ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TITOLO DELL' ELABORATO

Quantum Supervised Learning: algoritmi e implementazione

Tesi in

Data Mining

Relatore:
Prof. Claudio Sartori

Candidato:
Nicolò Cangini

Correlatori:
Antonio Macaluso
Prof. Stefano Lodi

Sessione III
Anno Accademico 2017/18

Keywords: Quantum Computing, Quantum Algorithms, Quantum Machine Learning, Supervised Machine Learning, Qiskit

Prefazione

“Non c’è motivo per ogni individuo di avere un computer in casa propria”. Fu la dichiarazione di Kenneth Olsen, allora direttore della Digital Equipment Corporation, nel 1977 durante la convention della World Future Society. Inutile sottolineare il fallimento di tale previsione; di fatto, al giorno d’oggi è praticamente impossibile suppore che una persona di questa generazione, non disponga abitualmente di uno smartphone, un tablet, un computer portatile o un qualsiasi altro dispositivo elettronico capace di processare informazione, dotato quindi di potenza di calcolo.
L’evoluzione tecnologica alla quale abbiamo assistito negli ultimi trent’anni è indiscutibile quanto straordinaria è l’importanza che l’informazione ha acquisito in qualsiasi ambito applicativo, quasi al punto da doverci chiedere quanto è condizionata la vita quotidiana dal concetto di elaborare l’informazione digitale che ci lasciamo alle spalle interagendo con il mondo.
Dove nasce questa assidua necessità di calcolare sempre più informazioni e risolvere problemi sempre più profondi e complessi al punto da raggiungere il limite della capacità delle macchine, sfidando sempre di più i limiti fisici e meccanici dei calcolatori?

Scopo di questa tesi è sintetizzare e introdurre i concetti più importanti legati alla computazione quantistica che definiscono le basi di un nuovo modello computazionale per l’elaborazione dell’informazione e come tale progresso tecnologico e scientifico possa garantire un’ulteriore evoluzione nel mondo dei calcolatori con nuovi strumenti e macchine più performanti capaci di garantire il miglioramento delle prestazioni nelle aree pratiche, come il Machine Learning, dando la possibilità di esplorare soluzioni ad oggi irraggiungibili a causa della complessità computazionale che caratterizza alcuni problemi che la computazione classica, alla quale siamo abituati, non è e non sarà mai in grado di risolvere.

Indice

Capitolo 1

Introduzione

Il primo calcolatore capace di trovare soluzioni all'interno di un numero incredibilmente grande di combinazioni fu chiamato Colossus. Una invenzione degli inglesi nel corso della Seconda Guerra Mondiale con il fine di decifrare i messaggi prodotti dall'ingegnosa Enigma, costruita dai tedeschi per criptare lo scambio di informazioni militari. Fu il punto di partenza per la rivoluzione tecnologica che ha portato alla modellazione dell'idea che ancora oggi abbiamo di computer. Ne seguì quello che divenne la teoria classica della computazione, basato su un modello astratto definito dal matematico Alan Turing, che in un articolo del 1396, sviluppò le regole e i principi che una macchina universale, chiamata la Macchina di Turing, doveva seguire per portare a termine in un numero finito di cicli una determinata operazione, obbedendo alle leggi della fisica classica.

Questi principi furono elaborati successivamente da John von Neumann negli anni '40 e rimangono ancora oggi praticamente immutati nonostante l'enorme sviluppo tecnologico che ha permesso la realizzazione di calcolatori nettamente più performanti, veloci e potenti, rispetto a quelli realizzabili inizialmente. Il sorprendente sviluppo delle tecnologie e dell'industria dei semiconduttori e dei processori è stato scandito a partire dal 1965 dalla famosa Legge di Moore. Gordon Moore, cofondatore di Intel e pioniere nell'ambito dei circuiti integrati, stabilì che circa ogni diciotto mesi era possibile ottenere il doppio delle prestazioni offerte da un processore raddoppiando il numero di transistor presenti all'interno del chip; fatto che ha portato negli anni un rimpicciolimento sempre più accentuato delle dimensioni dei transistor (i processori odierni arrivano a 14 nanometri), costringendo le grandi case produttrici a scontrarsi con problemi legati non più all'architettura o ai materiali ma bensì ai fenomeni fisici di interferenza non più trascurabili che si osservano nel microscopico. Infatti rimpicciolendo sempre di più i componenti interni, diminuisce lo spazio fisico nel quale la corrente, e quindi gli elettroni, possono muoversi. Queste particelle sub-atomiche, occupando uno spazio fisico compreso tra 1 e 2

nanometri, fanno sì che sorgano problemi legati ai margini di tolleranza all'interno dei gate e degli stessi transistor complicando la possibilità di un ulteriore ridimensionamento. Proprio queste dimensioni sempre più microscopiche, determinano il fatto di poter descrivere la propagazione degli elettroni solo attraverso le leggi della meccanica quantistica. È ragionevole pensare che senza alcun dubbio il rispetto della legge di Moore cesserà di valere nel 2020, quando probabilmente si raggiungerà il limite dimensionale di 7 nanometri e gli effetti quantistici non saranno più trascurabili, obbligando un netto cambio di strategia a livello architetturale e tecnologico che ha spostato l'attenzione sulla computazione in cloud, suggerendo l'aumento delle prestazioni dei server e dei grandi centri di calcolo, sfruttabili da qualsiasi utente semplicemente con l'utilizzo di una connessione a banda larga: il computer di casa diventa quindi un interfaccia sul quale lavorare ma il centro della computazione rimane esterno all'utilizzatore, che visualizzerà risultati computati altrove in totale trasparenza.
Non per questo si è diminuito lo sforzo e la ricerca di una nuova macchina intelligente, una nuova forma di computazione più veloce e performante capace di risolvere, magari, quei problemi definiti classicamente impossibili o possa fornire supporto computazionale in quegli ambiti dove la mole di calcolo supera la disponibilità tecnologica e temporale. Nasce dunque il concetto di quantum computing, di macchine quantistiche capaci di risolvere determinati problemi ad una velocità nettamente maggiore rispetto al modello classico sfruttando un nuovo modello computazionale che non rispetta più le leggi classiche ma rispetta a pieno regime quelle quantistiche.

Da dove derivano queste proprietà e questi principi della materia e del mondo dei *quanti* che sembrano farci intravedere un nuovo traguardo tecnologico? Già verso la fine dell'Ottocento, uno dei fenomeni più studiati e sotto stretta osservazione da parte degli scienziati era la radiazione di corpo nero. Il contorno scientifico era ovviamente orchestrato dalle leggi della fisica classica, per la quale, dopo le affermate formule sull'elettromagnetismo di Maxwell, la luce era concepita come un elemento continuo descrivibile attraverso un'equazione d'onda. Nel dicembre del 1900, durante un esperimento sui filamenti delle lampadine, il fisico tedesco Max Planck si imbatté su un fatto sorprendente, che lo portò a dover ammettere, spinto dai fatti sperimentali, che l'energia della luce e di tutte le altre forme di radiazione elettromagnetica potesse essere emessa e assorbita dalla materia soltanto in frammenti discreti, dei "pacchetti" di energia, che prendono il nome di 'Quanti'. Questi oggetti rappresentarono una vera e propria rottura

radicale con gli schemi interpretativi dati dalla fisica classica e aprirono nuovi orizzonti alla ricerca.

I maggiori scienziati dell'epoca usufruirono del concetto di *Quanto* per effettuare nuove scoperte, ad esempio Albert Einstein se ne servì per dimostrare nel suo primo articolo pubblicato l'effetto fotoelettrico, nel quale si assumeva che gli elettroni potevano essere ricavati da una superficie metallica piana quando colpita da un raggio luminoso di una certa intensità. Rispettando le leggi e le teorie di Maxwell sull'elettromagnetismo, l'aumentare dell'ampiezza dell'onda luminosa avrebbe aumentato l'energia cinetica dei fotoelettroni emessi, mentre aumentando la frequenza sarebbe aumentata la corrente misurata. Gli esperimenti dimostrarono una certa incoerenza con i risultati attesi e proprio Albert Einstein diede una spiegazione esaustiva dei fatti assumendo che la luce è un insieme di fotoni. Con il tempo si è dimostrato che tutta la materia nell'universo manifesta comportamenti corpuscolari e ondulatori.

Il *Quanto* fu per quegli anni, qualcosa che sconvolse la mente dei luminari della fisica teorica, in quanto, gli effetti atomici, microscopici che ne conseguivano, non rispettavano i dogmi classici della fisica. Per anni, si svilupparono teorie al limite tra fisica e filosofia, seguite da esperimenti mentali che diedero vita a dibattiti sulla esistenza o meno del *Quanto* e nacquero correnti di pensiero distinte sull'interpretazione di quella che prese il nome di meccanica quantistica.

Fondamentalmente, alla fine degli anni '40, da un lato vi era Einstein, che con Schrödinger era sostenitore di una fisica indipendente dall'osservazione, più realistica e dove qualsiasi cosa è causale, una fisica incompleta, alla ricerca di quello che fu definito il Santo Graal, ossia una teoria unica, unificatrice del macro e del micro, valida in qualsiasi ambito applicativo. Dall'altra sponda di questa battaglia intellettuale c'erano Bohr e Heisemberg, sostenitori invece che il comportamento e le equazioni della meccanica quantistica fossero esaustive, che fosse stata creata una teoria completa, per la quale non era possibile trovare una connessione con i fenomeni macroscopici dell'universo e dove valeva il principio di indeterminazione, per il quale era impossibile misurare dettagliatamente fenomeni microscopici in quanto le stesse tecnologie utilizzate per la misurazione alterano il valore reale del fenomeno, in altre parole, un fenomeno quantistico non esisterebbe fino al momento della sua osservazione (grossa divergenza rispetto alla causalità di Einstein).

Ancora oggi, nonostante i progressi in materia, la meccanica quantistica è qualcosa di parzialmente incompreso, un territorio difficile da esplorare per il quale non esiste una teoria esatta e completa. Si continua la ricerca di una teoria dei campi che possa unificare

tutte le forze fisiche in gioco e realizzare così quella scoperta che rimase un punto interrogativo per molti.

Ovviamente i nuovi concetti definiti dalla meccanica quantistica stanno alla base della computazione quantistica e di quella che viene definita *quantum information*, un'area di ricerca che negli ultimi anni è cresciuta e ha portato a piccoli grandi risultati, come dispositivi quantistici capaci di registrare l'informazione in singoli atomi, anche se ancora affetti dal grande nemico del quantum computing: la *decoerenza*. La realizzazione di computer quantistici reali e la possibilità esistente di testarli in cloud, testimoniano che il settore è in grande fermento.

Nonostante esplorare il significato dei singoli principi quantistici non sia l'obiettivo di questa tesi, se ne introdurranno alcuni dei quali rendono così affascinante e futuristico questo campo e questo nuovo modello di computazione.

Capitolo 2

Quantum Computing

Nella seguente sezione sono introdotti i concetti basici della teoria quantistica necessari per capire più approfonditamente il funzionamento degli algoritmi quantistici e in che modo, alcuni degli effetti quantistici, possano essere sfruttati all'interno di determinati algoritmi applicati a specifici problemi del computer science al fine di ottenere giovamenti e migliorie rispetto alla controparte classica.

2.1 Bit e Qubit

Nello studio dell'informatica classica, definiamo come unità fondamentale della computazione il bit. Questo elemento può, in un preciso istante di tempo, trovarsi in uno dei due stati fondamentali 0 o 1 rappresentando un sistema a due dimensioni detto binario. Sappiamo inoltre che avendo a disposizione *n* bit siamo in grado di rappresentare 2^n stati differenti, ognuno dei quali sarà associato ad un intero $k \in \{0, 1, \dots, 2^{n-1}\}$ descritto in notazione binaria nella forma:

$$k = k_{n-1}2^{n-1} + \cdots + k_1 2 + k_0 = \sum_{i=0}^{n-1} k_i 2^i \,;\; k_i \in \{0,1\}.$$

L'informazione complessiva è gestita e processata in qualsiasi contesto attraverso registri di bit e gate classici capaci di effettuare operazioni e modificarne lo stato.
La computazione quantistica introduce una nuova unità fondamentale che prende il nome di qubit. Fisicamente viene rappresentato con un sistema microscopico a due livelli come

lo spin di una particella, la polarizzazione di un singolo fotone o due stati di un atomo ottenibili cambiando il livello energetico di un suo elettrone.
Se vogliamo descriverlo matematicamente possiamo definirlo come un vettore unitario descritto in uno spazio vettoriale di Hilbert complesso bidimensionale($\mathbb{C}^2$).
Per rappresentare gli elementi di uno spazio vettoriale complesso è conveniente utilizzare la notazione di Dirac (notazione standard della meccanica quantistica). Secondo tale notazione, $|v\rangle$ (simbologia *ket*) indica un generico elemento dello spazio vettoriale e $|i\rangle$ è l'i-esimo elemento della base ortonormale canonica. Se $|k\rangle = \sum_i \alpha_i |i\rangle$ e $|w\rangle = \sum_i \beta_i |i\rangle$ allora il prodotto scalare tra i vettori si indica con $\langle v|w\rangle$ dove il vettore riga è $\langle v|$ (simbologia *bra*) in modo che il prodotto formi un *bracket*. Definendo due vettori:

$$\begin{pmatrix}1\\0\end{pmatrix} \quad \begin{pmatrix}0\\1\end{pmatrix}$$

e associandoli rispettivamente agli stati $|0\rangle$ e $|1\rangle$, essi formano una base *ortonormale*, cioè una base *ortogonale* di vettori aventi *norma* uno, nota come *base computazionale standard*.
Possiamo inoltre dare una definizione degli stati attraverso la forma matriciale (vettori colonne), ottenendo la seguente rappresentazione:

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

In questo caso, i due vettori appena introdotti corrispondono esattamente agli stati classici 0 e 1. È d'obbligo a questo punto marcare la principale differenza con il bit classico: un qubit, oltre a potersi trovare in uno degli stati fondamentali, potrà trovarsi contemporaneamente anche in un'altra qualsiasi combinazione di entrambi gli stati base.
Se definiamo $|\psi\rangle$ la seguente combinazione lineare:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

dove α e β rappresentano numeri complessi tali che valga:

$$|\alpha|^2 + |\beta|^2 = 1$$

allora $|\psi\rangle$ è un possibile stato del qubit la cui notazione algebrica equivalente sarà:

$$|\psi\rangle = \alpha\begin{pmatrix}1\\0\end{pmatrix} + \beta\begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}\alpha\\\beta\end{pmatrix}.$$

Che equivale a dire che $|\psi\rangle$ si trova in una sovrapposizione di stati. Quando abbiamo a che fare con un bit classico possiamo sempre stabilire con assoluta certezza in quale dei due stati esso si trovi, nel caso di un qubit non possiamo determinare con altrettanta precisione il suo stato quantistico, ossia i valori esatti di α e β. La meccanica quantistica ci dice che soltanto attraverso l'effettiva misurazione del sistema otterremo un valore discreto del qubit, più precisamente si dice che lo stato collasserà nello stato $|0\rangle$ con probabilità $|\alpha|^2$ o in $|1\rangle$ con probabilità $|\beta|^2$. Proprio per questa ragione, i due valori α e β prendono il nome di **ampiezze di probabilità** (amplitudes).
Una prima semplice e basilare sovrapposizione è definita dallo stato:

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle),$$

il quale avrà una rilevanza sostanziale nella fase applicativa.
Dunque per ora possiamo immaginare che fino al momento della sua effettiva misurazione, un qubit avrà una probabilità del 50% $\left({}^{1}/_{2}\right)$ di trovarsi nello stato $|0\rangle$ e un altro 50% $\left({}^{1}/_{2}\right)$ di trovarsi in $|1\rangle$; come se lanciando una moneta essa continuasse a girare su sé stessa fino al momento in cui la guardiamo e ne osserviamo il valore.

2.1.1 Interpretazione Geometrica e Sfera di Bloch

Per ottenere una visualizzazione utile del qubit possiamo immaginare che tutti i possibili stati siano posizionabili sulla superficie di una sfera di raggio unitario chiamata Sfera di Bloch, della quale i due poli rappresentano rispettivamente i due stati fondamentali. Questa rappresentazione intuitiva è alle volte molto utile per interpretare le operazioni effettuate sul singolo qubit.

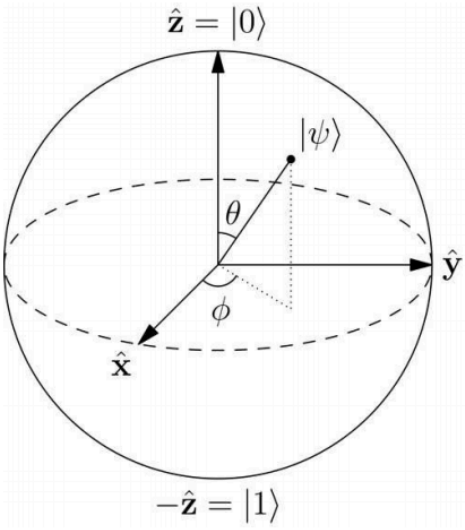


Figura 1-Sfera di Bloch usata per la rappresentazione spaziale del qubit

Prendendo come riferimento l'immagine rappresentata in figura, è possibile stabilire una corrispondenza biunivoca tra la rappresentazione di un generico stato del qubit:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

e la sua rappresentazione sulla superficie della sfera in $\mathbb{R}^3$:

$$|\psi\rangle = \cos(\theta/2)\,|0\rangle + e^{i\varphi}\sin(\theta/2)\,|1\rangle$$

dove θ e φ rappresentano i numeri reali che identificano le coordinate sferiche del punto. Precedentemente abbiamo detto che un qubit è rappresentato da un vettore in uno spazio vettoriale complesso; richiamiamo quindi la definizione di numero complesso:

$$z = a + ib$$

con $i = \sqrt{-1}$, a definita come parte Reale di z e b parte Immaginaria. La *norma* o *modulo* di z è $|z| = \sqrt{a^2 + b^2}$ e il *coniugato* di z è $z^* = a - ib$. Un numero complesso $z \in \mathbb{C}$ si può vedere come un punto nel piano complesso in un semplice grafico bidimensionale dove l'asse *x* rappresenta i numeri reali e l'asse *y* lo spazio complesso. Essendo le ampiezze degli

stati di un qubit numeri complessi tali per cui $|\alpha|^2 + |\beta|^2 = 1$, possiamo fornire una loro rappresentazione in coordinate polari sfruttando la formula di Eulero:

$$e^{i\varphi} = \cos(\varphi) + isin(\varphi)$$

possiamo riscrivere l'equazione del qubit come

$$|\psi\rangle = e^{i\gamma}(\cos(\theta/2)\,|0\rangle + e^{i\varphi}\sin(\theta/2)\,|1\rangle)$$

dove φ, θ e γ sono numeri reali. Inoltre, potendo trascurare il termine di fase globale, senza valore aggiunto dal punto di vista fisico, in quanto non ha effetti osservabili per il principio quantistico di misurazione [1], l'osservazione degli stati $e^{i\varphi}|\psi\rangle$ e $|\psi\rangle$ risulteranno quindi del tutto equivalenti.
Possiamo dunque validare la rappresentazione di un qubit, corrispondente ad un punto della Sfera di Bloch come:

$$|\psi\rangle = \cos(\theta/2)\,|0\rangle + e^{i\varphi}\sin(\theta/2)\,|1\rangle$$

Questa descrizione apparentemente astratta che abbiamo ottenuto, ha una sua corrispondenza nel mondo fisico reale: un qualsiasi sistema fisico con almeno due livelli di energia discreti e sufficientemente separati può effettivamente essere utilizzato per la rappresentazione del qubit.

2.1.2 Principio di Misurazione

Prima di proseguire con ulteriori componenti facciamo un inciso sull'importanza che l'operazione di misurazione ha nel mondo quantistico in relazione alla sovrapposizione di stati di un sistema, infatti proprio gli effetti della *misurazione* rappresentano uno dei postulati della meccanica quantistica. Per rendere chiara l'idea immaginiamo che una particella sia dotata di un numero finito possibile di stati base e che tale particella li possegga tutti contemporaneamente fin quando non avviene l'evento della misurazione che

farà ottenere uno degli stati base con probabilità uguale al quadrato del coefficiente associato a tale stato. Immaginando la matrice *O* come la matrice contenente tutti i valori osservabili (autovalori) che attua sullo stato quantistico $|\psi\rangle$, l'effetto della misurazione sarà quello di discretizzare tra tutti gli stati possibili. Descrivendo l'evento della misurazione con una freccia, quello che abbiamo è:

$$O|\psi\rangle \rightarrow \lambda_i|a_i\rangle$$

Dove $|a_i\rangle$ può essere interpretato come quello stato preciso in cui il sistema si trova nell'istante della misurazione, ossia in cui è avvenuto un *collasso* della sovrapposizione $|\psi\rangle$ ad un preciso stato $|a_i\rangle$. È dunque evidente la differenza tra il concetto classico di misurazione e quello quantistico, riassunto di seguito con una serie di assunzioni che le caratterizzano.
In una *misurazione classica*:

- teoricamente, la misurazione lascia il sistema inalterato, nello stesso stato in cui si trovava al principio;
- il risultato della misurazione è predicibile, significa che se l'esperimento effettuato venisse ripetuto rispettando le condizioni, ci aspetteremmo esattamente lo stesso risultato.

Queste due assunzioni mutano passando al mondo quantistico, dove la *misurazione quantistica* risulta essere:

- un'operazione irreversibile che trasforma lo stato generale (sovrapposizione) $|\psi\rangle$ in un'autovettore $|a_i\rangle$. Oppure che lo stato $|\psi\rangle$ collassa nell'autovettore $|a_i\rangle$.
- Il risultato della misurazione è incerto ma sempre appartenente ad uno degli autovalori λ_i delle misure osservabili in *O*. L'autovalore sarà misurato con probabilità $|\langle a_i|\psi\rangle|^2$, dove $|a_i\rangle$ è l'auto vettore corrispondente all'autovalore λ_i.

2.2 Registri quantistici

Esattamente come un insieme di bit classici è memorizzato in un *registro*, una collezione di *n* qubit è immagazzinato in una struttura che chiamiamo *registro quantistico* di dimensione *n*. Assumendo che tutti gli *n* qubit di un determinato insieme si trovino in uno dei due stati fondamentali, possiamo rappresentare esattamente tutti i 2^n stati disponibili, esattamente come accade nella codifica binaria classica. Per la rappresentazione di registri e unione di qubit si utilizza il **prodotto tensore** '⊗', un operatore che combina spazi vettoriali di una certa dimensione per generarne dei più grandi; tale operatore è dunque applicabile anche ad oggetti rappresentati in forma matriciale e questo tornerà utile quando parleremo di gate e della loro applicazione su singoli qubit.
Formalmente, e come descritto per un qubit, un registro quantistico di *n* qubit è un elemento dello spazio di Hilbert 2^n-dimensionale, con base computazionale formata da 2^n registri a *n* qubit tali per cui:

$$|i_1\rangle \otimes |i_2\rangle \otimes |i_3\rangle \otimes \dots |i_n\rangle$$

Se il nostro sistema dispone di due qubit possiamo quindi costruire una base computazionale dello spazio degli stati formata dai seguenti vettori:

$$|0\rangle \otimes |0\rangle = |00\rangle = |0\rangle$$

$$|0\rangle \otimes |1\rangle = |01\rangle = |1\rangle$$

$$|1\rangle \otimes |0\rangle = |10\rangle = |2\rangle$$

$$|1\rangle \otimes |1\rangle = |11\rangle = |3\rangle$$

che in notazione algebrica corrispondono a:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

Richiamando la proprietà di sovrapposizione degli stati, possiamo ricavare una caratteristica fondamentale di questi registri, ossia il poter memorizzare contemporaneamente tutti gli stati simultaneamente. Per farlo è necessario mettere in sovrapposizione i singoli qubit:

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\otimes|0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + |2\rangle).$$

In questo caso abbiamo applicato la sovrapposizione solo al primo qubit, in modo da ottenere simultaneamente lo stato 00 e lo stato 10. Ripetendo il processo su tutti gli *n* qubit, faremmo in modo di preparare un registro a *n* qubit a contenere contemporaneamente tutti gli 2^n stati disponibili. In quel caso la sovrapposizione avrebbe la forma data da:

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\otimes\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\otimes \dots \otimes\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

tante volte quanti i qubit moltiplicati tra loro con il prodotto tensoriale.
Si noti che per mezzo della sovrapposizione e del prodotto tensoriale lo stato viene sempre normalizzato per avere lunghezza unitaria. Un altro dettaglio importante è che effettuando la misurazione su uno dei qubit del registro (ad esempio il primo) e ottenendo valore 1, lo stato generale del sistema otterrà un collasso parziale in tutti quei possibili stati aventi il primo qubit a valore 1, eliminando quelli di valore 0.
A scopo descrittivo si introducono anche altre rappresentazioni equivalenti spesso utilizzate in letteratura per la rappresentazione degli stati quantistici del sistema:

$$x_0|00\dots0\rangle + x_1|00\dots01\rangle + \dots + x_{n-2}|01\dots11\rangle + x_{n-1}|11\dots11\rangle,$$

$$x_0|0\rangle + x_1|1\rangle + x_2|2\rangle + x_3|3\rangle + x_4|4\rangle + x_5|5\rangle + \dots + x_{n-1}|2^n - 1\rangle,$$

$$\sum_{i=0}^{n-1} x_i|i\rangle.$$

2.3 Entanglement

Dopo aver introdotto il qubit e i registri quantistici, è fondamentale descrivere una ulteriore proprietà legata ai possibili stati in cui può trovarsi il sistema; essi rappresentano un'eccezione in quanto non sono ritrovabili nei componenti della fisica classica. Questi stati chiamati *entangled* rappresentano quelle possibili configurazioni di *n* qubit componenti che non hanno un proprio stato ben definito ma solamente la loro combinazione ne rappresenta uno concreto. In altre parole significa che uno *stato entangled* non può essere descritto come prodotto tensore degli stati dei singoli componenti.
Questa proprietà, propria della fisica quantistica, permette che stati entangled siano connessi a prescindere dalla distanza fisica che gli separa di modo che una misurazione o un'operazione su uno dei componenti fornisca all'istante informazioni sulla sua coppia.
Un classico esempio per spiegare questa proprietà è dato dal seguente stato:

$$|00\rangle + |11\rangle,$$

in questo caso non esiste un modo di rappresentare lo stato attraverso il prodotto tensore dei due singoli qubit, in quanto non esistono dei coefficienti $\alpha_1, \alpha_2, \beta_1, \beta_2$, tali per cui valga:

$$|00\rangle + |11\rangle = \big(\alpha_1|0\rangle + \beta_1|1\rangle\big)\otimes\big(\alpha_2|0\rangle + \beta_2|1\rangle\big).$$

L'entanglement è alla base della risoluzione di alcuni di quei problemi informatici non riproducibili classicamente e della possibilità di ottenere un aumento esponenziale nella capacità di calcolo. Nel seguito verrà descritto come è possibile a livello circuitale ottenere questo genere di sovrapposizioni attuando sui singoli qubit.

2.4 Porte Logiche Quantistiche

Dopo aver introdotto una rappresentazione basica degli stati quantistici e dei registri, passiamo ad una descrizione delle principali porte logiche quantistiche elementari che sono

state realizzate e più comuni in letteratura, con le quali è possibile realizzare circuiti e implementare algoritmi che, allo stesso modo degli operatori classici con i bit, gestiscono le operazioni aventi operandi composti da qubit. Uno stato entrante in un gate di un circuito quantistico ne uscirà con un altro stato descrivendo così un'evoluzione nel tempo dello stato del qubit.
I gate quantistici devono soddisfare due criteri fondamentali:

- La somma dei quadrati delle norme delle probabilità deve essere conservata rimanendo uguale ad uno anche dopo l'applicazione del gate;
- Deve esserci reversibilità: ogni evoluzione dello stato quantistico deve essere reversibile.

Un altro aspetto fondamentale di cui tener conto nella definizione dei gate quantistici è quello di dover creare un insieme di operatori capaci di racchiudere l'universalità delle operazioni eseguibili con il calcolatore. Una volta raggiunto questo scopo, ne segue automaticamente la possibilità di ricreare ogni operazione classica attraverso un modello quantistico, il che dimostra l'equivalenza dei due modelli di computazione.
Di seguito sono introdotti gli operandi a singolo qubit e a ingressi multipli.

2.4.1 Gate Single Qubit

Richiamando gli operatori classici, ne abbiamo a disposizione uno solo per operare su un singolo elemento, il NOT, che semplicemente nega il valore corrente del bit invertendo lo stato del in entrata (NOT(0)→1, NOT(1)→0). Questa operazione è semplice da realizzare anche nel caso quantistico per quanto riguarda gli stati fondamentali, si dovrà inoltre introdurre la possibile gestione di quegli stati che essendo sovrapposti, sono caratterizzati dai coefficienti α e β; intuitivamente l'applicazione della porta logica NOT al qubit, ne scambierà i due coefficienti.
Immaginando di rappresentare in forma vettoriale il qubit, e definendo la matrice corrispondente al NOT quantistico come:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix},$$

è facilmente verificabile che applicando tale porta a un qubit nella forma $\alpha|0\rangle + \beta|1\rangle$ otterremo, seguendo la notazione vettoriale:

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}.$$

È di fondamentale importanza ricordare che la validità della condizione di normalizzazione rimanga verificata anche su qualsiasi qubit si trovi all'uscita di una qualsiasi porta logica. Non tutte le matrici 2x2 sono infatti validi operatori applicabili a un qubit, *l'unitarietà* è quella proprietà delle matrici che garantisce la trasformazione di un vettore unitario in un altro vettore sempre unitario e che quindi rispetta le condizioni di un gate.
Contrariamente al caso classico esistono altre due porte quantistiche che utilizzano un solo bit; la prima è la porta *Z* definita dalla seguente matrice:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix},$$

la quale opera cambiando di segno esclusivamente alla componente $|1\rangle$.
Un'altra è la matrice unitaria:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix},$$

che mappa $|0\rangle$ in $i|1\rangle$ e $|1\rangle$ in $-i|0\rangle$.
in generale *X*, *Y* e *Z* sono chiamate matrici di Paoli e fisicamente rappresentano le componenti x, y, z dello spin di un elettrone.
L'ultima e importantissima porta a singolo qubit è chiamata porta di Hadamard, definita:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

generalmente è la più utilizzata e si incarica di trasformare uno stato base in una sovrapposizione tale che il nuovo stato si trovi con un 50% di probabilità in uno dei due stati fondamentali. Ad esempio, applicando *H* a |0⟩ o |1⟩ si otterrebbe:

$$H\begin{bmatrix}1\\0\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}},$$

$$H\begin{bmatrix}0\\1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\-1\end{bmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Rispettivamente queste due nuove rappresentazioni di sovrapposizioni vengono definite |+⟩ e |-⟩ sempre in relazione alla posizione della Sfera di Bloch.
Si osserva che applicando due volte consecutive la porta di Hadamard si ritorna allo stato di partenza; sapendo infatti che nella Sfera di Bloch tale operazione comporta una rotazione di 90° attorno all'asse y seguita da una riflessione sul piano *x*, *y*, applicando a |0⟩ la rete H^2, il risultato finale rimane inalterato.

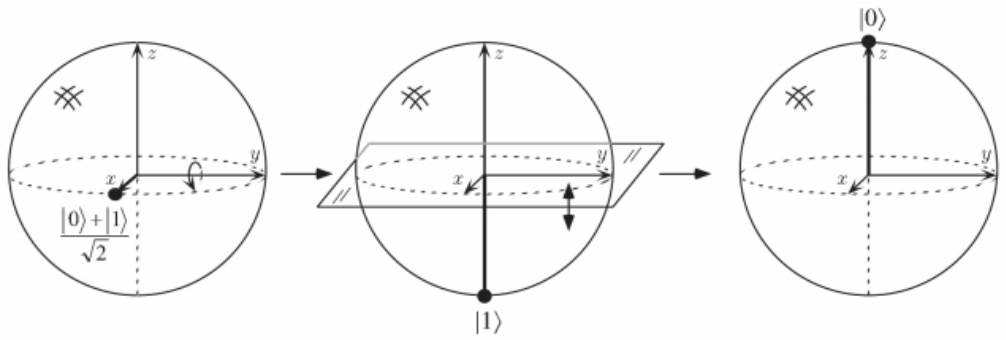


Figura 2-Effetto della doppia applicazione dell'operatore Hadamard

Riassumendo, abbiamo visto che la tecnologia quantistica, a differenza di quella classica, ci permette di definire infinite porte logiche a singolo qubit. Infatti è possibile immaginare ogni operazione come serie di rotazioni specifiche attorno agli assi definendo correttamente i valori degli angoli e rispettando l'unitarietà della matrice operando.

2.4.2 Gate Multi Qubit

Nella rappresentazione classica, un registro è composto da una serie di bit che rappresentano gli operatori nel momento in cui effettuiamo una operazione che vede coinvolti due registri; in questo caso avvengono un numero finito di operazioni unitarie sfruttando le porte logiche classiche a due bit che per inciso sappiamo essere: AND, OR, XOR, NAND, NOR. Dalla teoria dei circuiti è noto che ogni funzione booleana che siamo in grado di progettare, può essere equivalentemente rappresentata ed eseguita attraverso l'utilizzo di porte AND e NOT, quindi sono realizzabili con sole porte NAND che per questo motivo rappresentano un *insieme universale*.

Analogamente, nel quantum computing, siamo capaci di formulare una porta universale attraverso l'utilizzo del *NOT controllato*, o più correttamente chiamato **CNOT**. In comparazione alle porte classiche il CNOT rappresenterebbe un XOR, operazione capace di rappresentare solamente un sottoinsieme della totalità delle funzioni booleane.

Passando l'attenzione alle caratteristiche intrinseche del CNOT, esso è dotato di due qubit in ingresso, rispettivamente definiti *controllo* e *bersaglio* (o *target*); dunque nel caso il qubit controllo si trovi nello stato zero allora il target viene lasciato inalterato, al contrario, se il qubit controllo è nello stato uno, allora il target viene invertito. Tale trasformazione può essere scritta $|A, B\rangle \mapsto |A, B \oplus A\rangle$.

La sua rappresentazione circuitale è mostrata di seguito:

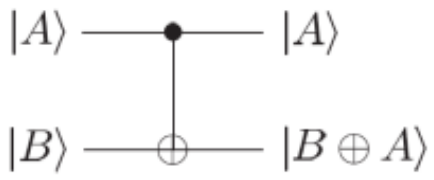


Figura 3-Rappresentazione grafica del circuito che implementa un controlled-NOT

mentre la rappresentazione matriciale dell'operatore è:

$$U_{CN} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

dove effettivamente possiamo notare come gli ultimi due stati vengano rispettivamente invertiti.
Numericamente parlando abbiamo la seguente operazione:

$$|00\rangle \rightarrow |00\rangle$$

$$|01\rangle \rightarrow |01\rangle$$

$$|10\rangle \rightarrow |11\rangle$$

$$|11\rangle \rightarrow |10\rangle$$

Una delle proprietà fondamentali delle porte quantistiche, in particolare del CNOT e di tutte le porte viste a singolo qubit, è quella di essere invertibili, infatti a differenza delle porte classiche XOR e NAND generalmente irreversibili, permettono di ottenere l'input avendo a disposizione il valore di output. Combinando opportunamente CNOT e porte a singolo qubit, otteniamo l'insieme dei gate necessari per definire un insieme universale, capace dunque di inglobare le operazioni sufficienti alla rappresentazione di tutte le porte logiche quantistiche e quindi l'universalità delle operazioni quantistiche.

2.5 Circuiti Quantistici

Dopo aver visto la rappresentazione delle porte logiche fondamentali per la realizzazione di operazioni e funzioni, passiamo ora a descrivere la realizzazione dei circuiti quantistici. Esistono ovviamente differenze sostanziali a livello hardware con quelli classici, soprattutto per quanto riguarda la rappresentazione grafica ed il significato intrinseco che ne deriva. Un primo semplicissimo circuito è quella della misurazione dello stato in cui si trova un qubit. Come detto precedentemente, nonostante il nostro qubit sia dato dalla forma $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, all'evento della misurazione ciò che vediamo è un bit classico M con uno stato preciso 0 o 1 dettato dalle probabilità rispettive $|\alpha|^2$ e $|\beta|^2$. Per questo motivo nella rappresentazione della misurazione, il bit classico è rappresentato con due linee.

Solitamente ogni circuito comincia con una serie di *n* qubit inizializzati nello stato |0⟩, ciò implica che applicando la misurazione si ottiene sicuramente il risultato |0⟩.

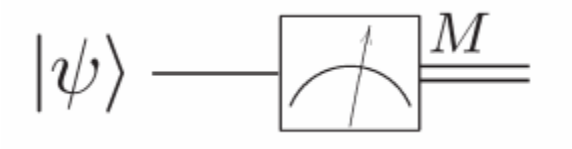


Figura 4-Rappresentazione grafica della misurazione in un circuito quantistico

Un altro semplice circuito molto utile ai fini pratici è quello chiamato di *scambio* (**SWAP**), rappresentato graficamente nella seguente immagine.

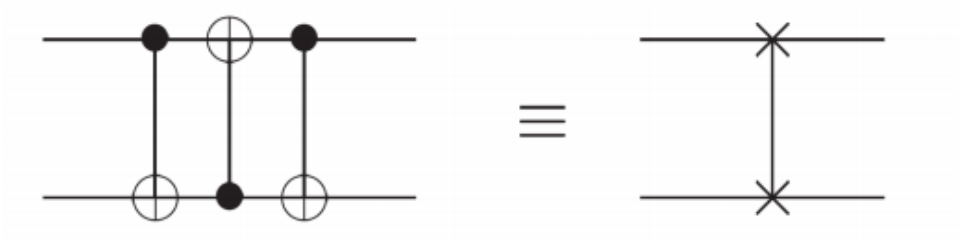


Figura 5-Equivalenza di tre controlled-NOT per ottenere un gate SWAP

Questo circuito, già a livello grafico, mostra come tramite l'utilizzo ripetuto del gate CNOT realizzi facilmente l'operazione di prende in ingresso un registro di due qubit e scambiarne i rispettivi stati. Praticamente, lo scambio si realizza con l'applicazione di tre gate CNOT e seguendo l'evoluzione partendo da un registro nella forma $|a, b\rangle$ viene applicato il primo gate con qubit di controllo a e target b, dunque con la prima operazione si ottiene la seguente sostituzione:

$$|a, b\rangle \rightarrow |a, a \oplus b\rangle,$$

a questo punto il secondo qubit oggetto del rimpiazzo viene preso come controllo dal secondo CNOT nel quale il target è rappresentato da a, di conseguenza tramite la seconda operazione otteniamo:

$$|a, a \oplus b\rangle \rightarrow |a \oplus (a \oplus b), a \oplus b\rangle,$$

dove essendo $a \oplus (a \oplus b) = b$, applichiamo un ultimo CNOT avente come qubit controllo b e come target $a \oplus b$ dove con la semplificazione per cui $(a \oplus b) \oplus b = a$, otteniamo l'effettivo scambio dei due qubit:

$$|b, a \oplus b\rangle \rightarrow |b, (a \oplus b) \oplus b\rangle = |b, a\rangle.$$

Lo SWAP è spesso utilizzato a seguito dell'applicazione di determinate operazioni per riottenere l'ordine effettivo dei qubit. Si riporteranno esempi significativi nei paragrafi successivi.

