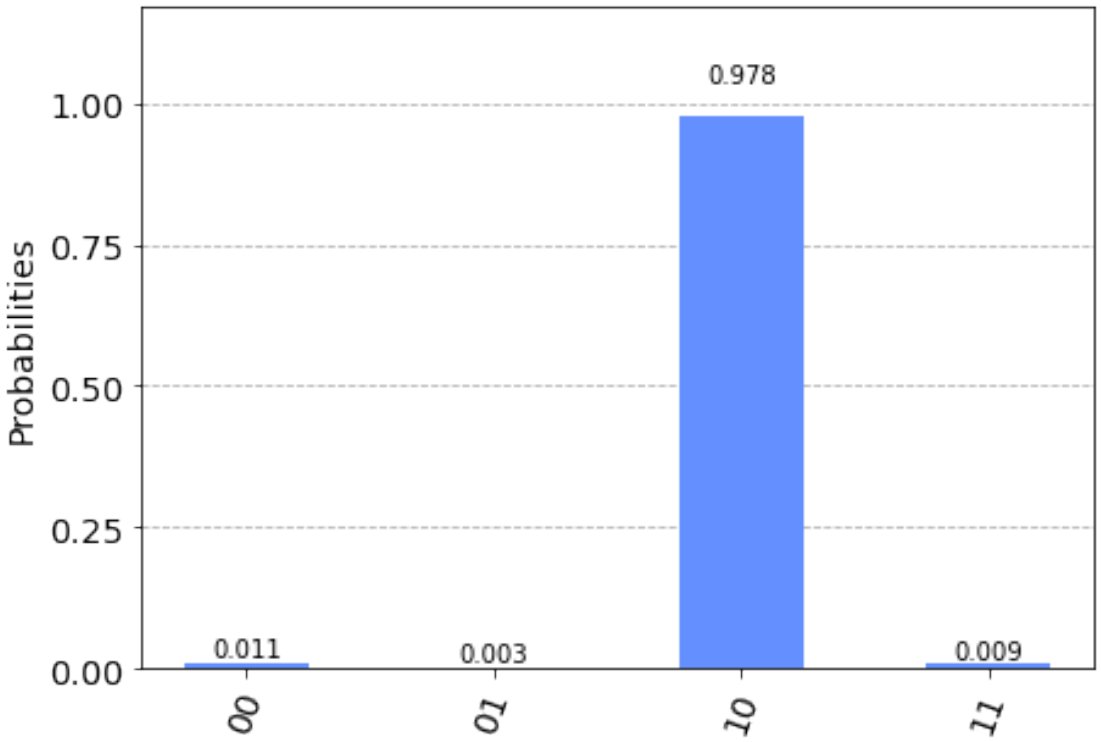
```
    print(res)

    backend = Aer.get_backend('aer_simulator')
    backend.shots = 512

    qc_res = create_qaoa_circ(J, h, res.x)

    counts = backend.run(qc_res, seed_simulator=10).result().get_counts()
    counts = invert_counts(counts)

    return counts
```

Problema A

[9]:
```
# Inizializzazione della variabile S contenente i sottoinsiemi di un insieme X␣
 ↪di oggetti
S = [{1,2}, {1}]

# Lista contenente l'insieme di oggetti
X = [1,2]

# Matrice di incidenza K
K = calculate_incidence_matrix(S, X)

J, h = calculate_parameters(S, K, X)
print("K:", K)
print("J:", J)
print("h:", h)

# Livelli del circuito quantistico
P = 3

counts = optimize(J, h, P)
plot_histogram(counts)
```

```
K: [[1, 1], [1, 0]]
J: {(0, 1): 0.5}
h: [-0.5, 0.0]
     fun: -0.48876953125
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 78
  status: 1
 success: True
       x: array([0.82965513, 1.41491612, 0.7442845 , 1.71654496, 0.19278985,
       1.37047721])
```

[9]:

