Risolvere Sudoku con l'algoritmo di Grover

01-Introduzione al problema

1.1-Sudoku

1.1.1-Regole

La traduzione di Sudoku è "numero unico", questo ha senso in quanto lo scopo del gioco è completare tutte le celle di una griglia 9x9 con numeri che non sono ripetuti. Spieghiamo meglio: per un insieme di numeri che vanno da 1 a 9, abbiamo una griglia composta da 9x9 celle, questo perché, appunto, dobbiamo utilizzare tutti i numeri ad una cifra che stanno nell'intervallo $[1-9]$ (estremi inclusi).
All'interno di ogni cella potremo scegliere di posizionare un solo numero a una cifra che l'intervallo $[1-9]$ ci fornisce, ma questo numero che scegliamo deve rispettare alcune regole fondamentali per la riuscita del Sudoku.
La prima regola che ci viene fornita è il fatto che il numero che scegliamo per una cella precisa non deve ripetersi per tutta la riga e per tutta la colonna di cui la cella fa parte. In aggiunta, oltre a ciò, dobbiamo, una volta conclusa l'intera riga o colonna, avere per tutte le celle che si susseguono sulla riga o colonna l'insieme di tutti i numeri dell'intervallo a una cifra rappresentabili. Ma le regole non finiscono qui, in quanto all'interno della tabella 9x9 vengono definite anche delle matrici 3x3, e in queste dovremo avere l'insieme di tutti i numeri presenti nell'intervallo $[1-9]$ senza essere mai ripetuti.
Solitamente la griglia iniziale viene popolata con alcune cifre, note come "indizio".

1.1.2-Problematiche

Possiamo dire ad oggi che il problema di risoluzione del Sudoku è un problema del tipo NP-completo. Cosa significa? Che l'esistenza di un algoritmo che risolva questo gioco logico in tempo polinomiale è ignoto.
Vediamo un piccolo esempio di calcolo della soluzione: se dovessi applicare un tipo di soluzione *Generate and Test*, il funzionamento sarebbe che, per ogni cella vuota, inserisco un numero e verifico che le regole del gioco vengano rispettate. Se non succede, riprovo con un'altra combinazione di numeri. Possiamo già intuire che la complessità di questa tipologia di soluzione sarà eccessiva. Infatti, se noi abbiamo una tabella 9x9, quindi un totale di 81 celle da riempire, supponiamo che 31 delle celle siano già complete con gli indizi che abbiamo citato precedentemente; ne rimangono da riempire esattamente 50. A questo punto, il numero di combinazioni possibili da testare è 9^{50}, circa $5 * 10^{47}$.
Ma quale sarebbe il tempo di risoluzione previsto? Supponendo che ci voglia 1 nano-secondo per formulare una possibile combinazione, il tempo previsto sarà:

- In un anno abbiamo $3.154 * 10^{16}$ nano-secondi.
  Se, per trovare una soluzione valida, nel caso peggiore, dovessi testare tutte le possibili soluzioni (cioè $5 * 10^{47}$), e per ogni soluzione dovessi impiegare 1 nano-secondo, abbiamo che:
- Per tutte le possibili soluzioni impiego un tempo di $5 * 10^{47}$ nano-secondi,

- che in anni si traduce in $5 * 10^{47} / 3.154 * 10^{16} = 10^{31}$ anni.
  Dunque, possiamo capire che ci vorrebbe veramente un tempo esagerato per trovare una possibile soluzione.

1.1.3-Perché utilizzare i computer quantistici?

I computer quantistici potrebbero offrire vantaggi significativi nella risoluzione del Sudoku, o di problemi NP-completi in generale, rispetto ai computer classici. Ecco alcune motivazioni per cui dovremmo considerare l'utilizzo dei computer quantistici per trovare una soluzione valida:

I computer quantistici sfruttano la **sovrapposizione quantistica**, che consente a un singolo qubit di rappresentare simultaneamente più stati. In pratica, invece di testare ogni possibile combinazione di numeri in sequenza come un computer classico, un computer quantistico potrebbe esplorare più combinazioni in parallelo. Ciò ridurrebbe drasticamente il tempo necessario per trovare una soluzione rispetto ai metodi tradizionali.

L'**entanglement quantistico** consente a qubit separati di essere correlati tra loro in modo che un'operazione su uno possa influenzare istantaneamente gli altri. Questo fenomeno potrebbe, in teoria, consentire una ricerca più rapida e una risoluzione più efficiente di problemi complessi, come il Sudoku, dove è necessario considerare molteplici vincoli e condizioni in parallelo.

In breve, per i problemi che richiedono enormi quantità di tempo per essere risolti dai computer classici (come nel caso delle combinazioni esponenziali nel Sudoku), un computer quantistico potrebbe essere in grado di ridurre drasticamente il tempo di esecuzione. In pratica, una soluzione che potrebbe richiedere migliaia di anni su un computer classico potrebbe, in teoria, essere trovata in tempi molto più brevi grazie ai benefici del calcolo quantistico.

02-Proposta della soluzione con un algoritmo quantistico

2.1-Vantaggi dell'algoritmo quantistico: Grover

Come già accennato nel capitolo precedente, uno dei vantaggi di sviluppare un algoritmo con un computer quantistico rispetto a uno classico è la velocità con cui quello quantistico effettua una ricerca all'interno di una struttura dati.
Se ci concentriamo sull'algoritmo di Grover, possiamo dire che esso può accelerare quadraticamente un problema di ricerca non strutturato.

Supponiamo di avere un elenco di n elementi e, tra di questi, dobbiamo individuarne uno che abbia determinate caratteristiche che soddisfano la nostra esigenza. Solitamente, in un computer classico, per la ricerca all'interno di una struttura dati, tipo array, nel caso peggiore

dobbiamo verificare tutti gli n elementi prima di trovare quello di cui necessitiamo. In media, invece, dovremmo verificare $n/2$ elementi prima di trovare quello desiderato.

Parlando invece di computer quantistici, abbiamo che la caratteristica dell'algoritmo di Grover è quella di implicare un'accelerazione quadratica per trovare l'elemento con le caratteristiche di cui abbiamo bisogno. Questo significa che possiamo trovarlo in circa $\sqrt{n}$.
Banalmente, possiamo dedurre che si tratta di un notevole risparmio di tempo, soprattutto se le grandezze delle strutture dati in cui dobbiamo effettuare la ricerca diventano sempre più grandi.

2.2-Generalizzazione del problema con Grover

Sappiamo che l'algoritmo di Grover viene diviso in 3 parti fondamentali. **La preparazione dello stato iniziale** si occupa di creare lo spazio che viene utilizzato per la ricerca della soluzione, in breve, un range di stati in cui si può trovare una soluzione al problema. L'**oracolo** è una funzione che identifica le soluzioni che vengono considerate corrette per il problema dato. Infine, l'**operatore di diffusione** si occupa di "ingrandire" le risposte trovate, in modo che vengano risaltate e successivamente misurate al termine dell'algoritmo.

2.2.1-Preparazione dello stato iniziale

Il primo passaggio per applicare l'algoritmo di Grover è la preparazione dello spazio in cui l'oracolo dovrà fare la ricerca per trovare la soluzione al nostro problema. Sostanzialmente, la nostra struttura dati deve contenere tutti i possibili stati/elementi in cui la nostra soluzione potrebbe trovarsi.

2.2.2-Creazione dell'oracolo

Il secondo passaggio, quello più importante, è l'oracolo. Questo ha il compito di contraddistinguere la soluzione finale da tutti gli altri stati. E come? Semplicemente applica una fase negativa agli stati della soluzione. Vediamo un esempio per un qualsiasi stato $|x\rangle$:

$$U_\omega|x\rangle = \begin{cases} |x\rangle & \text{if } x \neq w \\ -|x\rangle & \text{if } x = w \end{cases}$$

dove w è l'elemento che vogliamo come soluzione.
L'oracolo viene, dunque, rappresentato come una matrice diagonale, dove la voce corrispondente all'elemento che soddisfa la soluzione avrà una fase negativa.
La semplicità dell'oracolo sta nel fatto che è facilmente possibile convertire un problema in un oracolo. Infatti, ci sono molti problemi computazionali in cui è relativamente facile **verificare** una soluzione, ma in cui è veramente difficile **trovarne** una soluzione, come il Sudoku.

2.2.3-Operatore di diffusione

L'ultimo passaggio, dopo che l'oracolo ha contrassegnato la risposta negandola, si occupa di enfatizzare le soluzioni corrette, aumentando la probabilità di trovarle al termine dell'algoritmo. L'operatore di diffusione è, dunque, una combinazione di due riflessioni (una specifica operazione matematica che modifica l'ampiezza di uno stato invertendola rispetto a un determinato riferimento):

- **Riflessione intorno alla media delle ampiezze**: ogni stato ha un'ampiezza associata (un numero che determina la sua probabilità). L'operatore calcola la media di tutte queste ampiezze e riflette ogni ampiezza rispetto a questa media. Possiamo dedurre, quindi, che ampiezze basse diventano ancora più basse e quelle alte ancora più alte;
- **Riflessione rispetto allo stato iniziale**: serve a reindirizzare l'amplificazione delle ampiezze verso la struttura dello stato iniziale.

2.3-Grover per Sudoku 2x2

Come descritto precedentemente, abbiamo 3 fasi essenziali per sviluppare la soluzione. Nella parte della **preparazione dello stato iniziale**, dato che abbiamo un problema composto da $2x2$ celle, è ragionevole scegliere 4 bit (uno per ogni cella) che restituiranno un numero pari a 2^4 possibili soluzioni.