Possiamo rappresentare un altro componente generalizzato chiamato *controlled-U*, esso può essere visto come una estensione della già descritta porta CNOT ad un livello più circuitale, di fatto viene rappresentato con il caratteristico cerchietto nero il qubit di controllo mentre tutti gli *n* qubit target sono inseriti come input nel modulo U, che rappresenta una operazione unitaria da effettuare sugli *n* qubit. Quando introdurremo algoritmi più complessi questa rappresentazione sarà molto utile.

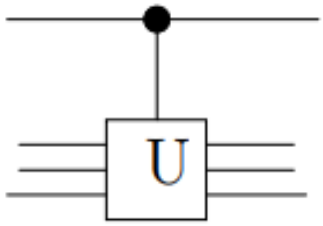


Figura 6-Generalizzazione di un'operazione controllata su n qubit

Altra importante proprietà che viene alla luce con l'introduzione ai circuiti quantistici, è l'impossibilità, a differenza del modello classico, di realizzare la copia di un qubit. Questo fatto è verificato dal *Teorema del No-Cloning*, per il quale è impossibile creare un circuito utilizzando un CNOT nel quale un qubit di controllo contenente $|\psi\rangle$ venga copiato nel qubit target posto inizialmente nello stato $|0\rangle$. Di fatto questo funzionerebbe per i bit classici o per un qubit, solo nel caso in cui si trovi in uno stato fondamentale ma non per un qualsiasi sovrapposizione nella forma $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. In quest'ultimo caso, il circuito descritto nell'ipotesi rappresenterebbe inizialmente lo stato $|\psi\rangle|0\rangle$ ed essendo il nostro obiettivo quello di trovarci nel qubit target $|\psi\rangle$, ossia il risultato finale $|\psi\rangle|\psi\rangle$, esso varrebbe:

$$|\psi\rangle|\psi\rangle = \alpha^2|00\rangle + \alpha\beta|01\rangle + \alpha\beta|10\rangle + \beta^2|11\rangle,$$

è diverso rispetto al risultato $\alpha|00\rangle + \beta|11\rangle$ valido solo nel caso in cui $\alpha\beta = 0$. Da qui si evince l'impossibilità di copiare un qubit.

2.5.1 Stati di Bell

Passiamo alla descrizione di circuiti quantistici più complessi, utilizzati soprattutto per la costruzione di stati computazionali che non hanno alcuna rappresentazione nella controparte classica e che vengono applicati per realizzare fenomeni paradossali secondo le leggi della fisica classica. Stiamo parlando di quegli stati definiti *entangled*, precedentemente enunciati tra le caratteristiche uniche di un qubit e della meccanica quantistica.
Un primo circuito permette la trasformazione dei quattro stati computazionali appartenenti a due qubit in altrettanti quattro *stati* chiamati *di Bell* o *coppie EPR* (dai nomi Einstein, Podolsky, Rosen che per primi ne interpretarono le straordinarie proprietà).

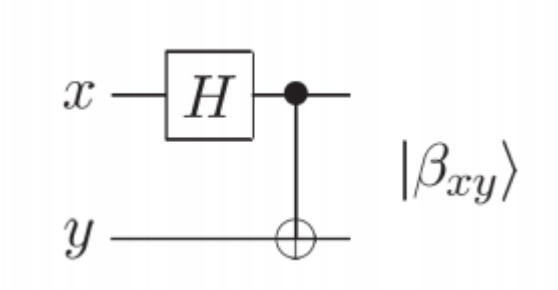


Figura 7-Circuito per ottenere uno stato di Bell (Entangled)

Come abbiamo visto nella descrizione dei gate a un qubit, la porta Hadamard (H nella rappresentazione grafica) può essere utilizzata per ottenere una sovrapposizione di stati. Se immaginiamo un esempio dove *x* è composto da un qubit, viene generata una sovrapposizione del tipo $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, successivamente $|x\rangle$ agisce come qubit di controllo per il CNOT che inverte il target *y* solo nel caso in cui il controllo si trovi nello stato 1. Gli

stati entangled possono essere rappresentati nella forma β_{xy}, e prendono appunto il nome di stati di Bell o EPR.
Applicando differenti combinazioni di qubit si ottengono i seguenti possibili stati:

$$|00\rangle \rightarrow |\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}},$$

$$|01\rangle \rightarrow |\beta_{01}\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}},$$

$$|10\rangle \rightarrow |\beta_{10}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}},$$

$$|11\rangle \rightarrow |\beta_{11}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}.$$

Il fatto più sorprendente e importante dal punto di vista applicativo è che una misura sul secondo qubit da sempre lo stesso risultato ottenuto dalla misura del primo: le misure sono perfettamente correlate, al 100%.
L'applicazione degli stati entangled è molto utile per la descrizione di fenomeni particolari come il Teletrasporto quantistico.

2.5.2 Teletrasporto Quantistico

Quando si parla di teletrasporto quantistico, ci si riferisce ad una tecnica utilizzabile per trasportare stati quantistici da due differenti località spaziali sfruttando solo la trasmissione di bit classici; intuitivamente si può pensare a questa tecnica come ad una soluzione al problema dell'impossibilità di copiare lo stato di un qubit descritta in precedenza, effettivamente non poter copiare un qubit non significa che non possa essere teletrasportato a qualsiasi distanza.
Per interpretare meglio il significato del metodo e i problemi ai quali può essere applicato immaginiamo una classica situazione di comunicazione tra Alice e Bob. L'obbiettivo di

Alice è far conoscere all'amico Bob lo stato di un qubit che possiede. Inizialmente Alice non ne conosce lo stato e per il *Teorema del no-cloning* sappiamo che le è impossibile eseguirne una copia; come ulteriore vincolo, Alice può inviare a Bob solamente informazione in forma classica, quindi utilizzando bit classici con valori 0 o 1. Il problema sarebbe impossibile se non fosse che i due amici possiedono un ulteriore qubit di una coppia EPR (entanglement) generata precedentemente senza alcun vincolo di utilizzo. Questa ipotesi fondamentale e l'applicazione delle proprietà degli stati entangled risolvono il problema per mezzo del circuito:

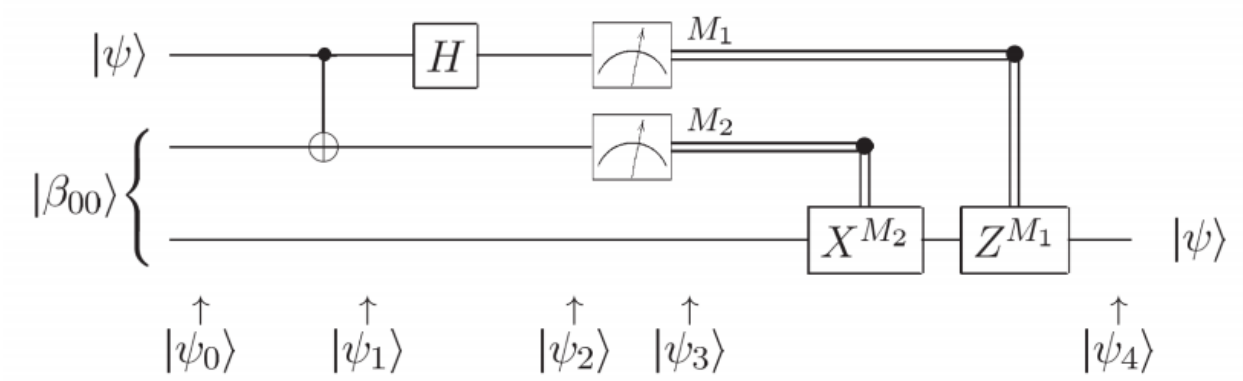


Figura 8-Circuito per la realizzazione del Teletrasporto quantistico

Consideriamo di voler trasmettere un qubit nella forma $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ per il quale sia Alice che Bob ignorano i valori dei coefficienti α e β; nello stato di input del sistema raffigurato Alice possiede il qubit $|\psi\rangle$ e la sua parte del qubit appartenente alla coppia EPR, mentre l'ultima linea raffigura il qubit appartenente a Bob facente parte dell'entanglement.

$$|\psi_0\rangle = |\psi\rangle|\beta_{00}\rangle = \frac{1}{\sqrt{2}}[\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|00\rangle + |11\rangle)],$$

dunque dopo l'operazione del primo CNOT applicato al suo qubit della coppia EPR, Alice si trova con:

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}[\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|10\rangle + |01\rangle)],$$

di seguito il primo qubit di Alice entra in una porta Hadamard facendole ottenere una ulteriore sovrapposizione e uno stato del sistema nella forma:

$$|\psi_2\rangle = \frac{1}{2}[\alpha(|0\rangle + |1\rangle)(|00\rangle + |11\rangle) + \beta(|0\rangle - |1\rangle)(|10\rangle + |01\rangle)],$$

ora se raccogliamo i qubit di Alice e riscriviamo l'equazione, possiamo scriverla come:

$$|\psi_2\rangle = \frac{1}{2}[|00\rangle(\alpha|0\rangle + \beta|1\rangle) + |01\rangle(\alpha|1\rangle + \beta|0\rangle) + |10\rangle(\alpha|0\rangle - \beta|1\rangle) + |11\rangle(\alpha|1\rangle - \beta|0\rangle)].$$

A questo punto Alice effettua l'operazione di misurazione dei due qubit potendo ottenere quattro possibili differenti configurazioni (00, 01, 10, 11), inoltre proprio per effetto della misurazione di Alice dei due suoi qubit, anche il qubit posseduto da Bob collassa nello stato corrispondente al risultato della misurazione per effetto dell'entanglement. I possibili stati saranno dati da:

$$00 \rightarrow |\psi_3\rangle = \alpha|0\rangle + \beta|1\rangle,$$

$$01 \rightarrow |\psi_3\rangle = \alpha|1\rangle + \beta|0\rangle,$$

$$10 \rightarrow |\psi_3\rangle = \alpha|0\rangle - \beta|1\rangle,$$

$$11 \rightarrow |\psi_3\rangle = \alpha|1\rangle - \beta|0\rangle.$$

Quindi Alice comunica i due bit misurati (*n*, *m*) attraverso l'utilizzo di un canale di telecomunicazione classico. Al riceverli, Bob potrà ricavare il valore originale di $|\psi\rangle$ attraverso l'applicazione di un circuito formato da X^n e Z^m del quale si attiveranno nessuna, una delle due o entrambe le porte a seconda della configurazione di input ricevuta. Questa tecnica oltre a non violare il concetto di no-cloning, rispetta anche la relatività ristretta, in quanto nonostante l'apparente capacità di comunicazioni istantanee a velocità maggiori di quelle della luce, l'utilizzo di un canale classico per lo scambio di

informazione(classica) fa si che non si raggiunga nessuna violazione della famosa legge di Einstein.

2.6 Computazione Classica e Quantistica

Dopo aver visto una basica introduzione dei circuiti quantistici, passiamo ad analizzare la differenza sostanziale tra classici e quantistici accennata trattando i gate: *l'irreversibilità*.
Una funzione è detta reversibile quando conoscendo la sua uscita, si può sempre determinare la sua entrata, dunque la reversibilità implica la conservazione dell'informazione. Sapendo che le porte classiche come AND, XOR, NAND sono irreversibili, in quanto è impossibile recuperare l'informazione in entrata, e avendo stabilito che le porte quantistiche, godendo delle proprietà di unitarietà, sono sempre reversibili dobbiamo definire correttamente tutto ciò che necessitiamo per affermare che abbiamo a disposizione un modello computazionale alternativo (Quantum Computing) che sia in grado di inglobare tutte le possibili computazioni classiche.
Il primo obiettivo è quello di rappresentare tutte le computazioni classiche come **trasformazioni unitarie**, cioè farle equivalere a computazioni quantistiche, eliminando dunque l'irreversibilità che le contraddistingue e renderle reversibili.
Una qualsiasi computazione classica irreversibile si può trasformare in una computazione equivalente ma reversibile usando la *porta di Toffoli*. Classicamente essa è reversibile ed opera su tre bit dei quali due sono bit di controllo che non vengono mai modificati dal gate e un bit target che invece viene invertito solamente quando entrambi i controlli valgono uno altrimenti il risultato non subisce variazioni:

$$(a, b, c) \rightarrow (a, b, c \oplus ab) \rightarrow (a, b, c)$$

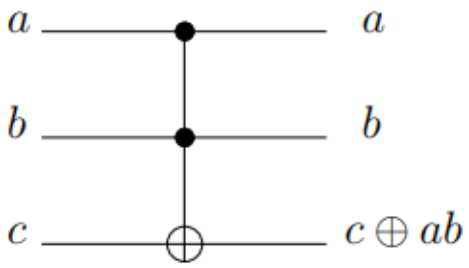


Figura 9-Rappresentazione della Porta di Toffoli

La porta di Toffoli è universale per le computazioni classiche reversibili, attraverso il suo utilizzo è possibile trasformare una qualsiasi computazione classica irreversibile in una reversibile. Così come per NAND e FANOUT, la costruzione di un qualsiasi circuito reversibile per un'operazione classica *f* mediante la porta di Toffoli comporta l'utilizzo di alcuni bit di servizio in input (*Ancilla* bits, nome che sarà ricorrente nel campo applicativo) e in output (*Garbage*). Eliminati tali supporti il circuito esegue la trasformazione.

$$(x, y) \mapsto \big(x, y \oplus f(x)\big).$$

In particolare la porta di Toffoli può essere implementata come circuito quantistico: analogamente vengono utilizzati tre qubit che operano seguendo lo stesso comportamento descritto per la controparte classica. La permutazione è racchiusa nella matrice unitaria:

$$U_{Toffoli} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Quello che la matrice fa è esattamente invertire il terzo qubit se entrambi i qubit di controllo si trovano nello stato |1⟩. Per esempio, l'effetto dell'applicazione allo stato |110⟩ sarebbe di ottenere lo stato |111⟩ e viceversa. La porta di Toffoli quantistica sarà utilizzata per la simulazione di porte classiche invertibili, assicurando la simulazione corretta di qualsiasi circuito classico attraverso un calcolatore quantistico.

Capitolo 3

Algoritmi Quantistici

Dopo aver descritto le principali proprietà del computer quantistico e parte dei componenti per rappresentare e manipolare l'informazione, entriamo nel dettaglio delle caratteristiche computazionali di questo modello parlando dei vantaggi che si posso ottenere con la sua applicazione ed in particolare degli algoritmi quantistici più importanti presenti in letteratura che rendono così performante la computazione quantistica e che, come vedremo nel dettaglio nei prossimi capitoli, porteranno ad uno speed-up ed un miglioramento delle prestazioni relativo alla complessità computazionale in confronto agli algoritmi classici, soprattutto nell'ambito della gestione di grosse quantità di informazioni come nel campo del Machine Learning e dei problemi di ottimizzazione.

Negli ultimi anni, grazie allo sviluppo e al perfezionamento delle tecnologie quantistiche, gli studi dei ricercatori si sono prettamente focalizzati su come le tecnologie e la computazione quantistica possano essere utilizzate per incrementare le performance degli algoritmi di Data Mining e Machine Learning, di fatto come è stato introdotto nel capitolo riguardante la computazione quantistica, essa sfrutta alcuni effetti quantistici come la sovrapposizione di stati e l'entanglement per risolvere alcuni set di problemi più velocemente di un computer classico. Nonostante i computer quantistici siano ancora in fase di sperimentazione, gli algoritmi quantistici sono stati ampiamente studiati negli ultimi due decenni. Questi algoritmi riguardano problemi di ottimizzazione che possono essere utilizzati per accelerare gli algoritmi di Machine Learning. Ovviamente uno dei problemi fondamentali per un 'non fisico' sta nella necessità di acquisire conoscenze delle teorie quantistiche e di notazioni scientifiche di non facile comprensione che possono creare barriere cognitive sull'interpretazione, sui benefici e sui limiti dell'uso di questi nuovi algoritmi.

In questa sezione verranno introdotte le classi di complessità alle quali un problema può appartenere in base alle caratteristiche risolutive dopodiché si analizzeranno alcuni dei

principali algoritmi quantistici presenti in letteratura maggiormente applicati come subroutine agli algoritmi di Machine Learning.
Saranno analizzati:

- Algoritmo di Deutsch
- Algoritmo Deutsch-Jozsa
- Algoritmo di ricerca di Grover;
- Algoritmo quantistico della Trasformata di Fourier;
- Algoritmo quantistico di Stima di Fase;
- Algoritmo di Ordinamento
- Algoritmo di Shor per la fattorizzazione.

Teoricamente, ogni versione quantistica è almeno quadratamente o esponenzialmente più veloce della sua controparte classica. Come utilizzato in precedenza e come solitamente si osserva in letteratura per misurare l'efficienza di un algoritmo viene utilizzata la complessità temporale con la notazione O-grande. Questa notazione spiega quanto rapidamente la complessità temporale di un algoritmo cresce quando il numero di elementi da processare è arbitrariamente grande, dove per numero di elementi si fa riferimento alla dimensione dell'input, al numero di gate quantistici o al numero di iterazioni utilizzate a seconda di quale fattore influisca realmente sull'efficienza dell'algoritmo in esame.
Per ogni algoritmo verrà fatta una breve introduzione seguita da una spiegazione step by step dell'esecuzione su un semplice esempio dopodiché se ne descriverà a livello generale la parte quantistica e infine le conclusioni riguardo alla complessità computazionale e ulteriori commenti rilevanti.

3.1 Parallelismo Quantistico

L'utilizzo di questo nuovo modello computazionale permette di sfruttare una sua caratteristica fondamentale chiamata *parallelismo quantistico*. Fondamentalmente permette ad un algoritmo applicato su un circuito quantistico di valutare una funzione $f(x)$ su più valori di x contemporaneamente.

Consideriamo una qualsiasi funzione booleana con dominio e codominio descritto da un bit come:

$$f(x): \{0,1\} \mapsto \{0,1\}.$$

Per effettuare il calcolo di $f(x)$ utilizzando la computazione quantistica è necessario, come visto nel capitolo precedente, definire una trasformazione $f(x)$ nella forma di una trasformazione unitaria U_f. Tale trasformazione è rappresentata come un circuito composto da una sequenza di porte logiche quantistiche che sarà applicata ad uno stato di input $|x, y\rangle$ trasformandolo nello stato $|x, y \oplus f(x)\rangle$, detto registro target. Il circuito idealizzato è mostrato in Figura 9:

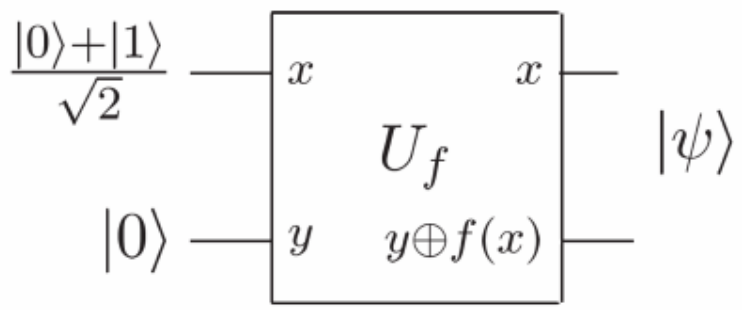


Figura 10-Circuito per l'ottenimento del parallelismo quantistico

Ponendo y=0, lo stato finale del secondo qubit conterrà esattamente $f(x)$. Se nel circuito in figura mettiamo come valore di x uno stato in sovrapposizione, ottenuto tramite l'applicazione di un Hadamard, lo stato finale che ne deriva dall'applicazione di U_f sarà:

$$\frac{|0, f(x)\rangle + |1, f(x)\rangle}{\sqrt{2}}.$$

Lo stato $|\psi\rangle$ contiene quindi informazioni sia sul valore di $f(0)$ che su $f(1)$. Questo genere di parallelismo sull'applicazione di una funzione non è replicabile classicamente, dove per effettuare un calcolo simultaneo vengono eseguiti più circuiti contemporaneamente con diversi valori di x. Il metodo appena descritto può essere generalizzato per il calcolo di funzioni su un numero arbitrario di configurazioni utilizzando una estensione della porta di Hadamard nota come *trasformata di Walsh-Hadamard*. Presi n qubit, vengono applicate n

porte Hadamard, una per ognuno, in modo da ottenere una sovrapposizione equiprobabile di 2^n stati. Per esempio, se vogliamo applicarla a due qubit, il risultato che otteniamo è:

$$H^{\otimes 2} = H \otimes H = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle + |1\rangle}{\sqrt{2}} = \frac{|00\rangle + |01\rangle + |10\rangle + |11\rangle}{2}.$$

In generale, l'estensione della trasformata ad *n* qubit inizialmente nello stato $|0\rangle$ è:

$$H^{\otimes n} = \frac{1}{\sqrt{2^n}} \sum_x |x\rangle,$$

dove ogni valore di *x* rappresenta una delle possibili 2^n configurazioni binarie. Si osservi che per ottenere un numero 2^n di configurazioni sono necessarie solamente *n* porte di Hadamard. Una volta ottenuta la sovrapposizione degli stati e il bit output per la valutazione della funzione lasciato nello stato $|0\rangle$ applichiamo il circuito U_f:

$$\frac{1}{\sqrt{2^n}} \sum_x |x\rangle|f(x)\rangle.$$

Il parallelismo quantistico non permette però di ottenere tutti i valori calcolati per ogni *x* con una sola misurazione, infatti proprio per il principio di misurazione quantistica di cui abbiamo descritto l'effetto, otterremo il valore della funzione per un singolo valore di *x*. Per sfruttare al meglio l'informazione nascosta all'interno della sovrapposizione degli stati dobbiamo usufruire del concetto di *interferenza*, che è alla base del successo di alcuni dei più famosi algoritmi standard. Come vedremo nella descrizione di essi, la potenzialità alla base degli algoritmi quantistici va sfruttata applicando le giuste trasformazioni per ottenere risultati più efficienti di quelli ottenibili tramite circuiti classici.

3.2 Algoritmo di Deutsch

L'algoritmo di Deutsch mostra come sia possibile ricavare determinate proprietà da una valutazione parallela della funzione su tutti i suoi input, sfruttando l'interferenza tra gli stati e applicando un'estensione del circuito descritto nel parallelismo quantistico. Poniamoci l'obbiettivo di valutare se i valori della funzione sono uguali per ogni *x*, in questo caso la funzione è costante, oppure se i valori sono diversi e quindi la funzione è bilanciata.

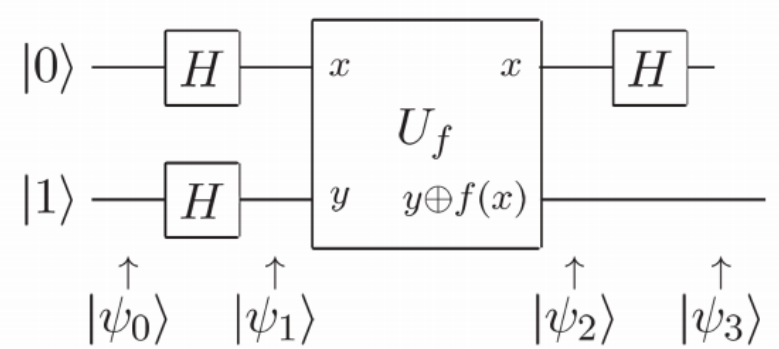


Figura 11-Circuito rappresentante l'algoritmo di Deutsch

Prendiamo come stato iniziale una sovrapposizione di stati di due qubit, inizialmente posti a |0⟩ e |1⟩ rispettivamente tali per cui:

$$|\psi_0\rangle = |0\rangle \otimes |1\rangle,$$

applichiamo a ciascuno una porta Hadamard, trasformando il sistema iniziale in:

$$|\psi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}},$$

proseguiamo con la trasformazione U_f su $|\psi_1\rangle$ in modo che rimanga valida la seguente equivalenza:

$$U_f|\psi_1\rangle = U_f\left[|x\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}\right].$$

Risolvendo l'operatore U_f otteniamo l'equazione:

$$\frac{1}{\sqrt{2}}[|x, 0 \oplus f(x)\rangle - |x, 1 \oplus f(x)\rangle],$$

$$\begin{cases} |x\rangle \dfrac{|0\rangle - |1\rangle}{\sqrt{2}} & se\ f(x) = 0 \\ -|x\rangle \dfrac{|0\rangle - |1\rangle}{\sqrt{2}} & se\ f(x) = 1 \end{cases}$$

Osserviamo che l'unica differenza è relativa al segno sulla variabile $|x\rangle$, quindi possiamo esprimere questo fatto con un semplice trucco algebrico:

$$|x\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} \xrightarrow{U_f} (-1)^{f(x)}|x\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Da qui possiamo ora avere due possibili scenari a seconda dei risultati che otteniamo dai due distinti valori della funzione $f(0)$ e $f(1)$.

$$|\psi_2\rangle = \pm \begin{cases} \dfrac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \dfrac{|0\rangle - |1\rangle}{\sqrt{2}} & se\ f(0) = f(1) \\ \dfrac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes \dfrac{|0\rangle - |1\rangle}{\sqrt{2}} & se\ f(0) \neq f(1) \end{cases}$$

Applicando un Hadamard al primo qubit e considerando alcune semplificazioni passiamo nello stato:

$$|\psi_3\rangle = \pm \begin{cases} |0\rangle \otimes \dfrac{|0\rangle - |1\rangle}{\sqrt{2}} & se\ f(0) = f(1) \\ |1\rangle \otimes \dfrac{|0\rangle - |1\rangle}{\sqrt{2}} & se\ f(0) \neq f(1) \end{cases}$$

$$\begin{cases} f(0) \oplus f(1) = 0 & se\ f(0) = f(1) \\ f(0) \oplus f(1) = 1 & se\ f(0) \neq f(1) \end{cases}$$

$$|\psi_3\rangle = \pm |f(0) \oplus f(1)\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Effettuando la misurazione del primo qubit otteniamo un risultato che equivale ad aver ottenuto una proprietà globale della funzione, cosa che nella controparte classica avrebbe necessitato almeno due valutazioni per ottenere lo stesso risultato, quindi più lento.

3.3 Algoritmo di Deutsch-Jozsa

Possiamo estendere l'algoritmo analizzato in precedenza a funzioni booleane su *n* bit della forma $f:\{0,1\}^n \longmapsto \{0,1\}$ e supponiamo di sapere che la funzione *f* può essere costante oppure bilanciata assumendo valore 1 nella metà degli input e 0 nell'altra metà.
Un algoritmo classico, impiegherebbe, nel caso peggiore, almeno $2^{n-1}+1$ valutazioni della funzione per stabilirne la classe d'appartenenza. L'algoritmo quantistico di Deutsch-Jozsa ci permette ancora una volta di stabilirlo in un solo step. La rappresentazione del circuito:

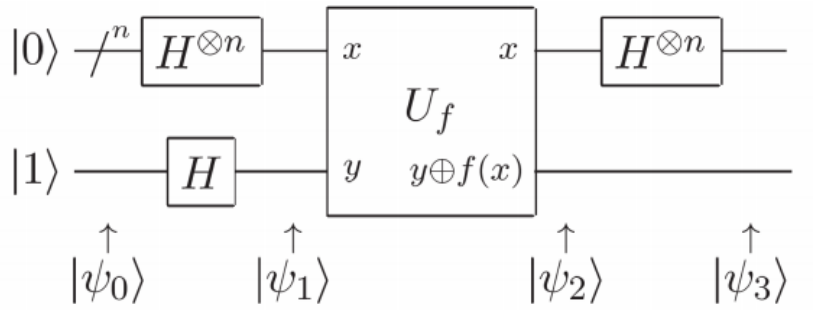


Figura 12-Circuito rappresentante l'algoritmo di Deutsch-Jozsa

indica un input *x* caratterizzato da *n* qubit inizialmente nello stato $|0\rangle$ che formeranno il registro dei dati, mentre un qubit nello stato $|1\rangle$ sarà utilizzato come target per contenere il risultato di $f(x)$:

$$|\psi_0\rangle = |0\rangle^{\otimes n}|1\rangle.$$

A $|\psi_0\rangle$ viene applicata la trasformazione di Walsh-Hadamard per realizzare una sovrapposizione equiprobabile dei 2^n stati. Potendo indicare *x*, *y* come stringhe di bit della forma $\{0,1\}^n$,possiamo utilizzare la rappresentazione:

$$H^{\otimes n}|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{y\in\{0,1\}^n} (-1)^{x\cdot y}|y\rangle,$$

ed in particolare:

$$H^{\otimes n}|0\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle,$$

per cui i relativi stati del sistema ad ogni step del circuito saranno:

$$|\psi_1\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}},$$

$$|\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{f(x)}|x\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}},$$

ed infine riapplicando *n* porte di Hadamard:

$$|\psi_3\rangle = \frac{1}{2^n} \sum_{y\in\{0,1\}^n} \sum_{x\in\{0,1\}^n} (-1)^{x\cdot y+f(x)}|y\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

A questo punto vediamo che l'ampiezza dello stato $|0\rangle^{\otimes n}$ è data da $\frac{1}{2^n}\sum_x(-1)^{f(x)}$, dunque nel caso la funzione sia costante e uguale a 0 in tutti i punti sarà +1, se costante e uguale a1 sarà -1, quindi all'atto della misurazione misureremo tutti qubit nello stato 0 mentre se la funzione è bilanciata ci sarà almeno un qubit diverso da zero dovuto ai contributi positivi e negativi che risultano in un'ampiezza nulla.
Diversamente dal caso classico dove la soluzione al problema è deterministica e dipende esponenzialmente dal numero di bit, nel caso quantistico si ottiene uno speed-up della velocità dovuto alla necessità di una singola esecuzione.

3.4 Algoritmo di ricerca di Grover

L'algoritmo quantistico di ricerca di Grover consiste nel trovare un elemento x in un insieme di possibili soluzioni tale che una certa funzione o condizione $P(x)$ sia vera. Ad esempio, un problema di ordinamento di un array equivale alla ricerca di una permutazione che soddisfi la particolare relazione d'ordine che si desidera è che quindi soddisfi la condizione di partenza.

Nel caso generale di un problema di ricerca non strutturato, ossia per il quale non si conosce la struttura dello spazio delle soluzioni, l'approccio classico che si può applicare è quello che controlla la condizione $P(x)$ su ognuno degli elementi x scelti casualmente dall'insieme delle possibili soluzioni e che quindi esplora sequenzialmente ogni combinazione. Se tale insieme ha dimensione *N*, allora l'algoritmo dovrà svolgere O(*N*) valutazioni della funzione $P(x)$. Se invece di un computer classico utilizzassimo una macchina quantistica, il problema sarebbe risolvibile oltre che con un ridotto margine di errore, con un numero di valutazioni di P uguale a $O(\sqrt{N})$ grazie all'applicazione dell'algoritmo di Grover.

Questo metodo è stato dimostrato ottimale per problemi di ricerca completamente non strutturati e le sue applicazioni più importanti e utili si riscontrano nella risoluzione di problemi NP-completi.

Assumiamo che il numero delle combinazioni candidate del problema di ricerca sia $N = 2^n$, dove *n* in un problema reale è di grandissime dimensioni e supponiamo che ogni elemento dell'insieme sia una stringa di *n* bit. Assumiamo inoltre che il numero di soluzioni reali sia esattamente *M* e che all'interno del problema esista un oracolo che ha il compito di determinare se una determinata sequenza di *n* bit è soluzione oppure no. Come nei casi precedenti, matematicamente possiamo rappresentare l'oracolo come una trasformazione unitaria che implementa una funzione booleana particolare *f* definita come $f: \{0,1\}^n \mapsto \{0,1\}$, dove $f(x) = 1$ significa che *x* è una soluzione mentre $f(x) = 0$ no.

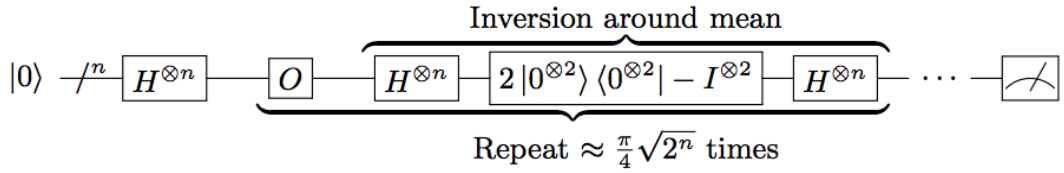


Figura 13-Circuito rappresentante l'algoritmo di Grover, con evidenziati gli step che compongono l'inversione intorno alla media e l'intero passaggio da reiterare per aumentare il valore della soluzione.

Ridefiniamo il problema secondo la notazione di Dirac:

$$O\colon |x\rangle|y\rangle \longmapsto |x\rangle|y \oplus f(x)\rangle,$$

dove *x* appartiene ovviamente all'insieme $\{0,1\}^n$ mentre $|y\rangle$ rappresenta un singolo qubit che nel caso venga inizializzato al valore 0, diventa 1 quando $f(x) = 1$. Se invece il qubit $|y\rangle$ venisse preparato nella sovrapposizione di stati $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$, allora l'azione dell'oracolo *O* è quella di invertire le ampiezze degli stati $|x\rangle$ che sono soluzioni mentre quelli che non lo sono vengono lasciati invariati. Questo procedimento può essere scritto come:

$$O\colon |x\rangle\frac{|0\rangle - |1\rangle}{\sqrt{2}} \longmapsto (-1)^{f(x)}|x\rangle\frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

Dato che il qubit $|y\rangle$ non viene mai modificato dall'oracolo, si può decidere di ignorarne il valore, trasformando dunque l'azione precedente in:

$$O\colon \sum_{x=0}^{n-1} \alpha_x |x\rangle \longmapsto \sum_{x=0}^{n-1} (-1)^{f(x)} \alpha_x |x\rangle .$$

L'inizializzazione dell'algoritmo di Grover prevede di preparare l'input nello stato:

$$|\psi\rangle = |0\rangle^{\otimes n} = |00 \ldots 0\rangle,$$

quindi viene applicata una trasformazione di Walsh-Hadamard per ottenere una sovrapposizione equiprobabile di stati:

$$|\psi\rangle H^{\otimes n} = \frac{1}{\sqrt{N}} \sum_{x=0}^{n-1} |x\rangle.$$

A questo punto abbiamo ottenuto tutte le possibili combinazioni di input per l'oracolo con peso uguale per ogni sovrapposizione. Applichiamo iterativamente per un numero appropriato di volte l'operatore di Grover *G*, definito come:

$$G = OH^{\otimes n}P_0H^{\otimes n},$$

Il primo passaggio è caratterizzato dall'azione dell'oracolo, che come descritto in precedenza è quello di invertire le ampiezze di quegli stati che sono soluzioni ma da solo non sarebbe sufficiente a riconoscere gli elementi ricercati poiché il segno dell'ampiezza non ha effetto sulla probabilità della misurazione:

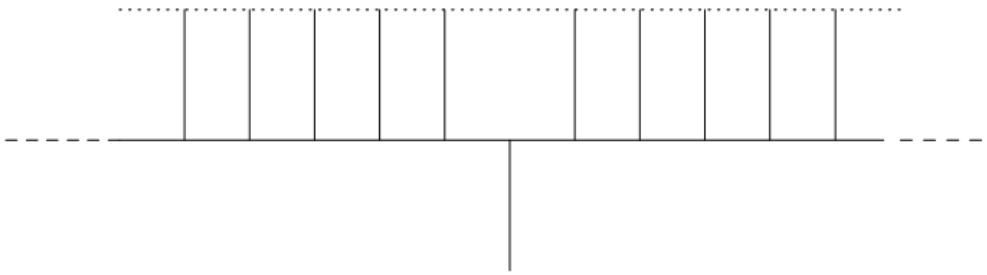


Figura 14-Effetto dell'azione dell'Oracolo sulle ampiezze degli stati

Si aggiungono perciò una serie di gate quantistici che hanno il compito di amplificare notevolmente le ampiezze di quegli elementi ricercati. L'operazione $H^{\otimes n}P_0H^{\otimes n}$ prende il nome di 'inversione intorno alla media' poiché il suo effetto è quello di amplificare le ampiezze degli stati soluzione che erano state invertite dall'oracolo, innalzandole del doppio sopra la media di tutte le altre ampiezze.
In questo caso P_0 equivale ad uno shift di fase di −1 su tutti gli stati computazionali diversi da $|0\rangle$, cioè:

$$P_0 \colon |x\rangle \longmapsto \begin{cases} |x\rangle & x = 0 \\ -|x\rangle & x > 0 \end{cases},$$

e si verifica facilmente che $P_0 = 2|0\rangle\langle 0| - I$ e che $H^{\otimes n}P_0H^{\otimes n} = (2|\psi\rangle\langle\psi| - I)$.
Applicando l'ultimo operatore ad uno stato si ottiene:

$$(2|\psi\rangle\langle\psi| - I)\sum_{x=0}^{n-1}\alpha_x|x\rangle = \sum_{x=0}^{n-1}(2A - \alpha_x|x\rangle,$$

$$con\ A = \sum_{x=0}^{n-1} \frac{\alpha_x}{N} = \frac{1}{N} \sum_{x=0}^{n-1} \alpha_x$$

Ripetendo l'applicazione di *G*, ossia l'oracolo e l'inversione intorno alla media all'incirca i_{max} volte, con:

$$i_{max} = \lceil \sqrt{N} \rceil,$$

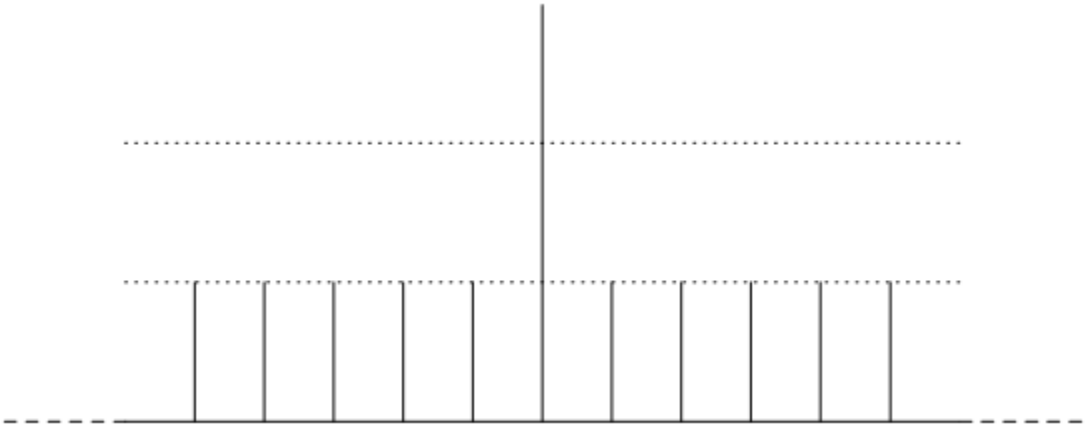


Figura 15-Rappresentazione dell'inversione attorno alla media dello stato soluzione

Aumentiamo l'ampiezza dello stato soluzione, consolidando le probabilità di ottenerlo effettuando la misurazione. Geometricamente l'effetto dell'operatore di Grover è quello di una doppia rotazione dello stato soluzione attorno all'asse σ:

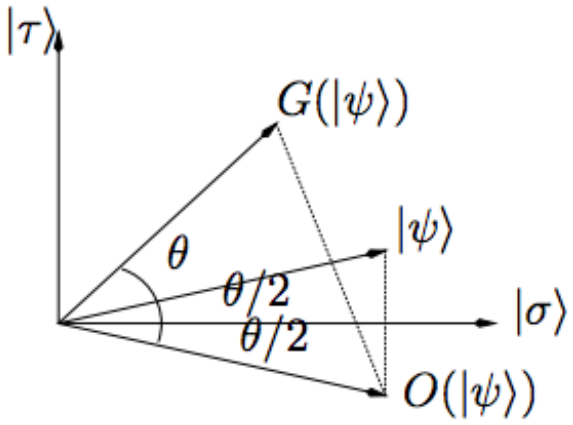


Figura 16-Interpretazione geometrica dell'iterazione dell'operatore G

La complessità dell'algoritmo di Grover è data essenzialmente dal numero di iterazioni dell'operatore *G* che si sceglie di eseguire e risulta essere nell'ordine $O\left(\sqrt{2^n}\right)$, partendo da un algoritmo classico di ricerca con complessità computazionale $O(2^n)$ si ottiene con Grover uno speed-up quadratico per gli algoritmi di ricerca.