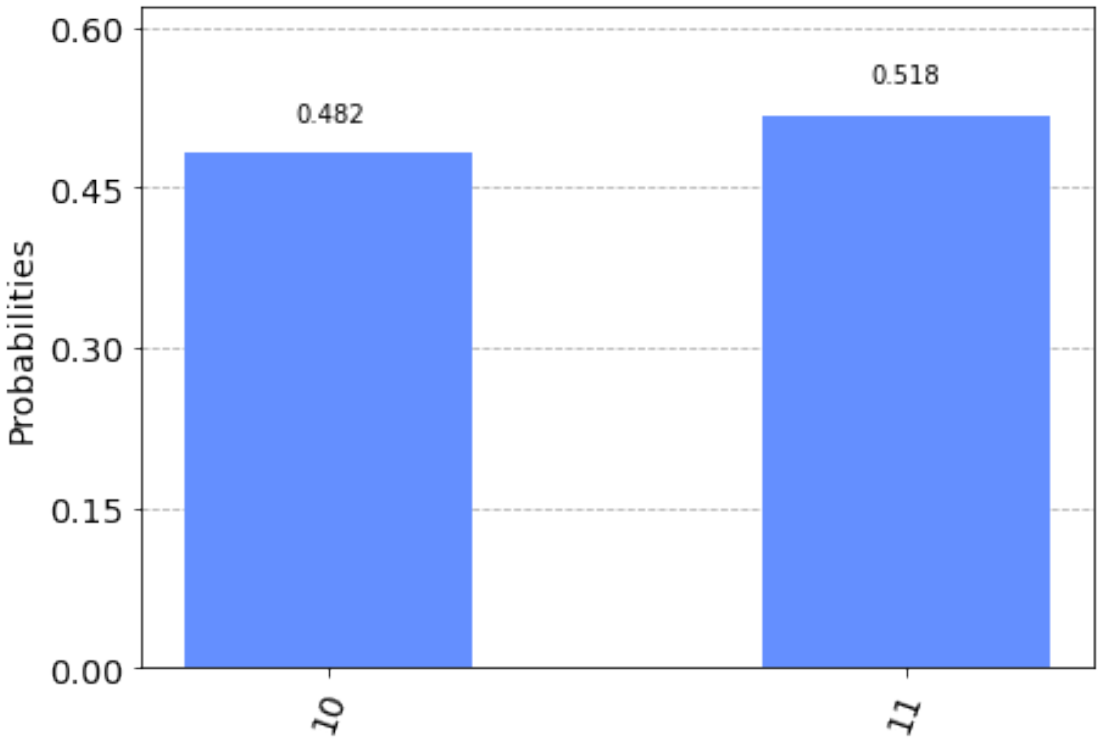


Problema B

[10]:
```
# Inizializzazione della variabile S contenente i sottoinsiemi di un insieme X
 ↪di oggetti
S = [{1,2}, {}]

# Lista contenente l'insieme di oggetti
X = [1,2]

# Matrice di incidenza K
K = calculate_incidence_matrix(S, X)

J, h = calculate_parameters(S, K, X)
print("K:", K)
print("J:", J)
print("h:", h)

# Livelli del circuito quantistico
P = 3

counts = optimize(J, h, P)
plot_histogram(counts)
```

```
K: [[1, 0], [1, 0]]
J: {(0, 1): 0.0}
h: [-1.0, 0.0]
     fun: -1.0
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 47
  status: 1
 success: True
       x: array([1.24745991, 0.98212886, 0.97544766, 0.96735079, 0.91679204,
       1.05000883])
```

[10]:



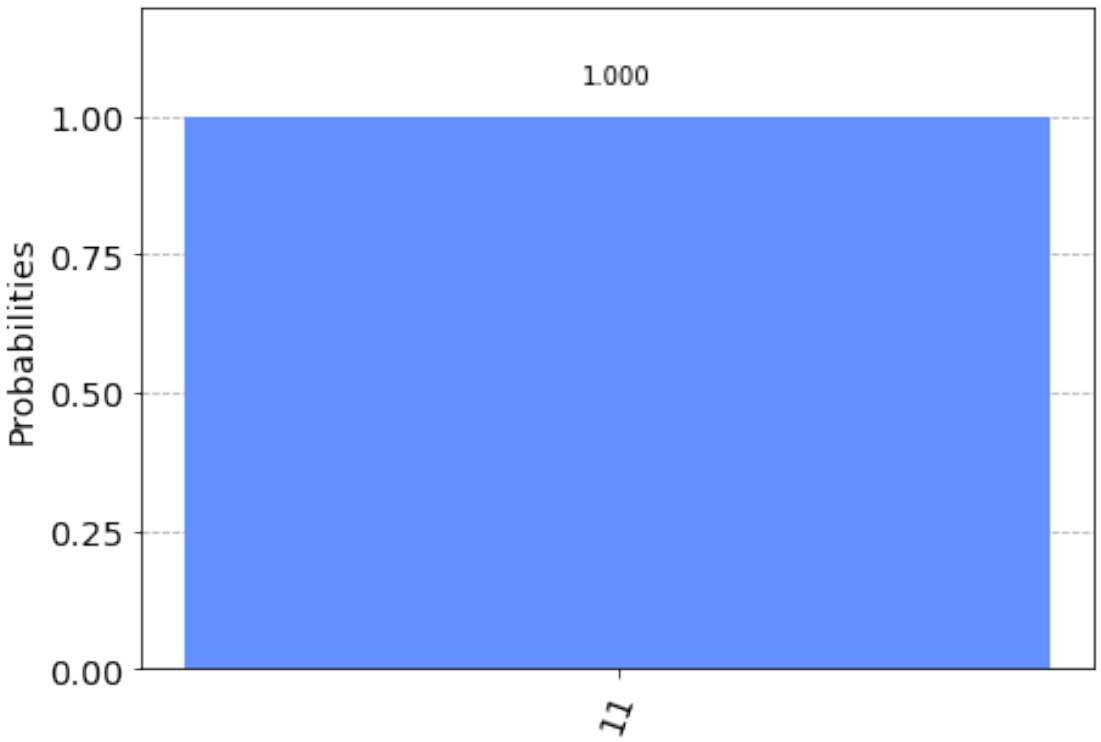
Problema C

[11]:
```
# Inizializzazione della variabile S contenente i sottoinsiemi di un insieme X␣
 ↪di oggetti
S = [{1}, {2}]

# Lista contenente l'insieme di oggetti
X = [1,2]

# Matrice di incidenza K
K = calculate_incidence_matrix(S, X)
```

```
J, h = calculate_parameters(S, K, X)
print("K:", K)
print("J:", J)
print("h:", h)

# Livelli del circuito quantistico
P = 3

counts = optimize(J, h, P)
plot_histogram(counts)
```

```
K: [[1, 0], [0, 1]]
J: {(0, 1): 0.0}
h: [-0.5, -0.5]
     fun: -1.0
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 67
  status: 1
 success: True
       x: array([1.88982433, 0.96892666, 0.94466872, 2.32782903, 1.85230386,
       0.92567929])
```

[11]:

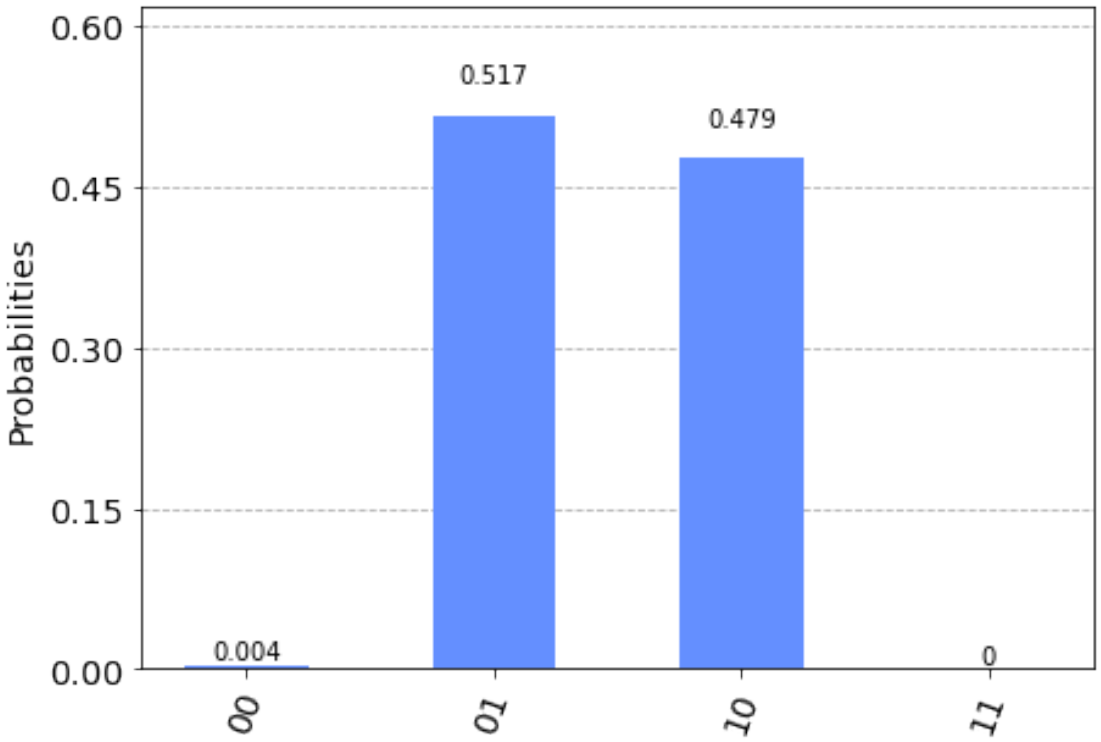


Problema D

[12]:
```
# Inizializzazione della variabile S contenente i sottoinsiemi di un insieme X di oggetti
S = [{1,2}, {1,2}]

# Lista contenente l'insieme di oggetti
X = [1,2]

# Matrice di incidenza K
K = calculate_incidence_matrix(S, X)

J, h = calculate_parameters(S, K, X)
print("K:", K)
print("J:", J)
print("h:", h)

# Livelli del circuito quantistico
P = 3

counts = optimize(J, h, P)
plot_histogram(counts)
```

```
K: [[1, 1], [1, 1]]
J: {(0, 1): 1.0}
h: [0.0, 0.0]
     fun: 0.0009765625
   maxcv: 0.0
 message: 'Optimization terminated successfully.'
    nfev: 45
  status: 1
 success: True
       x: array([1.        , 1.        , 1.        , 1.06984264, 0.89633223,
       1.        ])
```

[12]:



1.5 Riferimenti

- Tutorial di qiskit su QAOA - https://qiskit.org/textbook/ch-applications/qaoa.html
- QAOA per Exact Cover - https://arxiv.org/pdf/1912.10495.pdf
- Calcolo dei coefficienti - https://research.chalmers.se/publication/520988/file/520988_Fulltext.pdf
- Exact Cover to Ising - https://arxiv.org/pdf/2005.05275.pdf
- Ising formulations of many NP problems - https://arxiv.org/pdf/1302.5843.pdf