Come prima cosa, a questo punto, serve capire quante iterazioni l'algoritmo ha bisogno di fare per arrivare a uno stato che comprenda le soluzioni esatte per il Sudoku. Abbiamo bisogno di utilizzare la seguente formula:

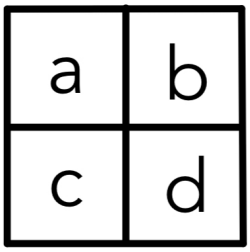
$$t \approx \left\lfloor \frac{\pi}{4} \sqrt{\frac{N}{M}} \right\rfloor,$$

dove N è il numero di tutte le possibili combinazioni di soluzione e M è il numero di soluzioni valide per il Sudoku.

Come ultima cosa, si deve pensare a inizializzare i 4 stati che occuperanno le relative 4 celle del gioco logico, utilizzando banalmente una porta di Hadamard, che ci riserverà per ogni stato, con probabilità $\frac{1}{\sqrt{2}}$, il qubit $|0\rangle$ o il qubit $|1\rangle$.

All'interno della parte dell'**oracolo**, abbiamo la necessità di trovare quali soluzioni sono corrette per il nostro sistema. Come descritto precedentemente, il Sudoku ha delle regole ben precise, che chiameremo clausole. Dobbiamo dunque controllare che esse vengano rispettate da tutti i qubit per arrivare a una prima soluzione.

Avendo un Sudoku della forma:

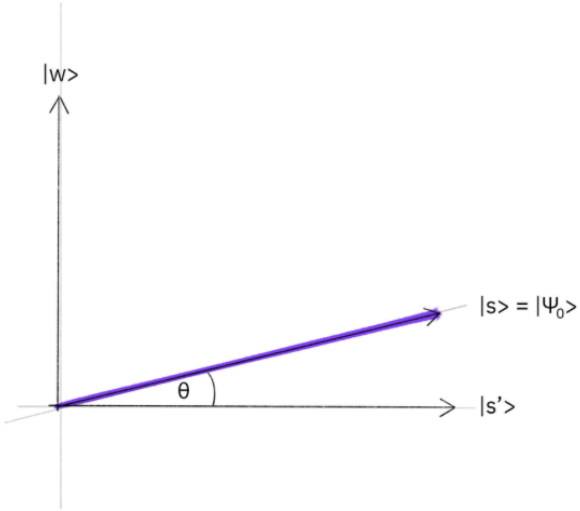


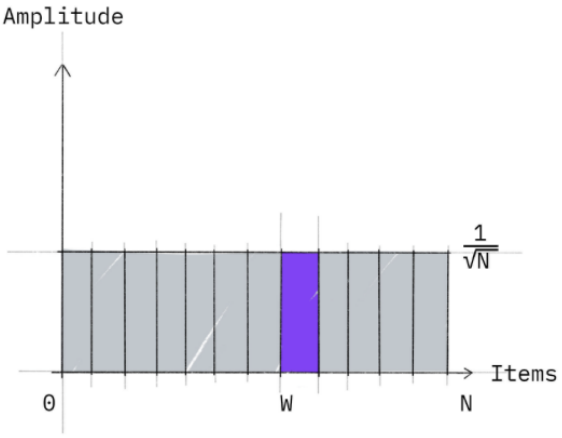
Dobbiamo applicare le regole del gioco. Quindi, avremo, ad esempio, un insieme di clausole dove $\{[a \neq b], [a \neq c], [b \neq d], [c \neq d]\}$. Queste sono le regole essenziali perché l'algoritmo ritorni una soluzione valida.

Abbiamo poi l'**operatore di diffusione**, che, successivamente all'oracolo, in cui vengono restituite, in questo caso, 2 possibili soluzioni valide, ha il compito di enfatizzarle per darle più probabilità di verificarsi.
Proprio in questa fase entra in gioco l'**azione** che l'**algoritmo di Grover** ci fornisce. Infatti, applichiamo in questa sezione le porte

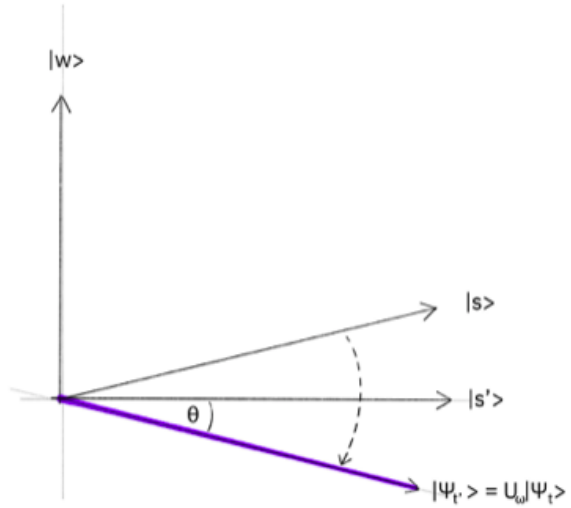
$$H^{\otimes n} Z_{OR} H^{\otimes n},$$

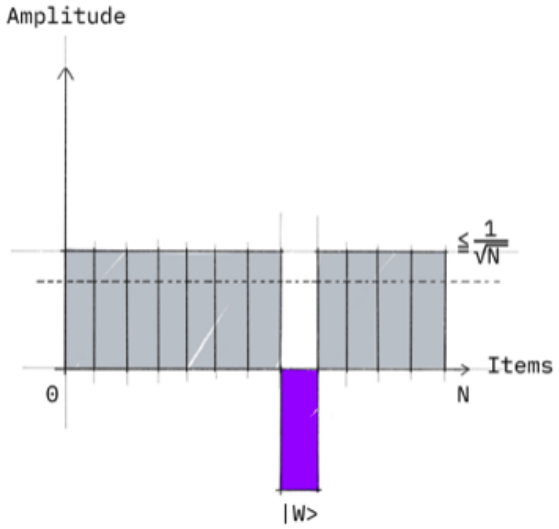
che ci servono per la procedura di *amplificazione dell'ampiezza*, un meccanismo geometrico di due riflessioni, che generano una rotazione in un piano bidimensionale.
Il primo passaggio si occupa della sovrapposizione uniforme dello stato iniziale, che si costruisce con le porte $H^{\otimes n}|0\rangle^n$, dove le ampiezze di tutte le soluzioni si trovano alla stessa altezza, come a destra nel seguente grafico, mentre a sinistra abbiamo un piano 2D attraversato dai vettori $|w\rangle$ (vettore della soluzione accettata) e $|s'\rangle$, dove
$|s\rangle = \sin\theta|w\rangle + \cos\theta|s'\rangle$ e $\sin\theta = \langle s|w\rangle = \frac{1}{\sqrt{N}}$.



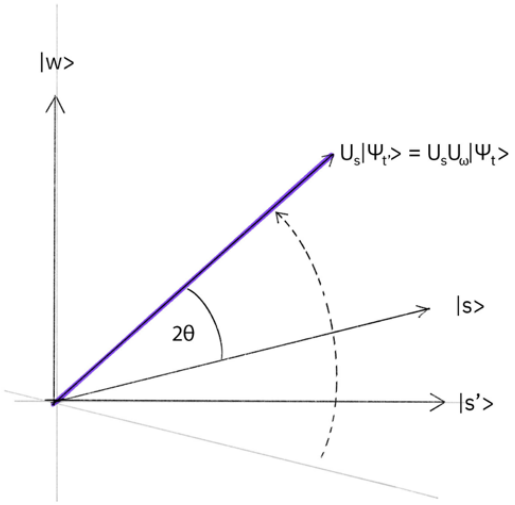


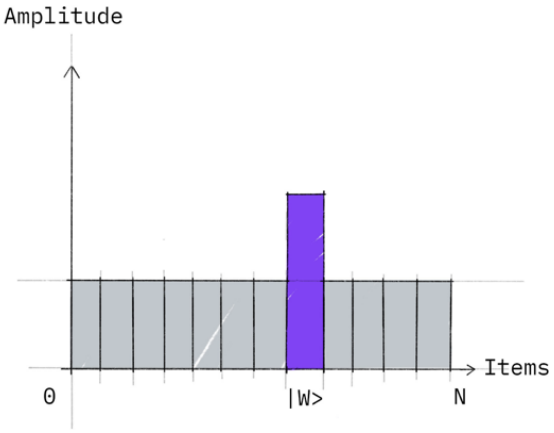
Ill secondo passaggio applica la riflessione dell'oracolo allo stato $|s\rangle$. Questa trasformazione viene eseguita sullo stato $|w\rangle$ (stato che esprime la soluzione esatta), facendolo diventare negativo. Questo implica che la media delle altezze di tutte le possibili soluzioni si abbassa. Vediamo i grafici:





Ora non ci resta che l'ultimo passaggio, in cui viene applicata la diffusione, che si occupa di riportare lo stato $|w\rangle$ nello stato originale (positivo). Tuttavia, dato che, sottraendo la media a tutte le soluzioni, incluso $|w\rangle$, quest'ultimo si trova nello stato negativo e gli viene sottratta la media, esso diventa ancora più negativo. Successivamente, riportandolo nello stato originale (positivo), esso avrà un'ampiezza più amplificata rispetto a tutti gli altri stati che non sono vere soluzioni.





03-Implementazione in Qiskit

Il codice di riferimento mostra un risolutore di un Sudoku, più precisamente un quadrato latino, $2x2$ introducendo un modello di rumore che prevede la depolarizzazione, il rilassamento termico e l'errore sulla misura.

Per ottenere tutte le possibili soluzioni del gioco logico(2 in questo caso), il codice implementa l'algoritmo di Grover precedentemente trattato.

3.1-Panoramica generale

Il codice modella in modo non modulare la risoluzione di un sudoku S con $n = 2$ celle per lato con un alfabeto:

$$\Sigma = \{0, 1\}$$

Nella dashboard utente vi sono dei flags che permettono di configurare il risolutore cambiando i rumori che si vogliono introdurre, le iterazioni di calcolo e iterazioni di Grover.