3.5 Trasformata di Fourier Quantistica

Molti problemi fisici e matematici sono risolvibili proiettando il problema in un altro di cui si conosce la soluzione, questo si fa applicando ad esempio la trasformata di Laplace, quella di Legendre e altre ancora. Una delle più note ed utilizzate è la trasformata di Fourier, metodo matematico per passare dal dominio del tempo a quello delle frequenze, la cui rappresentazione discreta può essere definita come:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i j(k/N)},$$

dove i valori di $x = (x_0, x_1, \dots, x_{N-1})$ rappresentano i valori che vengono trasformati. Parlando della Quantum Fourier Transform, utilizziamo la stessa idea concettuale con la unica differenza che i vettori oggetto dell'operatore *x* e *y* rappresentano stati computazionali quantistici della forma:

$$x = \sum_{j=0}^{N-1} x_j \, |j\rangle,$$

$$y = \sum_{k=0}^{N-1} y_k |k\rangle.$$

Applicando l'operatore per ottenere y otteniamo la seguente azione sulle componenti di *x*:

$$\sum_{j=0}^{N-1} x_j \, |j\rangle \longrightarrow \sum_{k=0}^{N-1} y_k \, |k\rangle$$

$$= \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \sum_{j=0}^{N-1} x_j e^{2\pi i j \left(\frac{k}{N}\right)} |k\rangle = \sum_{j=0}^{N-1} x_j \left(\frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} x_j e^{2\pi i j \left(\frac{k}{N}\right)} |k\rangle \right).$$

Quello che otteniamo osservando l'ultima equivalenza è in altre parole una distribuzione di ogni ampiezza x_j di una specifica base $|j\rangle$ su tutte le basi vettoriali $|k\rangle$ rimanendo nello stesso spazio di Hilbert.

Passiamo alla descrizione di come poter rappresentare ed utilizzare la Quantum Fourier Transform nella computazione quantistica. Supponiamo di disporre di una base computazionale $|j\rangle$ di dimensione $N = 2^n$, dove n è il numero di qubit utilizzati per la rappresentazione. Ogni stato computazionale è quindi descritto in rappresentazione binaria come:

$$j: \mathbb{N} \longrightarrow \{0,1\}^n$$

$$j = [j_1 j_2 \dots j_n] \equiv \sum_{i=1}^{n} j_i 2^{n-1}.$$

Introduciamo in oltre la funzione binaria (0.):

$$0.: \{0,1\}^m \longrightarrow (0,1),$$

$$0.(j_1 j_2 \dots j_m) = \sum_{i=1}^{m} j_i 2^{-m}.$$

Seguendo alcuni step matematici partendo dalla rappresentazione di $|j\rangle$ ricaviamo la rappresentazione della trasformata:

$$|j\rangle \longrightarrow \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} x_j e^{2\pi i j \left(\frac{k}{2^n}\right)} |k\rangle$$

$$= \frac{1}{\sqrt{2^n}} \sum_{k_1=0}^{1} \dots \sum_{k_n=0}^{1} e^{2\pi i j \left(\sum_{l=1}^{n} k_l 2^{-l}\right)} |k_1 \dots k_n\rangle$$

$$= \frac{1}{\sqrt{2^n}} \sum_{k_1=0}^{1} \dots \sum_{k_n=0}^{1} \bigotimes_{l=1}^{n} e^{2\pi i j k_l 2^{-l}} |k_l\rangle$$

$$= \frac{1}{\sqrt{2^n}} \bigotimes_{l=1}^{n} \left[\sum_{k_l=0}^{n} e^{2\pi i j k_l 2^{-l}} |k_l\rangle \right]$$

$$= \frac{1}{\sqrt{2^n}} \bigotimes_{l=1}^{n} \left[|0\rangle + e^{2\pi i j 2^{-l}} |1\rangle \right].$$

Se sviluppato, otteniamo il prodotto:

$$\frac{\left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle\right) \dots \left(|0\rangle + e^{2\pi i 0.j_1 j_2 \dots j_n}|1\rangle\right)}{\sqrt{2^n}}.$$

Da questa rappresentazione si può ricavare il circuito che implementa la trasformazione, dove l'input è un registro di *n* qubit ognuno rappresentante una cifra binaria della rappresentazione di $j = [j_1 j_2 \dots j_n]$.

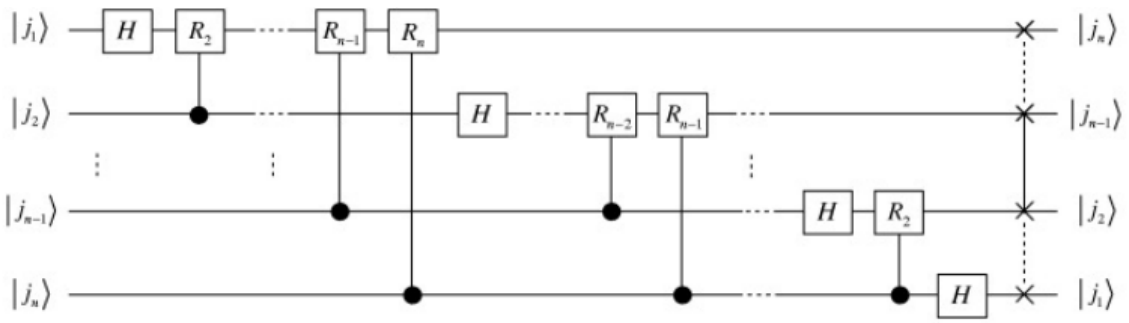


Figura 17-Circuito che implementa la Quantum Fourier Transform

Applicando Hadamard al primo qubit si ottiene:

$$\frac{\left(|0\rangle + e^{2\pi i 0.j_1}|1\rangle\right)}{\sqrt{2}}|j_2 \dots j_n\rangle,$$
$$con \begin{cases} j_1 = 1 \longrightarrow e^{\pi i} = 1 \\ j_1 = 0 \longrightarrow e^{0} = 1 \end{cases}$$

Tutte le altre operazioni che vengono impiegate nel circuito sono trasformazioni unitarie della forma:

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i\left(\frac{1}{2^k}\right)} \end{bmatrix}.$$

Utilizzando queste rotazioni come operazioni controllate, ci permettono di aggiungere il bit di indice k alla fase del coefficiente di $|1\rangle$. Proseguiamo quindi applicando tanti *controlled-Ri* quanto il numero di qubit *n* al primo ($i = 2, \dots, n$). Proseguiamo applicando al secondo qubit tutte le *controlled-Ri* di indice successivo e si continua in questo modo fino ad ottenere lo stato finale:

$$\frac{\left(|0\rangle + e^{2\pi i 0.j_1 j_2 \dots j_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.j_2 \dots j_n}|1\rangle\right) \dots \left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right)}{\sqrt{2^n}}.$$

Per ritrovare la trasformata di $|j\rangle$ è necessario applicare $n/2$ operazioni di SWAP per riportare i fattori nell'ordine giusto. In totale il numero di operazioni per effettuare il calcolo della QFT è nell'ordine di $O(n^2)$, mentre il calcolo della trasformata di Fourier su un computer classico usando la FFT (Fast Fourier Transform) richiede un numero di operazioni dell'ordine $O(n2^n)$, esponenzialmente più elevato della versione quantistica.
È da sottolineare che l'applicazione di tale operatore fa sì che il risultato sia codificato nelle ampiezze delle sovrapposizioni degli stati del sistema, dunque non direttamente osservabili o accessibili mediante le misurazioni; gli usi della Quantum Fourier Transform risultano infatti di tipo indiretto ma fondamentali per la realizzazione di tutti quegli algoritmi quantistici che fanno ottenere uno speed-up esponenziale rispetto alla controparte classica. Tale trasformazione è infatti essenziale per alcuni degli algoritmi più famosi e importanti introdotti di seguito.

3.6 Stima di Fase

La Trasformata di Fourier è alla base della procedura chiamata *Phase Estimation* o *stima della fase* che permette di stimare gli autovalori di una matrice unitaria *U*. Essendo *U* unitaria, possiamo rappresentare l'autovalore nel seguente modo:

$$\lambda = e^{2\pi i\varphi}$$

dove φ è un qualche valore che dobbiamo stimare compreso tra 0 e 1.
L'uso di questo algoritmo è parte integrante di altri dal momento che determinati problemi possono essere ricondotti alla stima delle fasi; uno dei più importanti di cui si parlerà successivamente è l'algoritmo per la fattorizzazione.
Il circuito è mostrato in figura e di seguito ne è descritta l'implementazione.

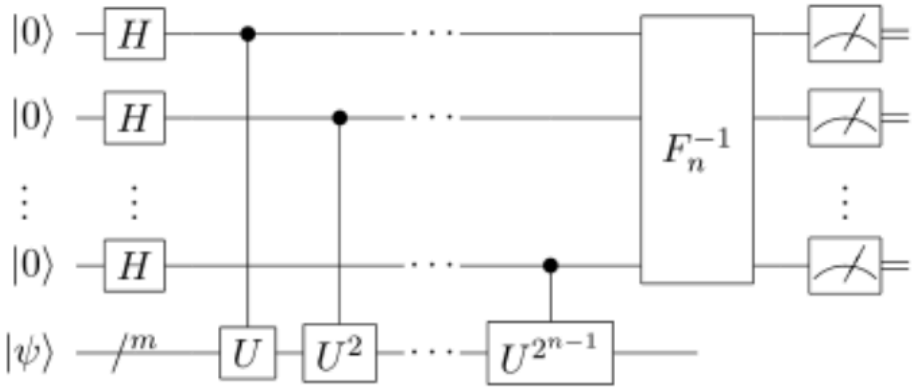


Figura 18-Circuito che implementa l'algoritmo di Stima delle Fasi

Consideriamo quindi un operatore unitario *U* e un suo autovettore $|\psi\rangle$ che ha autovalore $e^{2\pi i\varphi}$:

$$U|\psi\rangle = e^{2\pi i\varphi}|\psi\rangle.$$

Supponiamo di disporre di un registro $|j\rangle$ di *t* qubit inizialmente tutti nello stato $|0\rangle$ mentre il secondo registro contiene l'autovettore $|\psi\rangle$. Nel primo step si applicano *t* porte Hadamard per ottenere la sovrapposizione:

$$|j\rangle = \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle\,|\psi\rangle.$$

Ogni qubit del primo registro viene utilizzato come controllo per effettuare l'operazione controlled-U^{2^k}. L'effetto di tale sequenza è di trasformare $|j\rangle|\psi\rangle$ in $|j\rangle U^j|\psi\rangle$. Ricordando la rappresentazione binaria $j = [j_1 j_2 \dots j_n] \equiv \sum_{i=1}^{n} j_i 2^{n-1}$ notiamo che U^j viene applicato solamente quando j=1, infatti:

$$U^j = U^{j_t 2^0} U^{j_{t-1} 2^1} \dots U^{j_2 2^{t-2}} U^{j_1 t-1}.$$

Se $|\psi\rangle$ è un autovettore di U con autovalore λ, allora $|\psi\rangle$ è anche autovalore di U^{2^k} con autovalore λ^{2^k}, quindi otteniamo:

$$|j\rangle U^j|\psi\rangle = |j\rangle e^{2\pi i\varphi}|\psi\rangle = e^{2\pi i\varphi}|j\rangle|\psi\rangle$$

$$= \left[\left(e^{2\pi i\varphi j_1 2^{t-1}}|j_1\rangle\right) \dots \left(e^{2\pi i\varphi j_{t-1} 2^1}|j_{t-1}\rangle\right)\left(e^{2\pi i\varphi j_t 2^0}|j_t\rangle\right)\right]|\psi\rangle.$$

Siccome ogni $|j_t\rangle$ è in sovrapposizione di stati e $\varphi = [0.\varphi_1\varphi_2 \dots \varphi_t]$possiamo esprimere il risultato precedente come:

$$\frac{\left(|0\rangle + e^{2\pi i 0.\varphi_1\varphi_2\dots\varphi_t}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.\varphi_2\dots\varphi_t}|1\rangle\right) \dots \left(|0\rangle + e^{2\pi i 0.\varphi_t}|1\rangle\right)}{\sqrt{2^t}}|\psi\rangle$$

$$\bigotimes_{k=0}^{t-1} \frac{|0\rangle + e^{2\pi i 2^k\varphi}|1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2^t}} \sum_{y=0}^{2^t-1} e^{2\pi i\varphi(y/2^t)}|y\rangle.$$

Che corrisponde all'espressione ottenuta con l'applicazione della QFT al vettore $|\varphi\rangle$. A questo punto possiamo applicare il secondo passo dell'algoritmo che è quello di applicare l'inversa della trasformata di Fourier. Essa produce $|\varphi_1\varphi_2 \dots \varphi_t\rangle$, rappresentazione in t bit che possiamo misurare per ottenere il valore esatto di φ a meno di un errore δ dipendente dal numero di bit t fissato. Questo valore può essere scelto in base al margine di errore ammissibile che si sceglie di tollerare. Scelto un ε tale per cui il φ che otterremo sarà

approssimato fino all'*n*-esimo bit con probabilità 1- ε possiamo valutare il numero di bit come:

$$t = n + \left\lceil log\left(2 + \frac{1}{2\varepsilon}\right)\right\rceil.$$

3.7 Algoritmo di Ordinamento

In questo caso specifico, trovare l'ordine di un numero, si riferisce al problema di trovare, dati due interi positivi *x* e *N* senza fattori in comune e tali che $x < N$, l'*ordine* di *x* modulo *N* ossia il più piccolo intero *r* tale che valga:

$$x^r = 1(mod\ N).$$

Per la risoluzione di questo problema non si conoscono algoritmi classici capaci di trovare *r* in tempo polinomiale, mentre con gli strumenti della computazione quantistica abbiamo un algoritmo che risolve il problema con una complessità di $O((\log N)^3)$.
Il principio base dell'algoritmo è quello di applicare la stima delle fasi all'operatore unitario *U*, definito:

$$U|y\rangle = \begin{cases} |xy(mod\ N)\rangle & \text{se } 0 \leq y \leq N-1 \\ |y\rangle & \text{se } N \leq y \leq 2^L - 1 \end{cases}$$

dove $y \in \{0,1\}^L$ con *L* numero di qubit che lo compongono. Se *r* è l'ordine di *x* modulo *N* ed esistono valori *s* tali per cui $0 \leq s \leq r-1$, allora ogni vettore scritto nella seguente forma:

$$U|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-\frac{2\pi isk}{r}} |x^k(mod\ N)\rangle,$$

è un'autovalore di *U* con autovalore $e^{\frac{2\pi is}{r}}$. Infatti se vale l'ipotesi iniziale $x^r = 1(mod\ N)$, si può verificare che:

$$U|u_s\rangle = \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1} e^{-\frac{2\pi isk}{r}}|x^{k+1}(mod\ N)\rangle$$

$$= \frac{1}{\sqrt{r}}\sum_{k=1}^{r} e^{-\frac{2\pi is(k-1)}{r}}|x^{k}(mod\ N)\rangle$$

$$= \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1} e^{-\frac{2\pi is(k-1)}{r}}|x^{k}(mod\ N)\rangle$$

$$= e^{\frac{2\pi is}{r}}|u_s\rangle.$$

Poiché prendendo l'ultima sommatoria $\frac{1}{\sqrt{r}}\sum_{k=0}^{r-1} e^{-\frac{2\pi is(k-1)}{r}}|x^{k}(mod\ N)\rangle$ il valore che si ottiene per $k = 0$ risulta $e^{\frac{2\pi is}{r}}|1(mod\ N)\rangle$ ed è esattamente quello che si ottiene risolvendo la sommatoria $\frac{1}{\sqrt{r}}\sum_{k=1}^{r} e^{-\frac{2\pi is(k-1)}{r}}|x^{k}(mod\ N)\rangle$ con $k = r$. Se lo sviluppiamo risulta:

$$e^{-\frac{2\pi is(r-1)}{r}}|x^{r}(mod\ N)\rangle = e^{-2\pi is}e^{\frac{2\pi is}{r}}|1(mod\ N)\rangle = e^{\frac{2\pi is}{r}}|1(mod\ N)\rangle.$$

Se si è in grado di ricreare uno stato simile a quello appena descritto, potremmo applicare la procedura di stima delle fasi per determinare s/r e dal quale ricavare *r.* Le difficoltà non sono banali in quanto bisogna risolvere diversi problemi prima di passare all'applicazione. Il primo è dovuto alla preparazione dello stato $|u_s\rangle$ in quanto definito in funzione di *r* che stiamo cercando; un aiuto viene dal fatto di sfruttare l'equivalenza dello stato $|1\rangle$ con una corrispondenza lineare di tutti gli autovettori $|u_s\rangle$ di *U* per $0 \leq s \leq r-1$:

$$\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}|u_s\rangle = \sum_{k=0}^{r-1}\left(\frac{1}{r}\sum_{s=0}^{r-1} e^{-\frac{2\pi isk}{r}}\right)|x^{k}(mod\ N)\rangle.$$

L'espressione tra parentesi, per $k > 0$, è la serie geometrica $e^{-\frac{2\pi ik}{r}}$ che converge a 0, quindi per ogni $k > 0$ l'equazione ritorna 0, mentre per $k = 0$ è uguale a 1. Pertanto la combinazione degli autovettori $|u_s\rangle$ coincide con lo stato $|1\rangle$. Vale quindi l'equivalenza:

$$\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}|u_s\rangle = |1\rangle.$$

Un altro problema è gestire l'esecuzione della sequenza di operazioni controlled-U^{2^k}, a differenza di quando abbiamo parlato dell'algoritmo di stima delle fasi, dove le operazioni controllate erano viste come scatole nere senza alcun dettaglio particolare, in questo specifico problema, questa sequenza di operazioni corrisponde alla moltiplicazione modulo *N* del secondo registro per *x* elevato a potenza uguale al contenuto del primo registro.
In altre parole, se $|k\rangle$, $|u\rangle$ sono gli stati del primo e secondo registro rispettivamente, allora dopo l'applicazione delle controlled-U^{2^k} operazioni si ottiene:

$$|k\rangle|u\rangle \longmapsto |k\rangle U^{k_t 2^{t-1}} U^{k_{t-1}2^{t-2}} \dots U^{k_1 2^0}|u\rangle$$

$$= |k\rangle|x^{k_t 2^{t-1}} x^{k_{t-1}2^{t-2}} \dots x^{k_1 2^0} u(mod\ N)\rangle$$

$$= |k\rangle|x^{k_t 2^{t-1}+k_{t-1}2^{t-2}+\cdots+k_1 2^0} u(mod\ N)\rangle$$

$$= |k\rangle|x^k u(mod\ N)$$

Questa operazione può essere realizzata usando lo schema di computazione reversibile con un numero di operazioni elementari dell'ordine di complessità $O(L^3)$.
Infine l'ultimo problema da risolvere è come ricavare *r* dalla stima delle fasi effettuata che chiamiamo φ_s. Una possibile soluzione è usare la tecnica delle frazioni continue, che permettono di approssimare un qualsiasi numero reale con una sequenza di numeri razionali nella forma:

$$[a_0, a_1, a_2, \dots, a_p] = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\dots + \cfrac{1}{a_p}}}}.$$

Senza entrare nel dettaglio della teoria dei numeri, consideriamo il problema al quale siamo interessati, per il quale possiamo stabilire che $^{s}/_{r}$ è un convergente del numero razionale φ_s, ossia appartiene ai $[a_0, a_1, a_2, \dots, a_j]$ di φ_s, se e solo se si sceglie una precisione $n = 2L + 1$.

Disponendo di questi risultati possiamo completare l'algoritmo per trovare l'ordine *r* di un numero positivo *x* modulo *N*. Calcoliamo con il metodo delle frazioni continue un convergente $^{s}/_{r}$ di φ_s tale che siano verificati i requisiti:

$$\begin{cases} |\varphi_s - {}^{s}/_{r}| < \dfrac{1}{2^{2L+1}}, con\ 2L + 1 = n \\ r \leq 2^L \end{cases}$$

Il candidato *r* deve essere controllato testando se $x^r = 1(mod\ N)$; se il test ha successo allora *r* è l'ordine di *x* modulo *N*, altrimenti l'algoritmo fallisce e deve essere rieseguito.

È di fondamentale importanza che *r* e *s* non abbiano fattori comuni, altrimenti l'algoritmo delle frazioni continue potrebbe restituire un fattore di *r* invece che *r* stesso. Aumentando la dimensione di *N*, la probabilità che *r* e *s* siano co-primi è di almeno $^{1}/_{\log N}$, è quindi sufficiente lanciare l'algoritmo un numero adeguato di volte per ottenere con altra probabilità un caso con *r* e *s* co-primi; in particolare lanciando l'algoritmo $O(L)$ volte, ha successo con probabilità $(1 - \varepsilon)(1 - {}^{1}/_{N})$, con un costo complessivo di $O(L^4)$ operazioni.

3.8 Algoritmo di Shor

La più importante applicazione dell'algoritmo di Shor è dovuta alla possibilità di risolvere il problema della fattorizzazione, che consiste nel trovare, dato un numero intero positivo dispari e composto *N*, i fattori primi nei quali è scomponibile.

Le proprietà matematiche alla base della fattorizzazione e le difficoltà computazionali degli algoritmi classici per risolvere il problema sono alla base dei sistemi crittografici ad oggi più diffusi, come ad esempio il sistema RSA.

Questo proprio perché non esiste un algoritmo polinomiale capace di effettuare la fattorizzazione di numeri interi grandi; classicamente il più performante è il 'number field sieve' che richiede comunque un tempo superpolinomiale nel numero di cifre O(log *N*) in *N*. Nel 1994, Peter Shor [5] pubblicò un algoritmo quantistico in grado di fattorizzare un intero in tempo polinomiale con complessità $O\big((logN)^2(\log\log N)(\log\log\log N)\big)$.

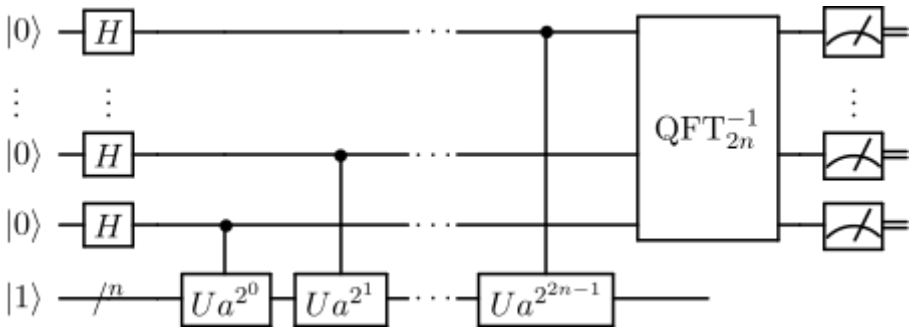


Figura 19-Circuito che implementa l'algoritmo di Shor per la fattorizzazione

Come si può vedere in Figura, l'algoritmo di Shor parte da uno stato quantistico iniziale definito da:

$$|00\ldots0\rangle|00\ldots01\rangle.$$

Stato che come abbiamo visto nel paragrafo anteriore possiamo modificare sostituendo appropriatamente il valore del qubit nello stato $|1\rangle$:

$$|0\rangle^{\otimes t}\frac{1}{r}\sum_{s=0}^{r-1}\sum_{k=0}^{r-1}e^{-\frac{2\pi isk}{r}}|x^k(mod\ N)\rangle,$$

l'applicazione degli Hadamard gate porta il sistema nello stato:

$$\frac{1}{\sqrt{2^t}}\sum_{j}|j\rangle\frac{1}{r}\sum_{s=0}^{r-1}\sum_{k=0}^{r-1}e^{-\frac{2\pi isk}{r}}|x^k(mod\ N)\rangle,$$

proseguiamo con il circuito applicando l'operatore di modulo *N*:

$$\frac{1}{\sqrt{2^t}}\sum_j |j\rangle \frac{1}{r}\sum_{s=0}^{r-1}\sum_{k=0}^{r-1} e^{-\frac{2\pi isj}{r}-\frac{2\pi isk}{r}} |x^k (mod\ N)\rangle,$$

effettuando la misurazione sul secondo registro (quello sul quale non abbiamo applicato Hadamard), lo stato diventa:

$$\frac{1}{r}\sum_{s=0}^{r-1}\left[\frac{1}{\sqrt{2^t}}\sum_j e^{-\frac{2\pi isj}{r}} |j\rangle\right] e^{-\frac{2\pi isk}{r}} |x^k (mod\ N)\rangle,$$

e applicando l'inversa della trasformata di Fourier trasformiamo il contenuto della parentesi quadra in $\widetilde{s/r}$, dal quale procedendo con il metodo delle frazioni continue possiamo trovare il valore di *r:*

$$\frac{1}{r}\sum_{s=0}^{r-1} |\widetilde{s/r}\rangle e^{-\frac{2\pi isk}{r}} |x^k (mod\ N)\rangle.$$

Ad esempio giunti a questo punto, potremmo trovarci nel seguente stato:

$$\widetilde{s/r} = 0.010110 = \frac{32*0+16*1+8*0+4*1+2*1+1*0}{64} = \frac{22}{64},$$

$$\frac{22}{64} = \frac{1}{2+\frac{20}{22}} = \frac{1}{2+\frac{1}{1+\frac{1}{10}}} \cong \frac{1}{3} \Longrightarrow r = 3;$$

oppure:

$$\widetilde{s/r} = 0.010011 = \frac{32*0+16*1+8*0+4*0+2*1+1*1}{64} = \frac{19}{64},$$

$$\frac{19}{64} = \frac{1}{3+\frac{7}{19}} = \frac{1}{3+\frac{1}{2+\frac{5}{7}}} = \frac{1}{3+\frac{1}{2+\frac{1}{1+\frac{2}{5}}}} = \frac{1}{3+\frac{1}{2+\frac{1}{1+\frac{1}{2+\frac{1}{2}}}}}$$

$$\cong \frac{8}{27} \Longrightarrow r = 27;$$

Dato che il valore di *r* è un'approssimazione, l'algoritmo potrebbe fallire, per questo è necessaria la verifica del valore tramite $x^r = 1(mod\ N)$.
L'algoritmo per la fattorizzazione consiste più in generale di cinque passi di cui soltanto uno necessita il supporto del computer quantistico, mentre tutti gli altri possono essere eseguiti con computazione classica. Per il calcolo di un fattore non banale di un numero intero *N* di *L* bits, dispari e composto, si eseguono i seguenti passaggi:

Come abbiamo anticipato la parte quantistica della fattorizzazione risiede nel terzo step, con l'applicazione dell'algoritmo quantistico di Shor, il cui circuito è riportato di seguito:

1) Se *N* è pari, restituire il fattore 2 e ripetere;
2) Scegliere un numero casuale *x*, tale che $1 < x < N$;
3) Utilizzando l'algoritmo di Euclide calcolare $f_{nb} = MCD\ (x, N)$. Se $f_{nb} > 1$ allora f_{nb} è un fattore non banale di *N* altrimenti procedere con il passo 4;
4) Applicazione dell'algoritmo quantistico di Shor per trovare l'ordine *r* di *x* modulo *N*; tale per cui *r* verifichi $x^r = 1(mod\ N)$;
5) Se *r* è dispari oppure $x^{\frac{r}{2}} = N - 1(mod\ N)$, allora ritorna al passo 1;
6) Se *r* è pari e $x^{\frac{r}{2}} \neq N - 1(mod\ N)$, calcola con l'algoritmo di Euclide $MCD\ (x^{\frac{r}{2}} - 1, N)$ e $MCD\ \left(x^{\frac{r}{2}} + 1, N\right)$; se uno dei due interi calcolati risulta essere un fattore non banale di *N*, verificando: $x^r - 1 = \left(x^{\frac{r}{2}} - 1\right)\left(x^{\frac{r}{2}} + 1\right) = N = 0(mod\ N)$. Se verificato l'algoritmo termina con successo altrimenti va ripetuto dal passo 1.

L'implementazione dell'algoritmo di Shor è oggi impossibilitata in quanto non è possibile costruire un hardware quantistico sufficiente per la fattorizzazione di grandi numeri. Questo possibile futuro successo nel campo della computazione quantistica comporterebbe un

radicale cambiamento tecnologico in quanto verrebbe ridefinito il concetto di sicurezza dell'informazione attuale.

Capitolo 4

Machine Learning

Il Machine Learning è quel ramo della Computer Science che si occupa di riconoscere pattern significativi e apprendere da un insieme di dati con l'obiettivo di effettuare predizioni corrette o utilizzare l'apprendimento per la classificazione di input futuri; può essere anche considerato come una specie di intelligenza artificiale utilizzabile come supporto alle decisioni in analisi governative e di business, su referti medici, riguardo la gestione di rischi finanziari e tanti altri campi nei quali le decisioni e l'ottimizzazione delle risorse è basata sullo studio delle informazioni digitali di cui si dispone. Quelli appena citati sono soltanto alcuni esempi ricorrenti delle tante aree applicative sulle quali il Machine Learning può essere applicato con successo e che dimostrano la versatilità e i vantaggi dell'utilizzo di queste tecniche.

Uno degli scenari più tipici è quello dedicato alla classificazione di istanze. Il problema prevede di avere a disposizione un dataset contenente una quantità limitata di oggetti, associabili a vettori di informazioni, dei quali conosciamo le *features* e la classe di appartenenza alla quale tale oggetto è stato precedentemente assegnato proprio sulla base del conoscimento che si ha delle sue caratteristiche. Utilizzando questa base iniziale di conoscimento possiamo dunque costruire un modello di apprendimento, o classificatore, che sarà capace di catalogare nuovi oggetti introdotti nel sistema dei quali non conosciamo a priori la classe di appartenenza; il classificatore esplorerà la possibile soluzione sulla base di misurazioni sulle *features* del vettore e basandosi sul conoscimento ottenuto dall'esperienza precedente. Un buon classificatore è dunque quello capace di classificare accuratamente nuovi elementi inseriti all'interno del dataset con un margine di errore prossimo allo zero.

Un'altra definizione che descrive correttamente l'azione del Machine Learning, è quella di definirlo come quella scienza che si occupa di far attuare i computer senza dover essere esplicitamente programmati, ma utilizzando una forma di *autoapprendimento*, dove con

questo termine si indica un algoritmo capace di fornire alla macchina la capacità di apprendere dai dati che ha a disposizione; una chiara definizione è quella data da Mitchell nel 1997:

'A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E'*.

La letteratura mette a disposizione una varietà infinita di algoritmi di apprendimento, classificatori e valutatori di le performance che hanno reso il campo del Machine Learning ricco di soluzioni alternative applicabili a qualsiasi esigenza e permettendo di intervenire su quei problemi che sono troppo difficili da risolvere con soluzioni progettate e scritte da essere umani. Da un punto di vista scientifico e filosofico il Machine Learning è molto interessante perché lo sviluppo delle nostre conoscenze in questo campo è direttamente proporzionale al conoscimento dei principi che stanno alla base dell'intelligenza.

4.1 Supervised Machine Learning

Nel Machine Learning Supervisionato, l'obiettivo è trovare una sufficientemente buona approssimazione $f'(x)$ della funzione$f(x)$, la quale è alla base della relazione di predizione tra input e output. Per completezza consideriamo che il sistema possa eseguire una decisione con margine di errore ε e quindi considerare il modello $Y = f(x) + \varepsilon$.

Una formulazione standard di un problema di apprendimento supervisionato è quello di classificazione: l'obiettivo del sistema è quello di apprendere (approssimare il comportamento di) una funzione che mappi un vettore d'entrata in una delle *n* possibili classi attraverso l'osservazione di vari input-output precedentemente passati al sistema e che sono stati parametri della funzione partecipando attivamente all'apprendimento del sistema.

Senza soffermarsi troppo sulle specifiche implementative di ogni metodologia esistente in letteratura, si elencano i principali approcci di Machine Learning Supervisionato:

- **Tree-based algorithms** (Alberi Decisionali): il metodo degli alberi decisionali è spesso utilizzato per approssimare funzioni obiettivo a valori discreti, nelle quali la

funzione di apprendimento è presentata sotto forma di un albero decisionale che ne facilita la lettura umana, nel quale spesso sono implicitamente rappresentati insiemi di *regole if-then* che ne definiscono il comportamento. Questi metodi sono oggi uno dei più utilizzati come strumento di apprendimento, in un range di problemi che va dal supporto alla diagnosi medica fino alla valutazione di accettare o meno il rischio di credito fronte a una richiesta di prestito monetario.

- **Regression-based**: algoritmi che cercano di modellare esplicitamente la relazione intrinseca tra variabili di input o variabili indipendenti e output, tipicamente nella forma di equazioni parametriche nelle quali gli stessi parametri sono stimati a partire dai dati iniziali. Questi metodi spesso forniscono esplicite stime delle misure di associazione tra elementi, che possono essere aggiustate per nuovi input applicando una stima di errore standard che può essere definita differentemente a seconda del paradigma di modellazione utilizzato.

- **Instance-based**: questi metodi utilizzano la memorizzazione di uno specifico dataset precompilato, di cui si conoscono caratteristiche e classificazione delle singole istanze, per utilizzarlo in una seconda fase come fonte di partenza per l'apprendimento, ossia come training set. Una volta completato l'apprendimento si suppone che il sistema sia capace di classificare una nuova istanza di questo problema. A fronte di una nuova richiesta il sistema utilizzerà quindi tutte le informazioni di cui dispone per determinare il livello di relazione della nuova istanza e determinarne la classe di appartenenza.

- **Bayesian methods**: basati sul famoso teorema di Bayes, assume un modello probabilistico che consente di catturarne l'incertezza in maniera predeterminata valutando le probabilità associate ai risultati. Vengono prevalentemente sfruttati per la risoluzione di problemi legati alla necessità di effettuare diagnosi e predizioni future.

- **Radial Basis Functions and Kernels methodss**: i metodi *Kernel* mappano i dati all'interno di uno spazio dimensionale di più grandi dimensioni ricercando in questo nuovo spazio dimensionale la possibilità di ottenere una più facile separazione dei dati, una più chiara distinzione o una loro migliore strutturazione. È importante

tenere presente che non ci sono limiti sulla forma di questo mapping che può persino raggiungere, teoricamente, uno spazio di dimensioni infinite. Si ricorda infine l'esistenza di una proprietà matematica conosciuta con il nome di '*Kernel trick'* che permette di eliminare esplicitamente il calcolo della funzione di mappatura dei dati.

- **Ensemble methods**: l'idea base di questi metodi è quella di costruire un modello di predizione combinando insieme le forze di un insieme di più semplici modelli base creando una sorta di modello ibrido. Questa strategia può essere adottata per raggiungere due scopi: il primo è sviluppare una popolazione di diverse tipologie di apprenditori (learners) che giovano tutti dello stesso training set, e secondo, combinarli successivamente tutti insieme per dare vita al classificatore composto vero e proprio.

- **Artificial Neural Netwok**: uno dei modelli più famosi, basato sull'idea di utilizzare una modellazione ispirata alle reti neurali biologiche per realizzare l'apprendimento simulando il comportamento di un cervello umano. Utilizzando gli input, si vogliono realizzare combinazioni lineari degli stessi per derivarne nuove caratteristiche che possono essere utilizzate per la modellazione della funzione obiettivo come una funzione non lineare di tali caratteristiche. Esistono varie versioni differenti di questo modello, differenziate principalmente dalla caratteristica di essere *feedforward* o *backpropagation*, dove nel secondo caso, il risultato di un input è riutilizzato come fonte di informazione per migliorare la fase di apprendimento. Il risultato è un metodo di apprendimento molto potente con una vastissima area applicativa.

4.2 Unsupervised Machine Learning

Nel Machine Learning non Supervisionato ogni problema comincia con il dover osservare le caratteristiche dei dati senza avere a disposizione alcuna misurazione o informazione della classe d'appartenenza o più in generale dell'outcome. L'obbiettivo diventa dunque quello di descrivere con che criterio i dati sono distribuiti, organizzati o clusterizzati

all'interno del sistema. La computazione deve quindi cercare di costruire una fedele rappresentazione dei dati iniziali che possa essere usata come base per effettuare previsioni o decisioni riguardanti informazioni future.
Il Machine Learning non Supervisionato può essere interpretato come uno strumento utilizzabile per trovare pattern significativi all'interno e nascosti dietro ciò che inizialmente sarebbe interpretato come rumore senza alcuna struttura. I due macro modelli appartenenti a questa area sono:

- **Dimensinality reduction**: esistono diversi problemi per i quali i vettori che rappresentano i dati misurati per ogni istanza sono composti da moltissimi valori, appartenendo così ad una dimensione grandissima dello spazio vettoriale, ma potremmo avere buone ragioni di pensare che i dati significativi si trovino in un collettore di dimensione inferiore all'originale. In altre parole, possiamo immaginare che parte dei dati ricavati siano in realtà ottenuti come derivazione o combinazione di un sottoinsieme di risorse di rilevanza più significativa che tipicamente non possono essere ricavate in nessuna forma dalle altre. Creare questo sottoinsieme di dimensione spaziale più piccola equivale dunque a trovare una selezione di caratteristiche primarie non riconducibili ad altre. La riduzione della dimensionalità può inoltre essere interpretata come quel processo effettuabile per ottenere un insieme di gradi di libertà utilizzabili per riprodurre gran parte delle variabili presenti nel dataset.

