# Implementazione di un algoritmo di ricerca del minimo mediante computazione quantistica

Simone Allegra

Febbraio 2024

## 1 Introduzione

In questo documento vengono spiegati i passaggi per una corretta implementazione dell'algoritmo quantistico proposto da Christoph Dürr e Peter Høyer riguardo la ricerca di un valore minimo in un array non strutturato di dimensione N contenente valori numerici.

La soluzione proposta è descritta nel dettaglio nell'articolo pubblicato nel Luglio del 1996 dal titolo *A quantum algorithm for finding the minimum*, in cui sono presenti le dimostrazioni che provano il vantaggio in termini di complessità dell'algoritmo rispetto la soluzione che si ottiene in computazione classica, passando da una difficolta dell'ordine $O(N)$ ad una $O(\sqrt{N})$.

L'implementazione realizzata di cui fa capo questo testo è di libera consultazione sulla piattaforma GitHub[1].

Progetto realizzato per il corso di Quantum Computer Programming tenuto dal professore S. Faro, UniCt. Considerata la finalità didattica dello studio, si cerca di implementare gli algoritmi utilizzati in maniera personalizzata piuttosto che utilizzare quelli già presenti nella libreria di qiskit.

## 2 Soluzione Classica

Nella computazione classica è necessario che vengano analizzati tutti i valori appartenenti all'array in esame: scelto un valore iniziale arbitrario è necessario che esso venga confrontato con tutti gli elementi, aggiornando dunque la variabile di supporto al valore più piccolo trovato.

Da questa trattazione è assiomatico dimostrare come tale algoritmo abbia una complessità di ordine $O(N)$, cioè dell'intera dimensione del vettore.

[1] ttps://github.com/simoneallegra/MinimumQuantumSearching

Ai fini dimostrativi è stata realizzata una piccola implementazione di questa soluzione per valutare le migliorie alla complessità e la correttezza dei risultati ottenuti dalla soluzione realizzata mediante computazione quantistica.

```
def classical_min_searching(db):
    aux = 0
    for i in range(len(db)):
        if db[i] < db[aux]:
            aux = i
    return db[aux], i
```

Listing 1: Implementazione classica

# 3 Approccio quantistico

L'implementazione si basa sull'algoritmo di Grover: il funzionamento di Grover si basa sul principio di interferenza quantistica. L'algoritmo sfrutta le proprietà dei qubit, le unità di informazione quantistiche, per esplorare simultaneamente diverse possibili soluzioni.

All'algoritmo di Grover va passato in input un opportuno *Oracle*, strumento ampiamente utilizzato nella computazione quantistica per rappresentare le soluzioni di un problema; in pratica mediante questo schema, si cerca di ridurre un problema di ricerca delle soluzione in un problema di verifica. Condizione necessaria per il riutilizza delle risorse è la reversibilità del *Quantum Gate* che rappresenta l'Oracle.

Per l'implementazione è stata utilizzata la libreria di qiskit, che combina le funzionalità del linguaggio python con una serie di funzionalità tipiche della computazione quantistica, offrendo un'interfaccia per l'elaborazione ad un reale computer quantisco dell'IBM, oltre alla possibilità di simulare il tutto su macchine classiche.

## 3.1 Oracle

L'oracle implementato ha come input un oracle che è composto da due sezioni: un boolean oracle che mappa, per ogni indice, il relativo valore, ed una funzione, denominata con la lettera $P$ come funzione di confronto parallelo tra il valore dell'indice scelto di riferimento per la i-esima iterazione dell'algoritmo e tutti i valori precedentemente mappati nell'oracle.

```
def Generate_Oracle(db: list, bits_of_value: int):
    N = len(db)
    bits_index = int(math.ceil(math.log2(N)))

    qc = QuantumCircuit(bits_index + bits_of_value)

    for index, value in enumerate(db):

        str_index =
           str(bin(index))[2:].zfill(bits_index)
        for i in range(len(str_index)):
            if str_index[i] == "0":
                qc.x(i)
        for i in range(bits_of_value):
            if value[i] == "1":
                qc.mcx(list(range(bits_index)),
                   bits_index + i)
        for i in range(len(str_index)):
            if str_index[i] == "0":
                qc.x(i)
        qc = qc.to_gate(label="Oracle")
        return qc
```