> Per Grover è possibile inserire 0 per lasciare calcolare il numero di iterazioni suggerite al software secondo la legge descritta nel capitolo 2.3-Grover per Sudoku 2x2.

Quindi, dopo aver preparato $n \times n$ qubits che rappresentano le celle, e quindi lo spazio di una possibile combinazione del sudoku $qubits = \{q_1, q_2, q_3, q_4\}$ e 4 ancille $Ancille = \{an_1, an_2, an_3, an_4\}$ che rappresentano i risultati delle condizioni dell'oracolo, il circuito inizializza i qubits in superposition tramite la porta H e procede attraverso l'algoritmo di Grover sottoponendo i qubit all'oracolo.

$$S = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

L'oracolo, procede confrontando tutte le possibile coppie (ab, bc, cd, bd) attraverso porta "XOR" costruita con l'applicazione di una porta X (NOT) al target, seguita da una porta MCX (restituisce l'ancilla negata se tutti i [control value] sono uguali a 1) e infine applica nuovamente la porta X per riportare il valore target all'origine. Queste operazioni sono utili per capire se i qubit che confronto sono uguali.
Di seguito viene riportato la parte di codice che si occupa di definire l'oracolo come descritto:

```
# Compare two qubits using an ancilla qubit
def compare(qc, control_qubit, target_qubit, ancilla_qubit):
    """Perform controlled comparison between two qubits."""
    qc.x(target_qubit)
    qc.mcx([control_qubit, target_qubit], ancilla_qubit)
    qc.x(target_qubit)

# Perfomr a XOR operation to two given qubits and returns result in ancilla
def XOR(qc, qubit_1, qubit_2, ancilla):
    """Perform XOR operation using ancilla qubits."""
        compare(qc, qubit_1[0], qubit_2[0], ancilla[0])
    compare(qc, qubit_2[0], qubit_1[0], ancilla[1])
    qc.cx(ancilla[0], ancilla[1])
    qc.barrier()

# Define the oracle for Grover's algorithm
def oracle(qc):
    """Mark the desired states using XOR gates."""
    XOR(qc, qubits[0], qubits[1], ancilla_ab)
    XOR(qc, qubits[0], qubits[2], ancilla_ac)
    XOR(qc, qubits[1], qubits[3], ancilla_bd)
    XOR(qc, qubits[2], qubits[3], ancilla_cd)
```

Come visto nel codice, il susseguirsi di "porte" XOR, ci serve per garantire che tutte le clausole di cui abbiamo parlato nel capitolo 2.3-Grover per Sudoku 2x2 vengano rispettate per ottenere uno stato dal circuito che sia una soluzione valida per il Sudoku.

In questa fase vengono quindi testate tutte le possibili combinazioni N_k che appartengono allo spazio delle combinazioni N, con k equivalente al numero di combinazioni possibili.

$$k = n! \cdot (n-1)!$$

$$N_k \in N \Leftrightarrow N_k \cap \{N - N_k\} = \emptyset$$

dove ad esempio nel nostro caso

$$N = \{N_0 = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, N_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \ldots, N_k = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}\}.$$

Tuttavia, essendo interessati solo allo spazio delle soluzioni accettabili M, condizioniamo i valori tramite un diffusore, il quale praticherà un'inversione rispetto la media, che andrà ad amplificare le misurazioni delle soluzioni accettabili.

```
# Perform inversion about the mean
def inversion_about_mean(qubits, target_qubit):
    """Perform inversion about the mean operation."""
    qc.h(target_qubit[0])
    qc.mcx([qubit for qubit in qubits if qubit != target_qubit],
target_qubit[0])
    qc.h(target_qubit[0])

# Diffuser operation for Grover's algorithm
def diffuser(qc):
    """Apply the diffuser transformation for Grover's algorithm."""
    apply_superposition(qubits)
    flip_qubits(qubits)

    inversion_about_mean(qubits, qubits[0])

    flip_qubits(qubits)
    apply_superposition(qubits)

    qc.barrier()
```

Questo è ottenuto applicando l'azione di Grover definita nel capitolo 2.3-Grover per Sudoku 2x2 dove appunto andiamo a dedicare la porta H riportando i qubits in una sovrapposizione uniforme e successivamente X nega tutti i qubits, a questo punto non ci resta che inverte il valore di probabilità di ogni stato rispetto alla media, amplificando gli stati precedentemente marcati dall'oracolo applicando le porte nel seguente ordine - H(applicata al qubit target trasforma lo stato in una base di lavoro), MCX(introduce un'inversione condizionale sugli stati, questo contribuisce a riflettere il vettore di stato rispetto alla media) e H(ripristina il qubit target nella base standard) - ripristino lo stato iniziale dei qubit applicando all'inverso le due porte iniziali H e X.

3.2-Rumore

Siccome l'obbiettivo è quello di creare una simulazione di un caso reale, prima della stampa dei risultati viene costruito e applicato un modello di rumore per simulare le condizioni reali di un circuito quantistico.

3.2.1-Modello dell'errore

Il codice di riferimento, permette all'utente di scegliere un modello puro, quindi senza errori o scegliere fra depolarizzazione tramite una probabilità, che rappresenta una variazione casuale di uno stato di un qubit per via delle imperfezioni hardware delle porte; oppure un rilassamento $T1$ e $T2$ che rappresentano rispettivamente qubits che perdono lo stato di eccitazione o fluttuazioni dovute all'interazione con l'ambiente circostante.

```
# --- noise: DEPOLARIZATION ---
if depolarization_noise_on:
    one_qubit_error = depolarizing_error(param=0.02, num_qubits=1)
    two_qubit_error = depolarizing_error(param=0.03, num_qubits=2)

    custom_noise_model.add_all_qubit_quantum_error(one_qubit_error, ['x',
'h', 'u3'])
    custom_noise_model.add_all_qubit_quantum_error(two_qubit_error, ['cx'])

# --- noise: RELAXATION ---
if relaxation_noise_on:
    # Apply relaxation error only to 1-qubit gates (not multi-qubit gates)
    T1s = np.random.normal(50e3, 10e3, 4)
    T2s = np.random.normal(70e3, 10e3, 4)

    T2s = np.array([min(T2s[j], 2 * T1s[j]) for j in range(4)])

    # Instruction times (in nanoseconds)
    time_u1 = 0   # virtual gate
    time_u2 = 50  # (single X90 pulse)
    time_u3 = 100 # (two X90 pulses)
    time_cx = 300
    time_reset = 1000  # 1 microsecond
    time_measure = 1000 # 1 microsecond

    # QuantumError objects
    errors_reset = [thermal_relaxation_error(t1, t2, time_reset) for t1, t2
in zip(T1s, T2s)]
    errors_measure = [thermal_relaxation_error(t1, t2, time_measure) for t1,
t2 in zip(T1s, T2s)]
    errors_u1  = [thermal_relaxation_error(t1, t2, time_u1) for t1, t2 in
zip(T1s, T2s)]
    errors_u2  = [thermal_relaxation_error(t1, t2, time_u2) for t1, t2 in
zip(T1s, T2s)]
    errors_u3  = [thermal_relaxation_error(t1, t2, time_u3) for t1, t2 in
zip(T1s, T2s)]
    errors_cx = [[thermal_relaxation_error(t1a, t2a, time_cx).expand(
                 thermal_relaxation_error(t1b, t2b, time_cx))
                 for t1a, t2a in zip(T1s, T2s)]
```

```
                for t1b, t2b in zip(T1s, T2s)]

    for j in range(4):
        custom_noise_model.add_quantum_error(errors_reset[j], "reset", [j])
        custom_noise_model.add_quantum_error(errors_measure[j], "measure",
[j])
        custom_noise_model.add_quantum_error(errors_u1[j], "u1", [j])
        custom_noise_model.add_quantum_error(errors_u2[j], "u2", [j])
        custom_noise_model.add_quantum_error(errors_u3[j], "u3", [j])
        for k in range(4):
            custom_noise_model.add_quantum_error(errors_cx[j][k], "cx", [j,
k])
```