- **Clustering**: probabilmente il più famoso e utilizzato metodo di Machine Learning non supervisionato; clusterizzare indica il processo di raggruppamento di oggetti, basandosi esclusivamente sui dati che lo descrivono e sulle relazioni esistenti tra i vari oggetti. L'obiettivo è quello di ottenere un numero di insiemi nei quali le istanze che vengono racchiuse nello stesso gruppo siano il più possibile similari o relazionate tra di loro e il più possibile differenti dagli oggetti appartenenti agli altri insiemi. Dunque, ottenere una maggiore similarità all'interno di un gruppo e una maggiore differenza tra insiemi diversi indicherà una migliore o più definita distinzione tra i cluster realizzati dal sistema.

4.3 Deep Learning

Le metodologie di Machine Learning citate fino ad ora, nonostante rappresentino un ampio insieme di soluzioni ad una vasta gamma di problemi, non hanno successo nel risolvere le problematiche centrali dell'Intelligenza Artificiale, come ad esempio il riconoscimento vocale o quello degli oggetti. A questo proposito nasce e si sviluppa il Deep Learning, proprio come supporto agli innumerevoli fallimenti avvenuti in passato nel tentativo di risolvere per mezzo di algoritmi tradizionali questi problemi difficili da sormontare.
Di fatto, generalizzare nuove istanze diventa esponenzialmente più difficile quando si lavora con dati caratterizzati da uno spazio dimensionale sempre più grande e i meccanismi di Machine Learning tradizionali utilizzati per generalizzare sono limitati e alle volte insufficienti per apprendere le complicate funzioni che si incontrano avendo a che fare con grandi spazi dimensionali. Queste dimensioni spaziali sono la principale causa di un costo computazionale elevato che insieme ad ulteriori problematiche da superare hanno portato il Deep Learning ad essere lo strumento progettato per il loro superamento.
Tra i metodi principali abbiamo:

- **Deep Feedforward Networks**: chiamate anche più comunemente reti neurali o percettroni multistrato, rappresentano il modello di riferimento del Deep Learning. L'obiettivo di una rete *feedforward* è di approssimare una funzione f^*. Mentre un normale classificatore della forma $Y = \hat{f}(x)$ associa una istanza *x* ad una categoria *Y*, una rete feedforward definisce un mapping $Y = f(x, \vartheta)$ e apprende i valori dei parametri ϑ che meglio approssimano la funzione f^*. Il modello viene definito feedforward in quanto l'informazione iniziale *x* subisce una serie di computazioni intermedie usate per la definizione dell'output *Y*; non esistono quindi connessioni di *feedback* nelle quali gli outupt dell'algoritmo vengono forniti nuovamente in ingresso come fonte di informazione addizionale.

- **Convolutional Networks**: sono un esempio di reti neurali specializzate sul processo di dati che hanno una topologia a griglia ben nota: un esempio pratico può essere quello delle immagini intese come una struttura 2D di pixel. Il nome stesso del metodo indica che il successo applicativo di queste reti è basato sull'utilizzo

massivo della *convoluzione* come operazione matematica ed il loro utilizzo nel campo applicativo ha ottenuto un successo straordinario.

- **Recurrent and Recursive Networks**: sono una famiglia di reti neurali dedicate al processamento di dati sequenziali. A differenza delle Convolutional Network, specializzate in strutture dati a griglia come un'immagini, questo tipo di reti neurali è specializzato sulle sequenze di dati della forma $x_1, x_2, \dots, x_n$ ed è capace di scalare facilmente su sequenze di grandi dimensioni proprio come una Convolutional scala abilmente quando ha a che fare con immagini di grandi dimensioni. La maggior parte delle Recurrent Networks può anche processare sequenze di lunghezza variabile.

- **Autoencoder**: è una rete neurale addestrata per cercare di copiare l'input nel suo output. Internamente, possiede uno strato nascosto h che descrive il codice usato per la rappresentazione dell'input. La rete può quindi essere vista come costituita di due parti: una prima che definisce una funzione *encoder* $h = f(x)$ e una seconda, con una funzione *decoder* definita $r = g(x)$ che produce invece una ricostruzione. Tradizionalmente, queste strategie venivano utilizzate per effettuare la riduzione della dimensionalità o per l'apprendimento di feature; recentemente invece, le teoriche connessioni tra autoencoder e modelli a variabili latenti, hanno portato questi metodi all'applicazione nei casi più all'avanguardia della modellazione generativa; possono essere immaginati come casi particolari di reti feedforward programmabili con le stesse tecniche di apprendimento basate sul gradiente ma a differenza delle feedforward, le autoencoder possono anche essere programmate con la *ricircolazione*, un algoritmo di apprendimento basato sulla comparazione delle attivazioni della rete sull'input originale.

4.4 Esempi addizionali di Machine Learning

Oltre a queste macro aree di problemi di Machine Learning descritte nei paragrafi precedenti, esistono altre tipologie di algoritmi di apprendimento organizzati su una

tassonomia che si basa sul genere di risultati che si vogliono ottenere dall'applicazione dell'algoritmo.
Tra i più comuni si richiamano:

- **Semi-supervised learning**: metodo che combina assieme entrambi le tipologie di esempi (supervisionato e non supervisionato) per generare differenti tipi di classificatori specifici di alcuni problemi.

- **Reinforcement learning**: in questo caso l'algoritmo apprende un modello di comportamento attraverso l'osservazione dei fatti che accadono nell'intorno. Ogni nuova azione genera di conseguenza un qualche impatto nell'enviroment il quale fornisce un nuovo feedback che viene utilizzato dall'algoritmo come ulterirore strumento di apprendimento.

- **Transduction**: modello molto simile a quello supervisionato ma con la differenza che non ha l'obbiettivo di costruire esplicitamente una funzione per effettuare la classificazione, ma cerca invece di prevedere un nuovo output sulla base di più training input, training output e nuovi input.

Capitolo 5

Quantum Machine Learning

Negli ultimi anni i ricercatori stanno investendo sempre più risorse nel capire in che misura il quantum computing possa aiutare a migliorare le prestazioni degli algoritmi classici di Machine Learning. L'idea base di quello che si vuole ottenere è un'esecuzione di quegli algoritmi con costo computazionale elevato e delle loro sub-routine, più efficiente con l'applicazione del computer quantistico, traducendo le logiche originali all'interno della teoria quantistica.

Da quando il volume globale dei dati raccolti è in continuo aumento del 20% ogni anno, parallelamente sta crescendo con grande rapidità la necessità di trovare approcci innovativi per la loro elaborazione e questo sta portando molte aree di studio a valutare più profondamente le potenzialità della computazione quantistica proprio per l'ottimizzazione degli algoritmi di Machine Learning. Recentemente, molti lavori condotti soprattutto da fisici ed esperti di computer science, hanno dimostrato l'impressionante potenza computazionale dei sistemi quantistici per il processamento dei dati dovuto alle proprietà e alle caratteristiche delle teorie quantistiche e ad oggi siamo già in possesso di numerosi esempi di come gli algoritmi quantistici facciano ottenere uno speed-up rispetto ai migliori metodi classici utilizzati sullo stesso problema grazie all'accesso a subroutine non banali come la simulazione Hamiltoniana, amplificazione delle ampiezze ed estimazione di fase, di cui abbiamo accennato nel Capitolo 3. Molti esperti garantiscono che sia soltanto una questione di tempo per disporre delle tecnologie sufficientemente stabili per testare tutte le proposte teoriche di cui disponiamo.

Questa nuova area di ricerca dedicata al Quantum Machine Learning potrebbe davvero nascondere quel potenziale necessario per rivoluzionare il punto di vista che abbiamo dell'elaborazione intelligente dei dati. Molte intuizioni sono semplici, basti pensare che il compito del Machine Learning è manipolare e classificare un quantitativo enorme di dati e informazioni che tipicamente sono ordinati in array, quindi vettori, ed array di array,

prodotti tensori, due strutture che il quantum computing è capace di manipolare anche in spazi dimensionali di dimensioni elevate.
La situazione attuale dei metodi e del loro approccio dal punto di vista quantistico può essere riassunta come in tabella:

Metodi di Machine Learning	Approccio Quantistico
k-nearest neighbour Support Vector Machines k-means clustering	Calcolo efficiente della distanza nel quantum computer
Reti Neurali Alberi Decisionali	Prime esplorazioni dei modelli quantistici
Sistemi Lineari	Algoritmo HHL
Teoria Bayesiana	Riformulazione nel linguaggio dei sistemi quantistici aperti

Tabella 1-Riassunto della situazione Quantistica dei metodi Classici di Machine Learning

Il fatto che il calcolo della distanza sia già un'operazione funzionante e ottimizzata suggerisce la possibilità di guardare con particolare attenzione tutti quei metodi di classificazione che fanno della misura della distanza tra istanze del dataset il punto cardine dell'algoritmo.
Nei seguenti paragrafi verranno introdotti i metodi elencati, dandone una breve introduzione del modello classico per poi spiegare lo stato attuale dello sviluppo quantistico sulla base dei risultati presenti in letteratura.

5.1 Quantum k-nearest-neighbour

Uno dei metodi più semplici e standard che implementano la classificazione è l'algoritmo k-nearest neigbour, nel quale abbiamo un training set di vettori descritti secondo diverse caratteristiche e il valore della classe di appartenenza così come un vettore di input $\vec{x}$ del quale invece non conosciamo la classe e vogliamo identificarla in base alle k istanze del

training set che si trovano in un suo intorno. Questa metodologia si basa sull'assunzione che caratteristiche simili codifichino lo stesso genere di esempi è vero per moltissime applicazioni anche se la scelta del valore k non è sempre facile e può influenzare significativamente i risultati. Un valore troppo grande può significare la perdita di località dell'informazione finendo con una valutazione sull'intero training set, mentre un valore troppo piccolo aumenta il rischio di overfitting nel risultato.

Per la misurazione della distanza si utilizzano molto spesso il prodotto scalare, la distanza euclidea o quella di Hamming. Esistono diverse varianti dell'algoritmo che permettono di semplificare le misurazioni effettuandole sui centrodi delle classi e non sulle istanze del training set, oppure ancora pesare l'influenza della distanza indipendizzando l'algoritmo dal parametro k. Rimane il fatto che metodi come il k-nearest neighbour sono basati sulla misura della distanza per valutare la similarità tra vettori.

Gli sforzi per tradurre l'algoritmo in una versione quantistica sono concentrati dunque sulla valutazione efficiente della distanza con metodi quantistici; Aïmeur, Brassard and Gambs [15] introducono l'idea di utilizzare la sovrapposizione o *fedeltà*, che si basa sulla norma di due stati quantistici $|a\rangle$ e $|b\rangle$ come quantità di riferimento per misurare la similarità. La *fedeltà* può essere ottenuta attraverso l'applicazione di una semplice routine quantistica spesso definita '*swap test*'.

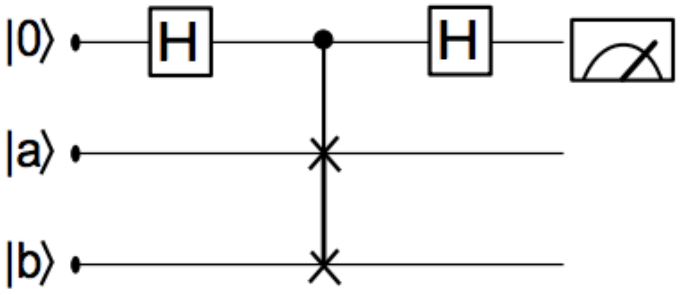


Figura 20-Circuito che realizza lo swap test

Dato un sistema quantistico $|a, b, 0_{ancilla}\rangle$ contenenti i due stati quantistici di partenza e un registro ancilla inizializzato nello stato $|0\rangle$, viene applicata una porta Hadamard sull'ancilla per metterlo in sovrapposizione, poi un controlled-swap gate è applicato su $|a\rangle$ e $|b\rangle$ per scambiarne gli stati sulla condizione che l'ancilla valga $|1\rangle$. Infine è applicato un ulteriore Hadamard in modo da ottenere lo stato finale:

$$|\psi_f\rangle = \frac{1}{2}|0\rangle(|a,b\rangle + |b,a\rangle) + \frac{1}{2}|1\rangle(|a,b\rangle - |b,a\rangle),$$

per il quale la probabilità di misurare l'ancilla qubit nello stato $|0\rangle$ è data da:

$$P(|0_{ancilla}\rangle) = \frac{1}{2} + \frac{1}{2}|\langle a|b\rangle|^2.$$

Dunque una probabilità uguale a ½ indica che i due stati $|a\rangle$ e $|b\rangle$ non si sovrappongono affatto (sono ortogonali), mentre la probabilità uguale a 1 ci dice che abbiamo una totale sovrapposizione.

Basandosi sullo swap test, Lloyd, Mohseni and Rebentrost [11] hanno recentemente proposto un modo per ricavare la distanza tra due vettori di numeri reali *n*-dimensionali $\vec{a}$ e $\vec{b}$ attraverso una misurazione quantistica; più precisamente essi calcolano il prodotto scalare tra due stati per valutare la fedeltà $|\langle\psi|\phi\rangle|^2$. Per farlo vengono preparati, in maniera molto efficiente, i due stati nella forma:

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(|0,\vec{a}\rangle + |1,\vec{b}\rangle\right),$$

$$|\phi\rangle = \frac{1}{\sqrt{|\vec{a}|^2 + |\vec{b}|^2}}\left(|\vec{a}||0\rangle - |\vec{b}||1\rangle\right),$$

e attraverso lo swap test eseguono la valutazione di $|\langle\psi|\phi\rangle|^2$. Il segreto risiede nello sfruttare la definizione di stato quantistico data da Seth Lloyd e dai suoi collaboratori, che utilizzano una rappresentazione dei dati alternativa codificando l'informazione classica all'interno della norma di uno stato quantistico:

$$\langle x|x\rangle = |\vec{x}|^{-1}\vec{x}^2 \Longrightarrow |x\rangle = |\vec{x}|^{-\frac{1}{2}}\vec{x}.$$

Con questa definizione l'identità:

$$|\vec{a} - \vec{b}|^2 = \left(|\vec{a}|^2 + |\vec{b}|^2\right)|\langle\psi|\phi\rangle|^2$$

rimane valida.

La distanza classica tra due vettori $\vec{a}$ e $\vec{b}$ può di conseguenza essere trovata attraverso un semplice swap test di stati accuratamente preparati in anticipo. Lloyd, Mohseni and Rebentrost applicano questa procedura per ottenere una versione quantistica di un algoritmo *k*-nearest neighbour basato sull'uso dei centroidi; gli stessi autori hanno verificato che nonostante le operazioni necessarie alla realizzazione degli stati quantistici coinvolti, questo metodo quantistico è più efficiente rispetto all'ottenere lo stesso valore in un computer classico.
Wiebe, Kapoor and Svore [13] usano ugualmente lo swap test per calcolare il prodotto scalare tra due vettori per valutarne la distanza, tuttavia utilizzano una rappresentazione differente dell'informazione classica sugli stati quantistici ottenendo uno speed-up quadratico comparato con la versione classica dell'algoritmo. In generale il quantum machine learning fornisce uno speed-up esponenziale rispetto a tutti gli algoritmi classici che coinvolgono la valutazione della distanza e del prodotto scalare tra vettori di grosse dimensioni.

5.2 Quantum Support Vector Machine

L'algoritmo delle Support Vector Machine (SVM) viene utilizzato per la discriminazione lineare di insiemi e rappresenta una sottocategoria della classificazione basata su pattern. L'obiettivo della discriminazione lineare è trovare un *iperpiano* che sia il miglior discriminatore tra due regioni spaziali che identificano due classi differenti e che in futuro possa essere utilizzato come strumento per la classificazione di nuovi input.
Per definizione l'iperpiano di discriminazione migliore è quello che massimizza la distanza tra se stesso e i punti più vicini di ogni classe, chiamati *support vector*.
La soluzione si ottiene applicando un modello matematico di ottimizzazione per trovare il massimo margine tra l'iperpiano e i support vector; ad esempio nel caso bidimensionale le condizioni di separazione sono date da:

$$\begin{cases} \vec{w}\,\vec{v}_i + b \geq 1 & c_i = 1, \\ \vec{w}\,\vec{v}_i + b \leq -1 & c_i = -1, \end{cases}$$

per ogni support vector $\vec{v}_i$ appartenente al training set di classe $c_i \in \{1, -1\}$; uno dei metodi principali classicamente utilizzati per la risoluzione di problemi di ottimizzazione come questo è il metodo di Lagrange.
Per la struttura matematica del problema, le SVM appartengono ad una classe più grandi di metodi definiti 'kernel methods'; questo fatto è di fondamentale importanza in quanto il calcolo del *kernel* può diventare estremamente dispendioso in termini di risorse computazionali al crescere del numero di istanze all'interno del dataset; per questa ragione è molto importante trovare un metodo di valutazione del prodotto scalare (che deve essere calcolato durante la fase di ottimizzazione della funzione obiettivo) efficace che possa essere applicato a questo genere di modelli.

Qui è dove entra in gioco la computazione quantistica; Rebentrost, Mohseni e Lloyd [12] hanno dimostrato che la valutazione di un prodotto scalare può essere sviluppata più velocemente su un calcolatore quantistico, dato lo stato quantistico:

$$|\psi\rangle = \frac{1}{\sqrt{N_\psi}} \sum_{i=1}^{N} |\vec{x}_i|\, |i\rangle|x^i\rangle; \quad N_\psi = \sum_{i=1}^{N} |\vec{x}_i|^2,$$

dove lo stato iniziale può essere costruito utilizzando il *Quantum Random Access Memory* come spiegato approfonditamente in [16] e dove gli $|x^i\rangle$ formano una base computazionale 2^n-dimensionale dello spazio dei vettori di training così che ogni vettore di training $|v^c\rangle$ può essere rappresentato con una sovrapposizione $|v^c\rangle = \sum \alpha_i |x^i\rangle$. È possibile allora effettuare la valutazione del prodotto scalare necessaria alla classificazione considerando che la valutazione quantistica di un prodotto scalare si basa sul fatto che gli stati quantistici sono normalizzati come:

$$\langle x^i|x^j\rangle =, \frac{\vec{x}_i \cdot \vec{x}_j}{|\vec{x}_i|\left|\vec{x}_j\right|}.$$

La matrice kernel *K* dei prodotti scalari può essere calcolata tenendo una traccia parziale della corrispondente matrice di densità $|\psi\rangle\langle\psi|$ sugli stati $|x^i\rangle$:

$$tr_x[|\psi\rangle\langle\psi|] = \frac{1}{\sqrt{N_\psi}} \sum_{i,j=1}^{N} \langle x^i|x^j\rangle |\vec{x}_i|\left|\vec{x}_j\right| |i\rangle\langle j| = \frac{\widehat{K}}{tr[K]}.$$

Rebentrost, Mohseni e Lloyd, sempre in [12], propongono un possibile utilizzo della valutazione del prodotto scalare non solo per ottenere la matrice kernel ma anche un suo possibile utilizzo per la classificazione di pattern, valutando il prodotto scalare $\vec{w}\ \vec{v}_i$ per la condizione di separazioni.

5.3 Quantum Clustering

Il Clustering si occupa di dividere un set di dati non previamente classificati in *k* subset di elementi realizzando autonomamente una classificazione in base alle caratteristiche delle istanze; proprio per questo motivo rappresenta la più importante metodologia all'interno dell'apprendimento non supervisionato (Unsupervised Learning) in quanto non sfrutta un training set per la preparazione iniziale. Solitamente, le metodologie classiche di clustering sfruttano come misura di dissimilarità tra due vettori la distanza Euclidea:

$$\left(\vec{a} - \vec{b}\right)^2 \ con\ \vec{a}, \vec{b}\ \in\ \mathbb{R}^n.$$

In letteratura, l'algoritmo classico di Clustering più famoso è il *k-means*, che assegna iterativamente i dati alla classe in cui la distanza rispetto al centroide risulta minima, all'inizio i centroidi sono scelti casualmente, ad ogni iterazione viene ricalcolata la posizione del centroide basandosi sui risultati di quella precedente fino ad ottenere k insiemi separati, dove *k* è un intero scelto anch'esso a priori come parametro dell'algoritmo. L'algoritmo ha sempre una terminazione anche se non è detto che la soluzione incontrata sia la ottimale; il k-means è affetto da diverse problematiche inerenti la scelta della funzione di distanza da utilizzare, la scelta del numero di cluster e dei punti di partenza e infine come gestire la presenza di cluster vuoti e outliers (dati sporchi che portano a risultati distorti).
L'idea base per sfruttare la computazione quantistica nei problemi di Clustering è applicare quella che viene definita in meccanica quantistica *Fedeltà* come misura di distanza:

$$Fid(|\psi\rangle, |\phi\rangle) = |\langle\psi|\phi\rangle|^2.$$

In letteratura sono presenti varie routine per effettuare clustering con computazione quantistica, per esempio Aïmeur, Brassard, Gilles and Gambs [8] utilizzano due subroutine per applicare una versione del *k-means*.
Lloyd, Mohseni and Rebentrost in [11], presentano una possibile implementazione del *k-means* basto su *quantum computing adiabatico*: un'alternativa all'implementazione di gate unitari e che prova ad aggiustare continuamente i parametri del sistema quantistico in un processo adiabatico che trasferisce uno stato fondamentale facile da preparare in uno stato fondamentale nel quale è codificato il risultato della computazione. Nell'articolo vengono forniti alcuni step e viene menzionato come la computazione adiabatico possa essere di supporto al problema della ricerca iniziale dei primi centroidi.

5.4 Quantum Neural Network

Una rete neuronale è ispirata al modello biologico dell'elaborazione delle informazioni del cervello; può essere sintetizzato come un grafo costituito da un insieme di elementi x_m legati tra loro da connessioni pesate con parametri w_{ml} che rappresentano l'equivalente delle sinapsi di un modello neuronale biologico.

$$x_l = \sum_{m=1}^{N} w_{ml} x_m$$

Una funzione di attivazione definisce il valore di un neurone basandosi sul valore corrente di tutti gli altri stati pesati dai valori w_{ml} e la dinamica della rete si sviluppa con il continuo aggiornamento dei neuroni attraverso la funzione di attivazione. Questo genere di modello può essere visto come un vero e proprio strumento computazionale e la sua programmazione può essere fatta attraverso il settaggio dei pesi w_{ml} e l'utilizzo di una funzione di attivazione che codifichi una certa relazione tra input e output.
Per la classificazione di pattern di solito si considera una rete neuronale feed-forward, nella quale i neuroni sono organizzati in strati (layers) ed ogni livello nutre con le sue informazioni quello successivo. Un insieme di valori iniziali vengono utilizzati per

alimentare l'input layer e dopo i successivi aggiornamenti su ogni strato, è possibile leggere il valore finale sull'output layer.
Spesso le reti neuronali feed-forward utilizzano come funzione di attivazione una *sigmoide*:

$$sgm(a,k) = \frac{1}{(1-e^{-ka})}.$$

La rete viene inizializzata con un set di input e l'output iniziale viene confrontato con i valori attesi per aggiustare il valore dei pesi per minimizzare l'errore della classificazione. Se un appropriato set di pesi è dato, questo genere di reti sono in grado di classificare nuovi input estremamente bene.
Nonostante vari approcci e idee nel tentativo di adattare le reti neuronali alla computazione quantistica, non si conosce una proposta concreta che descriva un metodo di classificazione quantistico con reti neuronali sufficientemente performante e funzionante. Trovare questo adattamento rimane tuttora una delle più interessanti sfide.

5.5 Alberi Decisionali Quantistici

Gli alberi decisioni sono probabilmente uno dei classificatori più intuitivi da immaginare; secondo la risposta ad una domanda sulla base delle caratteristiche, si segue una ramificazione dell'albero fino ad arrivare ad una foglia, o nodo di terminazione, nel quale troviamo la classe di appartenenza. Più precisamente possiamo definirlo come un albero con un nodo radice che ad ogni livello di profondità ha altri nodi fino a quelli finali che abbiamo definito foglie. Ogni nodo, eccetto per le foglie, contiene una funzione di decisione che decide quale ramo seguire per scendere allo strato successivo, scartando l'ulteriore partizione.
Come tutti i classificatori del Machine Learning supervisionato, gli alberi decisionali sono costruiti partendo da un training set di dati con differenti caratteristiche, che devono essere utilizzate per suddividere l'insieme iniziale in sottoinsiemi il più efficacemente possibile. Questo è spesso eseguito utilizzando la misura dell'entropia di Shannon [17]

A seconda delle applicazioni, una soluzione ottimale per un albero decisionale dovrebbe essere limitata nel numero di nodi, ramificazioni e foglie.

Lu and Brainstein [18] propongono una versione quantistica per gli alberi decisionali che segue coerentemente la versione classica con la differenza nell'utilizzare stati quantistici per le caratteristiche dei vettori $|v\rangle^p = |v_1^p, v_2^p, \dots, v_n^p\rangle$ codificando n valori delle caratteristiche delle istanze in un unico stato quantistico. Ad ogni nodo dell'albero, l'insieme di training composto dagli stati quantistici è partizionato in un sottoinsieme da una procedura definita dagli stessi autori 'stima d'attributo'; la spiegazione data nell'articolo rimane molto enigmatica sul come effettivamente avviene il partizionamento in sottoinsiemi. Nonostante questa mancanza di dettaglio nella parte saliente dell'algoritmo, suggeriscono l'idea di utilizzare l'entropia di von Neumann per la partizione del grafo. Nonostante il primo passo sia stato compiuto, il potenziale del Quantum Decision Tree è ancora da stabilire con chiarezza.

5.6 Quantum HHL per risoluzione di Sistemi Lineari

Consideriamo ora il problema della risoluzione di sistemi di equazioni lineari; esso rappresenta un problema molto diffuso e conosciuto in quanto è alla base della risoluzione di moltissimi problemi scientifici ed ingegneristici.
Definiamo il problema dando un sistema ad *N* equazioni lineari, con *N* non conosciuto, che può essere espresso nella forma:

$$A\vec{x} = \vec{b},$$

dove $\vec{b}$ è il vettore delle soluzioni, *A* la matrice dei coefficienti delle equazioni e $\vec{x}$ il vettore che soddisfa tale relazione. Partiremo dal presupposto che la matrice *A* è Hermitiana per ottenere una dimostrazione più semplificata, ma è dimostrato che anche matrici non Hermitiane possono essere riportate al caso che descriviamo di seguito attraverso alcuni step matematici aggiuntivi. Inoltre dobbiamo scalare i vettori $\vec{b}$ e $\vec{x}$ come vettori unitari in modo tale che la loro norma sia uguale a 1 e che possano essere rispettivamente

rappresentati come statici quantistici $|b\rangle$ e $|x\rangle$; effettuata questa prima trasformazione possiamo rappresentare il problema come:

$$A|x\rangle = |b\rangle,$$

ed è possibile ricavare le soluzioni $|x\rangle$ date da:

$$|x\rangle = \frac{A^{-1}|b\rangle}{\|A^{-1}|b\rangle\|}.$$

L'algoritmo quantistico per la risoluzione di sistemi lineari è stato proposto per la prima volta da Harrow, Hassidim, Lloyd [19], da cui l'algoritmo prende il nome di HHL. Il problema principale sta nel fatto che la misurazione quantistica permette di osservare un solo stato e quindi non permette di ottenere tutti i valori di $\vec{x}$, per questo motivo è necessario recuperare i risultati partendo dal valore atteso di un operando M con $\vec{x}$ tale che $\vec{x}^{\dagger}M\vec{x}$. Questo è particolarmente utile quando si implementa su quantum computer dato che i risultati che si ottengono sono sempre sotto forma di probabilità rispetto ad un operatore. Consideriamo ora di disporre degli autovalori λ_j normalizzati ad 1 e delle autobasi $|u_j\rangle$ della matrice A, allora possiamo scrivere $|b\rangle$ come combinazione lineare delle autobasi di A:

$$|b\rangle = \sum_{j=1}^{N} \beta_j |u_j\rangle.$$

L'obiettivo dell'algoritmo diventa quello di ottenere $|x\rangle$, non come vettore reale delle soluzioni $\vec{x}$, ma un'approssimazione statistica nella forma:

$$|x\rangle = \sum_{j=1}^{N} \beta_j \frac{1}{\lambda_j} |u_j\rangle.$$

Raggiungere questo scopo è possibile grazie alla possibilità di ottenere la scomposizione spettrale di A, che essendo Hermitiana può essere rappresentata in funzione dei suoi autovalori e autovettori:

$$A = R^{\dagger}\Lambda R,$$

Questa scomposizione possiamo ottenerla attraverso una serie di step dell'algoritmo. Matematicamente l'equivalenza è valida grazie alle proprietà di cui gode *A* in quanto per ipotesi Hermitiana.
Potendo riscrivere il problema di partenza come:

$$R^{\dagger}\Lambda R|x\rangle = |b\rangle,$$

il *primo step* è quello di premoltiplicare entrambi i membri per la matrice *R* che contiene gli autovettori di *A*, e considerando che una matrice per la sua trasposta coniugata è l'identità trasformiamo l'equazione precedente in:

$$RR^{\dagger}\Lambda R|x\rangle = R|b\rangle$$

$$= \Lambda R|x\rangle = R|b\rangle.$$

Il *secondo step* prevede di premoltiplicare l'equazione per l'inversa della matrice degli autovalori Λ. In quanto Λ contiene gli autovalori di una matrice Hermitiana che sono sempre reali, l'inversa di una matrice diagonale a valori reali è definita come Λ^{-1}:

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \lambda_2 & 0 & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \lambda_n \end{bmatrix}, \qquad \Lambda^{-1} = \begin{bmatrix} \frac{1}{\lambda_1} & 0 & 0 & 0 \\ 0 & \frac{1}{\lambda_2} & 0 & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \frac{1}{\lambda_n} \end{bmatrix}.$$

Rappresentando il loro prodotto l'identità otteniamo il seguente sviluppo dell'equazione:

$$\Lambda^{-1}\Lambda R|x\rangle = \Lambda^{-1}R|b\rangle,$$

$$R|x\rangle = \Lambda^{-1}R|b\rangle.$$

Si prosegue applicando un *terzo step* dove, valendo per le matrici degli autovettori la proprietà: $R^{\dagger}R = RR^{\dagger} = Identità$, riusciamo ad isolare $|x\rangle$:

$$R^{\dagger}R|x\rangle = R^{\dagger}\Lambda^{-1}R|b\rangle,$$

$$|x\rangle = R^{\dagger}\Lambda^{-1}R|b\rangle.$$

In generale HHL permette di realizzare questi tre passaggi con algoritmi noti visti nel Capitolo 3 e con l'applicazione di trasformazioni quantistiche. Il primo step equivale all'applicazione della Stima delle Fasi utilizzando la trasformazione unitaria e^{iAt} e mappando gli autovalori λ_j in forma binaria all'interno di un registro quantistico per ottenere uno stato quantistico della forma:

$$|\psi\rangle = \sum_{j=1}^{N} \beta_j \, |\widetilde{\lambda_j}\rangle |u_j\rangle,$$

qui $\widetilde{\lambda_j}$ rappresenta la codifica binaria del j-esimo autovalore. A questo punto è necessario un qubit ancilla per effettuare su di esso delle rotazioni controllate sulla base degli autovalori λ_j. Questo secondo step, considerando il valore iniziale dell'ancilla $|0\rangle$ porta ad una sovrapposizione della forma:

$$|\psi\rangle = \sum_{j=1}^{N} \beta_j \left(\sqrt{1 - \frac{C^2}{{\lambda_j}^2}} |0\rangle_{ancilla} + \frac{C}{\lambda_j} |1\rangle_{ancilla} \right) |\lambda_j\rangle |u_j\rangle.$$

Il terzo ed ultimo step è composto dall'inversione della Stima delle Fasi, realizzabile applicando in ordine contrario i gate della fase uno, per questo viene spesso definito '*uncompute*'. Completato questo passaggio si ottiene uno stato finale:

$$|\psi\rangle = \sum_{j=1}^{N} \beta_j \left(\sqrt{1 - \frac{C^2}{{\lambda_j}^2}} |0\rangle_{ancilla} + \frac{C}{\lambda_j} |1\rangle_{ancilla} \right) |0\rangle |u_j\rangle.$$

Effettuando la misurazione dello stato un numero considerevole di volte e selezionando solamente i risultati con ancilla qubit a $|1\rangle$ possiamo stabilire la seguente proporzione:

$$|x\rangle \approx \sum_{j=1}^{N} C\left(\frac{\beta_j}{\lambda_j}\right)|u_j\rangle.$$

La possibilità di risolvere un sistema lineare con un algoritmo quantistico è molto interessante, soprattutto dato che rappresenta un caso in cui è possibile ottenere uno speed-up delle prestazioni di ordine esponenziale.

In generale, la complessità del problema dipende dalla dimensione *N* del vettore $\vec{b}$, ossia dal numero di soluzioni che l'algoritmo deve trovare per risolvere il sistema; classicamente il miglior risultato per la risoluzione di sistemi lineari si ottiene con l'eliminazione Gaussiana, che ha una complessità $O(N^3)$; ciò può essere risotto se abbiamo a che fare con una matrice *A* sparsa, raggiungendo un costo computazionale dell'ordine $O\left(\frac{sk}{\log \varepsilon} N \log N\right)$, dove *s* è la sparsità della matrice *A*, *k* il rapporto tra il più grande autovalore della matrice ed il più piccolo e ε la precisione.

L'algoritmo quantistico HHL è in grado di effettuare la computazione in $O\left(\frac{k^2 s^2}{\varepsilon} \log N\right)$.

Gli esperimenti pratici su computer quantistici sono ancora limitati dalla stabilità dei calcolatori e dal numero di qubit necessari per la codifica della matrice *A* oltre che a causa della decoerenza degli stati. La risoluzione di sistemi lineari rimane comunque uno degli aspetti più interessanti della computazione quantistica e del suo possibile utilizzo per ottenere giovamenti in problemi reali che il Machine Learning deve affrontare costantemente.

Capitolo 6

Sviluppo e Implementazione

In questo appartato verrà analizzata in dettaglio la libreria software utilizzata per la fase di sviluppo, fornendo alcuni dettagli di installazione e di primo approccio con il linguaggio. Si riporteranno parti di codice con annessa spiegazione di alcuni degli algoritmi utilizzati. Il codice completo sarà disponibile nei contenuti in appendice della tesi.

6.1 Qiskit

Qiskit è un framework open-source scritto in python, reso pubblico da IBM, che permette di lavorare su un computer quantistico affetto da rumore e creare simulazioni di semplici circuiti e algoritmi. L'obiettivo che rimane alla base di questo progetto è quello di espandere i suoi contenuti e le sue funzionalità per migliorarne l'utilizzo da parte degli utenti e soprattutto facilitare la ricerca sulle difficoltà, problematiche e sfide inerenti il quantum computing che oggigiorno affliggono questo nuovo modello computazionale.
Una prima importante caratteristica di questa piattaforma è la possibilità di effettuare test su differenti architetture definiti *backend* che possono essere suddivisi in due macroaree di utilizzo:

- **Simulatore su hardware classico**: possiamo eseguire circuiti ed algoritmi direttamente sul nostro sistema e su di esso verrà simulato il comportamento del quantum computer affetto da rumore ottenendo risultati simili a quelli di un quantum computer reale, oppure possiamo emularne il comportamento scartando gli effetti derivanti da errori che corrompono gli stati codificati. Questa seconda opzione rappresenta un'alternativa più didattica, utile per osservare il comportamento delle ampiezze dei qubit e il risultato dell'applicazione dei vari

operatori. Per gli esempi pratici è preferibile valutare un algoritmo sulla base dell'utilizzo del backend QASM, invece del backend State Vector, che emulando un comportamento reale, richiede un'esecuzione ripetuta più volte per procedere ad una valutazione più realistica e concreta dei risultati.

- **Simulazione su real device**: IBM mette a disposizione in cloud diversi calcolatori quantistici reali, ognuno caratterizzato da un chip con un numero fisso di qubit utilizzabili e caratteristiche intrinseche sulla velocità di calcolo e sulla tolleranza relativa agli errori. Su questi device è possibile lanciare piccoli script di dimensione limitata ad un numero finito di gate direttamente scrivendo codice oppure creando circuiti direttamente da una interfaccia grafica disponibile online [25].

L'intero framework di Qiskit è suddiviso in quattro elementi fondamentali, ognuno specificatamente dedicato ad un'area della computazione e dello studio del quantum computing. Possono essere riassunte in:

- **Terra**: è l'elemento fondamentale del framework in quanto fornisce tutti gli strumenti per la realizzazione di algoritmi direttamente al livello circuitale, dando la possibilità di utilizzare registri quantistici, classici e vari operatori già implementati. Permette inoltre di effettuare un accesso remoto ai real device e gestire la tipologia di simulazione.
- **Aer**: si occupa dei miglioramenti del simulatore, in particolare per quanto riguarda l'emulatore e il debugging dei programmi. L'obbiettivo è migliorarne lo sviluppo delle singole componenti per capire sempre più approfonditamente i limiti dei processori classici dimostrando anche fino a che punto e in che misura possono simulare il calcolo quantistico.
- **Ignis**: è la parte dedicata alla lotta contro i nemici del quantum computing: errori, rumore e decoerenza degli stati. Si occupa dunque di migliorare il comportamento dei gate, caratterizzazione dell'errore ed elaborazione in presenza di rumore con lo scopo di aumentare la tolleranza della computazione.
- **Aqua**: rappresenta la parte più applicativa del quantum computing, permette infatti di applicare la computazione a problemi e situazioni reali senza scendere a basso livello. Al momento l'utilizzo è limitato ad aspetti di Intelligenza Artificiale, ottimizzazione e chimica, per esplorare le possibilità e i benefici dell'uso di questo modello computazionale senza doversi preoccupare di come debba essere tradotto

il problema nel linguaggio del quantum computing. L'utilizzo non è semplice, in quanto è più che necessaria una solida base di conoscimento delle fondamenta del funzionamento dei sistemi.