Listing 2: Generazione Oracle booleano

Con la generazione dell'oracle booleano sopra descritto si intende creare una funzione capace di restituire in output il valore in base a quale indice è stato selezionato. L'algoritmo di Grover applicando la superimposizione ai qubit che rappresentano l'indice, sfrutta la potenza della computazione quantistica per esercitare una ricerca paralella, come se considerasse, durante il confronto, tutti i valori possibili invece di, come in computazione classica, eseguire il confronto due valori alla volta.

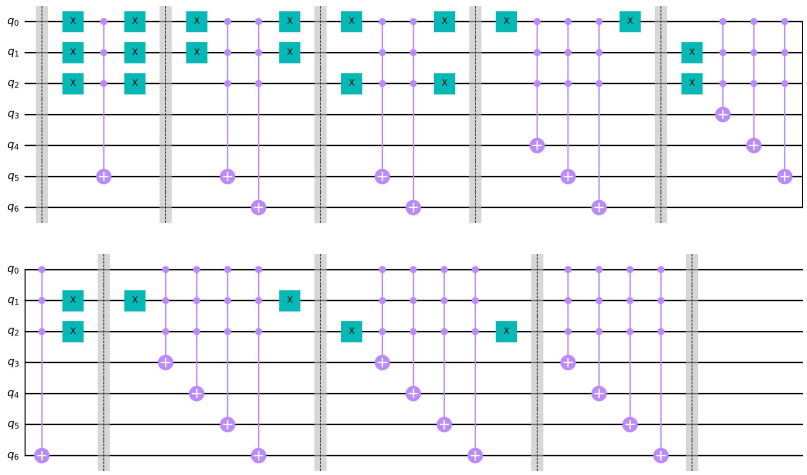


Figure 1: Esempio di un Oracle rappresentante il vettore input ['0010', '0011', '0011', '0111', '1111', '1111', '1111', '1111']

La seconda parte della funzione da passare come input all'algoritmo di Grover è quella utilizzata per il confronto: Il confronto si pone come obiettivo quello di individuare la presenza o meno di bit più significativi posti a 1, che indicano se i valore scelto per l'iterazione è maggiore o minore di qualche altro elemento all'interno dell'array.

La costruzione della funzione *Compare* ha come parametro il valore da dover confrontare con gli altri; tale costruzione fa in modo di confrontare gli zeri più significati del parametro d'ingresso con tutti gli altri valori. Il risultato del confronto si ottiene attraverso il fenomeno del phase kickback, dove viene posto un NOT controllato su una linea di qubit posta in —-¿ per eseguire una azione sulle linee di qubit controllate e poter dunque valutare quali sono gli indici che equivalgono ai risultati del confronto.

Si considera importante far notare come l'array di ingresso venga espanso con valori massimi rispetto lo spazio di bit consentito in modo da poter utilizzare Grover e non preoccuparsi del fatto che il numero di soluzioni deve essere inferiore della metà della lista analizzata per poter ottenere dei risultati. Una implementazione futura potrebbe prevedere l'implementazione di un *counting quantistico* al fine di determinare preventivamente le soluzioni dell'algoritmo

```
def Compare(n: int, k: int, yi: str):
    qc = QuantumCircuit(n+k+1)
    # Build chk
    chk = []
    zero_found = False
    b = False
    for i, s in enumerate(yi):
        chk.append(i)
        if s == '0':
            zero_found = True
        if s == '1' and zero_found:
            b = True
            break
    if not b:
        chk = [0]
    # --------
    for elem in chk:
        qc.x(n+elem)
    targets=[]
    [targets.append(n+elem) for elem in chk]

    qc.mcx(targets, n + k)
    for elem in chk:
        qc.x(n+elem)

    qc = qc.to_gate(label="Compare")
    return qc
```

Listing 3: Generazione Confronto

La presenza di più iterazioni è spiegata dal fatto che devono essere considerati bit sempre meno significati fino ad ottenere un valore che, confrontato con gli altri, non restituisca alcun minore.