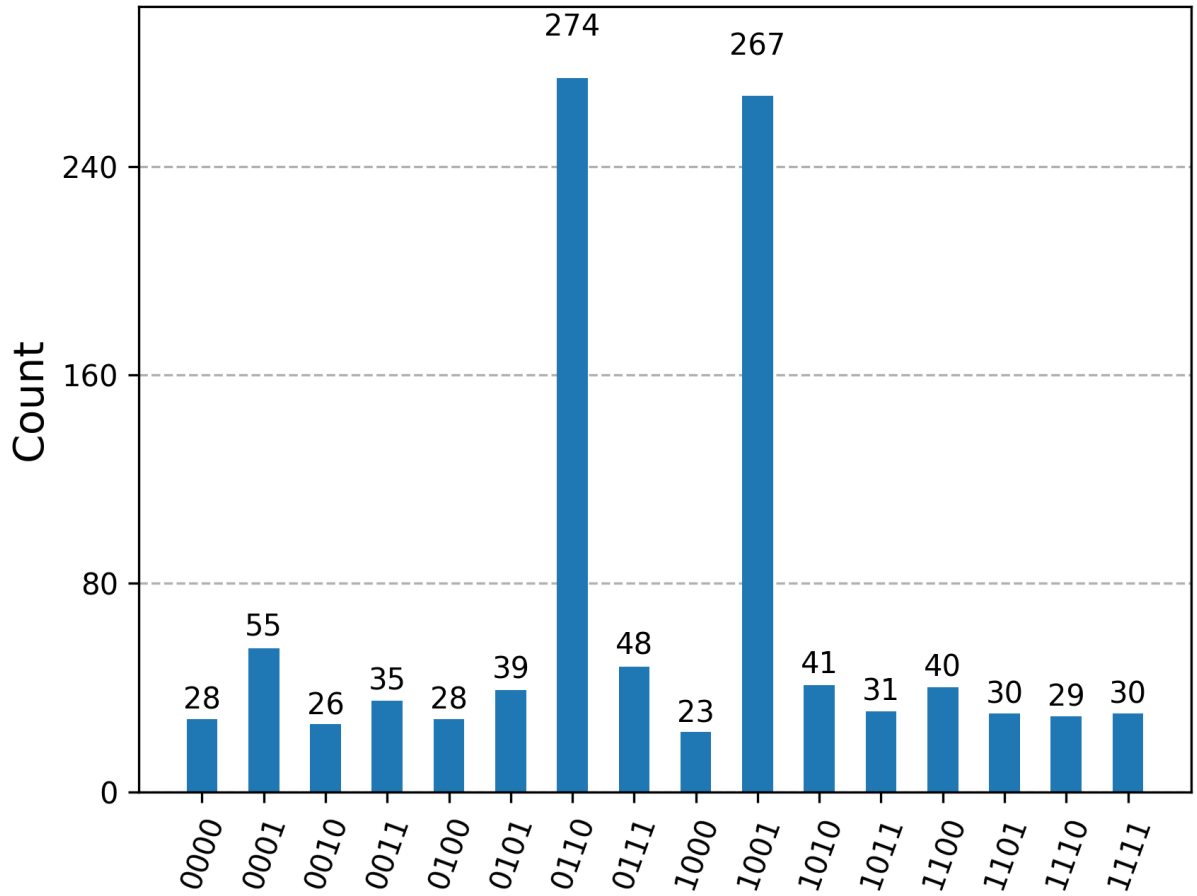
Inoltre, è possibile aggiungere ai modelli precedenti, un errore relativo alla lettura degli stati dei qubits che si verificano per imperfezioni hardware.

```
# --- noise: READOUT ---
if measurement_error_on:
    readout_error_a = ReadoutError([[0.90, 0.10], [0.05, 0.95]])
    readout_error_b = ReadoutError([[0.90, 0.10], [0.05, 0.95]])
    readout_error_c = ReadoutError([[0.90, 0.10], [0.05, 0.95]])
    readout_error_d = ReadoutError([[0.90, 0.10], [0.05, 0.95]])

    # Apply readout errors to measurements for each qubit
    custom_noise_model.add_readout_error(readout_error_a, [0])
    custom_noise_model.add_readout_error(readout_error_b, [1])
    custom_noise_model.add_readout_error(readout_error_c, [2])
    custom_noise_model.add_readout_error(readout_error_d, [3])
```

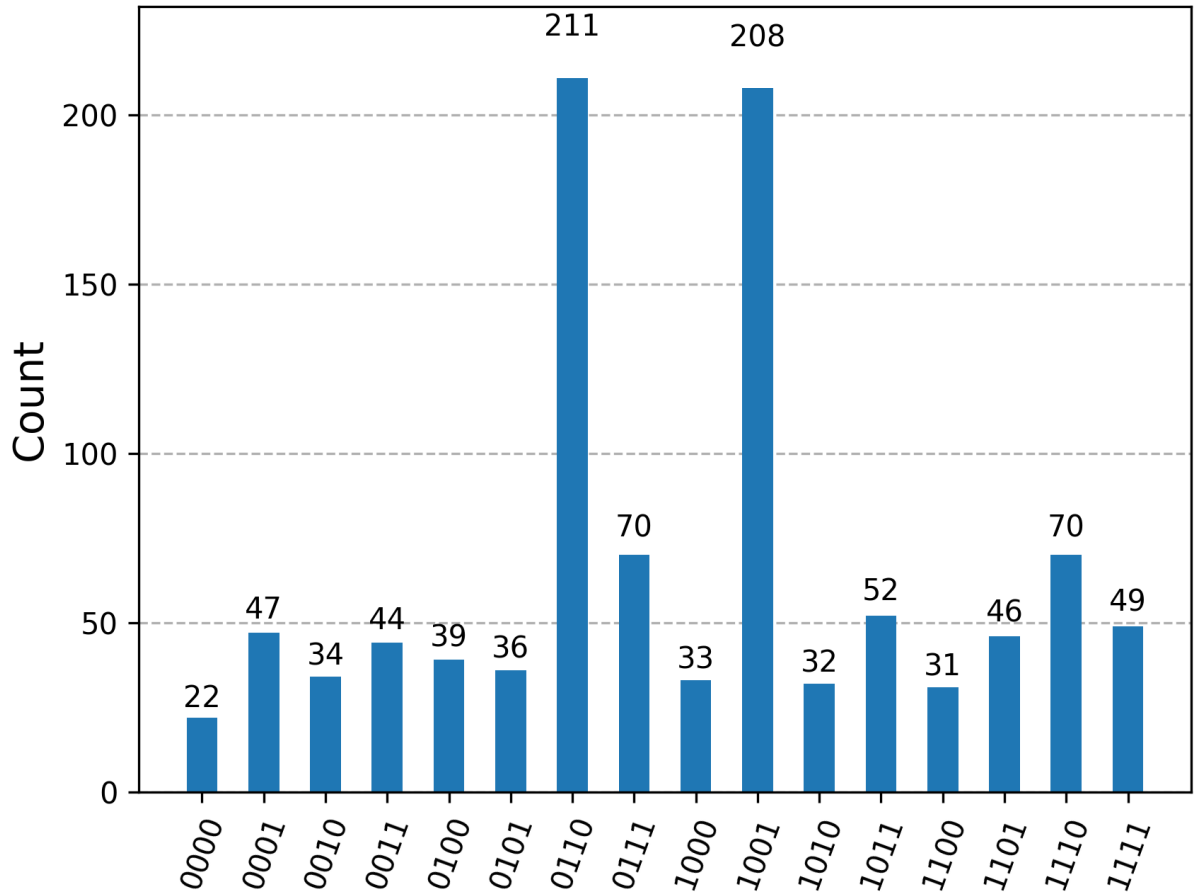
3.2.2-Comportamento del Rumore

Prenderemo ora in esame alcuni risultati notevoli al fine di mostrare la variazione sull'output che gli errori provocano sui dati.

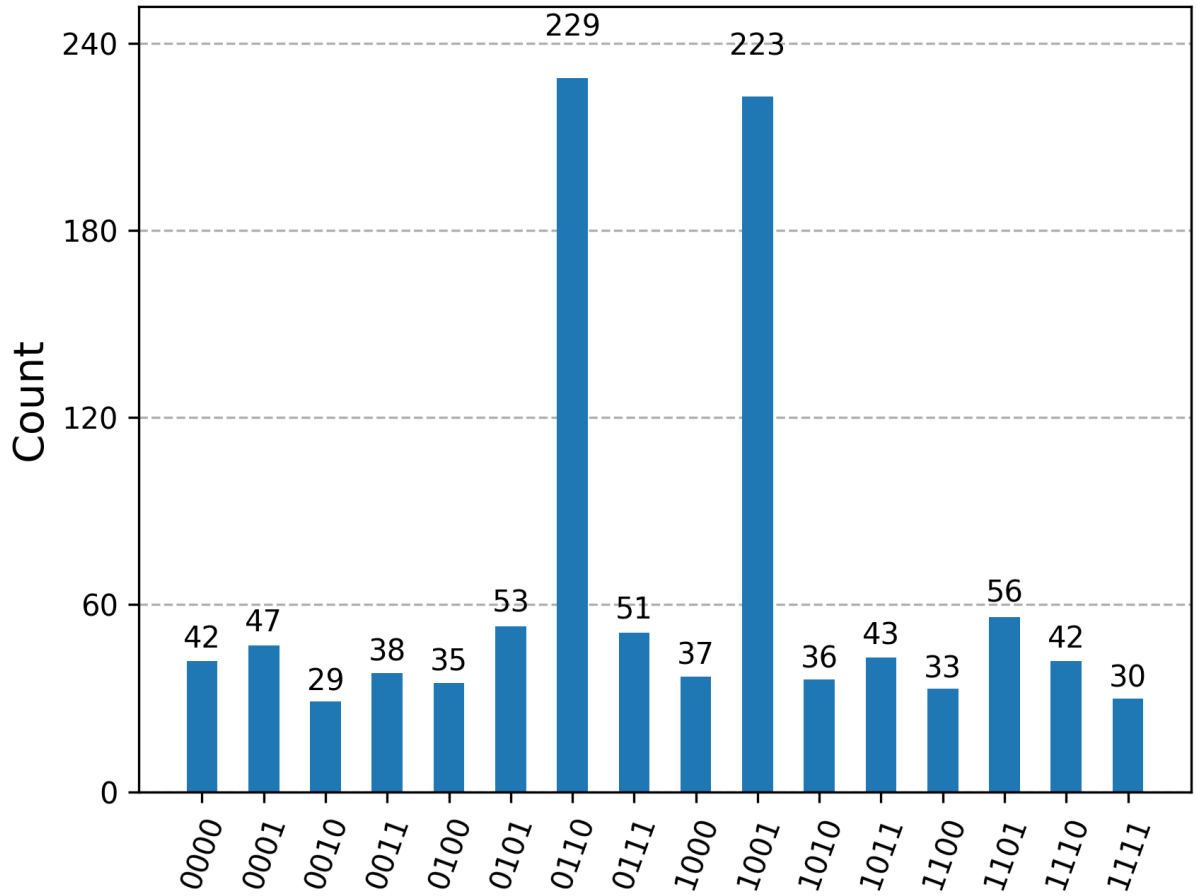
```
# Flags for enabling specific errors noises
depolarization_noise_on = False # Toggle depolarization noise
relaxation_noise_on = False     # Toggle relaxation noise
measurement_error_on = False    # Toggle measurement error

# Simulation configurations
simulation_shots = 1024       # Number of shots for the simulation
grover_iterations = 2         # Set 0 to use the suggested optimal value
```