In generale l'intero framework può essere riassunto come in Figura 21:

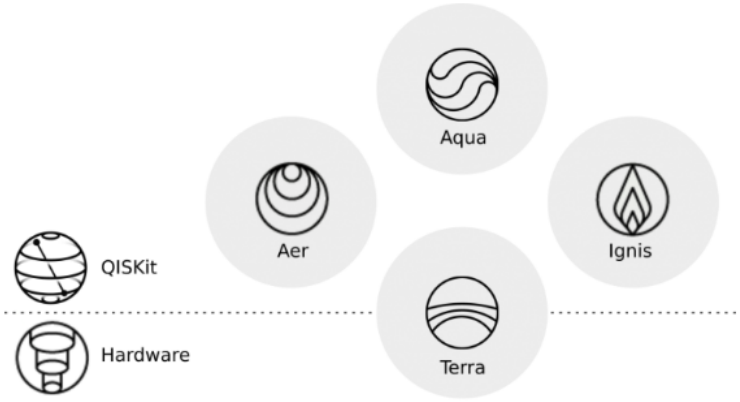


Figura 21-Architettura di Qiskit

Ai fini di questa tesi, abbiamo utilizzato prevalentemente Qiskit Terra, in quanto dispone del necessario per la completa definizione e simulazione di circuiti e algoritmi.
La libreria Qiskit è supportata da qualsiasi sistema operativo e necessita esclusivamente della *versione 3.5 di Python*. Per l'installazione è sufficiente eseguire da linea di comando:

```
pip install qiskit
```

Per l'utilizzo, se si lavora su Pycharm, è sufficiente importare nel progetto la libreria Qiskit che si troverà tra quelle disponibili una volta installata; se invece si utilizzano editor come Jupyter Notebook bisognerà solamente importare i moduli che vogliamo utilizzare direttamente sul file di lavoro.
A questo punto passiamo a fornire un'introduzione su come iniziare ad utilizzare Qiskit, suddividendone gli aspetti principali. Ogni volta che creiamo un circuito, o sviluppiamo un algoritmo, il flusso di lavoro consiste di due fasi salienti: la prima è quella della *costruzione del circuito* per rappresentare il problema che vogliamo risolvere, la seconda è quella di *esecuzione*, dove sceglieremo la metodologia in base alla selezione del backend. Completata l'esecuzione vengono infine raccolti i dati dei risultati che possono essere

rappresentati in forme grafiche o semplicemente stampati in console per effettuare le relative valutazioni.

6.1.1 Costruzione del Circuito e Gate

Per l'utilizzo delle porte predefinite nella libreria, creare circuiti e definire registri quantistici e classici è necessario importare i seguenti moduli all'interno del file:

```
from qiskit import QuantumCircuit,
          QuantumRegister, ClassicalRegister
```

La prima cosa da fare è definire un registro quantistico con un numero *n* di qubit sui quali svolgeremo tutte le operazioni che definiremo; è importante sottolineare che l'inizializzazione del registro setta automaticamente ogni qubit nello stato iniziale $|0\rangle$:

```
# Define quantum register
qr = QuantumRegister(n, "qr")
```

dove la variabile *n* indica il numero arbitrario di qubit che vogliamo gestire, mentre il secondo termine della funzione rappresenta il nome associato al registro ma non ha implicazioni pratiche nell'uso.
Possiamo ora definire effettivamente un circuito che sarà quindi composto dal registro quantistico sui quali qubit applicheremo in seguito gli operatori:

```
# Define quantum circuit
circuit = QuantumCircuit(qr, name="quantum_circuit")
```

A questo punto, creato il circuito, è possibile iniziare ad interagire direttamente sui singoli qubit. Tutte le trasformazioni applicate ad un singolo qubit possono essere espresse come l'applicazione di specifiche rotazioni di determinati valori angolari, questo porta alla

definizione di gate generalizzati nei quali possiamo definire l'entità della rotazione da applicare: tali operatori sono tre e rispettano le seguenti equivalenze:

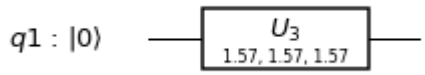
$$u_3(\theta, \phi, \lambda)$$

$$u_2(\phi, \lambda) = u_3(\pi/2, \phi, \lambda)$$

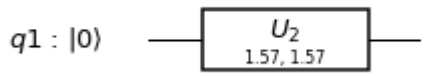
$$u_1(\lambda) = u_3(0, 0, \lambda).$$

Nella programmazione risultano anche molto utili per la realizzazione di operatori più complessi che non sono inizialmente disponibili all'interno della libreria. Le tre tipologie di rotazioni appena introdotte possono essere applicate al singolo qubit rispettivamente:

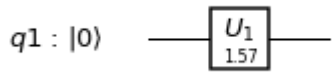
```
# u3 gate
circuit.u3(pi/2, pi/2, pi/2, qr[0])
print(circuit)
```




```
# u2 gate
circuit.u2(pi/2, pi/2, qr[0])
print(circuit)
```

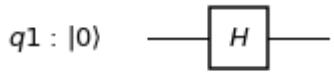



```
# u1 gate
circuit.u1(pi/2, qr[0])
print(circuit)
```



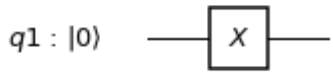
Uno dei gate più importanti e utilizzati è la porta di Hadamard, utilizzato per ottenere una sovrapposizione di stati equiprobabili. Equivale all'applicazione di $u_2(0, \pi)$ ma la libreria mette a disposizione la funzione:

```
# Hadamard gate
circuit.h(qr[0])
print(circuit)
```

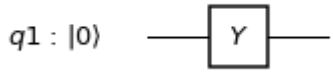


Altri gate a singolo qubit disponibili sono le matrici di Pauli X, Y e Z ognuna delle quali modifica lo stato del qubit esattamente come descritto nella parte teorica; nonostante le loro operazioni equivalgano rispettivamente a $X = u_3(\pi, 0, \pi)$, $Y = u_3(\pi, \pi/2, \pi)$, $Z = u_1(\pi)$, possono essere applicate come segue:

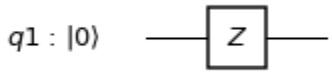
```
# X gate
circuit.x(qr[0])
print(circuit)
```




```
# Y gate
circuit.y(qr[0])
print(circuit)
```

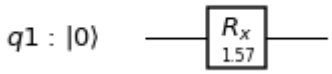



```
# Z gate
circuit.z(qr[0])
print(circuit)
```

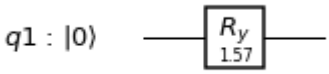


Sempre sul singolo qubit possono essere applicati altri gate di rotazione, questa volta considerando la rotazione attorno ad uno degli assi principali (x, y, z). Rispettivamente possono essere rappresentati sempre nella forma di altri operatori elementari, rispettivamente: $R_x = u_3(\theta, -\pi/2, \pi/2)$, $R_y = u_3(\theta, 0, 0)$, $R_z = u_1(\phi)$ ma possono essere eseguiti sul singolo qubit:

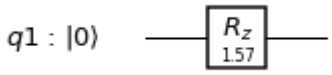
```
# Rx gate
circuit.rx(pi/2, qr[0])
print(circuit)
```




```
# Ry gate
circuit.ry(pi/2, qr[0])
print(circuit)
```



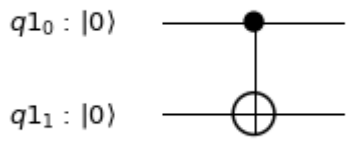

```
# Rz gate
circuit.rz(pi/2, qr[0])
print(circuit)
```



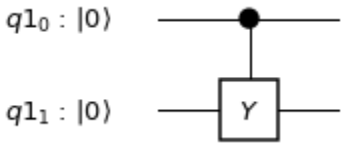
Passiamo ora alla descrizione di gate multi-qubit e di conseguenza ad utilizzare per gli esempi un circuito con registro quantistico a due. Solitamente questa tipologia di operatori viene definita *controlled gate*, in quanto un qubit è utilizzato come controllo per effettuare l'operazione su un target se e solo se il valore del controllo è |1⟩; un primo classico esempio già visto anche nella parte teorica è il controlled-NOT che equivale ad un controlled-X, da ciò possiamo estendere l'operazione controllata anche ad una qualsiasi matrice di Pauli (controlled-X, controlled-Y, controlled-Z), ad una porta Hadamard (controlled-Hadamard),

ad una rotazione attorno all'asse z (controlled-Rz) o una qualsiasi rotazione angolare (controlled-u1, controlled-u3):

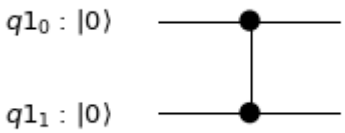
```
# Controlled-NOT o Controlled-X gate
circuit.cx(qr[0], qr[1])
print(circuit)
```



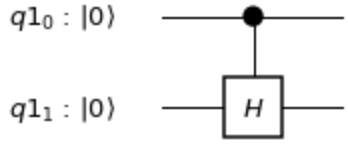

```
# Controlled-Y gate
ircuit.cy(qr[0], qr[1])
print(circuit)
```




```
# Controlled-Z gate
ircuit.cz(qr[0], qr[1])
print(circuit)
```

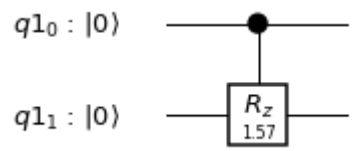



```
# Controlled-Hadamard gate
circuit.ch(qr[0],qr[1])
print(circuit)
```

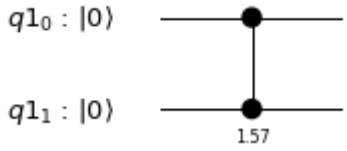



```
# Controlled-Z_Rotation gate
```

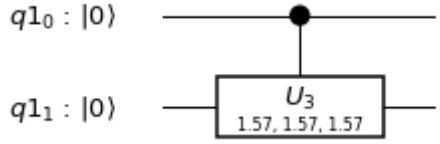
```
circuit.crz(pi/2, qr[0],qr[1])
print(circuit)
```




```
# Controlled-u1 gate
circuit.cu1(pi/2, qr[0],qr[1])
print(circuit)
```

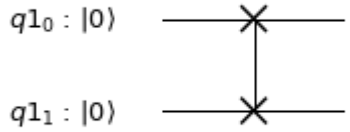



```
# Controlled-u3 gate
circuit.cu3(pi/2, pi/2, pi/2, qr[0],qr[1])
print(circuit)
```



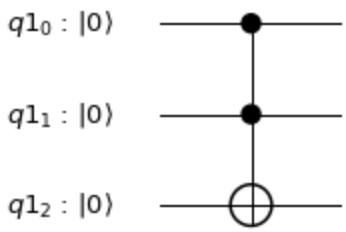
Un altro gate disponibile che abbiamo analizzato nella teoria è lo SWAP, che come abbiamo visto può essere rappresentato in maniera del tutto equivalente con l'utilizzo di tre controlled-X:

```
# SWAP gate
circuit.swap(qr[0], qr[1])
print(circuit)
```



Infine introduciamo una porta di fondamentale importanza per fini pratici e computazionali in quanto risulta molto utile per realizzare nuovi gate più complessi. La porta di Toffoli utilizza due qubit di controllo per effettuare l'operazione di negazione sul terzo, viene rappresentata nella più semplici delle ipotesi su un registro ad almeno tre qubit come:

```
# Toffoli gate
circuit.ccx(qr[0], qr[1], qr[2])
print(circuit)
```



Esistono ovviamente altri esempi di gate di minore rilevanza pratica ma comunque utilizzabili. Per un approfondimento sui gate quantistici si rimanda il lettore alla documentazione ufficiale di Qiskit [24].

6.1.2 Misurazione, esecuzione e Backend

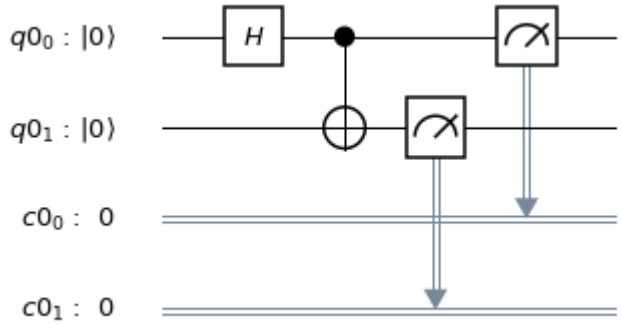
Nella sezione precedente abbiamo definito come realizzare la struttura di un circuito e operare sui qubit che lo compongono. Nel momento in cui vogliamo ottenere un effettivo risultato della computazione dobbiamo però effettuare una misurazione dello stato finale in cui si troverà il sistema dopo l'applicazione dei vari operatori. Per farlo abbiamo bisogno di un registro classico ad *n* bit, dove *n* è uguale al numero di qubit utilizzati pe il registro quantistico, in modo tale da poter misurare l'intero stato quantistico che, rispettando le regole descritte nel Capitolo 2 inerenti il concetto di misurazione, collasserà in uno e un solo stato finale. Dobbiamo quindi aggiungere alla definizione di registro quantistico quello classico e inserirlo nella definizione del circuito:

```
# Define quantum and classical registers on circuit
```

```
qr = QuantumRegister(n, "qr")
cr = ClassicalRegister(n, 'cr')
circuit = QuantumCircuit(qr, cr, name="quantum_circuit")
```

A questo punto il circuito non è altro che un insieme di qubit inizializzati a $|0\rangle$ che subiscono una serie di variazioni del loro stato in base ai gate utilizzati nel circuito, e un insieme di bit disponibili alla misurazione dello stato finale; l'operazione di misurazione deve essere sempre fatta come ultimo passaggio sul circuito in quanto, simulando il comportamento del quantum computer, una volta effettuata si perdono tutte le informazioni inerenti i singoli stati in possibile sovrapposizione.
Nell'esempio seguente riportiamo il circuito utilizzato per ottenere uno stato entangled tra il primo e il secondo qubit realizzando lo stato de Bell $|00\rangle \rightarrow |\beta_{00}\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$.

```
# Bell|00> state circuit
circuit.h(qr[0])
circuit.cx(qr[0],qr[1])
circuit.measure(qr, cr)
print(circuit)
```



Con il circuito definito e la misurazione inserita solo manca configurare il tipo di esecuzione che vogliamo attuare sul sistema; per prima cosa dobbiamo importare i moduli necessari all'esecuzione e ai backend disponibili:

```
from qiskit import execute, BasicAer
from qiskit import IBMQ
```

Ora possiamo scegliere su quale *backend* eseguire il nostro algoritmo: BasicAer mette a disposizione diversi approcci per la simulazione su un calcolatore classico sul quale possiamo emulare un comportamento ideale e privo di errori o una simulazione del comportamento di un calcolatore quantistico dove invece errore e rumore non sono trascurati; inoltre è anche disponibile la possibilità di eseguire il nostro algoritmo, rispettando determinate caratteristiche, su real device messi a disposizione in cloud da IBM grazie al modulo IBM.
Nei seguenti paragrafi vengono descritti separatamente i diversi approcci alla simulazione.

6.1.2.1 Simulazione su hardware classico

Il modo più semplice e immediato per avere un primo contatto con la simulazione degli algoritmi quantistici è quella di eseguire il codice su un computer classico. Qiskit mette a disposizione due backend a tale scopo: il primo, chiamato **Statevector simulator**, ci permette di eseguire il codice senza nessuna interferenza di errore o rumore, quindi quello che otteniamo è un vettore di numeri complessi rappresentante i valori delle ampiezze di ogni possibile configurazione di qubit; il suo scopo è prettamente didattico in quanto permette di interpretare il comportamento dei gate e del circuito senza interferenze sul risultato. Per utilizzare questa simulazione e visualizzarne i risultati è necessario definire il backend e lanciare l'esecuzione, dopodiché è possibile stampare i risultati ottenuti:

```
# Backend definition
backend = BasicAer.get_backend('statevector_simulator')
# Execution
job_sv = execute(circuit, backend)

# Printing results
result_sv = job_sv.result()
sv_out = result_sv.get_statevector(circuit, decimals=2)
print(sv_out)
```

La stampa dei risultati fornisce:

```
[0.71+0.j 0.  +0.j 0.  +0.j 0.71+0.j]
```

L'interpretazione è quella di leggere ogni stato nell'ordine standard $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ e osservarne l'ampiezza di ognuno. Giustamente e come previsto, in questo caso, avendo utilizzato il circuito che permette di ottenere lo stato di Bell $|00\rangle \rightarrow |\beta_{00}\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$, vediamo come i valori delle ampiezze sono correttamente distribuite sugli stati $|00\rangle$ e $|11\rangle$ con valore $\frac{1}{\sqrt{2}}$ e nullo per gli altri poiché eliminati per effetto dell'entanglement. Il risultato risulta dunque corretto e privo di imprecisione.

Il secondo metodo di simulazione su computer classico che permette invece di emulare il comportamento affetto da imprecisione del calcolatore quantistico. In Qiskit è definito come **QASM simulator**; il suo realismo è dovuto all'introduzione di rumore all'interno degli operatori e della misurazione. In questo caso è indispensabile inserire la misurazione per il funzionamento, infatti con l'esecuzione quello che si ottiene è il valore di un singolo stato e proprio per questo motivo è necessario definire il numero di volte che vogliamo che il calcolo venga ripetuto (shots), in modo da poter osservare il numero di occorrenze in cui una determinata configurazione è stata misurata.
Ripetiamo l'esecuzione del circuito per ottenere lo stato di Bell $|\beta_{00}\rangle$ con l'utilizzo di questo backend:

```
# Backend definition
backend = BasicAer.get_backend('qasm_simulator')
job_qsim = execute(circuit, backend=backend, shots=1024)

# Printing results
result_qsim = job_qsim.result()
print(result_qsim.get_counts())
```

La stampa dei risultati fornisce:

```
{'00': 569, '11': 455}
```

Dalla quale possiamo dedurre che, dopo aver lanciato il circuito e misurato lo stato finale, abbiamo ottenuto circa il 50% di volte uno dei due risultati attesi, ma non lo stesso valore. Questo metodo è in generale quello più utilizzato nel testare gli algoritmi poiché permette di emulare il comportamento del circuito su un problema reale, dove osservare il numero di occorrenze è utile per trarre conclusioni sui risultati.

6.1.2.2 Simulazione su hardware quantistico

Oltre alla simulazione su una piattaforma classica, Qiskit mette a disposizione gli strumenti per autenticarsi e interfacciarsi con IBM Q Experience, una piattaforma in cloud di IBM nella quale è possibile eseguire simulazioni su calcolatori quantistici reali.
Un primo metodo per farlo consiste nell'utilizzare **Open QASM** (Open Quantum Assembly Language) come linguaggio di codice per creare una connessione al dispositivo ed effettuare la computazione. Per farlo è necessario creare un account e ottenere un API_token visualizzabile nelle configurazioni dell'account online, che si utilizza per effettuare la connessione e la visualizzazione dei backend disponibili sui quali possiamo effettivamente lanciare il circuito che definiamo. Dobbiamo anche importare ulteriori moduli per valutare quale device risulta meno occupato e quindi disponibili a effettuare la computazione limitando i tempi di attesa. Il codice intero per la simulazione su real device si trova in Appendice VIII.:

```
from qiskit.providers.ibmq import least_busy, job_monitor

# Connection
APItoken= ####################
IBMQ.enable_account(APItoken)

# Print available backends and select the less busy one
print(IBMQ.backends(operational=True, simulator=False))
backend = least_busy((IBMQ.backends(operational=True,
                                    simulator=False))
print("The best backend is " + backend.name())
```

Stampando il listato dei backend possiamo attualmente visualizzare i seguenti device disponibili:

```
[<IBMQBackend('ibmqx4') from IBMQ()>,
<IBMQBackend('ibmqx2') from IBMQ()>,
<IBMQBackend('ibmq_16_melbourne') from IBMQ()>]
```

E applicando la funzione di selezione del meno occupato otteniamo il calcolatore con meno attesa. Proseguiamo ora con la parte inerente la definizione dell'esecuzione, per la quale nuovamente dobbiamo stabilire il numero di ripetizioni e il numero di crediti massimo che vogliamo spendere per l'utilizzo del quantum device (ogni giorno ne vengono assegnati 15, e una esecuzione de 1024 shots costa 3 crediti):

```
# Bell|00> state circuit
# ...
# Monitoring results
from qiskit.tools.monitor import job_monitor
job_exp = execute(circuit, backend=backend,
                                  shots=1024, max_credits=3)
job_monitor(job_exp)
```

Otterremo una serie di stati dopo l'inizializzazione dell'esecuzione:

```
Job Status: job is queued (1)

Job Status: job is actively running

Job Status: job has successfully run
```

Ora possiamo visualizzare i risultati stampandoli con:

```
# Printing results
result_exp = job_exp.result()
counts_exp = result_exp.get_counts(circuit)
print(counts_exp)
```

Con il quale otteniamo:

```
{'11': 427, '01': 90, '00': 464, '10': 43}
```

Risulta interessante osservare come una simulazione su un computer quantistico comporti risultati molto più realistici e sporchi.
Completata l'esecuzione e ogni operazione sui dati rilevati possiamo disconnetterci con il seguente comando:

```
# Disconnection
IBMQ.disable_accounts(token=APItoken)
```

Un ulteriore modo di interfacciarsi all'utilizzo in cloud di un vero quantum computer è il **Quantum Composer**: una interfaccia grafica disegnata per permettere agli utenti di creare algoritmi utilizzando un insieme di gate quantistici predefiniti ed eseguirli. Al momento per il Composer sono disponibili due chip a cinque qubit, rispettivamente chiamati:

- IBM Q 5 Tenerife [ibmqx4]
- IBM Q 5 Yorktown [ibmqx2]

Per entrambi sono disponibili informazioni inerenti la rappresentazione del circuito interno del chip, come sono interconnessi i qubit tra di loro e una serie di parametri interni inerenti la frequenza e l'errore accumulabile quando il circuito comincia ad espandersi con l'aumentare del numero dei gate inseriti. L'utilizzo è molto semplice e non necessita di spiegazioni dettagliate.
Riportiamo un esempio di come possiamo realizzare l'entanglement tra i primi due qubit per ottenere lo stato di Bell $|\beta_{00}\rangle$:

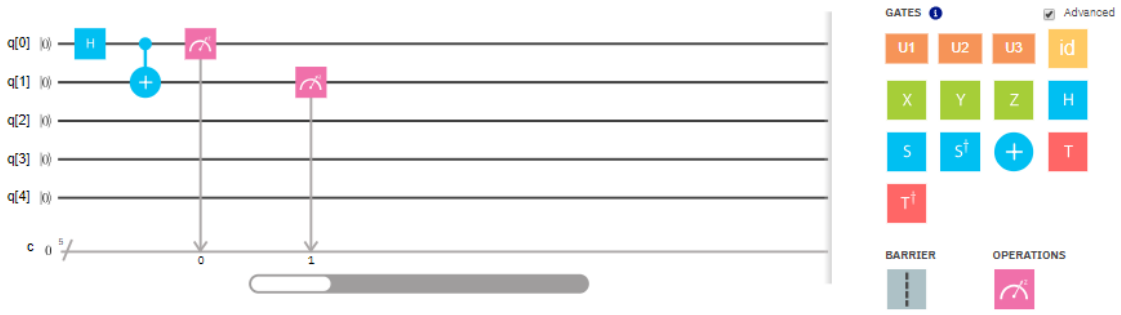


Figura 22-Interfaccia grafica del Quantum Composer di IBM

Preparato il circuito aggiungendo una porta Hadamard, un controlled-NOT e due operatori di misurazione possiamo lanciare l'esecuzione e aspettare che una volta terminata siano disponibili i risultati:

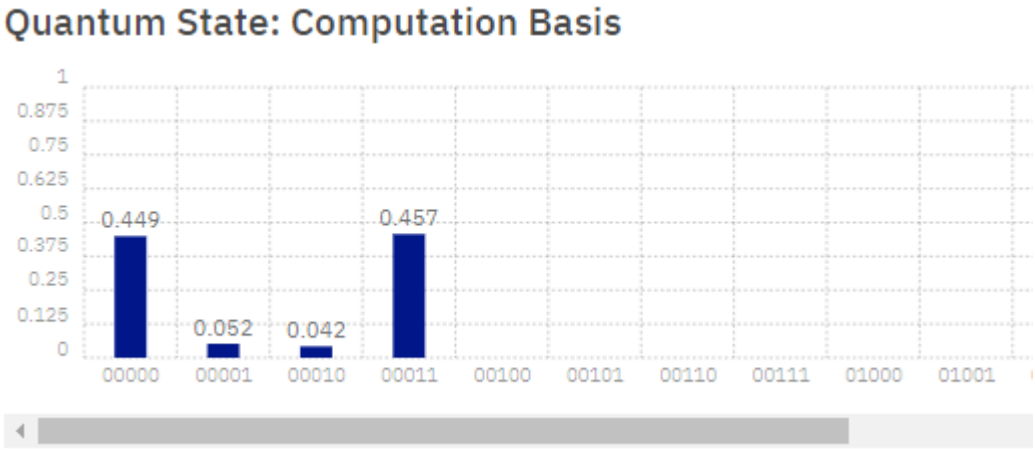


Figura 23-Report sull'esecuzione su computer quantistico del circuito per ottenere uno stato di Bell

La visualizzazione dei risultati in formato grafico è molto interessante ed utile per l'interpretazione. Nel seguente paragrafo verranno descritte le principali funzioni per ottenere anche nel codice il plotting dei risultati ottenuti.

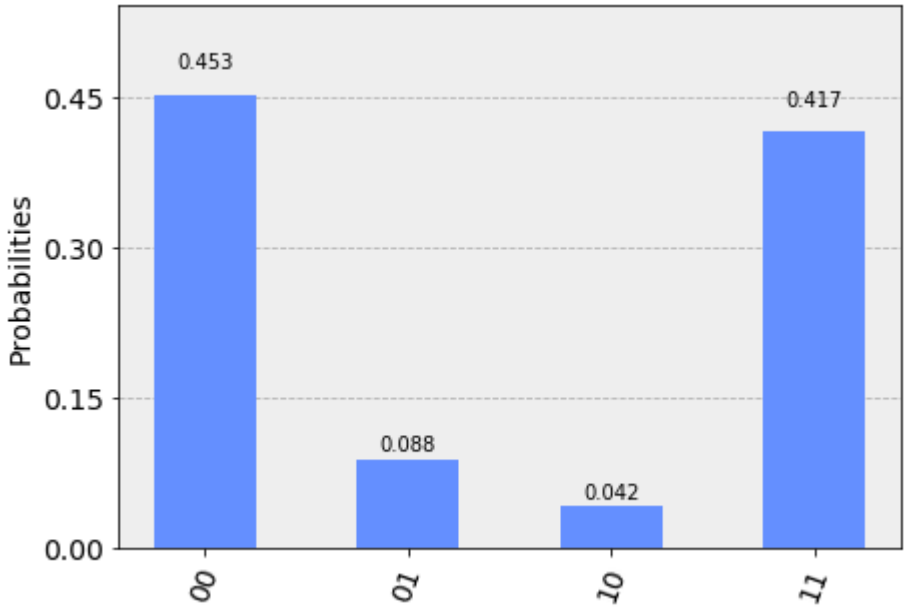
6.1.3 Visualizzazione dei Risultati

Un ultimo aspetto da descrivere, utile per l'interpretazione dei risultati, è quello di interpretare graficamente le misurazioni degli output. Fino ad ora abbiamo visto come definire un circuito quantistico in tutte le sue componenti e come eseguire una simulazione che rispecchi più fedelmente possibile una computazione quantistica. I risultati ottenuti possono essere plottati in forme di istogrammi per chiarire maggiormente gli effetti della computazione. Importiamo il modulo per disporre degli strumenti per il plotting:

```
from qiskit.tools.visualization import plot_histogram
```

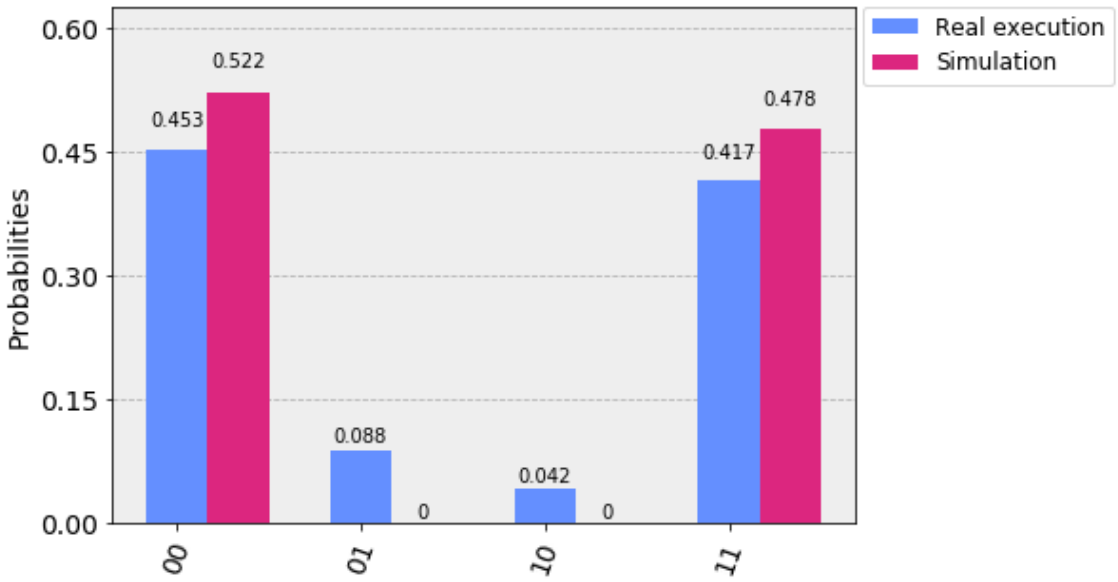
Riprendendo l'esempio descritto in precedenza dove abbiamo simulato su un quantum computer reale la realizzazione dello stato di Bell $|\beta_{00}\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$, eseguendo il comando seguente otteniamo la rappresentazione a istogrammi delle misurazioni rilevate:

```
# Bell|00> state circuit
# ...
result_exp = job_exp.result()
counts_exp = result_exp.get_counts(circuit)
# Plotting results
plot_histogram([counts_exp])
```



Un altro aspetto utile di questa visualizzazione è quella di poter comparare sullo stesso grafico due esecuzioni parallele dello stesso circuito o una esecuzione reale ed una fittizia. Per farlo effettuiamo una seconda computazione e plottiamo i dati nello stesso grafico:

```
# Simulation of the circuit
backend = BasicAer.get_backend('qasm_simulator')
result_qsim = execute(circuit, backend, shots=1024).result()
ccounts_qsim = second_result.get_counts()
legend = ['Real execution, 'Simulation']
plot_histogram([counts_exp, counts_qsim], legend=legend,)
```



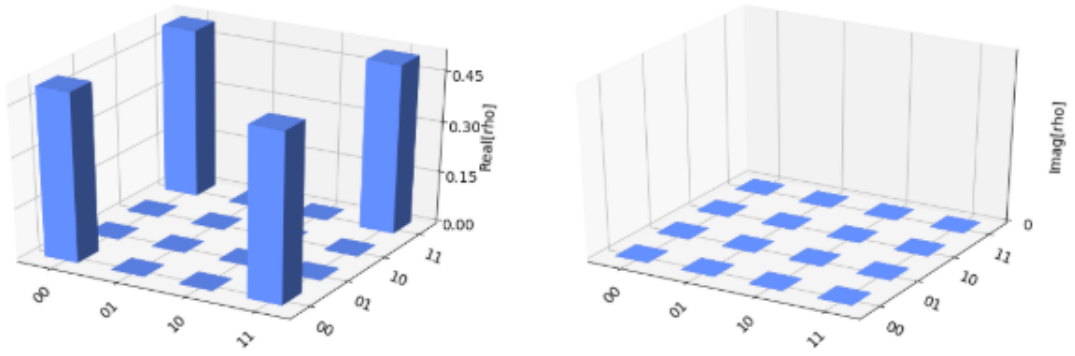
Già con questo semplice esempio possiamo osservare come diverse esecuzioni dello stesso algoritmo portino a risultati in linea ma differenti e come la simulazione non introduca la stessa quantità di errore e di difficoltà nell'interpretazione dei risultati rispetto al real device.

Altre possibili rappresentazioni sono utilizzabili sui risultati della simulazione Statevector, importabili con i seguenti moduli:

```
from qiskit.tools.visualization import plot_state_city,
```

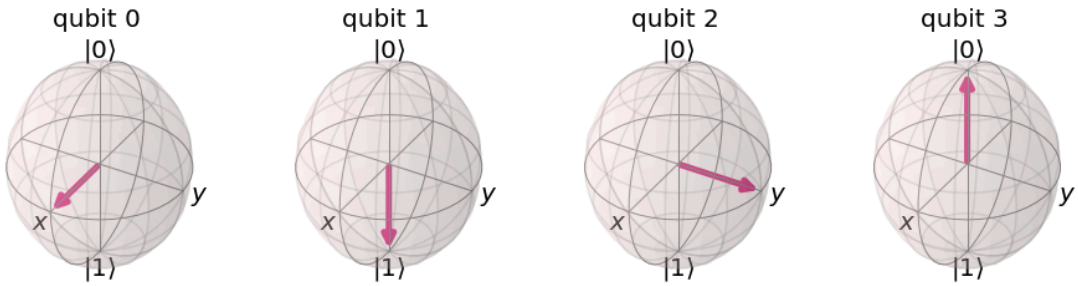
Permette di ottenere una rappresentazione tridimensionale dei valori delle ampiezze degli stati suddivisi in un grafico per la parte Reale ed uno per la parte Immaginaria. La loro utilità entra in gioco quando bisogna manipolare le matrici di densità e applicando la tomografia sul circuito. In generale si esegue con:

```
backend = BasicAer.get_backend('statevector_simulator')
job = execute(circuit, backend).
results = job.result()
outputstate = results.get_statevector(circuit)
plot_state_city(outputstate )
```



Un'ultima visualizzazione che può risultare utile soprattutto nelle fasi iniziali di approccio alla realizzazione dei circuiti e allo studio degli effetti dei singoli gate sulla variazione di stato del qubit, è quello di rappresentare tridimensionalmente la Sfera di Bloch appartenente ad ognuno dei qubit che compongono il registro. Nell'esempio che segue vediamo l'effetto di applicare su un registro a quattro qubit un gate differente per ognuno:

```
from qiskit.tools.visualization import
                          plot_bloch_multivector

n = 4
qr = QuantumRegister(n, "qr")
circuit = QuantumCircuit(qr, name="test_circuit")

circuit.h(qr[0])
circuit.x(qr[1])
circuit.u2(pi/2, 0,qr[2])
circuit.iden(qr[3])

# State Vector Simulation
backend = BasicAer.get_backend('statevector_simulator')
job = execute(circuit, backend)
results = job.result()
outputstate = results.get_statevector(circuit, decimals=5)
plot_bloch_multivector(outputstate)
```



Esistono ulteriori forme grafiche e comandi per gestire la rappresentazione dei risultati che otteniamo. Per la loro consultazione e approfondimento e per tutto il materiale disponibile sulla libreria Qiskit si rimanda il lettore alla documentazione ufficiale [100].

6.2 Implementazione di algoritmi quantistici

Dopo aver introdotto tutto il necessario per interfacciarsi con lo sviluppo di circuiti e algoritmi quantistici, passiamo ora a descrivere dal punto di vista implementativo alcuni degli algoritmi caratteristici del quantum computing dei quali abbiamo dato una versione teorica nel Capitolo 3. Per ognuno di essi si suddividerà il paragrafo in una breve introduzione iniziale dell'implementazione, seguita dall'intero codice Qiskit necessario per la fase di esecuzione. Questa ultima fase verrà presentata sotto forma di grafici e descrizione dei risultati. Infine si riporterà una stampa del circuito che realizza l'algoritmo.

6.2.1 Algoritmo di Grover

Come visto nella parte teorica, l'algoritmo di Grover può essere applicato ad un determinato problema di ricerca per individuare un elemento all'interno di un dataset. Compito dell'operatore di Grover è *identificare la soluzione* e *amplificarne l'ampiezza* per aumentare le probabilità di misurazione del circuito.