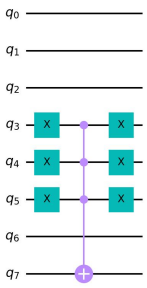


Figure 2: Generazione Confronto di esempio con input il valore 00010 o 00011

## 3.2 Grover

Parte fondamentale dell'algoritmo è la ricerca basata sull'algoritmo di Grover. Con questo approccio è possibile isolare le soluzioni e farne risaltare le probabilità di misura rispetto alle non-soluzioni. Tale algoritmo funziona sfruttando la non-perpendicolarità che si genera tra due vettori nell'ipersfera di Bloch, dovuta dalla presenza di un 1 del vettore obiettivo. Questa differenza dà la possibilità di ottenere la proiezione rispetto l'ascissa e la negazione dell'ampiezza del vettore obiettivo.

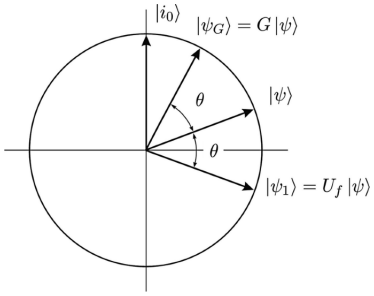


Figure 3: Visualizzazione geometrica di una singola iterazione dell'algoritmo di Grover

Dalla negazione dell'ampiezza eseguita, attraverso una *riflessione*, si può ottenere una amplificazione dell'ampiezza del vettore obiettivo. Si nota quindi che, agendo in maniera iterativa, si può ottenere una amplificazione dell'ampiezza significativa che equivale ad una probabilità di misura corretta del vettore obiettivo significativa.

Tale amplificazione viene eseguita sugli indici del vettore indicati come soluzioni della singola iterazione, ottenuta dall'applicazione degli oracle.

```
def Diffuser(n: int):
    qc = QuantumCircuit(n)
    for i in range(n):
        qc.h(i)
        qc.x(i)
    qc.h(n-1)
    qc.mcx(list(range(n-1)),n-1)
    qc.h(n-1)
    for i in range(n):
        qc.x(i)
        qc.h(i)
    qc = qc.to_gate(label='Diffuser')
    return qc
```

Listing 4: Funzione di riflessione (diffusione) utilizzata nell'algoritmo di Grover

L'algoritmo di Grover dunque è così completo, come mostrato nella prossima immagine Per eseguire l'algoritmo, alla prima iterazione viene preso un indice

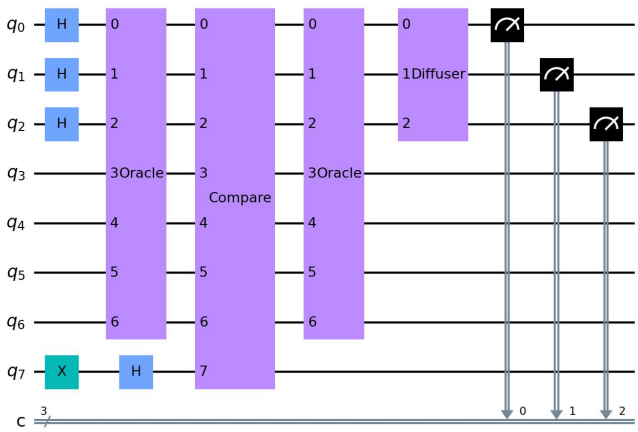


Figure 4: Algoritmo di Grover con il dettaglio delle parti di cui è composto l'oracle

casuale per il confronto, mentre per le iterazioni successive viene scelto casualmente (come scelta implementativa) tra quelli che sono i risultati della iterazione precedente.