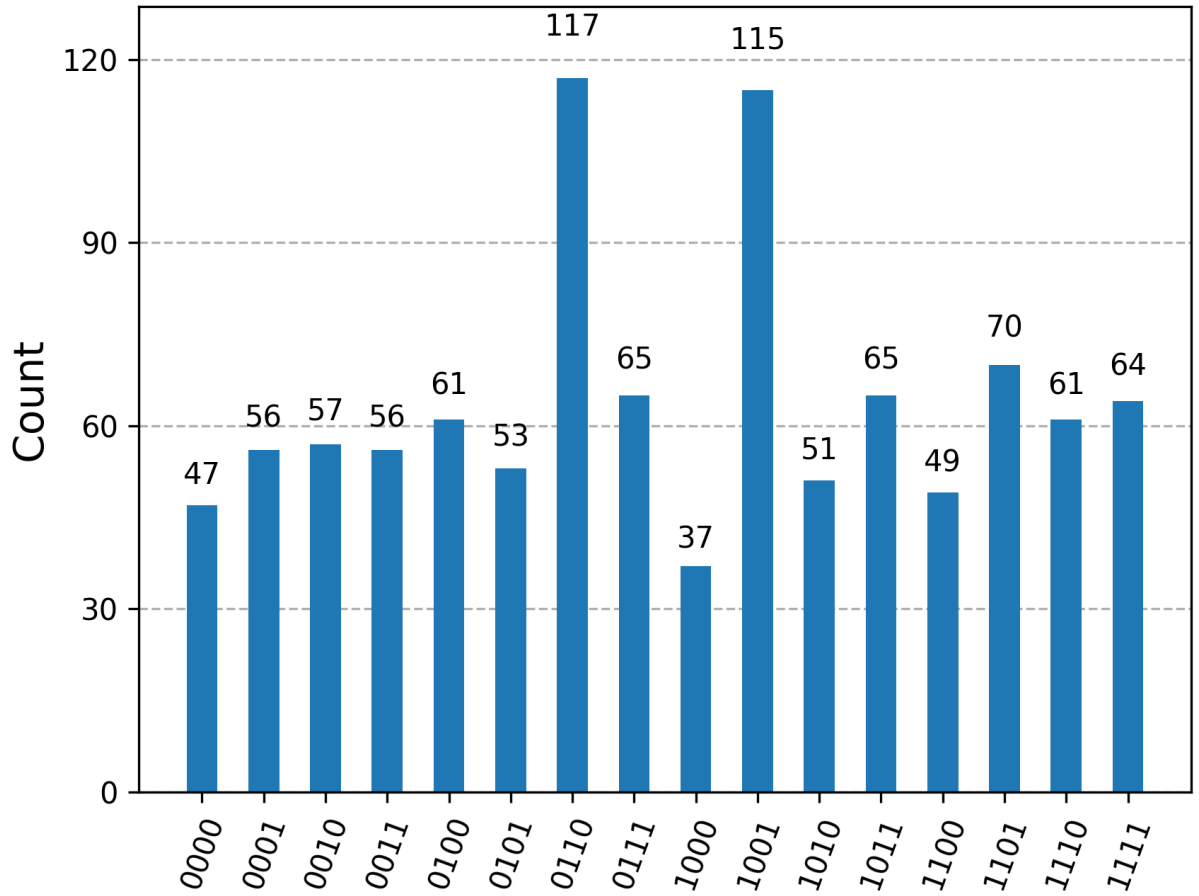

```
# Flags for enabling specific errors noises
depolarization_noise_on = False # Toggle depolarization noise
relaxation_noise_on = False     # Toggle relaxation noise
measurement_error_on = True     # Toggle measurement error

# Simulation configurations
simulation_shots = 1024       # Number of shots for the simulation
grover_iterations = 2         # Set 0 to use the suggested optimal value
```




```
# Flags for enabling specific errors noises
depolarization_noise_on = False # Toggle depolarization noise
relaxation_noise_on = True      # Toggle relaxation noise
measurement_error_on = True     # Toggle measurement error

# Simulation configurations
simulation_shots = 1024       # Number of shots for the simulation
grover_iterations = 2         # Set 0 to use the suggested optimal value
```




```
# Flags for enabling specific errors noises
depolarization_noise_on = True # Toggle depolarization noise
relaxation_noise_on = False    # Toggle relaxation noise
measurement_error_on = True    # Toggle measurement error

# Simulation configurations
simulation_shots = 1024        # Number of shots for the simulation
grover_iterations = 2          # Set 0 to use the suggested optimal value
```



Risulta evidente che il caso peggiore lo troviamo quando errori di misura e di depolarizzazione si combinano, mentre il caso migliore rimane comunque quello dove gli errori non vengano applicati. Ovviamente nel nostro caso il risultato non è compromesso, ma è necessario prenderne atto.

3.2.3 - Mitigazione degli errori

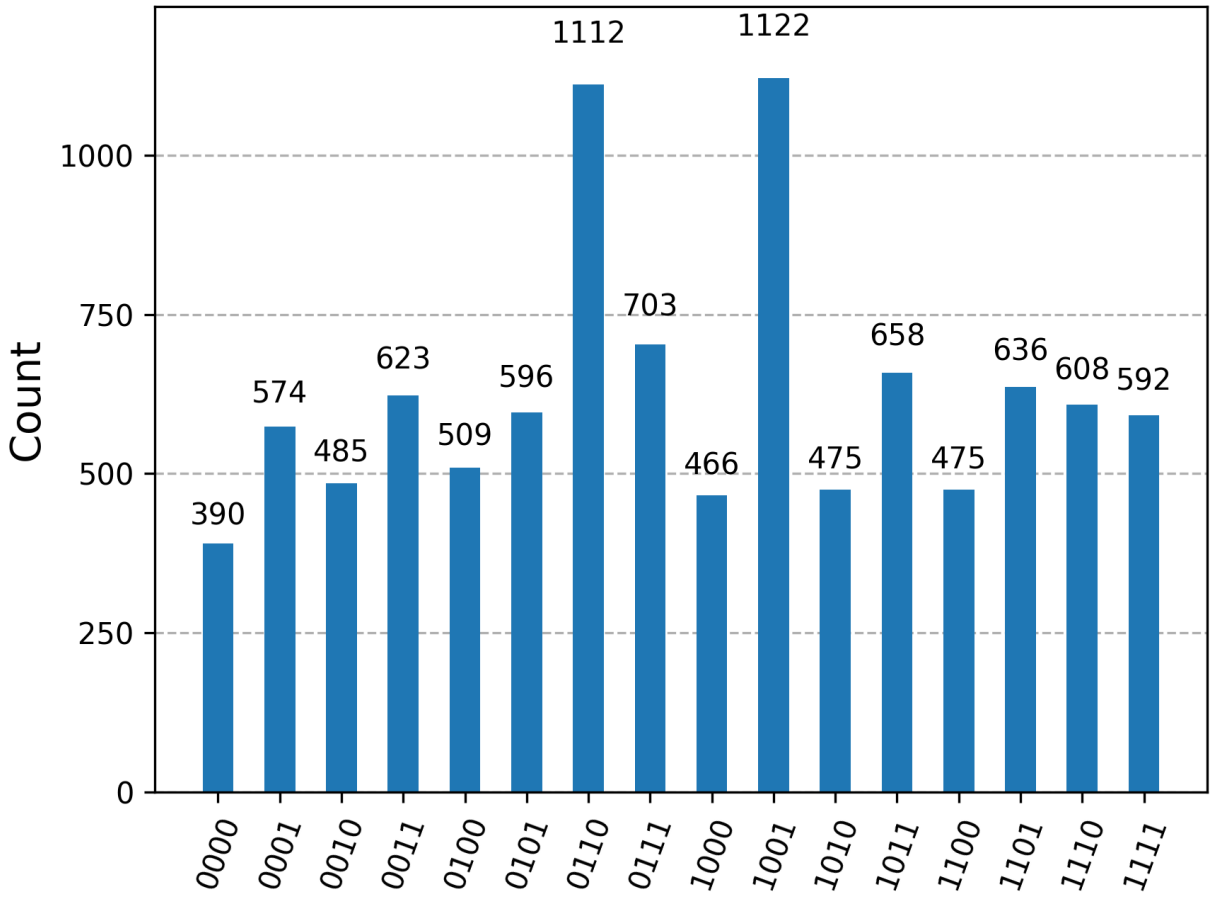
In generale per ridurre possibili errori di **depolarizzazione** e **rilassamento** che possono verificarsi nel concreto, sarebbe utile riuscire a ridurre il numero di livelli di porte che vengono introdotte per ridurre l'accumulo di rumore e, inoltre, anche la quantità di porte e il tempo delle operazioni per minimizzare la durata del circuito complessivo.
Un ulteriore tecnica per l'ottimizzazione degli errori per quanto riguarda la depolarizzazione è il **codice di Shor** che in sintesi ha cil compito di rappresentare un singolo qubit logico da più qubits fisici, questo significa che per ogni bit logico (0 o 1), il computer quantistico usa una "versione" distribuita di quel bit su un gruppo di qubits.
Il Codice di Shor codifica un singolo qubit logico in un **insieme di 9 qubits fisici**, dunque, l'informazione di un qubit viene distribuita in modo tale che anche se si verificano errori su uno o due dei qubits, è ancora possibile recuperare correttamente il valore del bit logico.

Un'altra versione, sarebbe utilizzare il **codice di Steane** che al posto di utilizzare 9 qubit fisici per uno logico ne utilizza solamente 7.