Prima di passare alla parte di implementazione dell'algoritmo è importante citare una difficoltà incontrata in questa implementazione. Per entrambe le fasi sopra citate, l'operatore di Grover necessita di una operazione *controlled-Z* che dipende dal numero di

qubit che decidiamo di utilizzare. Nel caso di due qubit, che ci permettono di realizzare quattro possibili stati, è sufficiente un controlled-Z a singolo controllo che come abbiamo avuto modo di vedere è presente tra gli operatori di Qiskit e quindi non genera problemi dal punto di vista implementativo.
Passando invece ad un esempio a tre qubit, con otto possibili stati rappresentabili, abbiamo bisogno di un operatore controlled-Z con due qubit di controllo (ccz), non disponibile tra gli operatori predefiniti. Seguendo la definizione teorica del circuito è stato possibile implementarlo e utilizzarlo per la fase di test:

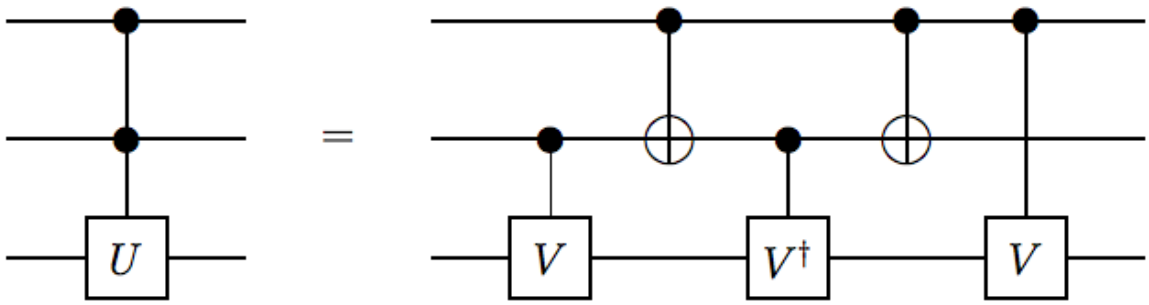


Figura 24-Equivalenza circuitale di un operatore con doppio controllo

Il codice che lo implementa è il seguente:

```
# Define number of qubits, normalization value and circuit
n = 3
qr = QuantumRegister(n, "qr")
norm = math.sqrt(2**n)

# Define ccz
def ccz():
    circuit.cu1((pi/norm), qr[1], qr[2])
    circuit.cx(qr[0], qr[1])
    circuit.cu1((-pi/norm), qr[1], qr[2])
    circuit.cx(qr[0], qr[1])
    circuit.cu1((pi/norm), qr[0], qr[2])
```

Successivamente abbiamo valutato la possibilità di utilizzare Grover su un sistema a quattro qubit e sedici stati disponibili, per il quale era ovviamente necessario implementare un controlled-Z con tre qubit di controllo (cccZ). Anche in questo caso disponendo di una rappresentazione teorica del circuito è stato possibile realizzare la funzione che lo implementa:

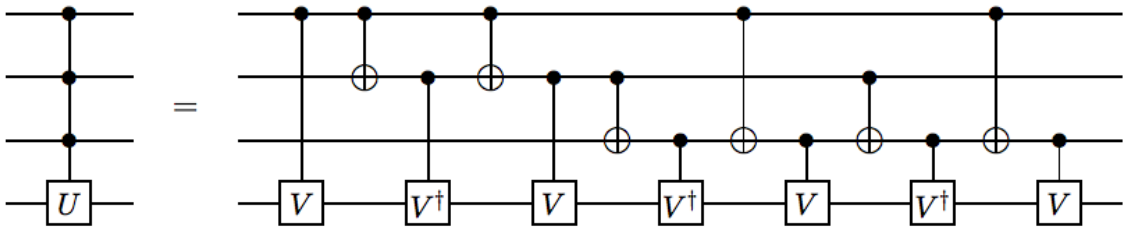


Figura 25-Equivalenza circuitale di un operatore con triplo controllo

Il codice per realizzare l'operatore è:

```
n = 4
norm = math.sqrt(2**n)
qr = QuantumRegister(n, "qr")

def cccz():
    circuit.cu1(pi / norm, qr[0], qr[3])
    circuit.cx(qr[0], qr[1])
    circuit.cu1(-pi / norm, qr[1], qr[3])
    circuit.cx(qr[0], qr[1])
    circuit.cu1(pi / norm, qr[1], qr[3])
    circuit.cx(qr[1], qr[2])
    circuit.cu1(-pi / norm, qr[2], qr[3])
    circuit.cx(qr[0], qr[2])
    circuit.cu1(pi / norm, qr[2], qr[3])
    circuit.cx(qr[1], qr[2])
    circuit.cu1(-pi / norm, qr[2], qr[3])
    circuit.cx(qr[0], qr[2])
    circuit.cu1(pi / norm, qr[2], qr[3])
```

Disponendo di questo risultato e non affrontando il caso a cinque qubit a causa di una notevole crescita della complessità del circuito, proseguiamo con la descrizione dell'algoritmo a quattro qubit. Nell'implementazione dell'operatore di Grover dobbiamo identificare il risultato che vogliamo e amplificarne l'ampiezza, per farlo la prima cosa è identificare il risultato che vogliamo trovare che, considerando una delle sedici stringhe di bit disponibili con quattro qubit, possiamo isolare applicando una porta X prima e dopo il cccZ sui qubit che vogliamo lasciare a 0. Ad esempio se vogliamo cercare la soluzione 0011, dobbiamo applicare un gate X sul terzo e quarto qubit, prima e dopo il cccZ:

```
def grover_operator():
# Prepare the state that the oracle'll find adding X gates
# on qubits you want to let in |0> state: example 0011.
    circuit.x(qr[2])
    circuit.x(qr[3])
    circuit.barrier()
    cccz()
    circuit.barrier()
    circuit.x(qr[3])
    circuit.x(qr[2])

    circuit.barrier()

    # Amplification step
    for i in range(n):
        circuit.h(qr[i])
    for i in range(n):
        circuit.x(qr[i])
    circuit.barrier()
    cccz()
    circuit.barrier()
    for i in range(n):
        circuit.x(qr[i])
    for i in range(n):
        circuit.h(qr[i])
```

La fase di amplificazione è sempre uguale a prescindere del numero di qubit a parte la variazione dell'operazione controllata.
Per eseguire l'algoritmo riportiamo di seguito l'intero codice:

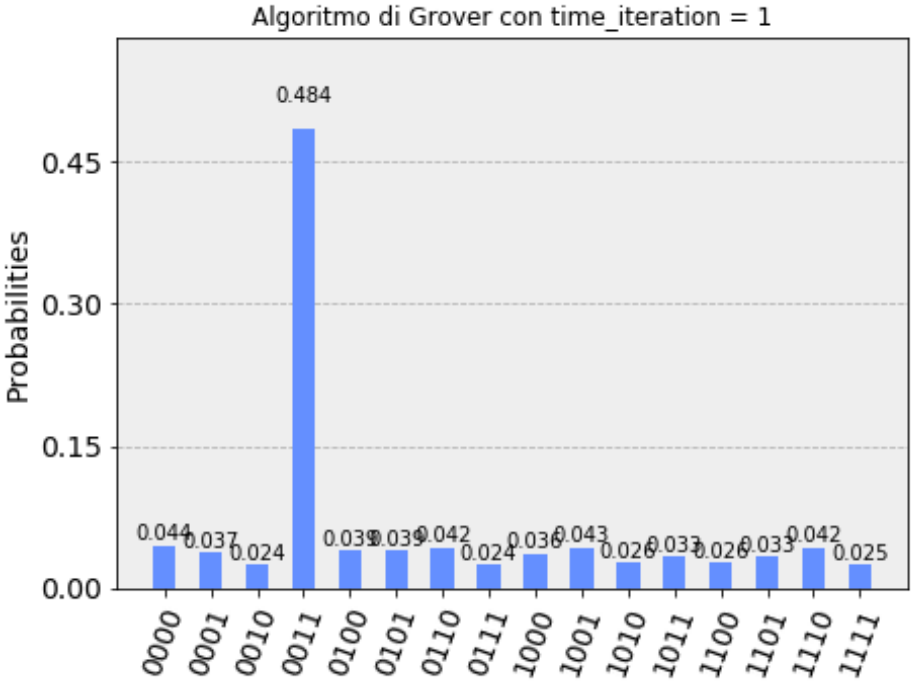
```
# Import packages
import numpy as np
from math import pi
from qiskit import ClassicalRegister,
           QuantumRegister, QuantumCircuit
from qiskit import execute, BasicAer, IBMQ
from qiskit.tools.visualization import plot_histogram

# Define registers and circuit
n = 4
norm = math.sqrt(2**n)
qr = QuantumRegister(n, "qr")
cr = ClassicalRegister(n, 'cr')
circuit = QuantumCircuit(qr, cr, name="Grover_4qubit")

# Hadamard over all qubit to obtein superposition
for i in range(n):
    circuit.h(qr[i])
circuit.barrier()

# Gate ccz and grover_operator implementation
def cccz():
def grover_operator():

# Define how many times you want iterate
# the Grover Operator[sqrt(2^n)]
time_iteration = 2
for i in range(time_iteration):
    grover_operator()
```

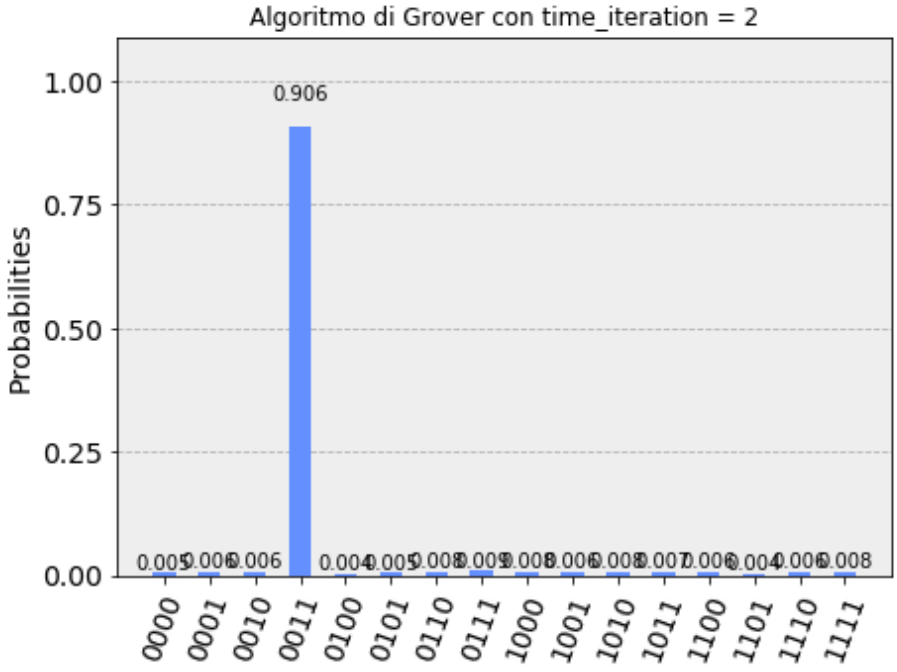
```
# Measurement
circuit.barrier()
for i in range(n):
    circuit.measure(qr[i], cr[i])
print(circuit)

# Statevector Simulation
backend_sv = BasicAer.get_backend('statevector_simulator')
job= execute(circuit, backend_sv)
results = job.result()
outputstate = results.get_statevector(circuit, decimals=2)
print(outputstate)

# QASM Simulation
backend = BasicAer.get_backend('qasm_simulator')
job = execute(circuit, backend=backend, shots=1024)
results = job.result()
answer = results.get_counts()
print(answer)

# Plot Results
plt = plot_histogram(answer)
plt.show()
```

Oltre alle porte Hadamard iniziali per ottenere la sovrapposizione degli stati, al circuito viene aggiunto la misurazione finale dopo aver iterato un numero finito di volte l'operatore di Grover.

Il numero di iterazioni può essere scelto prima di lanciare l'algoritmo e i grafici seguenti dimostrano come un numero maggiore di iterazioni aumenti l'ampiezza dello stato soluzione 1100 rispettando le aspettative della teoria.

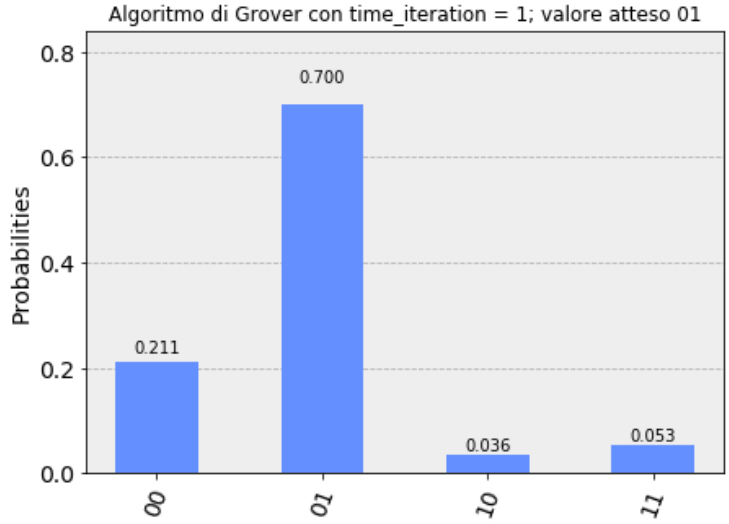
Algoritmo di Grover su backend QASM Simulator'

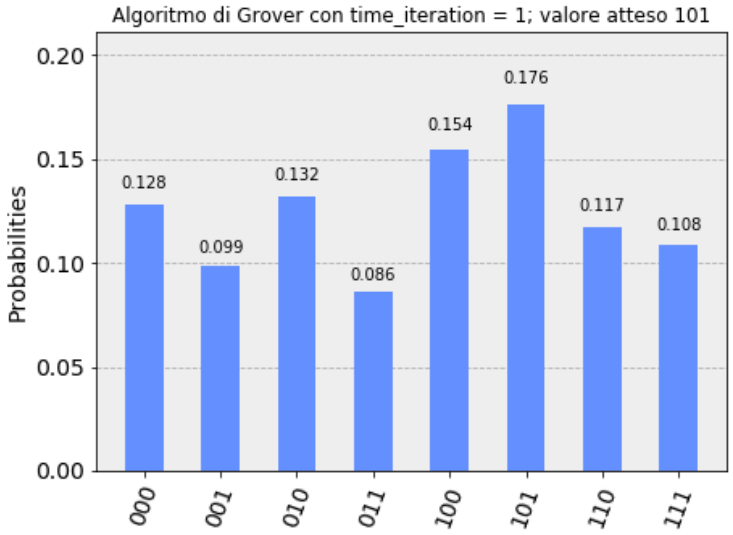


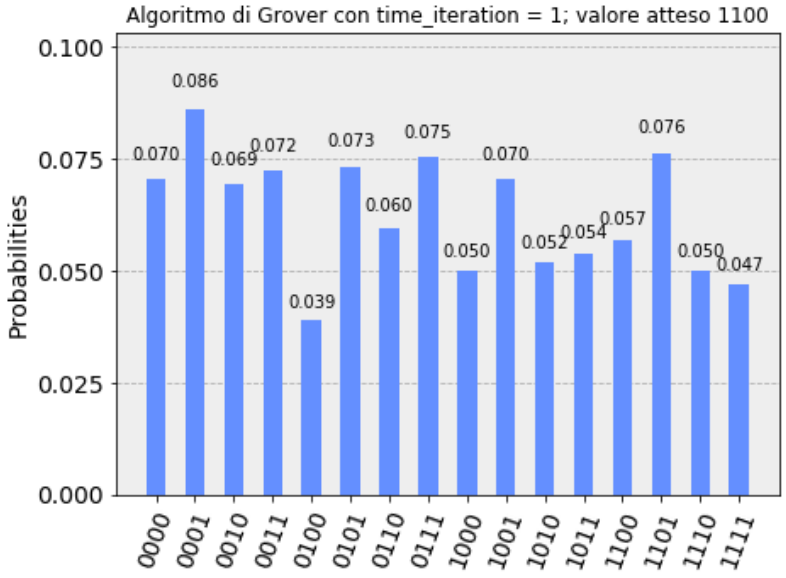


I risultati sono in linea con quelli attesi, vediamo che eseguire due iterazioni amplifica la probabilità di misurare il risultato corretto con probabilità maggiore del 90% mentre eseguendolo con solo una iterazione la misurazione sul risultato corretto avviene con meno del 50% di probabilità.

Algoritmo di Grover su backend reale 'ibmqx2' con 2, 3 e 4 qubit







Le cose cambiano eseguendo l'algoritmo su quantum computer reale, come possiamo osservare nella raccolta precedente, dove l'unico risultato soddisfacente è quello su un circuito a due qubit, aumentando a tre e a quattro la grandezza del registro e di conseguenza la complessità del circuito che implementa l'algoritmo, i risultati peggiorano incredibilmente, sottolineando lo scarso adattamento all'aumento delle dimensioni.
Il codice per le tre tipologie di implementazione, a due, tre e quattro qubit è disponibile insieme ai circuiti e raccolta dei risultati in Appendice I, II 2 III rispettivamente

6.2.2 Algoritmo Deutsch-Jouzsa

L'algoritmo di Deutsch-Jozsa è stato il primo a mostrare la vera separazione esistente tra l'approccio classico e quantistico alla risoluzione di un problema dimostrando una velocità esponenzialmente migliore rispetto ad ogni algoritmo classico deterministico. Essendo anche lì deterministico, produce sempre una risposta e tale risposta è sempre corretta.
L'algoritmo è caratterizzato da un oracolo, identificabile come *black box*, che implementa una funzione binaria dove dato un qualsiasi input, l'output è sempre una funzione costante con tutti i valori uguali a 0 o tutti 1, oppure bilanciata assumendo esattamente per la metà dei valori 0 e per l'altra metà 1. Lo scopo è quello di valutare la funzione con il minor numero di operazioni, cosa che classicamente avviene nel caso peggiore con $2^{n-1} + 1$ valutazioni della funzione.
Un primo step dell'implementazione è la valutazione casuale di una funzione, che supponiamo solo l'oracolo conosca:

```
oracleType = np.random.randint(2)
oracleValue = np.random.randint(2)

if oracleType == 0:
    print("The oracle returns a
                    constant value ", oracleValue)
else:
    print("The oracle returns a balanced function")
```

```
    # hidden parameter for balanced oracle n = number of qubit
    a = np.random.randint(1, 2**n)
```

A questo punto creiamo il nostro circuito a *n+1* qubit, dove *n* qubit comporranno il primo registro quantistico inizializzato a 0 e messo in sovrapposizione con l'applicazione di Hadamard mentre il secondo registro a un qubit dovrà essere prima settato su 1 con una porta X per poi essere messo in sovrapposizione. Questi primi step rappresentano esattamente l'inizializzazione del circuito vista nel Capitolo 3 parlando dell'algoritmo Deutsch-Jozsa.

```
# Creation of n+1 registers: x^n and y;
# As saw in theory we need to measure only x^n
n = 4
qr = QuantumRegister(n + 1)
cr = ClassicalRegister(n)
circuit = QuantumCircuit(qr, cr, name='DeutschJozsa')

# Create the superposition of all input queries in the first
# register by applying X to y and Hadamard to each qubit
circuit.x(qr[n])
for i in range(n+1):
    circuit.h(qr[i])
circuit.barrier()
```

Ora possiamo applicare l'oracolo, il cui compito è quello di modificare gli input in base al tipo di funzione:

```
# Application of Oracle operator
# If the oracleType is "0", it returns oracleValue
# for all input
if oracleType == 0:
    if oracleValue == 1:
        circuit.x(qr[n])
    else:
```

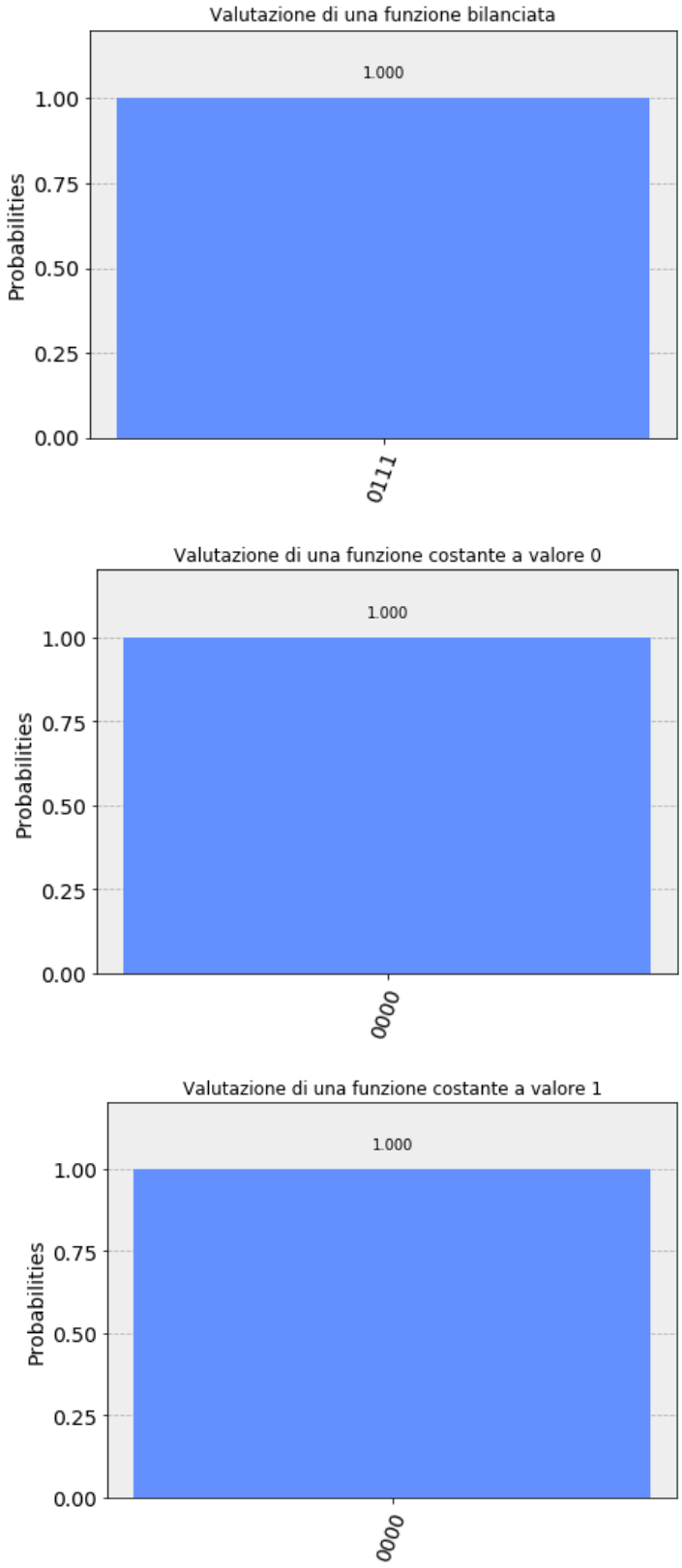
```
        circuit.iden(qr[n])
# If the oracleType is not "0", it returns the inner product
# of the input with a (non-zero bitstring)
else: for i in range(n):
        if (a & (1 << i)):
            circuit.cx(qr[i], qr[n])
circuit.barrier()
```

Infine si riapplica Hadamard su tutti gli input e se ne effettua la misurazione. Completato il circuito effettuiamo una prima valutazione con il backend di simulazione QASM Simulator:

```
# Apply Hadamard gates after querying the oracle
for i in range(n):
    circuit.h(qr[i])
# Measurement
circuit.barrier()
for i in range(n):
    circuit.measure(qr[i], cr[i])
print(circuit)

# QASM Simulation
backend = BasicAer.get_backend('qasm_simulator')
job = execute(circuit, backend=backend, shots=1024)
results = job.result()
answer = results.get_counts()
# Plot Probabilities Results
plt = plot_histogram(answer)
plt.show()
```

L'osservazione dei risultati sul simulatore è in linea con i risultati teorici: il margine di errore è nullo, e otteniamo la stringa di bit tutti a 0 solamente nel caso di funzione costante, mentre negli altri casi la funzione è sempre bilanciata.

Algoritmo di Deutsch-Jozsa su backend ‘Qasm simulator’ con 5 qubit



Codice dell’algoritmo, circuiti e ulteriori risultati si trovano in Appendice IV.

6.2.3 Quantum Fourier Transform

Un altro algoritmo di cui si è parlato nella parte teorica con una rilevante importanza dal punto di vista pratico è la Trasformata di Fourier Quantistica che rappresentando una trasformazione unitaria può essere implementata su circuito quantistico con una netta maggior efficienza rispetto all'implementazione classica dove sono necessari $O(n2^n)$ gate confronto agli $O(n^2)$ di quella quantistica. Questo guadagno esponenziale in efficienza è il cuore di molti algoritmi che ne sfruttano le potenzialità come ad esempio il Phase Estimation o l'algoritmo di Shor per la fattorizzazione, che un computer classico non può risolvere.
La sua implementazione è riconducibile ad una piccola funzione che applica un Hadamard ad un qubit ed esegue poi una rotazione controllata per ognuno degli altri qubit.

```
# Quantum Fourier Transform
def qft():
    for j in range(int(n/2)):
        circuit.swap(qr[j], qr[n-j-1])
    for j in range(n):
        for k in range(j):
            circuit.cu1(pi / float(2 ** (j-k)), qr[j], qr[k])
        circuit.h(qr[j])
```

Per testarne il comportamento ed il funzionamento la applichiamo ad un semplice circuito a due qubit del quale possiamo calcolare manualmente il risultato atteso e confrontarlo con quello dell'esecuzione. In questo scenario è facile applicare la matrice che descrive la trasformazione:

$$QFT_{2qubit} = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix},$$

che applicata allo stato $|00\rangle$, realizzabile semplicemente commentando l'operazione di rotazione e senza aggiungere gate al circuito in quanto corrisponde allo stato di inizializzazione dei qubit, risulta in:

$$QFT|00\rangle = \frac{1}{2}[|00\rangle + |01\rangle + |10\rangle + |11\rangle].$$

Il calcolo è facilmente verificabile osservando, non tanto il grafico delle probabilità, ma meglio ancora il risultato della simulazione con backend Statevector, dato che in questo modo abbiamo la possibilità di analizzare interamente il valore delle ampiezze, che ricordiamo essere un numero complesso.
Come sempre sviluppiamo il circuito definendone il numero di qubit e chiamando l'operazione della Trasformata di Fourier definita in precedenza:

```
# Define registers and circuit
n = 2
qr = QuantumRegister(n, "q")
cr = ClassicalRegister(n, "c")
circuit = QuantumCircuit(qr, cr, name="QFT_nqubit")

# Apply QFT
qft()
circuit.barrier()

# State vector simulation
backend_sv = BasicAer.get_backend('statevector_simulator')
job_sv = execute(circuit, backend_sv)
result_sv = job_sv.result()
outputstate = result_sv.get_statevector(circuit, decimals=2)
# Printing my initialization value on probability state
print(outputstate)
```

In questo caso l'esecuzione da:

```
[0.5+0.j 0.5+0.j 0.5+0.j 0.5+0.j]
```

Se invece calcoliamo la QFT di |01⟩, realizzabile applicando al principio un gate X sul secondo qubit, otteniamo la seguente trasformazione:

$$QFT|01\rangle = \frac{1}{2}[|00\rangle + \mathrm{i}|01\rangle - |10\rangle - \mathrm{i}|11\rangle].$$

Comparandolo con il risultato dello Statevector anche in questo caso il risultato combacia con quello atteso:

```
[ 0.5+0.j   0. +0.5j -0.5+0.j   0. -0.5j]
```

Per l'esempio a due qubit è stata verificata la correttezza dei risultati in tutti i casi. Per il codice completo e alcuni esempi del circuito su due e più qubit si rimanda alla sezione V dell'Appendice.

6.3 Algoritmi di Quantum Supervised Learning

Come è stato introdotto nel Capitolo 5, il quantum computing può essere teoricamente applicato a problemi di Machine Learning; entriamo più nello specifico su un problema di apprendimento supervisionato, dove il sistema predispone di un dataset di elementi dei quali si conoscono le caratteristiche espresse in forma vettoriale e la classe di appartenenza e ha l'obiettivo di essere capace di predire quella di un nuovo vettore di input. Una domanda centrale del Quantum Machine Learning è valutare e capire quanto l'utilizzo della computazione quantistica possa realmente accrescere le prestazioni dei metodi classici sfruttando metodi computazionali propri del quantum computing. Ad esempio, i metodi basati sulla distanza come *quantum k-nearest neighbour* e *clustering* sono basati su estensioni dell'amplificazione delle ampiezze [7, 8], mentre metodi basati su *quantum kernel* come *support vector machines* [9] fanno affidamento alle routine per l'inversione quantistica di matrici [11] o utilizzano oggetti matematici come la matrice di densità [12]. È importante sottolineare che il Quantum Machine Learning rimane un mezzo per emulare o adattare metodi appartenenti al Machine Learning classico e che sono stati realizzati su misura proprio per la computazione classica ma si può affermare con relativa certezza che se qualcuno può trovare un algoritmo quantistico efficiente, cioè con risorse che crescono polinomialmente in proporzione al numero n di qubit, allora è possibile manipolare le 2^n

ampiezze di un sistema quantistico in maniera 'super efficiente', ossia con risorse che aumentano in ragione del logaritmo $O(\log N)$ nelle dimensioni di uno spazio di Hilbert. Esistono pubblicazioni, per lo più teoriche, dove tutto questo è dimostrato e spesso ci si riferisce a questo metodo di codifica con il nome di Quantum Random Access Memory (QRAM), la quale carica parallelamente i vettori che caratterizzano le istanze in un registro di qubit e effettua una rotazione condizionata misurando il valore di un qubit definito *ancilla* per scrivere il valore effettivo nell'ampiezza.
Rimanendo nell'ambito di algoritmi di Quantum Machine Learning, proseguiremo con una possibile implementazione dell'algoritmo Quantum k-nearest neighbour, richiamando i concetti teorici necessari e cercando di rendere il codice il più semplice possibile dal punto di vista interpretativo; infine si passerà ad un ultimo esempio più complesso che descrive l'implementazione di un algoritmo per la risoluzione di sistemi lineari. In entrambi i casi verranno mostrati risultati e conclusioni dovute all'esecuzione.

6.3.1 Classificazione con Quantum k-nearest neighbour

Partiamo dal più semplice circuito quantistico implementabile mostrando come possa essere utilizzato come modello per la classificazione di istanze. L'idea di base è utilizzare l'interferenza quantistica tra differenti stati per valutare la misura di distanza e sfruttando il parallelismo quantistico effettuare la classificazione con una sola operazione.
Se conosciamo una efficiente maniera per la preparazione iniziale degli stati, l'algoritmo che utilizzeremo raggiunge lo stesso livello logaritmico di complessità in funzione del numero e della dimensione dei dati raggiunto in ugual misura da altri autori [9] che può, sempre in linea teorica, essere implementato anche su uno dei quantum computer reali di IBM descritti nei paragrafi precedenti.
La seguente routine sarà quindi suddivisa in due step: la prima dedicata alla preparazione iniziale degli stati del sistema, e la seconda di valutazione della distanza. Per la fase fondamentale di preparazione degli stati del sistema, dove vengono configurate le ampiezze dei singoli stati quantistici, si è scelto di utilizzare la funzione *initialize* di Qiskit, che permette di settare un numero complesso normalizzato su ogni stato del sistema. Questo permette una rappresentazione del codice che si distacca maggiormente dal livello fisico e fornisce una visione molto più intuitiva del funzionamento dell'algoritmo. Per la fase di

calcolo della distanza il circuito solo consiste di un Hadamard gate applicato sul primo qubit che permette di calcolare la distanza tra i dati sfruttando il parallelismo quantistico. Infine sono ovviamente applicati gli operatori di misurazione per ottenere il risultato dell'algoritmo.

Formalizziamo dunque il problema: dato un insieme di istanze che definiscono il training set $D = \{(x^1, y^1), \dots, (x^M, y^M)\}$ di input $x^m \in \mathbb{R}^N$ con la loro rispettiva classe $y^m \in \{0,1\}$, e un nuovo input del quale non conosciamo la classe di appartenenza $\tilde{x} \in \mathbb{R}^N$, trovare la classe $\tilde{y} \in \{0,1\}$ corrispondente al nuovo input.

Considerando che le nostre istanze saranno vettori bidimensionali con valori normalizzati all'interno di una sfera unitaria, possiamo proseguire con la parte di codice inerente la prima fase dell'algoritmo:

```
# Define registers and circuit
n = 4
qr = QuantumRegister(n, "qr")
cr = ClassicalRegister(n, 'cr')
circuit = QuantumCircuit(qr, cr, name="Qk-nn")

# Define all input vectors you want to classify.
# You can also change training vectors but keep
# respecting normalization

x_in = [-0.948, 0.318]
#x_in = [-0.549, 0.836]  # x' in publication
#x_in = [0.053, 0.999]  # x'' in publication

# Training Vectors
xt_1 = [0.000, -1.000] # class 0
xt_2 = [-0.789, -0.615] # class 1

# Normalization Factor based on training and input vectors
M = 2*x_in[0]**2 + 2*x_in[1]**2 + xt_1[0]**2 + xt_1[1]**2 +
    xt_2[0]**2 + xt_2[1]**2
```

```
# Amplitude value for each state
desired_vector = [
    1 / math.sqrt(M) * complex(x_in[0], 0),   # 0000
    1 / math.sqrt(M) * complex(xt_1[0], 0),   # 0001
    0,                                        # 0010
    0,                                        # 0011
    1 / math.sqrt(M) * complex(x_in[1], 0),   # 0100
    1 / math.sqrt(M) * complex(xt_1[1], 0),   # 0101
    0,                                        # 0110
    0,                                        # 0111
    0,                                        # 1000
    0,                                        # 1001
    1 / math.sqrt(M) * complex(x_in[0], 0),   # 1010
    1 / math.sqrt(M) * complex(xt_2[0], 0),   # 1011
    0,                                        # 1100
    0,                                        # 1101
    1 / math.sqrt(M) * complex(x_in[1], 0),   # 1110
    1 / math.sqrt(M) * complex(xt_2[1], 0),   # 1111
]

# Initialize Amplitudes into the system
circuit.initialize(desired_vector,
                   [qr[0], qr[1], qr[2], qr[3]])
circuit.barrier()
```

Per testare effettivamente il classificatore è essenziale una fase di standardizzazione dei dati in modo da avere varianza unitaria e media nulla. Questo è un metodo spesso utilizzato in Machine Learning in quanto permette di non risentire degli effetti di scala. Altro punto importante è normalizzare ogni feature del vettore all'unitarietà. Questa metodologia può essere di gran aiuto, in quanto permette di considerare esclusivamente l'angolo tra due punti. Questi step strategici di preparazione dei dati permettono di raggiungere a pieno regime la condizione di super efficienza dell'algoritmo.

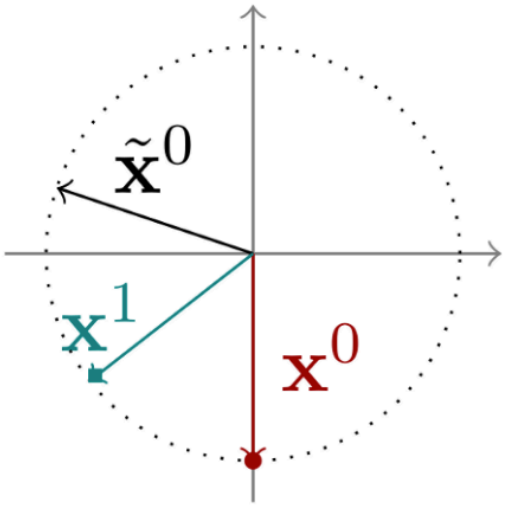


Figura 26-Rappresentazione nella sfera unitaria dei dati di training e del vettore che si vuole classificare

Con i valori dei dati normalizzati correttamente iniziamo la fase di configurazione delle ampiezze che è di fondamentale importanza e deve rispettare diverse regole. In primo luogo consideriamo il primo qubit (il meno importante) un *ancilla* qubit che ci servirà come appoggio in fase di misurazione e decidiamo di settare le informazioni del vettore di input sempre sull'ancilla a 0. I valori dei vettori del training devono invece sempre trovarsi con ancilla uguale a 1. Il quarto bit (il più significativo) è invece dedicato a codificare la classe, dovremmo assegnare correttamente i vettori di training alla corretta classe codificata di appartenenza, mentre come si può osservare nel codice, il vettore di input andrà inserito a prescindere dalla classe, ma in comparazione con ogni valore di training, rispettando la corrispondenza del secondo e terzo qubit, che possiamo pensare come una forma di codifica per gli indici delle feature da mettere in comparazione.

Ogni valore è normalizzato in modo che la sommatoria delle ampiezze al quadrato degli stati del sistema sia sempre uguale ad uno e la configurazione del codice permette di testare liberamente diversi vettori scegliendo i valori al principio.

L'idea di codificare le caratteristiche degli input e del training set nelle ampiezze di un sistema quantistico manipolandole successivamente tramite gate è la strategia responsabile nella maggior parte dei casi dell'ottenimento di uno speed-up esponenziale nel modello quantistico dell'algoritmo. Possiamo riferirci a questa trasformazione con il termine 'codifica delle ampiezze' per differenziarla dalla classica codifica tra bit classico e qubit. I vettori in forma classica $x \in \mathbb{R}^N$, dove $N = 2^n$ è la dimensione dello spazio vettoriale, che godono inoltre della proprietà $x^T x = 1$ e i cui valori possono essere scritti come $x =$

$(x_1, \dots, x_N)^T$ vengono quindi codificati con le 2^n ampiezze che descrivono lo stato di un sistema a n-qubit tale ottenendo:

$$|\psi\rangle = \sum_{i=0}^{N-1} x_i |i\rangle,$$

dove $|i\rangle$ rappresenta un registro di indice che associa il valore i-esimo del vettore classico con l'i-esimo stato della base computazionale.
Se il metodo utilizzato per la preparazione degli stati è sufficientemente preciso, il circuito quantistico di classificazione avrà un sistema quantistico di *n* qubit nella forma:

$$|\psi\rangle = \frac{1}{\sqrt{2MC}} \sum_{m=1}^{M} |m\rangle \left(|0\rangle|\psi_{\tilde{x}}\rangle + |1\rangle|\psi_{x^m}\rangle\right)|y^m\rangle,$$

dove l'equazione del sistema quantistico descrive una generalizzazione della preparazione degli stati effettuata e descritta in precedenza. Si puntualizza che in caso i dati vengano utilizzati in forma normalizzata, come nel nostro caso, è possibile settare *C* con valore 1 poiché tale variabile dipende esclusivamente dal preprocessamento dei dati.