L'applicazione ciclica di tale algoritmo porta ad un risultato con una complessità media stimata in accordo con l'obiettivo prefissato.

### 3.3 Codice e risultati

L'algoritmo di Grover è stato implementato come main di uno script python di cui segue l'estratto

```
n_chars = 4
    array_len = 4

    db = []
    db_int = []

    # Initialize memory
    for i in range(array_len):
        value = random.randint(0,2**(n_chars-1)-1)
        db_int.append(value)
        db.append(str(bin(value))[2:].zfill(n_chars))
    min = min(db_int)
    # -----------------

    ### Extends elements for Gorver algorithm
    for i in range(array_len):
        db.append('1' * n_chars)
    print(db)
    # ---------------------------------------

    ### Select random Yi
    first_index =
       int(random.randint(0,int((len(db)-1) / 2)))
    print(f"first index: {first_index}")
    value_yi = db[first_index]
    # ----------------

    bits_index = int(math.ceil(math.log2(len(db))))

    # Prepare parts of Grover that's no change during
       cycles
    U  = Generate_Oracle(db, n_chars)
    W = Diffuser(bits_index)

    while True:

        # GROVER Algorithm
        qc = QuantumCircuit(bits_index + n_chars + 1,
           bits_index)

        for i in range(bits_index):
            qc.h(i)
```

```
        qc.x(bits_index + n_chars)
        qc.h(bits_index + n_chars)

        P  = Compare(bits_index, n_chars, value_yi)

        qc = qc.compose(U)
        qc = qc.compose(P)
        qc = qc.compose(U)
        qc = qc.compose(W)

        for i in range(bits_index):
            qc.measure(i,i)
        # End GROVER Algorithm

        res = run_circuit(qc, 1000)
        qutils.print_circuit(qc)

        qutils.revcounts(res)

        found = False
        for key, value in res.items():
            if value > 240:
                end(int('0b'+ db[int('0b' +
                    key[::-1], base = 0)], base = 0),
                    min)

            if value > 100:
                found = True
                value_yi = db[int('0b'+key[::-1],
                    base = 0)]
                print(f"new index:
                    {int('0b'+key[::-1], base = 0)}")
                break

        if not found:
            end(int('0b'+ value_yi, base = 0), min)
```

Listing 5: Algoritmo implementato

Nonostante siano stati provati diversi input e situazione con ottimi risultati, nel codice proposto è riportato un esempio di ricerca del minimo in un array composto da 16 elementi con numeri esprimibili in 16 bit. Altro paramentro impostato per l'occasione è la soglia a cui si considera il valore della misura come unico risultato (nel caso specifico 240). Questo valore dipende dalla dimensione del circuito è questo punto non è stato trattato in tale implementazione. Quest'ultima soglia potrebbe essere comunque ignorata, in quanto l'algoritmo termina il suo operato in qualunque condizione a patto di non trovare

più soluzione chiare dal processo di Grover, considerando così l'i-esimo input analizzato come soluzione del problema.

L'output contiene sia il valore trovato mediante elaborazione quantistica sia approcciandosi in maniera classica, così da mostrare l'efficacia dell'algoritmo implementato:

```
$ python3 QuantumMinimumResearch.py
['0011100100110101', '0011010010000100',
   '0011011000111010', '0110100100101001',
   '0101001111111110', '0001010000011001',
   '0011011000011011', '0111111000111000',
   '0011110010111100', '0010101110001001',
   '0100111111011110', '0011110011000011',
   '0011001111111000', '0000010001000110',
   '0100001001101000', '0110011111111010',
   '1111111111111111', '1111111111111111',
   '1111111111111111', '1111111111111111',
   '1111111111111111', '1111111111111111',
   '1111111111111111', '1111111111111111',
   '1111111111111111', '1111111111111111',
   '1111111111111111', '1111111111111111',
   '1111111111111111', '1111111111111111']

[...]

Quantum Algorithm result: 1094
Classical minimum: 1094
```