Per quanto riguarda gli errori di misurazione, il più banale metodo per risolverlo, sarebbe quello di eseguire una misurazione ripetitiva per per lo stesso qubit, questo ridurrebbe la possibilità di errore. Di seguito riporto l'implementazione del codice applicato al nostro algoritmo che procede nell'ottimizzazione di questo errore:

```
# Flags for enabling specific errors noises
depolarization_noise_on = True # Toggle depolarization noise
relaxation_noise_on = False    # Toggle relaxation noise
measurement_error_on = True    # Toggle measurement error

# Simulation configurations
simulation_shots = 10024       # Number of shots for the simulation
num_repeats = 100              # Random reduction mitigation repeats
grover_iterations = 2          # Set 0 to use the suggested optimal value

...

if reduce_random:
    # Run the transpiled circuit with noise model
    all_counts = []
    for _ in range(num_repeats):
        result = backend.run(transpiled_circuit, shots=simulation_shots,
noise_model=custom_noise_model).result()
        all_counts.append(result.get_counts())

    average_counts = Counter()
    for counts in all_counts:
        average_counts.update(counts)

    # Normalize the counts
    total_shots = sum(average_counts.values())
    raw_data = {key: value / total_shots for key, value in
average_counts.items()}
```

Che riporta il seguente risultato a livello grafico:



Possiamo notare come le probabilità che le soluzioni accettate si verifichino è molto alta, ma il vero obiettivo è il fatto che le due probabilità delle due soluzioni sono veramente vicine con uno scarto di sole 10 unità che con un numero di *shots* elevato è difficile da ottenere.

Un ulteriore metodo più complesso, riguarda la **matrice di calibrazione** dove ogni dispositivo quantistico ha una probabilità di errore associata alla misurazione di ciascun stato, ad esempio le due probabilità principali che entrano in gioco in un errore di misurazione sono:

- $P(0 \to 1)$: probabilità che una misurazione su uno stato $|0\rangle$ dia erroneamente il risultato $|1\rangle$
- $P(1 \to 0)$: probabilità che una misurazione su uno stato $|1\rangle$ dia erroneamente il risultato $|0\rangle$

Queste probabilità di errore possono essere raccolte in una matrice di calibrazione, che descrive come la misurazione di ogni qubit può essere errata.

$$\begin{pmatrix} P(0 \to 0) & P(0 \to 1) \\ P(1 \to 0) & P(1 \to 1) \end{pmatrix}$$

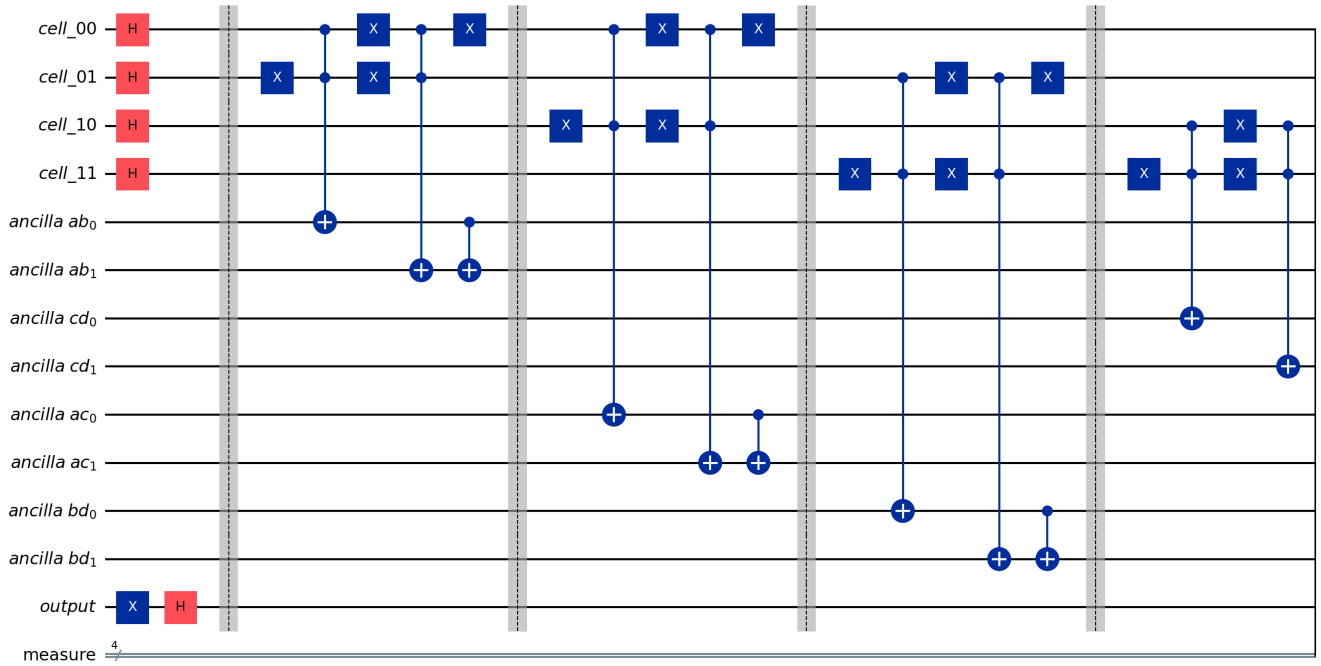
Questa matrice è importante perché ti permette di quantificare e capire la distorsione che si verifica durante le misurazioni, così da poterla correggere successivamente.

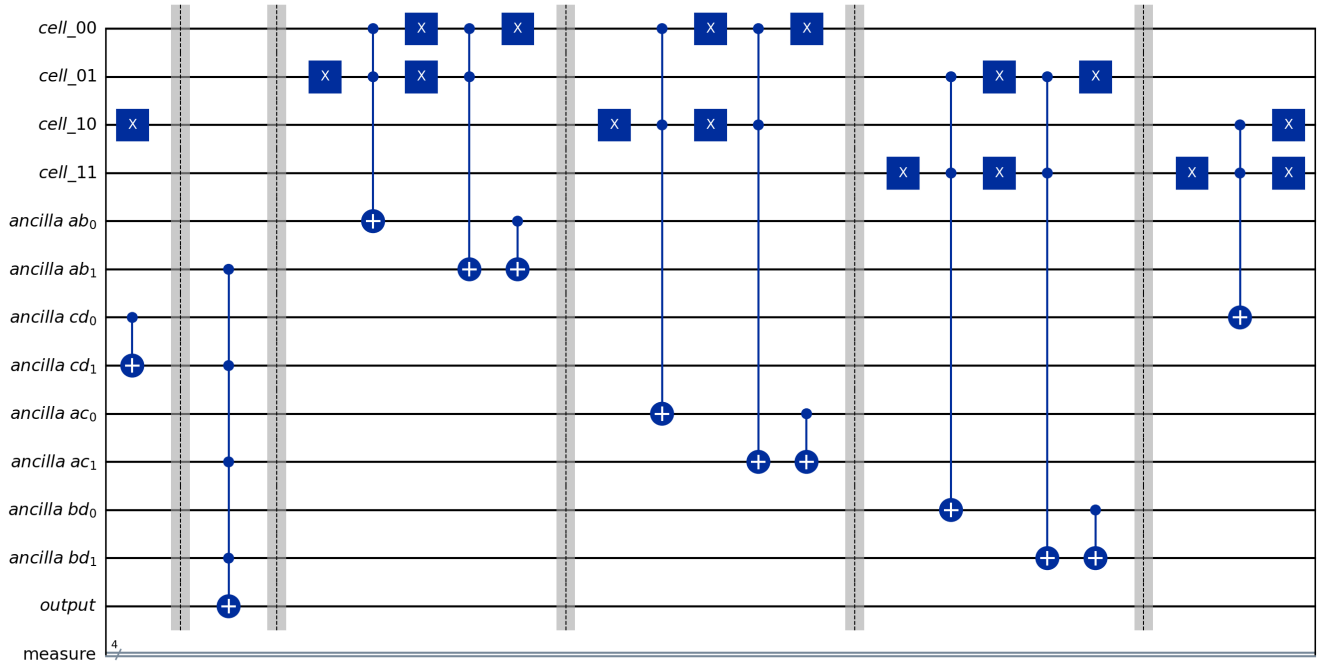
La matrice di calibrazione può essere usata per correggere gli errori di misurazione applicando un processo di **inversione**, per esempio, se si ottiene un risultato errato in base alle probabilità di errore descritte dalla matrice di calibrazione, l'inversione della matrice permette di aggiustare il risultato per restituire il valore che si sarebbe ottenuto senza l'errore.