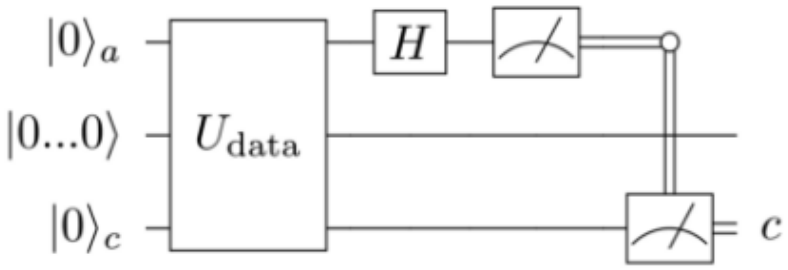


Figura 27-Semplificazione del circuito per realizzare la classificazione

Completata e descritta la prima fase di preparazione degli stati, il circuito necessita solo dell'applicazione di una porta Hadamard sul qubit ancilla per la realizzazione della fase di calcolo della distanza. Grazie a questo metodo si effettua l'interferenza tra le ampiezze del nuovo input e dei dati di training in modo da ottenere il seguente stato:

$$|\psi\rangle = \frac{1}{2\sqrt{M}} \sum_{m=1}^{M} |m\rangle \left(|0\rangle|\psi_{\tilde{x}+x^m}\rangle + |1\rangle|\psi_{\tilde{x}-x^m}\rangle\right)|y^m\rangle.$$

Il codice per la realizzazione di questo step con l'applicazione dell'Hadamard è:

```
# Hadamard on ancilla qubit
circuit.h(qr[0])
circuit.measure(qr, cr)
print(circuit)
```

A questo punto, avendo già aggiunto la misurazione sul circuito, dobbiamo pensare a come valuteremo la misurazione condizionata degli stati aventi il qubit ancilla $|0\rangle$, che si riferiscono quindi al vettore di input e che avviene con probabilità $p_{acc} = \frac{1}{4M}|\tilde{x} + x^m|^2$. Nel caso la misurazione abbia esito positivo il risultato sarà dato da:

$$|\psi\rangle = \frac{1}{2\sqrt{Mpacc}} \sum_{m=1}^{M} \sum_{i=1}^{N} |m\rangle(\tilde{x}_i + x_i^m)\,|i\rangle\,|y^m\rangle.$$

Le ampiezze pesano il valore del qubit classe $|y^m\rangle$ attraverso la distanza dell'm-esimo punto con il nuovo input e dovremmo valutarne i risultati sulla base delle differenti classi misurate ma con ancilla in stato $|0\rangle$.
A causa delle limitazioni implementative verificate con i test di altri algoritmi e del numero limitato di gate supportati dai quantum real device di IBM si è scelto di eseguire in prevalenza prove con backend Qasm Simulator. Inoltre come osservabile nel codice iniziale, sono utilizzate due feature per ogni istanze e per la realizzazione del training set ne saranno utilizzate soltanto due.
Il training set sarà dunque $D = \{(x^0, y^0), (x^1, y^1)\}$, dove i due vettori saranno:

- $x^0 = (0, -1)$ con $y^0 = 0$
- $x^1 = (-0.789, -0.615)$ con , $y^1 = 1$

Come vettore per la sperimentazione ne selezioneremo uno iniziale, ma possono essere condotte ulteriori prove con vettori normalizzati adeguatamente:

- $\tilde{x}' = (-0.948,\ 0.318)$ con $\tilde{y}' = 1$

Questo semplice caso può essere quindi gestito con l'utilizzo di quattro qubit, un'ancilla, uno per la rappresentazione della classe, uno per indicizzare i due training data e uno per rappresentare i due valori di ogni training e input data. Preparati i dati ed il circuito con gli step descritti fino ad ora eseguiamo l'algoritmo:

```
# QASM Simulation
backend = BasicAer.get_backend('qasm_simulator')
job = execute(circuit, backend, shots=1024)
result = job.result()
print(result.get_counts(circuit))
```

Dalla stampa dei risultati:

```
{'0100': 239, '1110': 98, '0000': 114, '1010': 3,
 '1011': 386, '0001': 100, '0101': 73, '1111': 11}
```

dobbiamo prima di tutto prendere tutti quelli che risultano con il qubit ancilla sullo stato 0, creando un primo sottoinsieme dei risultati; per farlo:

```
counts = result.get_counts()
total_samples = sum(counts.values())

# Define lambda function that retrieves only results where
# the ancilla is in |0> state

post_select = lambda sel_counts: [(state, occurences)
      for state, occurences in sel_counts.items()
            if state[-1] == '0']

# Create the new subselection of state
postselection = dict(post_select(counts))
```

```
sub_samples = sum(postselection.values())
print(f'Number of states with Ancilla qubitbit in|0> :
                                  { sub_samples}')
print(postselection)
print(f'Ancilla post-selection probability was
       found to be {postselected_samples / total_samples}')
```

Dobbiamo infine suddividere i risultati in base al quarto qubit che ci rappresenta la classe, quindi creiamo una ulteriore funzione lambda che separi il sottoinsieme di risultati in ulteriori due dipendenti dalla classe di appartenenza. Una volta ricavati, è sufficiente calcolare la probabilità di ottenere il risultato di una delle due classi sul numero totale di esempi con ancilla 0, e in base alla probabilità maggiore decidere la classificazione:

```
# Create subsets of element based on class qubit
retrieve_class = lambda binary_class: [occurences for state,
      occurences in postselection.items() if state[0] ==
            str(binary_class)]

# Printing number of occurences for each class
print(f'Occurences of Class 0 :  {sum(retrieve_class(0))}')
print(f'Occurences of Class 1 :  {sum(retrieve_class(1))}')

# Probability of classification
prob_class0 = sum(retrieve_class(0)) / postselected_samples
prob_class1 = sum(retrieve_class(1)) / postselected_samples
print(f'Probability for class 0 is {prob_class0}')
print(f'Probability for class 1 is {prob_class1}')

# Print the final result based on the probability
# obtained for each class 0 and 1
if prob_class0 > prob_class1:
    print(f"Classifying input x as class 0 with
                                        QASM Simulator")
elif prob_class0 < prob_class1:
    print(f"Classifying input x as class 1 with
```

```
                                        QASM Simulator")
else:
    print(f"Inconclusive. 50/50 results")
```

Una prova dell'algoritmo dimostra la corretta classificazione, ecco un possibile risultato ottenuto con i vettori definiti sopra:

```
{'0100': 408, '0001': 31, '1011': 238, '1110': 281,
'1111':8, '0000': 48, '0101': 5, '1010': 5}

Number of state with Ancilla bit |0> : 742
{'0100': 408, '1110': 281, '0000': 48, '1010': 5}


Ancilla post-selection probability was
                          found to be 0.724609375
Occurences of Class 0 :  194
Occurences of Class 1 :  385
Probability for class 0 is 0.33506044905008636
Probability for class 1 is 0.6649395509499136

Classifying input x as class 1 with noisy simulator backend
```

Possiamo affermare che la simulazione ha avuto successo in tutti i casi. Ovviamente il limite più grande dipende dal fatto di non poter effettuare un test significativo su quantum real device data la complessità del circuito dovuta alla codifica delle ampiezze. I troppi gate necessari rappresentano infatti ancora oggi un duro ostacolo alla fase di test reali degli algoritmi quantistici di Quantum Machine Learning.
Tutto il codice, l'esempio del circuito e la stampa di questo esempio sono riportati in Appendice VI.

6.3.2 Risoluzione di Problemi Lineari con HHL

Passiamo a questo punto alla descrizione della possibile implementazione dell'algoritmo HHL per la risoluzione di sistemi lineari. Come descritto nel relativo capitolo teorico, questo genere di problemi è alla base di tantissimi problemi matematici e ingegneristici ed ha una importanza sostanziale anche nell'ambito del Machine Learning.

Preso un problema $A\vec{x} = \vec{b}$, dove A è la matrice di ordine n dei coefficienti conosciuti, $\vec{b}$ è un vettore che contiene i termini noti delle equazioni anch'essi conosciuti a priori, l'obiettivo è trovare il vettore $\vec{x}$, non noto, che soddisfa l'equivalenza.

Il problema ha una grossa dipendenza dalla dimensione del vettore $\vec{b}$, quindi dal numero di variabili N che l'algoritmo deve trovare e che nei problemi reali risulta essere molto grande. Gli autori dell'algoritmo, Harrow, Hassidim e Lloyd in [19] descrivono approfonditamente gli step necessari allo sviluppo del circuito che possono essere suddivisi in tre parti del circuito:

- Stima delle Fasi;
- Rotazioni controllate su un qubit ancilla;
- Inversione della Stima delle Fasi (Uncompute);

e può essere rappresentato in forma circuitale come:

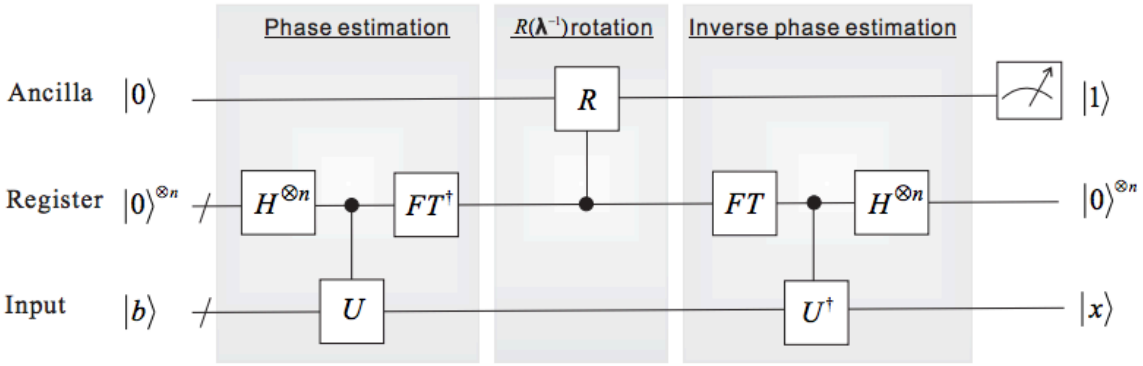


Figura 28-Circuito per la realizzazione dell'algoritmo HHL

Il primo qubit rappresenta un ancilla sul quale verranno effettuate rotazioni controllate e che servirà in fase di misurazione per la selezione dei risultati, il secondo registro è dedicato a salvare la codifica degli autovalori della matrice A dopo l'operazione di Stima delle Fasi

e verrà identificato con *C*, la sua dimensione dipende quindi da quella di *A* e da quella degli autovalori; infine il terzo registro è utilizzato per salvare i valori del vettore $\vec{b}$ e alla fine di tutto il processo dell'HHL, immagazzinerà l'approssimazione ottenuta di $\vec{x}$.

Per effettuare l'implementazione, prendiamo una matrice Hermitiana *A* 2x2 definita da:

$$A = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix},$$

e un vettore dei termini noti $\vec{b} = [0 \quad 1]$, in modo che sia facilmente rappresentabile come stato quantistico in quanto altro non è che lo stato $|1\rangle = [0 \quad 1]$. Possiamo calcolare a mano le soluzioni del sistema e gli autovalori della matrice *A*.
Le soluzioni saranno date dal sistema che risolve:

$$\begin{cases} 2x_1 - x_2 = 0 \\ -x_1 + 2x_2 = 1 \end{cases} \Longrightarrow \begin{cases} x_2 = 2x_1 \\ 3x_1 = 1 \end{cases} \Longrightarrow \begin{cases} x_1 = {}^1\!/_3 \\ x_2 = {}^2\!/_3 \end{cases}.$$

Gli autovalori della matrice *A* si calcolano invece con la seguente equivalenza:

$$det(-\Lambda I) = 0,$$

$$det\begin{bmatrix} 2-\lambda & -1 \\ -1 & 2-\lambda \end{bmatrix} = 0$$

$$(2-\lambda)^2 - 1 = 0 \Longrightarrow \lambda^2 - 4\lambda + 3 = 0 \begin{cases} \lambda_1 = 3 \\ \lambda_2 = 1 \end{cases}$$

Dati alla mano iniziamo a implementare l'algoritmo per valutare effettivamente i risultati attesi. Per la realizzazione del circuito sono sufficienti quattro qubit: uno per l'ancilla, uno per salvare $\vec{b}$ che come abbiamo detto è lo stato quantistico $|1\rangle$ e due qubit per immagazzinare l'informazione degli autovalori, infatti per rappresentare in bit 1 e 3 le rappresentazioni sono rispettivamente 0.01 e 0.11. Definiamo quindi un circuito a quattro qubit e quattro registri classici per la misurazione:

```
# Define registers and circuit
```

```
qr = QuantumRegister(4, name="qr")
cr = ClassicalRegister(4, name="cr")
circuit = QuantumCircuit(qr, cr, name="HLL_2x2")
```

Dalla teoria il primo spasso è quello di codificare il vettore $\vec{b}$ nello stato quantistico di $|b\rangle$ che avendolo definito come $|1\rangle$ sarà sufficiente un gate X sul terzo qubit:

```
# |b> =(0 1)
circuit.x(qr[3])
```

Iniziamo con applicare il primo step di Stima delle fasi con lo scopo di determinare gli autovalori della matrice *A* ed ottenere una rappresentazione del sistema del tipo:

$$|\psi\rangle = \sum_{j=1}^{N} \beta_j |u_j\rangle |\lambda_j\rangle.$$

Dove $|u_j\rangle$ sono gli autovettori della matrice *A* e $|\lambda_j\rangle$ i suoi autovalori espressi in forma binaria. Per farlo dobbiamo mettere in sovrapposizione di stati i due qubit che corrispondono al registro *C* e come da protocollo applichiamo l'evoluzione Hamiltoniana definita da:

$$\sum_{k=0}^{N-1} e^{iAk\frac{t_0}{N}}$$

con $t_0 = 2\pi$ e $N = 2^n$ dove *n* è il numero di qubit su cui si applica l'Hamiltoniana. Si applica infine l'inversa della Trasformata di Fourier sempre sul registro *C*
Il codice che lo esegue è:

```
# Phase Estimation and c-rotation to store the eigenvalue
circuit.h(qr[1])
circuit.h(qr[2])
circuit.barrier()
```

```
circuit.u1(pi, qr[1])
circuit.u1(pi/2, qr[2])
circuit.cx(qr[2], qr[3])
circuit.barrier()

# Inverse QFT
circuit.h(qr[1])
circuit.cu1(-pi / 2, qr[1], qr[2])
circuit.h(qr[2])
circuit.barrier()
```

A questo punto lo stato del sistema diventerà $\beta_1|01\rangle|u_1\rangle + \beta_2|11\rangle|u_2\rangle$, dove β_1 e β_2 sono coefficienti di $|b\rangle$ espressi in base agli autovalori di A. Sperimentalmente, il risultato è corretto, infatti valutando le amplitudes dopo la prima fase osserviamo il seguente risultato:

```
[ 0. +0.j  0. +0.j  0.5+0.j  0. +0.j  0. +0.j  0. +0.j
 -0.5+0.j  0. +0.j  0. +0.j  0. +0.j  0.5+0.j  0. +0.j
  0. +0.j  0. +0.j  0.5+0.j  0. +0.j]
```

Continuiamo con l'applicazione delle rotazioni controllate sull'ancilla qubit, questo processo serve a salvare nell'ampiezza dell'ancilla i valori $1/\lambda_j$. Avendo l'ancilla inizialmente settato a zero lo stato del sistema che otteniamo è:

$$|\psi\rangle = \sum_{j=1}^{N} \beta_j |\lambda_j\rangle|u_j\rangle \left(\sqrt{1 - \frac{C^2}{{\lambda_j}^2}} |0\rangle_{ancilla} + \frac{C}{\lambda_j} |1\rangle_{ancilla} \right).$$

Nel codice le rotazioni sull'ancilla sono definite come:

```
# (lamda^-1)-Rotation on ancilla qubit
circuit.x(qr[2])
circuit.cu3(pi / 16, 0, 0, qr[1], qr[0])
circuit.cu3(pi / 8, 0, 0, qr[2], qr[0])
circuit.barrier()
```

L'ultimo step viene chiamato Uncompute, e prevede l'applicazione dell'inversa delle operazioni effettuate prima delle rotazioni sull'ancilla. Abbiamo quindi una Trasformata di Fourier e le operazioni di rotazione sul registro *C* cambiate di segno e permettendo di rompere gli entanglement creati e tornare allo stato iniziale lasciando invariata la modifica sull'ancilla e ottenendo uno stato nella forma:

$$|\psi\rangle = \sum_{j=1}^{N} \beta_j |u_j\rangle \left(\sqrt{1 - \frac{C^2}{{\lambda_j}^2}} |0\rangle_{ancilla} + \frac{C}{\lambda_j} |1\rangle_{ancilla} \right).$$

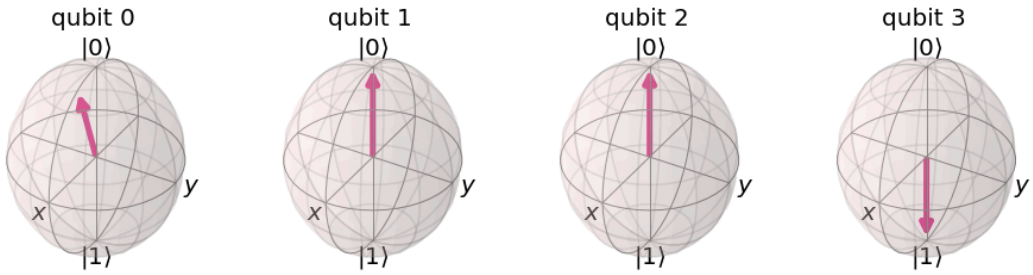
Il codice della terza fase definia Uncomputa è:

```
# Uncompute
circuit.x(qr[2])
circuit.h(qr[2])
circuit.cu1(pi / 2, qr[1], qr[2])
circuit.h(qr[1])
circuit.cx(qr[2], qr[3])
circuit.u1(-pi/2, qr[2])
circuit.u1(-pi, qr[1])
circuit.barrier()
circuit.h(qr[2])
circuit.h(qr[1])
circuit.barrier()
```

A questo punto eseguiamo il circuito con backend Statevector per visualizzare i valori delle amplitudes degli stati:

```
[-0.01912+0.j  0.09613+0.j  0.     +0.j  0.     +0.j
  0.     +0.j  0.     +0.j  0.     +0.j  0.     +0.j
  0.97606+0.j  0.19415+0.j  0.     +0.j  0.     +0.j
  0.     +0.j  0.     +0.j  0.     +0.j  0.     +0.j]
```

Inoltre una rappresentazione grafica dei singoli qubit mostra esattamente come sia verificata ciò che abbiamo detto. Tutti i qubit si trovano nello stato di partenza tranne il più a sinistra che rappresenta l'ancilla e che ha subito una variazione dovuta alle rotazioni.



Effettuare la misurazione a questo punto non è scontato. La prima cosa da fare è ovviamente misurare l'intero circuito per poi effettuare una seconda selezione sugli stati con ancilla uguale a 1 che rappresentano uno stato proporzionale a $|x\rangle$:

$$\sum_{j=1}^{N} C\left(\frac{\beta_j}{\lambda_j}\right)|u_j\rangle \approx |x\rangle$$

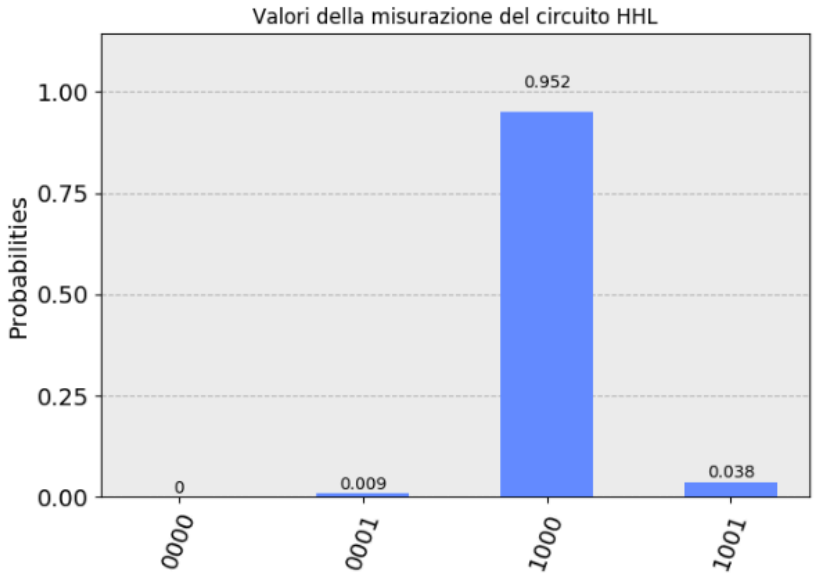
Lanciamo l'esecuzione su backend QASM ripetendola diecimila volte, in modo da ottenere una versione più approssimata possibile delle soluzioni:

```
# QASM Simulation
circuit.measure(qr, cr)
job = execute(circuit, backend=backend_qasm, shots=10000)
results = job.result()
answer = results.get_counts()
n_0 = answer['0001']
n_1 = answer['1001']
print('Occurrences in 0001: %d' % n_0)
print('Occurrences in 1001: %d' % n_1)
p_scale = float(n_0 / n_1)
print('Ratio: %d' % p_scale)
# Plot Results
plt = plot_histogram(answer)
```

```
plt.show()
```

L'output e il rispettivo istogramma che otteniamo sono:

```
Occurrences in 0001: 92
Occurrences in 1001: 384
Ratio: 0.239583
```



Dall'istogramma delle misurazioni dobbiamo escludere apriori gli stati in cui l'ancilla qubit è misurato uguale a 0. La probabilità di occorrenze dei due stati con ancilla a 1, ossia 0001 e 1001, approssimata eseguendo il circuito tantissime volte, è proporzionale ai risultati del vettore $\vec{x}$. Sperimentalmente possiamo verificare che il *ratio* dato dal rapporto al quadrato del numero di misurazioni dei due stati con ancilla a 1 è approssimativamente:

$$\left(\frac{P_{0001}}{P_{1001}}\right)^2 = \left(\frac{x_1}{x_2}\right)^2.$$

Verificando manualmente l'equivalenza otteniamo i seguenti risultati sperimentali, messi in comparazione con quelli teorizzati. L'esecuzione del circuito è stata ripetuta più volte su valori differenti del vettore dei termini noti $\vec{b}$, sfruttando quelli facilmente configurabili nella fase iniziale del circuito:

$\|b\rangle$	$\vec{x}_t$	$\vec{x}_s$	*Occorrenze*	*Ratio*
$\|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$	$\begin{pmatrix}0.33333\\0.66667\end{pmatrix}$	$\begin{pmatrix}0.32862\\0.67138\end{pmatrix}$	$0001 = 92$ $1001 = 384$	0.23958
$\|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$	$\begin{pmatrix}0.66667\\0.33333\end{pmatrix}$	$\begin{pmatrix}0.66274\\0.33726\end{pmatrix}$	$0001 = 390$ $1001 = 101$	3.861386
$\|+\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix}$	$\begin{pmatrix}0.70711\\0.70711\end{pmatrix}$	$\begin{pmatrix}0.72297\\0.69124\end{pmatrix}$	$0001 = 431$ $1001 = 394$	1.093909

Tabella 2-Riassunto dei risultati sperimentali dell'algoritmo HHL con due incognite

Per la visualizzazione dei differenti istogrammi inerenti ad ogni esecuzione con differente $|b\rangle$ si faccia riferimento all'Appendice VII.III.

Possiamo osservare che il *ratio* che otteniamo è relazionato con i valori delle *x* e ci permette in questo caso di valutare l'efficienza dell'approssimazione ricalcolando le soluzioni:

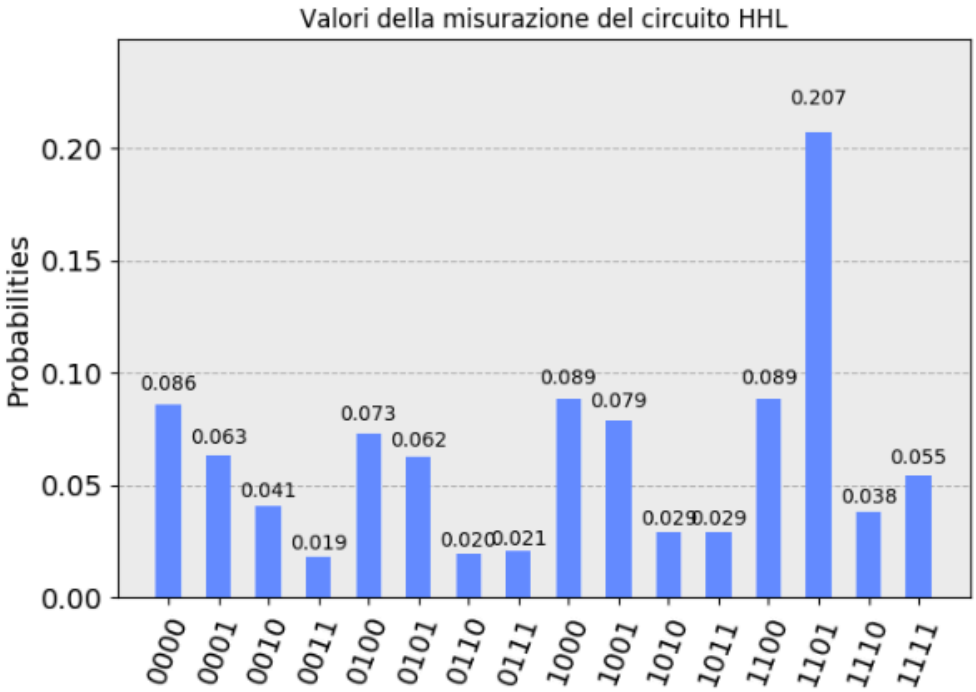
$$ratio^2 = \left(\frac{x_1}{x_2}\right)^2$$

$$\begin{cases} ratio * x_2 = x_1 \\ x_1 + x_2 = \sum_j \beta_j, \end{cases}$$

I risultati ottenuti dimostrano l'efficacia dell'algoritmo nel problema appena descritto. In particolare si noti l'importanza dell'efficienza nella fase di preparazione dei dati, spiegata approfonditamente in [20], passo fondamentale di ogni problema computazionale che può rallentare drammaticamente la velocità dell'algoritmo; ovviamente in questo caso abbiamo semplificato il più possibile la rappresentazione di |b⟩ utilizzando un solo qubit e eseguendo tra diversi test con diversi basi standard e ottenibili con un solo gate, anche per la matrice A si è utilizzata una rappresentazione minimale, con il registro C composto da due qubit per permettere la rappresentazione binaria dei primi tre numeri naturali, cosa fondamentale per la codifica degli autovalori.

Proviamo ora una esecuzione dell'algoritmo su quantum real device per valutare la coerenza della soluzione confronto a quella su QASM. Per farlo eseguiamo lo stesso codice utilizzando gli strumenti per accedere all'enviroment IBM descritti in precedenza. I risultati delle occorrenze degli stati e del ratio sono:

```
Occurrences in 0001: 65
Occurrences in 1001: 81
Ratio: 0.802469
```

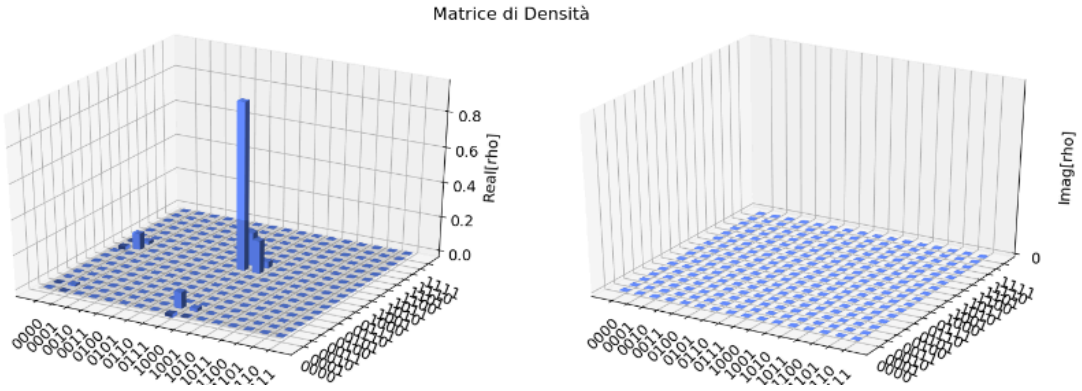


La differenza è evidente e i risultati ottenuti su real device sono chiaramente incorretti; la causa principale è di certo la dimensione del circuito e della conseguente decoerenza degli stati. Per la visualizzazione del circuito si rimanda all'Appendice VII.II.

Descriviamo infine un ultimo strumento applicato a questo problema disponibile nella libreria Qiskit. Come sappiamo la misurazione quantistica porta ad un collasso del sistema in uno degli *N* stati possibili, facendo perdere qualsiasi informazione sui veri valori degli stati in cui si trovano i qubit e le possibili sovrapposizioni. Lo stato a cui giungiamo un attimo prima della misurazione può essere descritto da una matrice di densità, rappresentabile nel nostro caso con la simulazione Statevector di Qiskit. Effettuandola sullo stato del sistema otteniamo la seguente rappresentazione:

```
# Statevector Simulation
backend_sv = BasicAer.get_backend('statevector_simulator')
job_sv = execute(circuit, backend_sv)
```

```
result_sv = job_sv.result()
outputstate_psi = result_sv.get_statevector(circuit, decimal
s=5)
Construct the density matrix from the state vector
outputstate_rho = outer(outputstate_psi) #
plt = plot_state_city(outputstate_rho)
plt.show()
```

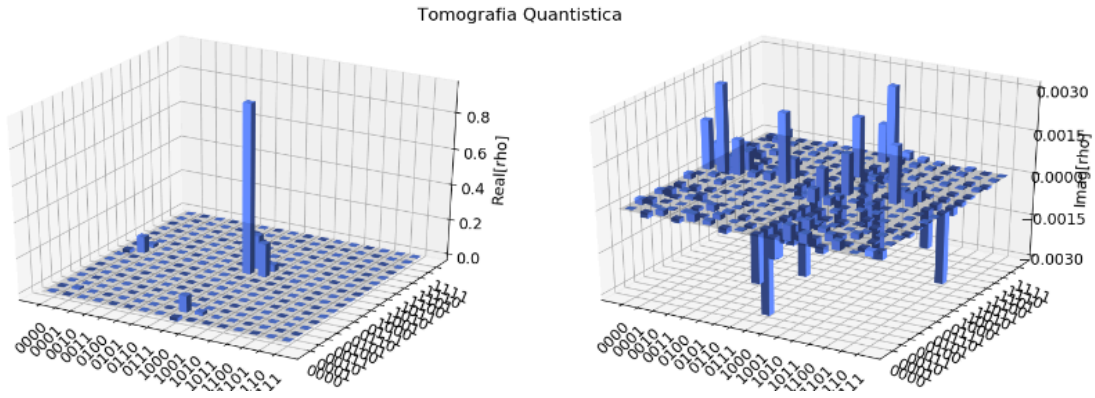


Questa matrice di densità può essere ricostruita utilizzando il modulo Tomography di Qiskit. In sostanza la tomografia quantistica degli stati è il processo per ricostruire uno stato quantistico o la sua matrice di densità di un determinato sistema attraverso una serie di misurazioni concrete che ricoprano interamente lo spazio di Hilbert del sistema, generalmente esplorando lo spazio definito da X, Y, Z. Lo scopo di questa ricostruzione è appunto quello di ottenere informazioni del sistema indipendentemente dalla misurazione, e il suo utilizzo trova applicazioni in molti problemi come ad esempio la ricostruzione di segnali ottici.
Per implementarla utilizziamo il codice:

```
# Construct state tomography set for measurement of qubits
bell_tomo_set = tomo.state_tomography_set([0, 1, 2, 3])
bell_tomo_circuits = tomo.create_tomography_circuits(circuit
, qr, cr, bell_tomo_set)
backend = BasicAer.get_backend('qasm_simulator')
shots = 5000
# Run the simulation
```

```
bell_tomo_job = execute(bell_tomo_circuits, backend=backend,
shots=shots)
bell_tomo_result = bell_tomo_job.result()
bell_tomo_data = tomo.tomography_data(bell_tomo_result, circ
uit.name, bell_tomo_set)
rho_fit = tomo.fit_tomography_data(bell_tomo_data)

# calculate fidelity, concurrence and purity of fitted state
F_fit = state_fidelity(rho_fit, outputstate_psi)
pur = purity(rho_fit)

print('Fidelity =', F_fit)
#print('concurrence = ', str(con))
print('purity = ', str(pur))
# Plot Results
plt = plot_state_city(outputstate_rho,
                      title='Matrice di Densità')
plt.show()
plt = plot_state_city(rho_fit,
                       title='Tomografia Quantistica')
plt.show()
```

In questo modo otteniamo la tomografia del sistema, un valore di fedeltà e uno di purezza rispetto allo stato reale:



```
Fidelity = 0.9954531929178873
purity =  0.991114820148999
```

Vediamo che la parte reale dello stato è stata ricostruita quasi perfettamente, la parte immaginaria è davvero minima e dovuta all'errore delle misurazioni. Si noti che anche nella parte reale è presente una componente di errore, che per ragioni di scala è inosservabile.
In questo caso si potrebbe utilizzare la tomografia per ottenere informazioni sui valori della soluzione del sistema, ma testandone l'applicazione, il tempo necessario all'elaborazione classica per effettuare l'intera ricostruzione è enorme in confronto alle dimensioni del problema. Questo si traduce in una totale dispersione dello speed-up teoricamente guadagnato dall'HHL. Anche in questo ultimo caso per la consultazione dell'algoritmo, del circuito e dei vari risultati ottenuti si rimanda alla sezione dell'Appendice VII.

Risultati e Conclusioni

Dopo un approfondito studio delle basi e degli strumenti che definiscono la computazione quantistica, degli algoritmi principali che definiscono la sua potenza computazionale e di un'analisi dei metodi di Machine Learning classici e di un loro possibile adattamento teorico a questo nuovo modello di computazione, abbiamo lavorato su uno dei più moderni e strutturati, per quanto ancora alle radici del suo possibile sviluppo, environment di implementazione di circuiti ed algoritmi quantistici. La fase di programmazione è risultata molto affascinante, nonostante gli ostacoli dovuti all'interpretazione di concetti astratti, per lo più teorici, legati al linguaggio della meccanica quantistica e difficili da assimilare senza una solida base di fisica e algebra lineare.

Nel complesso i risultati sono stati positivi, aver acquisito familiarità con questo nuovo modello computazionale è di per sé un ottimo traguardo, un punto di partenza solido considerando i possibili risultati e gli impatti che il quantum computing potrebbe avere nei prossimi anni, quando i real device avranno prestazioni migliori e gestiranno molta più informazione contemporaneamente. Dal punto di vista pratico, abbiamo infatti rilevato un'impossibilità nel testare algoritmi e semplici routine su un quantum computer, ottenendo risultati degni di nota soprattutto per quanto riguarda le simulazioni con backend QASM, dove si è dimostrato che i risultati sono decisamente in linea con quelli teorici attesi. La verifica e la conferma della possibilità di effettuare classificazione con algoritmi quantistici rappresenta una certezza implementativa il cui margine di crescita rimane legato quasi esclusivamente al miglioramento dell'hardware, che rimane un ambito di ricerca sulla quale vale la pena investire.

Nonostante le scarse prestazioni dei quantum real device, un altro aspetto soddisfacente di questa tesi è quello di aver avuto la possibilità di eseguire algoritmi su un vero e proprio quantum computer, verificandone il funzionamento per piccoli circuiti e confermando l'ostilità del più grande nemico attuale della computazione quantistica, la decoerenza degli stati quantistici del sistema. Quest'ultima è indubbiamente ciò che più di ogni altra cosa non ci permette di realizzare circuiti complessi, limitando il campo applicativo degli algoritmi come abbiamo dimostrato attraverso l'algoritmo di Grover, dove la coerenza dei

risultati rimane valida solamente se utilizzato un circuito a due qubit, mentre l'estensione a tre o a quattro qubit è già un grosso limite dovuto alla relativa complessità circuitale.
L'implementazione dell'HHL, nonostante i suoi requisiti per il funzionamento e il semplice esempio che abbiamo testato, è un'altra dimostrazione delle capacità che il quantum computing riserva e dei margini di miglioramento che potrà dare nell'ambito dell'informatica.
In conclusione, possiamo affermare che la Quantum Information rappresenta senza dubbi un'area della Computer Science in grande crescita, con un potenziale enorme, che continuerà ad acquisire valore soprattutto se parallelamente verranno raggiunti migliorie e risultati positivi sull'architettura delle macchine. Le teorie e le aspettative che descrivono il quantum computing come una rivoluzione informatica non sono più fantasie, ma iniziano a prendere forma in soluzioni il cui unico handicap è l'attesa di un calcolatore che ne supporti l'esecuzione. I computer classici che utilizziamo regolarmente sono indubbiamente macchine formidabili, ma sono diventate tali dopo decenni di ricerca, test, scoperte e miglioramenti che le hanno rese quello che sono; stanno indubbiamente anche raggiungendo il limite delle loro possibilità computazionali e non è un caso che proprio a questo punto siano iniziati i piccoli grandi progressi del quantum computing.
I computer quantistici esistono, li possiamo utilizzare e per questo dobbiamo continuare a studiare algoritmi quantistici che possano migliorare le performance degli attuali metodi di Machine Learning e che permettano la risoluzione di quello che oggi definiamo impossibile, poiché un'ulteriore rivoluzione informatica, l'era dei quantum computer, è solo una questione di tempo.