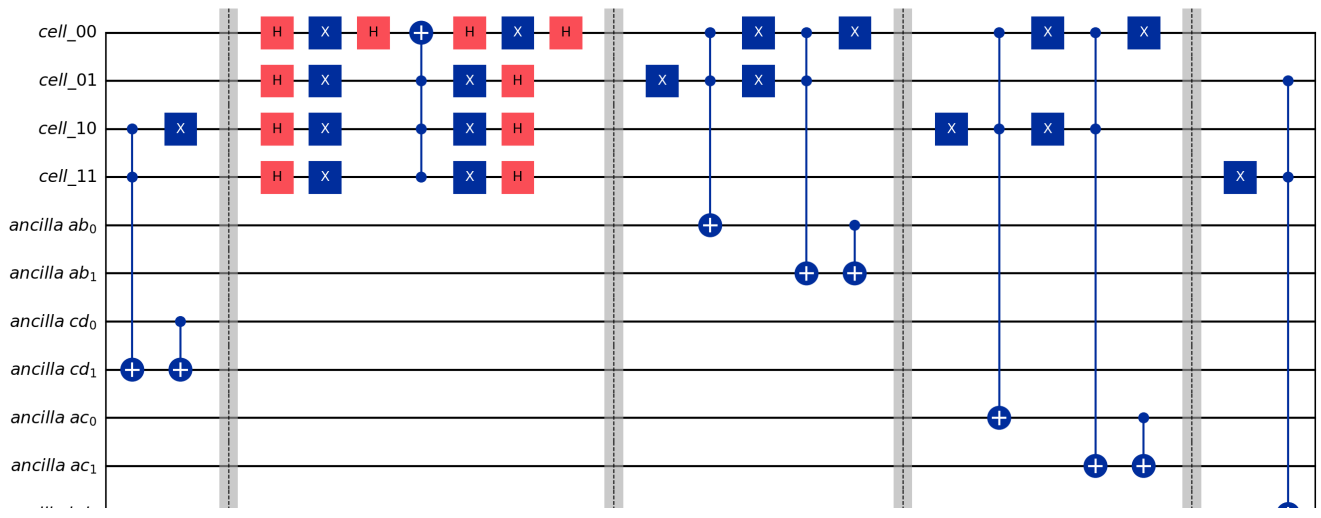
3.3-Output

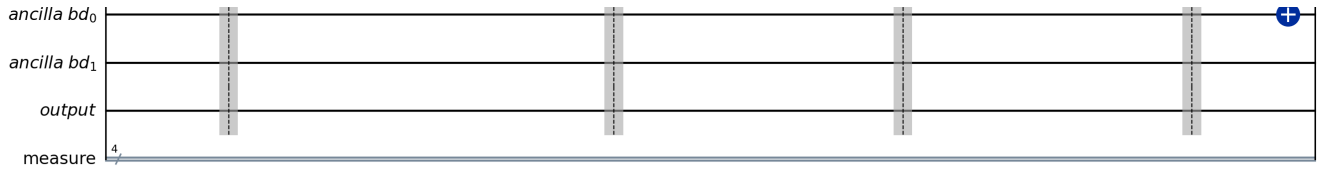
Oltre agli istogrammi che abbiamo visto nella sezione precedente, il codice produce anche l'immagine di un circuito e ci stampa la soluzione.
Nel nostro caso:

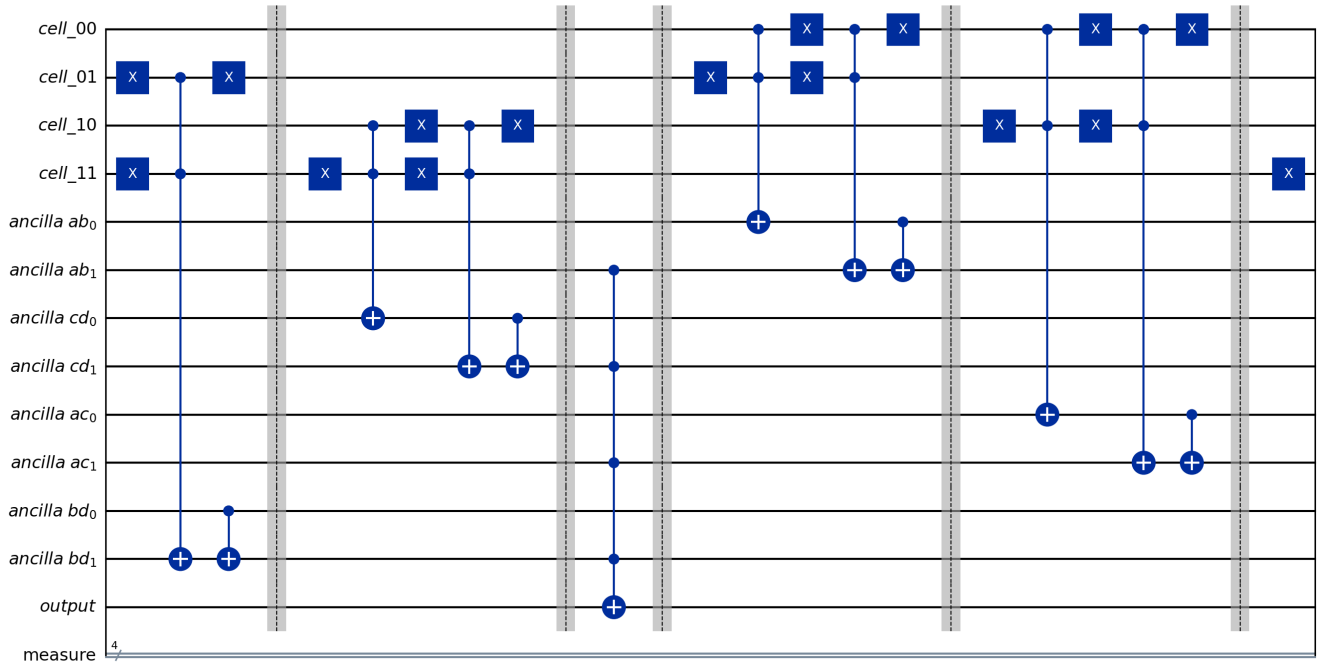
```
Solution 1:
[['1' '0']
 ['0' '1']]

 Solution 2:
 [['0' '1']
 ['1' '0']]
```

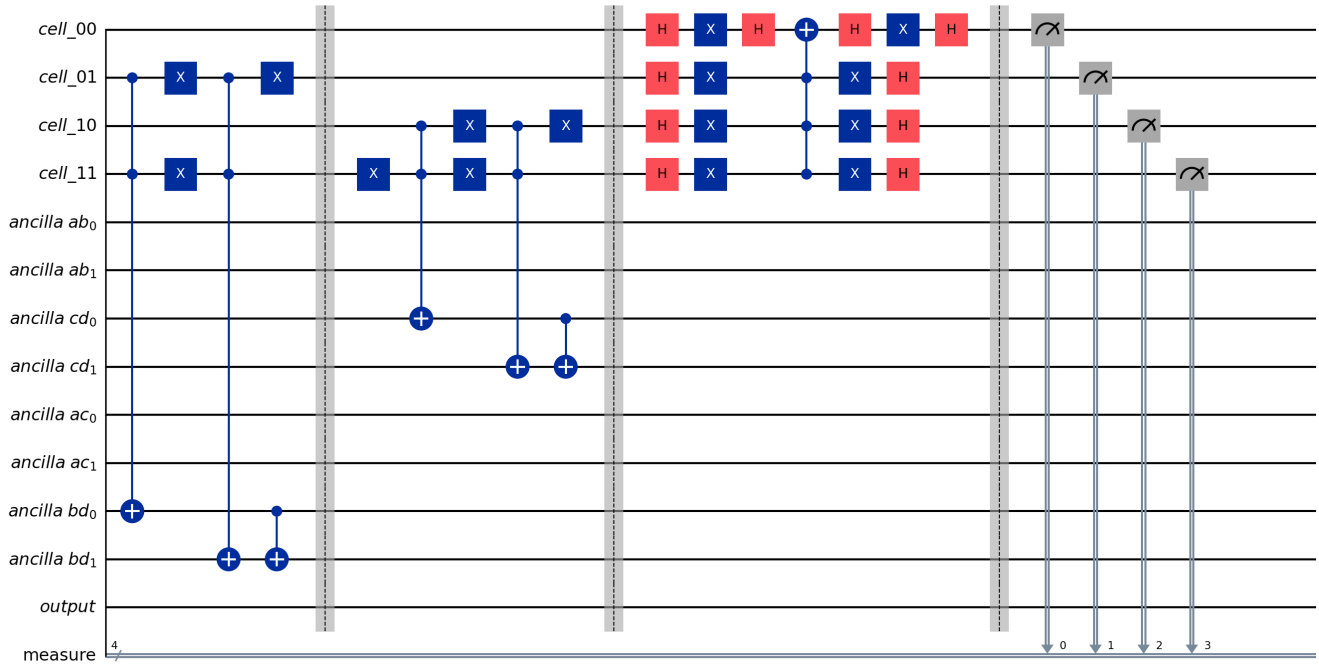

cell_00
cell_01
cell_10
cell_11
ancilla ab0
ancilla ab1
ancilla cd0
ancilla cd1
ancilla ac0
ancilla ac1
ancilla bd0
ancilla bd1
output
measure
4


cell_00
cell_01
cell_10
cell_11
ancilla ab0
ancilla ab1
ancilla cd0
ancilla cd1
ancilla ac0
ancilla ac1
ancilla bd0
ancilla bd1
output
measure
4


cell_00
cell_01
cell_10
cell_11
ancilla ab0
ancilla ab1
ancilla cd0
ancilla cd1
ancilla ac0
ancilla ac1







04-Prospettive di miglioramento e Conclusione

Dopo aver descritto l'implementazione dell'algoritmo di Grover applicato al Sudoku 2×2, è utile approfondire le strategie per ottimizzare il numero di iterazioni. Un approccio adattivo permette di migliorare l'efficienza complessiva dell'algoritmo, affrontando le sfide legate alla scelta statica

del numero di iterazioni e massimizzando la probabilità di successo anche in scenari con soluzioni ridotte o incerte.

4.1-Ottimizzazione del numero di Iterazioni: approccio adattivo

L'approccio adattivo si basa sull'idea di regolare dinamicamente il numero di iterazioni R durante l'esecuzione, anziché determinarlo a priori.
Questo metodo può essere particolarmente utile in scenari in cui il numero di soluzioni M è sconosciuto o in cui si cerca di massimizzare l'efficienza complessiva.

L'algoritmo di Grover ruota lo stato iniziale $|\psi\rangle$ verso lo stato delle soluzioni $|\beta\rangle$
in uno spazio N-dimensionale, utilizzando l'angolazione definita da:

$$\theta = 2arcsin\sqrt{\frac{M}{N}}$$

Il numero ottimale di iterazioni R è dato da:

$$R = \left\lfloor \frac{\pi}{4\theta} \right\rfloor \approx \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{M}} \right\rfloor$$

Tuttavia, se il valore di R è troppo grande, lo stato quantistico viene "ruotato troppo" nello spazio delle soluzioni, superando lo stato ideale $|\beta\rangle$ che rappresenta la configurazione ottimale. In termini pratici, ciò significa che il sistema perde allineamento con le soluzioni desiderate, riducendo la probabilità di successo. Questo effetto, noto come *overshooting*, provoca un ciclo in cui lo stato continua a oscillare attorno a $|\beta\rangle$, allontanandosi progressivamente dalla massima probabilità di trovare una soluzione.

D'altra parte, se R è troppo piccolo, il sistema non raggiunge un'ottimizzazione sufficiente, lasciando lo stato quantistico ancora distante da $|\beta\rangle$. In questo caso, la probabilità di successo rimane subottimale, poiché non si sfrutta appieno il potenziale dell'algoritmo.

Per ovviare a queste problematiche, l'approccio adattivo introduce una regolazione dinamica del numero di iterazioni. Invece di fissare R in anticipo, si procede con un numero iniziale ridotto di iterazioni e si verifica la presenza della soluzione dopo ogni ciclo. Se la soluzione non viene trovata, il numero di iterazioni viene gradualmente incrementato, garantendo un miglior avvicinamento progressivo a $|\beta\rangle$ senza rischiare di "sorpassarlo". Questo processo consente di massimizzare la probabilità di successo, adattandosi alle caratteristiche del problema e riducendo gli effetti negativi del sovra- o sotto-rotazione dello stato quantistico.

4.1.1-Procedura Adattiva

1. **Inizio con poche iterazioni**
   L'algoritmo parte con un numero minimo di iterazioni, ad esempio una sola. Dopo ogni ciclo, verifica se la soluzione è stata trovata. Se sì, l'esecuzione si conclude.
2. **Incremento progressivo**
   Se la soluzione non viene trovata, il numero di iterazioni viene aumentato gradualmente. Questo incremento può essere lineare (aggiungendo un numero fisso) o esponenziale (raddoppiando le iterazioni).
3. **Limitazione al massimo teorico**
   L'incremento continua fino a un limite massimo, calcolato per evitare di compromettere le probabilità di successo "ruotando troppo" lo stato quantistico.
4. **Verifica continua**
   Dopo ogni fase di iterazioni, si misura lo stato quantistico. Se la soluzione viene identificata, l'algoritmo si arresta. In caso contrario, il processo riprende con il numero di iterazioni successivo.

Questo approccio è particolarmente efficace per problemi con poche soluzioni valide o con incertezze nel numero di risultati possibili, adattandosi dinamicamente alle caratteristiche del problema per garantire prestazioni ottimali.

4.2-Ottimizzazione Alternativa: Approccio Basato su QAOA

Oltre all'algoritmo di Grover, un'alternativa più efficiente per risolvere problemi strutturati come il Sudoku 2×2 è rappresentata dall'utilizzo del **Quantum Approximate Optimization Algorithm (QAOA)**. Questo approccio si distingue per la capacità di sfruttare informazioni specifiche della struttura del problema, riducendo così la necessità di iterazioni e risorse computazionali.

4.2.1-Modellazione del Sudoku come Problema di Ottimizzazione

Sudoku 2×2 può essere formulato come un problema di ottimizzazione combinatoria, in cui l'obiettivo è minimizzare una funzione di costo che penalizza configurazioni non valide. In questo caso, ogni cella può contenere esclusivamente i valori 0 e 1, e tali valori possono essere rappresentati direttamente da variabili binarie.

La funzione obiettivo è costruita come segue:

1. **Regole sulle Righe e Colonne:** Viene assegnata una penalità se la somma dei valori in una riga o colonna è maggiore di 1, indicando la presenza di duplicati.
2. **Regola del Blocco:** La somma totale dei valori nel blocco 2×2 deve essere pari a 2, pena l'assegnazione di una penalità.

La funzione obiettivo complessiva può essere scritta come:

$$C(x) = C_{\text{righe}} + C_{\text{colonne}} + C_{\text{blocchi}}$$

dove $C_{\text{righe}}, C_{\text{colonne}}$ e C_{blocchi} rappresentano i contributi alle penalità.

4.2.2 Esecuzione dell'Algoritmo

1. **Stato Iniziale:** Lo stato iniziale è una sovrapposizione uniforme di tutte le configurazioni possibili della griglia binaria.
2. **Operatori Parametrizzati:** QAOA utilizza due operatori:
   - **Operatore costruttivo** basato su $C(x)$, che introduce una fase proporzionale alla funzione di costo, penalizzando gli stati con configurazioni non valide.
   - **Operatore di miscelazione**, che esplora lo spazio delle soluzioni.
3. **Ottimizzazione Variazionale:** I parametri degli operatori sono ottimizzati tramite un algoritmo classico per minimizzare la funzione di costo, massimizzando così la probabilità di ottenere configurazioni valide.

4.3-Confronto tra Algoritmo di Grover e QAOA nel Sudoku 2×2

L'applicazione dell'algoritmo di Grover al Sudoku 2×2 offre un approccio generico per identificare configurazioni valide attraverso la ricerca nello spazio delle soluzioni. Tuttavia, il suo principale svantaggio risiede nella dipendenza dalla progettazione di un oracolo ad hoc, che codifica esplicitamente le regole del Sudoku e identifica soluzioni valide.
Questo processo può essere complesso e richiedere significative risorse computazionali per scalare a problemi più grandi.
Inoltre, l'efficienza di Grover diminuisce con l'aumentare del numero di iterazioni necessarie, specialmente in presenza di un numero ridotto di soluzioni valide.

Al contrario, QAOA sfrutta una funzione di costo per penalizzare configurazioni non valide, rendendo il processo di risoluzione più naturale e adattivo.
Nel caso del Sudoku 2×2, QAOA può rappresentare un'opzione più efficiente, in quanto:

1. Riduce la necessità di progettare un oracolo esplicito.
2. Sfrutta la struttura del problema per guidare il sistema quantistico verso soluzioni valide, minimizzando iterazioni e risorse.

Per il Sudoku 2×2, dove lo spazio delle soluzioni è limitato e la complessità computazionale è contenuta, entrambi gli algoritmi sono implementabili con risorse quantistiche attualmente disponibili.
Tuttavia, QAOA mostra un vantaggio in termini di flessibilità, permettendo di adattarsi facilmente a varianti strutturate o leggermente modificate del problema.

4.4-Sudoku di Dimensioni Maggiori: Scalabilità e Sfide

Quando si passa a Sudoku di dimensioni maggiori, come il classico 9×9, le differenze tra Grover e QAOA diventano ancora più evidenti:

1. **Spazio delle Soluzioni**: Per un Sudoku 9×9, lo spazio delle soluzioni cresce esponenzialmente, rendendo l'algoritmo di Grover meno pratico a causa del numero di iterazioni necessario per esplorare un dominio così vasto. Al contrario, QAOA riduce l'esplorazione dello spazio sfruttando attivamente la struttura del problema tramite la funzione di costo.
2. **Progettazione dell'Oracolo**: Per Grover, la complessità nella progettazione dell'oracolo cresce notevolmente, richiedendo una codifica esplicita per gestire regole di riga, colonna e sottogriglia in una matrice 9×9. Con QAOA, l'espansione è più gestibile, poiché l'aggiunta di vincoli al modello di ottimizzazione è relativamente diretta.
3. **Efficienza e Robustezza**: La natura variazionale di QAOA lo rende intrinsecamente robusto rispetto a errori di calcolo e lo adatta meglio a implementazioni su hardware quantistico rumoroso, caratteristico della tecnologia attuale.

In sintesi, mentre Grover può essere utile per Sudoku di piccole dimensioni e scenari con poche soluzioni valide, la scalabilità verso Sudoku più grandi rende QAOA una scelta più naturale. La capacità di QAOA di adattarsi a problemi complessi e di sfruttare risorse computazionali classiche per ottimizzare parametri quantistici lo rende particolarmente vantaggioso nel contesto di problemi di ottimizzazione combinatoria di grande scala.

05-Conclusione

Nel caso specifico del Sudoku 2×2, l'approccio offerto da QAOA si dimostra generalmente più efficiente rispetto all'algoritmo di Grover. Questo vantaggio deriva dalla capacità di QAOA di sfruttare la struttura intrinseca del problema per ridurre il numero di iterazioni necessarie, rendendo il processo di risoluzione più diretto e adattabile. Inoltre, la progettazione dei circuiti richiesti da QAOA è più semplice e flessibile, facilitando l'implementazione pratica, soprattutto su hardware quantistico moderno.

Tuttavia, è importante notare che l'algoritmo di Grover presenta alcuni vantaggi distinti. In particolare, quando l'obiettivo è identificare tutte le soluzioni valide presenti nello spazio delle configurazioni, Grover si rivela una scelta più sistematica. La sua esplorazione esaustiva dello spazio delle soluzioni, sebbene richieda una complessità maggiore, può risultare vantaggiosa in contesti in cui la completezza delle soluzioni è cruciale.

In sintesi, la scelta tra QAOA e l'algoritmo di Grover dipende fortemente dalle esigenze specifiche del problema e dagli obiettivi di ottimizzazione. Mentre QAOA eccelle nella velocità e nella praticità per risolvere problemi strutturati come il Sudoku 2×2, Grover offre un approccio

complementare per la ricerca esaustiva di soluzioni in spazi complessi. Questa dualità evidenzia la versatilità degli algoritmi quantistici e il loro potenziale nell'affrontare una vasta gamma di sfide computazionali.

06-Bibliografia

Regole: Wikipedia
Problematiche: Esempio complessità
Algoritmo di Grover: GitHub
Codice algoritmo: Paper
Ottimizzazioni iterazioni: Grover Adaptive Search for Constrained Polynomial Binary Optimization
Ottimizzazione iterazioni: Quantum Computation and Quantum Information