Appendice

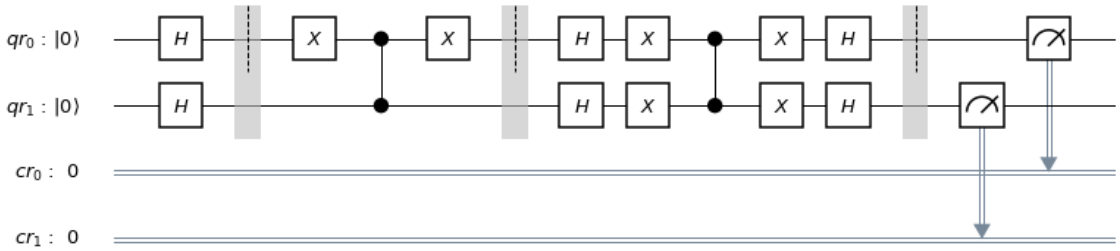
I. Algoritmo di Grover a 2 qubit con soluzione 10

```
# ******************************************************
#
# Author: Nicolò Cangini
# Project: Quantum Computation with Qiskit (IBM)
# File: Grover Algorithm on 2 qubit to find 10
# Sosftware: Python v3.7; Qiskit v0.7
# Date: March 2019
#
# ******************************************************

# Import packages
from qiskit import ClassicalRegister, QuantumRegister,
             QuantumCircuit
from qiskit import execute, BasicAer
from qiskit.tools.visualization import circuit_drawer,
                  plot_histogram, plot_bloch_multivector

# Define registers and circuit
n = 2
qr = QuantumRegister(n, "qr")
cr = ClassicalRegister(n, 'cr')
circuit = QuantumCircuit(qr, cr, name="Grover_2qubit")

# Hadamard on all qubits to create superposition
for i in range(n):
    circuit.h(qr[i])
circuit.barrier()

# Grover Operator
```

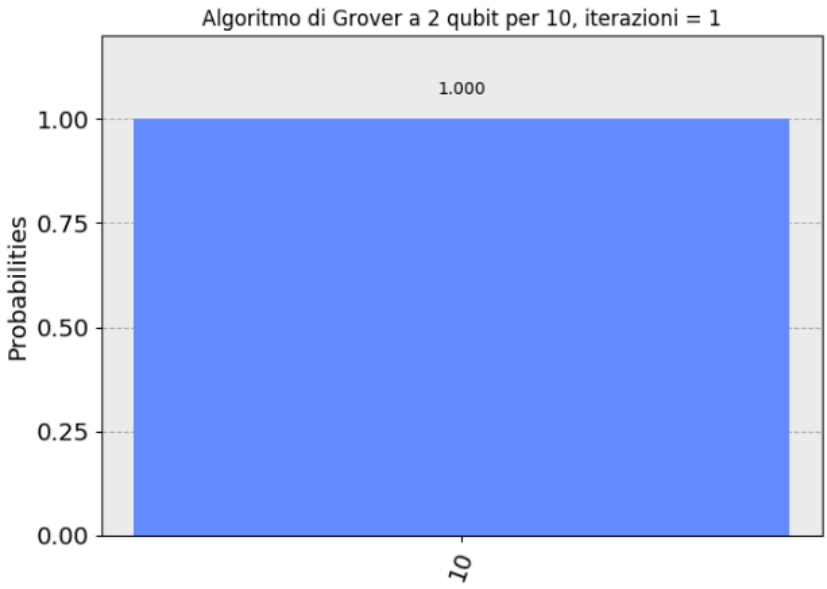
```
def grover_operator():
    # Oracle for 10
    circuit.x(qr[0])
    circuit.cz(qr[0], qr[1])
    circuit.x(qr[0])
    circuit.barrier()

    #Amplification
    circuit.h(qr[0])
    circuit.h(qr[1])
    circuit.x(qr[0])
    circuit.x(qr[1])
    circuit.cz(qr[0], qr[1])
    circuit.x(qr[0])
    circuit.x(qr[1])
    circuit.h(qr[0])
    circuit.h(qr[1])

# Number of Iteration to amplify the result
time_iteration = 1
for i in range(time_iteration):
     grover_operator()

# Measurement
circuit.barrier()
circuit.measure(qr, cr)
print(circuit)

# Statevector Simulation
backend_sv = BasicAer.get_backend('statevector_simulator')
job = execute(circuit, backend_sv)
results = job_sv.result()
outputstate = results.get_statevector(circuit, decimals=2)
print(outputstate)
```

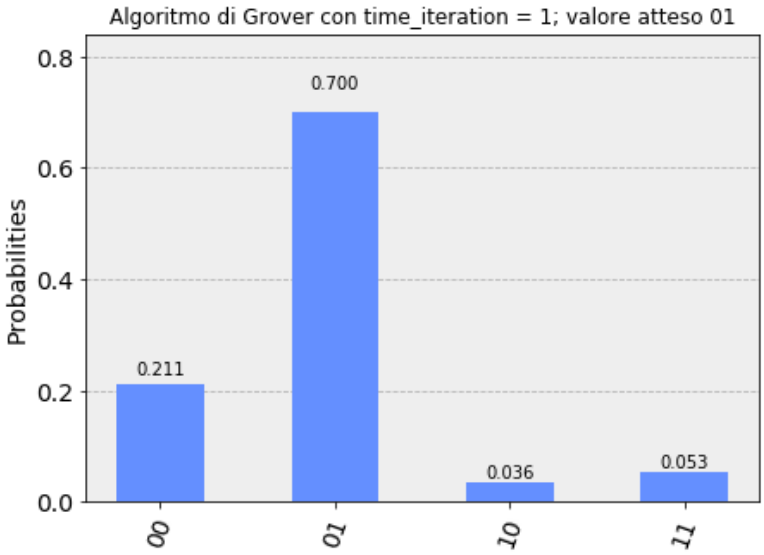
```
# QASM Simulation
backend = BasicAer.get_backend('qasm_simulator')
job = execute(circuit, backend=backend, shots=1024)
results = job.result()
answer = results.get_counts()
print(answer)

# Plot Probabilities Results
plt = plot_histogram(answer, title='Algoritmo di Grover
 a 2 qubit per 10, iterazioni = %d' % (time_iteration))
plt.show()
```

I.I. Citcuito dell'Algoritmo di Grover a 2 qubit con soluzione 10



I.II. Risultati sul simulatore e dell'esecuzione su real device





II. Algoritmo di Grover a 3 qubit con soluzione 101

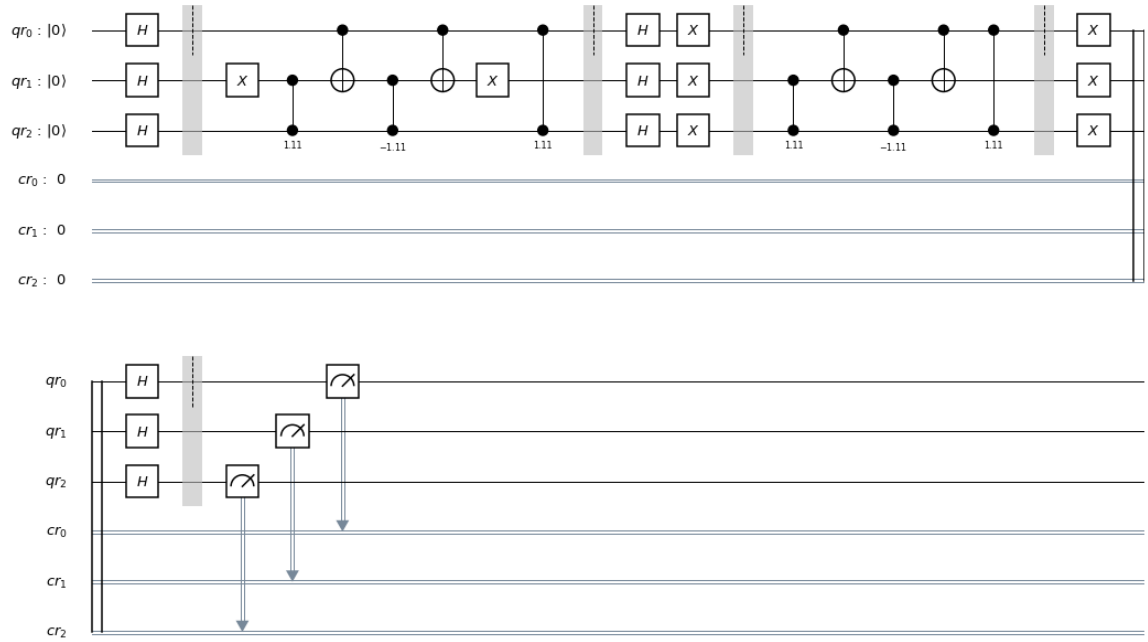
```
# ******************************************************
#
# Author: Nicolò Cangini
# Project: Quantum Computation with Qiskit (IBM)
# File: Grover Algorithm on 3 qubit to find 101
# Sosftware: Python v3.7; Qiskit v0.7
# Date: March 2019
#
# ******************************************************

# Import packages
from math import pi
import numpy as np
from qiskit import ClassicalRegister, QuantumRegister,
                                  QuantumCircuit
from qiskit import execute, BasicAer
from qiskit.tools.visualization import circuit_drawer,
                  plot_histogram, plot_bloch_multivector
```

```
# Define registers and circuit
n = 3
norm = np.sqrt(2**n)
qr = QuantumRegister(n, "qr")
cr = ClassicalRegister(n, 'cr')
circuit = QuantumCircuit(qr, cr, name="Grover_3qubit")

# Hadamard on all qubits to obtein superposition
for i in range(n):
    circuit.h(qr[i])
circuit.barrier()

# Gate ccz
def ccz():
    circuit.cu1((pi/norm), qr[1], qr[2])
    circuit.cx(qr[0], qr[1])
    circuit.cu1((-pi/norm), qr[1], qr[2])
    circuit.cx(qr[0], qr[1])
    circuit.cu1((pi/norm), qr[0], qr[2])

# Grover Operator
def grover_operator():
    # Oracle for 101
    circuit.x(qr[1])
    ccz()
    circuit.x(qr[1])
    circuit.barrier()

    # Amplification
    circuit.h(qr[0])
    circuit.h(qr[1])
    circuit.h(qr[2])
    circuit.x(qr[0])
    circuit.x(qr[1])
    circuit.x(qr[2])
```

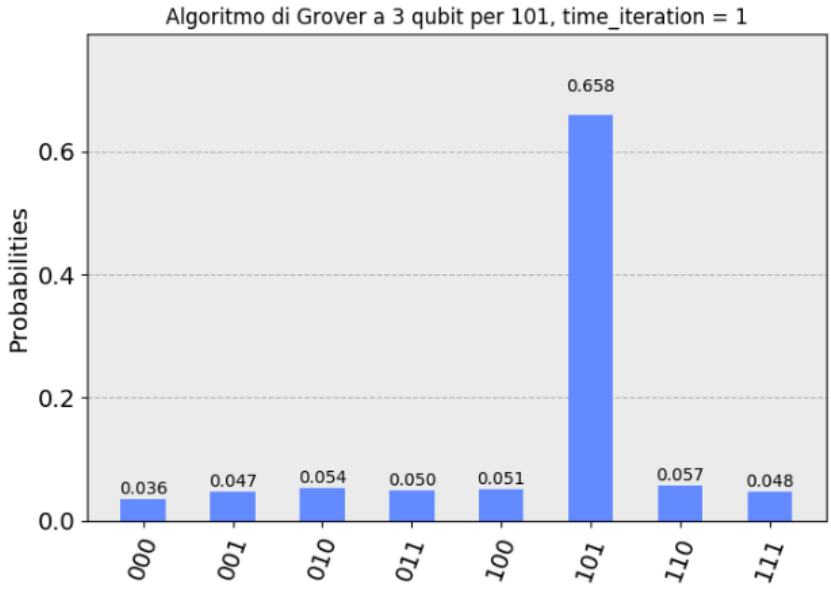
```
    circuit.barrier()
    ccz()
    circuit.barrier()
    circuit.x(qr[0])
    circuit.x(qr[1])
    circuit.x(qr[2])
    circuit.h(qr[0])
    circuit.h(qr[1])
    circuit.h(qr[2])

# Number of Iteration to amplify the result
time_iteration = 2
for i in range(time_iteration):
    grover_operator()

# Measurement
circuit.barrier()
circuit.measure(qr, cr)
print(circuit)

# Statevector Simulation
backend_sv = BasicAer.get_backend('statevector_simulator')
job = execute(circuit, backend_sv)
results = job.result()
outputstate = results.get_statevector(circuit, decimals=2)
print(outputstate)

# QASM Simulation
backend = BasicAer.get_backend('qasm_simulator')
job = execute(circuit, backend=backend, shots=1024)
results = job.result()
answer = results.get_counts()
print(answer)

# Plot Probabilities Results
```

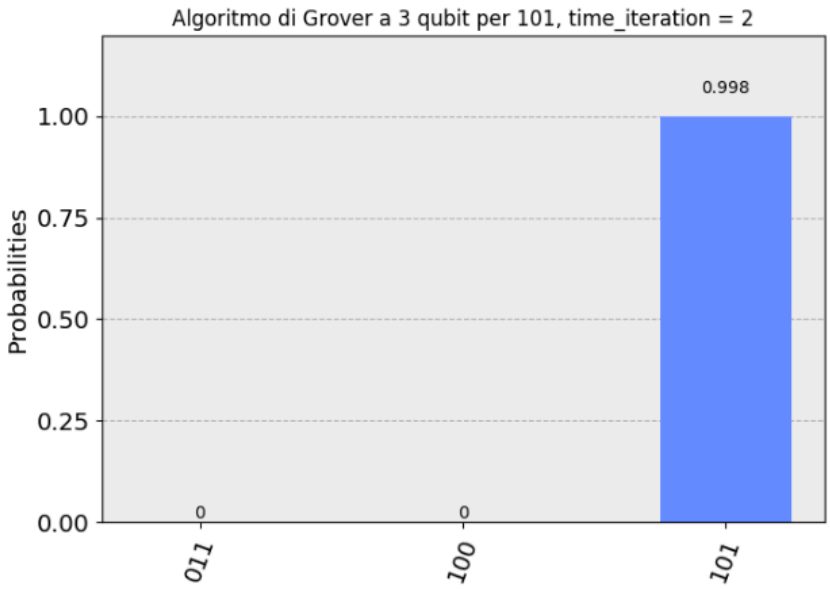
```
plt = plot_histogram(answer, title='Algoritmo di Grover a 3
qubit per 101, iterazioni = %d' % (time_iteration))
plt.show()
```

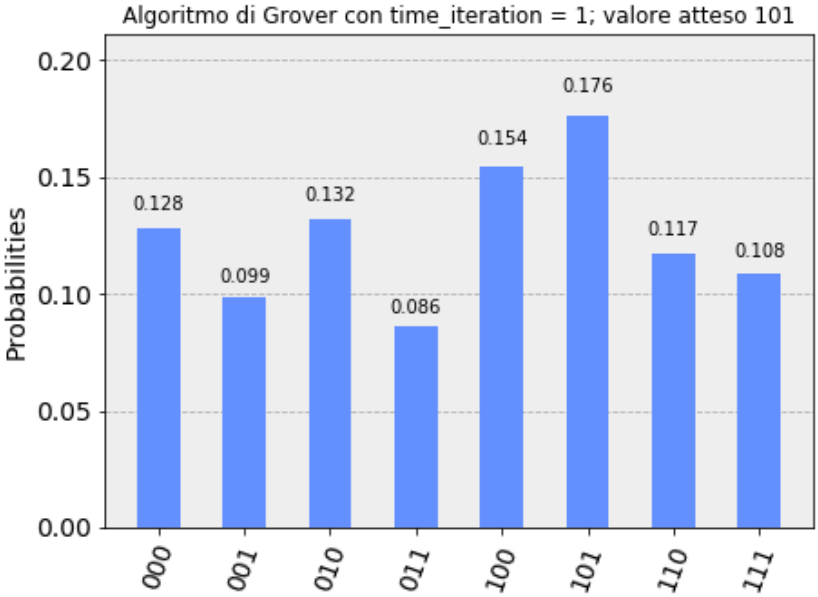
II.I. Circuito dell'Algoritmo di Grover a 3 qubit con soluzione 101



II.II. Risultati sul simulatore a varie iterazioni e dell'esecuzione su real device




Algoritmo di Grover a 3 qubit per 101, time_iteration = 2
Probabilities
1.00
0.75
0.50
0.25
0.00
0.998
0
0
011
100
101


Algoritmo di Grover con time_iteration = 1; valore atteso 101
Probabilities
0.20
0.15
0.10
0.05
0.00
0.128
0.099
0.132
0.086
0.154
0.176
0.117
0.108
000
001
010
011
100
101
110
111

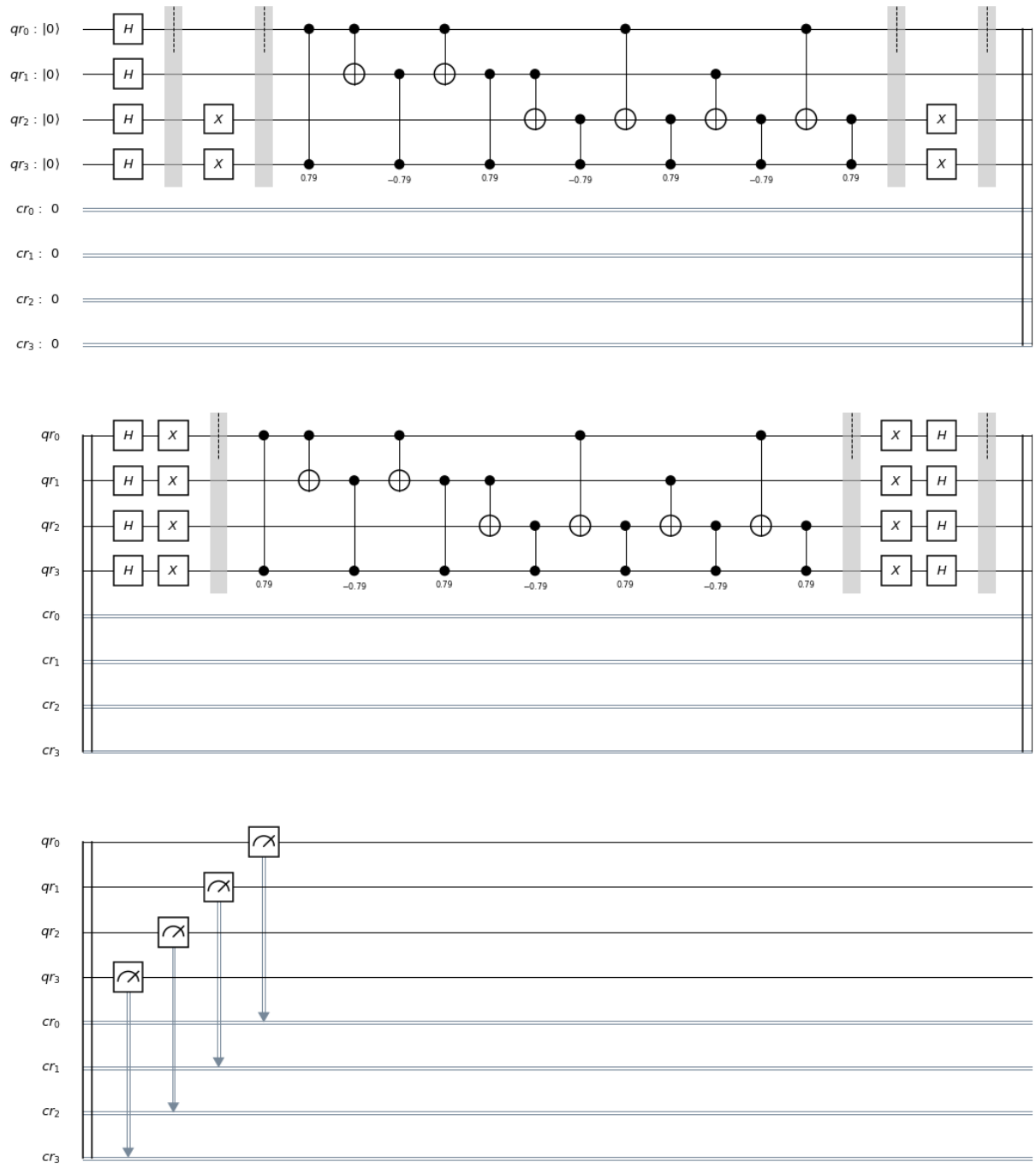
III. Algoritmo di Grover a 4 qubit con soluzione 0011

```
# ******************************************************
#
# Author: Nicolò Cangini
# Project: Quantum Computation with Qiskit (IBM)
# File: Grover Algorithm on 4 qubit to find 0011
# Sosftware: Python v3.7; Qiskit v0.7
# Date: March 2019
#
# ******************************************************

# Import packages
from math import pi
import numpy as np
from qiskit import ClassicalRegister, QuantumRegister, Quant
umCircuit
from qiskit import execute, BasicAer
from qiskit.tools.visualization import circuit_drawer, plot_
histogram, plot_bloch_multivector

# Define registers and circuit
n = 4
norm = np.sqrt(2**n)
qr = QuantumRegister(n, "qr")
cr = ClassicalRegister(n, 'cr')
circuit = QuantumCircuit(qr, cr, name="Grover_4qubit")

# Hadamard on all qubits to obtein superposition
for i in range(n):
    circuit.h(qr[i])
circuit.barrier()

# Gate cccz
def cccz():
```

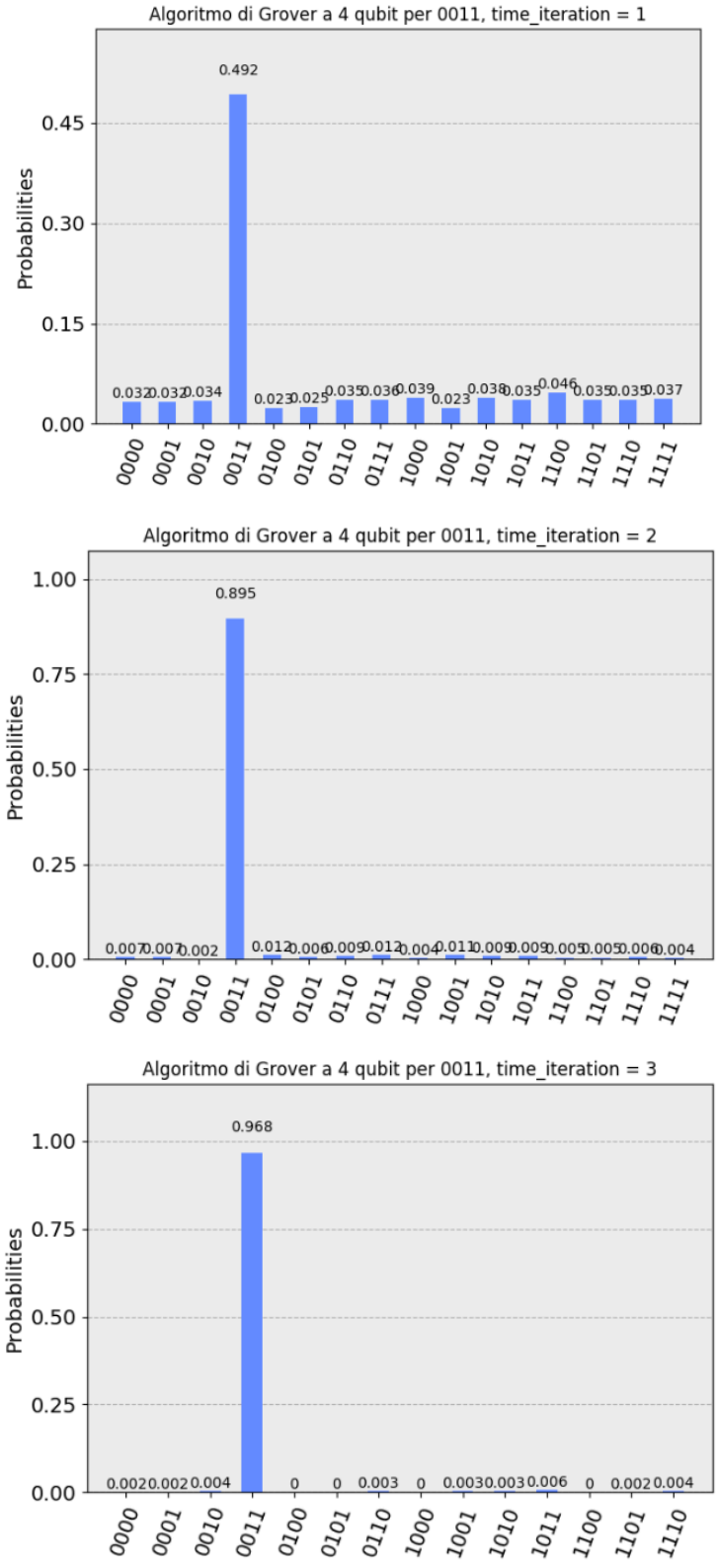
```
    circuit.cu1(pi / norm, qr[0], qr[3])
    circuit.cx(qr[0], qr[1])
    circuit.cu1(-pi / norm, qr[1], qr[3])
    circuit.cx(qr[0], qr[1])
    circuit.cu1(pi / norm, qr[1], qr[3])
    circuit.cx(qr[1], qr[2])
    circuit.cu1(-pi / norm, qr[2], qr[3])
    circuit.cx(qr[0], qr[2])
    circuit.cu1(pi / norm, qr[2], qr[3])
    circuit.cx(qr[1], qr[2])
    circuit.cu1(-pi / norm, qr[2], qr[3])
    circuit.cx(qr[0], qr[2])
    circuit.cu1(pi / norm, qr[2], qr[3])

# Grover Operator
def grover_operator():
    # Oracle for 0011
    circuit.x(qr[2])
    circuit.x(qr[3])
    cccz()
    circuit.x(qr[3])
    circuit.x(qr[2])
    circuit.barrier()

    # Amplification
    for i in range(n):
        circuit.h(qr[i])
    for i in range(n):
        circuit.x(qr[i])
    circuit.barrier()
    cccz()
    circuit.barrier()
    for i in range(n):
        circuit.x(qr[i])
    for i in range(n):
```

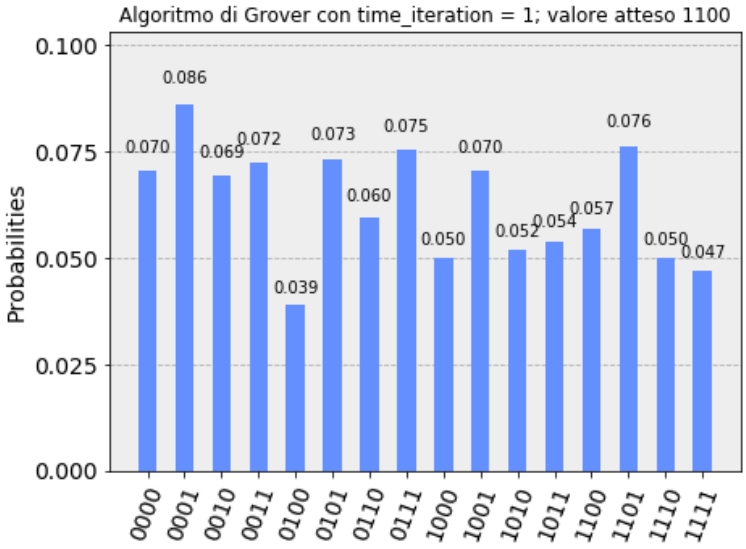
```
        circuit.h(qr[i])

# Number of Iteration to amplify the result
time_iteration = 1
for i in range(time_iteration):
    grover_operator()


# Measurement
circuit.barrier()
circuit.measure(qr, cr)
print(circuit)

# State Vector Simulation
backend_sv = BasicAer.get_backend('statevector_simulator')
job = execute(circuit, backend_sv)
results = job.result()
outputstate = results.get_statevector(circuit, decimals=2)
print(outputstate)

# QASM Simulation
backend = BasicAer.get_backend('qasm_simulator')
job = execute(circuit, backend=backend, shots=1024)
results = job.result()
answer = results.get_counts()
print(answer)

# Plot Results
plt = plot_histogram(answer, title='Algoritmo di Grover a
4 qubit per 0011, time_iteration = %d' % (time_iteration))
plt.show()
```

III.I. Circuito dell'Algoritmo di Grover a 4 qubit con soluzione 0011



III.II. Risultati sul simulatore a varie iterazioni e dell'esecuzione su real device





IV. Algoritmo di Deutsch-Jozsa a *n* qubit

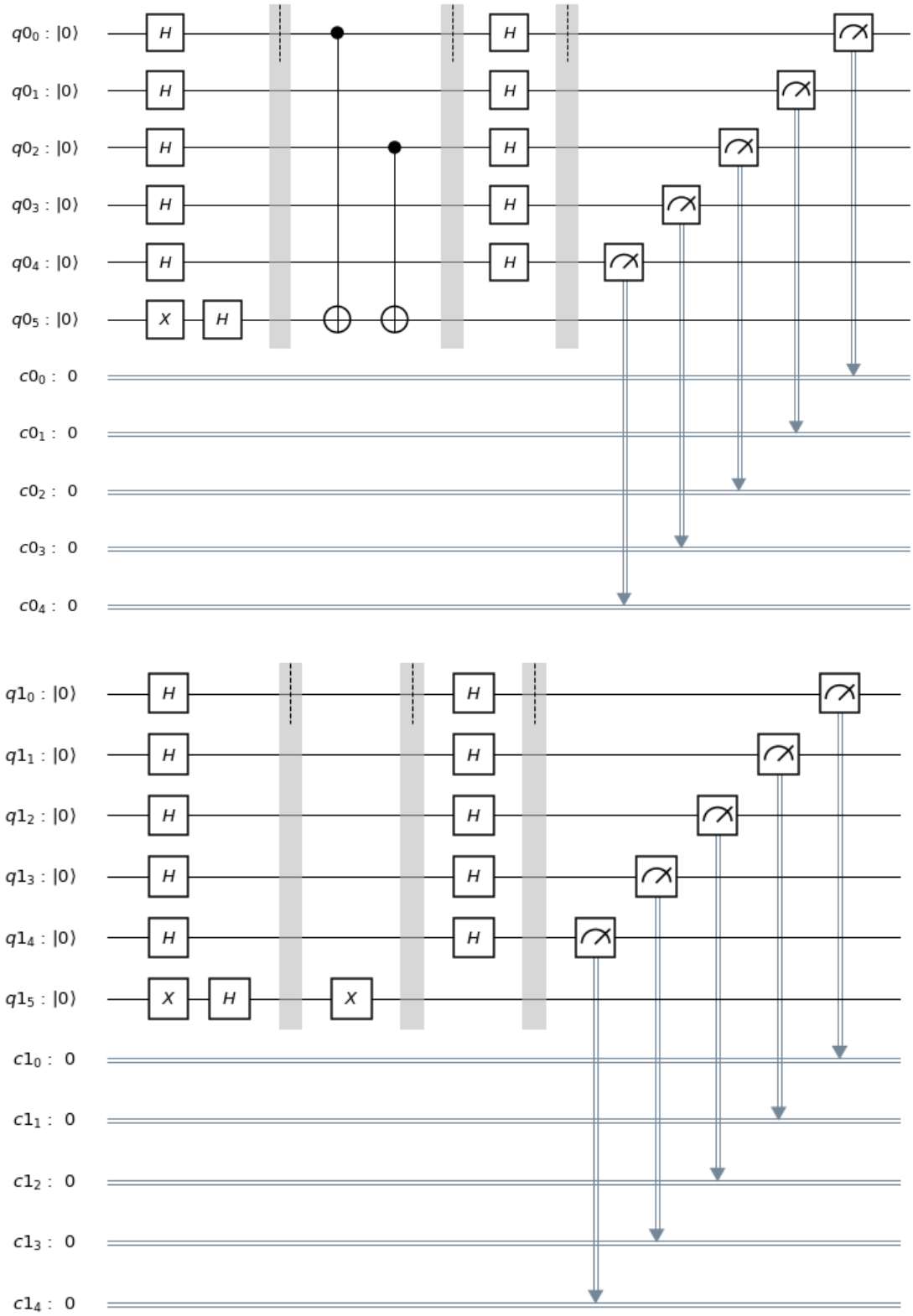
```
# ******************************************************
#
# Author: Nicolò Cangini
# Project: Quantum Computation with Qiskit (IBM)
# File: Deutsch-Jouzsa algorithm to verify if a
#       function is costant or balanced
# Sosftware: Python v3.7; Qiskit v0.7
# Date: March 2019
#
# ******************************************************

# Import packages
from math import pi
import numpy as np
from qiskit import ClassicalRegister, QuantumRegister, Quant
umCircuit
from qiskit import execute, BasicAer
from qiskit.tools.visualization import circuit_drawer, plot_
histogram, plot_bloch_multivector
```

```
# Length of the first register for querying the oracle
n = 5

# Choose a type of oracle at random. With probability half i
t is constant,
# and with the same probability it is balanced
oracleType = np.random.randint(2)
oracleValue = np.random.randint(2)
if oracleType == 0:
    print("The oracle returns a constant value ", oracleValu
e)
else:
    print("The oracle returns a balanced function")
    # this is a hidden parameter for balanced oracle based o
n number of qubit.
    a = np.random.randint(1, 2**n)

# Creating registers and circuit
# As saw in theory, n qubits for querying the oracle and one
qubit for storing the answer
qr = QuantumRegister(n + 1)
cr = ClassicalRegister(n)
circuit = QuantumCircuit(qr, cr, name='DeutschJozsa')

# Create the superposition of all input queries in the first
register and set the second to |1>.
circuit.x(qr[n])
for i in range(n+1):
    circuit.h(qr[i])
circuit.barrier()

# Application of Oracle operator
```

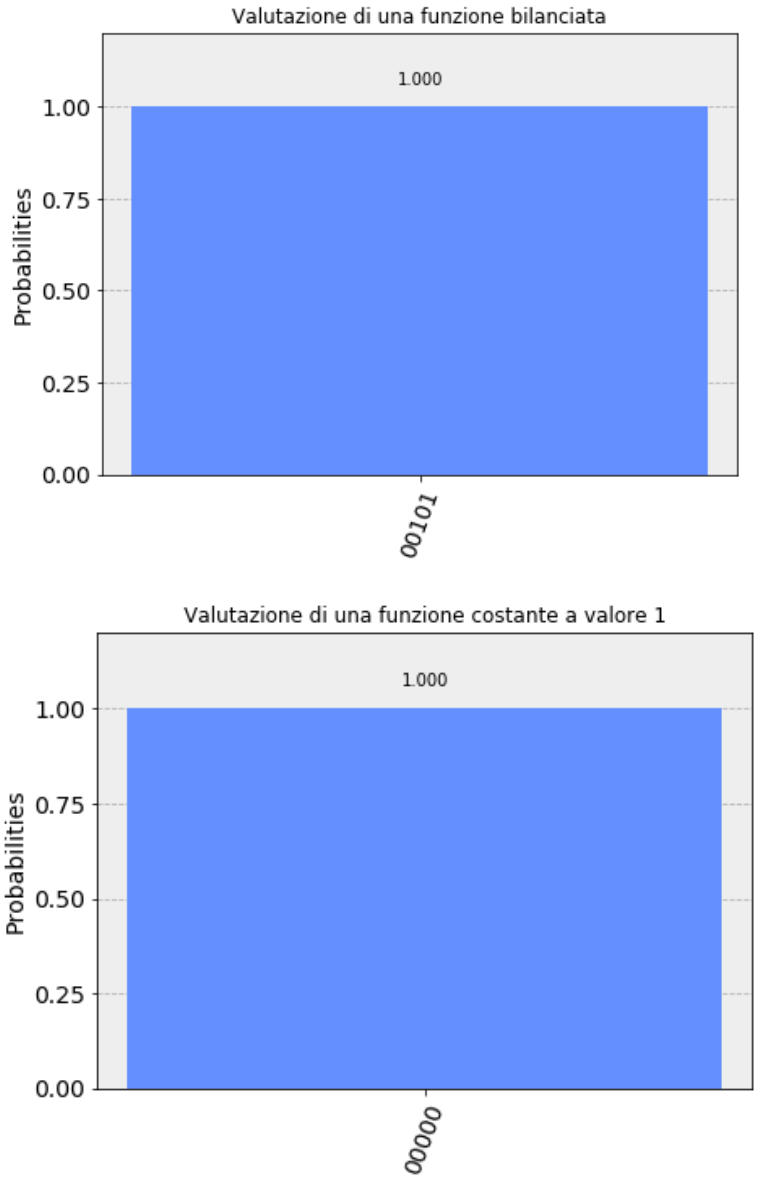
```
if oracleType == 0:  # If the oracleType is "0", the oracle
returns oracleValue for all input.
    if oracleValue == 1:
        circuit.x(qr[n])
    else:
        circuit.iden(qr[n])
else:  # Returns the inner product of the input with a non-z
ero bitstring
    for i in range(n):
        if (a & (1 << i)):
            circuit.cx(qr[i], qr[n])
circuit.barrier()

# Apply Hadamard gates after querying the oracle
for i in range(n):
    circuit.h(qr[i])
circuit.barrier()

# Measurement
for i in range(n):
    circuit.measure(qr[i], cr[i])
print(circuit)

# QASM Simulator
backend = BasicAer.get_backend('qasm_simulator')
job = execute(circuit, backend=backend, shots=1024)
results = job.result()
answer = results.get_counts()
# Plot Results
if oracleType == 0:
    title = "Valutazione di una funzione
                    costante a valore %d" % oracleValue
else:
    title = "Valutazione di una funzione bilanciata"
plt = plot_histogram(answer, title=title)
```

```
plt.show()
```

IV.I. Circuiti di Deutsch-Jozsa in base alla funzione



IV.II. Valutazione della funzione con l'Algoritmo di Deutsch-Jozsa dei circuiti in IV.I



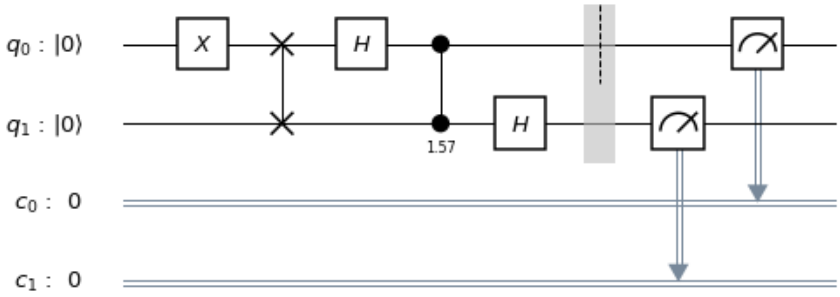
V. Quantum Fourier Transform a *n* qubit

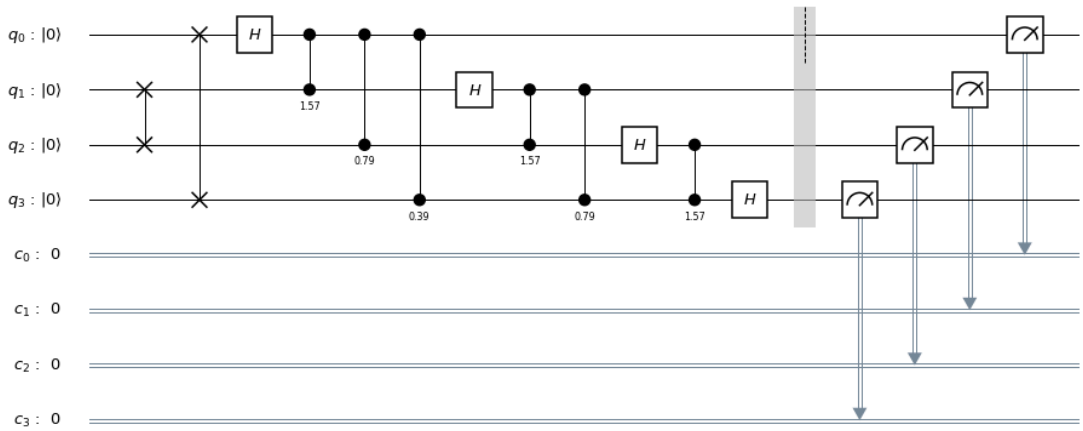
```
# ******************************************************
#
# Author: Nicolò Cangini
# Project: Quantum Computation with Qiskit (IBM)
# File: Quantum Fourier Transform on 2 qubit
```

```
# Sosftware: Python v3.7; Qiskit v0.7
# Date: March 2019
#
# ***************************************************

# Import packages
from math import pi
from qiskit import QuantumRegister, ClassicalRegister, Quant
umCircuit
from qiskit import execute, BasicAer
from qiskit.tools.visualization import plot_histogram, plot_
bloch_multivector

# Define circuit and numner of qubit
n = 2
qr = QuantumRegister(n, "q")
cr = ClassicalRegister(n, "c")
circuit = QuantumCircuit(qr, cr, name="QFT_nqubit")

# Quantum Fourier Transform
def qft():
    for j in range(int(n/2)):
        circuit.swap(qr[j], qr[n-j-1])
    for j in range(n):
        for k in range(j):
           circuit.cu1(pi / float(2 ** (j-k)), qr[j], qr[k])
        circuit.h(qr[j])

# Fourier Transform on 00, 01, 10 or 11
#circuit.x(qr[0])
#circuit.x(qr[1])
qft()
circuit.barrier()
circuit.measure(qr, cr)
print(circuit)
```

```
# Statevector Simulation
backend = BasicAer.get_backend('statevector_simulator')
job = execute(circuit, backend)
results = job.result()
outputstates = results.get_statevector(circuit, decimals=2)
print(outputstates)

# QASM Simulation
sim_backend = BasicAer.get_backend('qasm_simulator')
job = execute(circuit, sim_backend, shots=1024)
result = job.result()
print(result.get_counts(circuit))
# Plot Results
plt = plot_histogram(result.get_counts())
plt.show()
```

V.I Circuito della QFT a 2 qubit su 01 e a 4 qubit su 0000





VI. Classificatore k-nearest neighbour

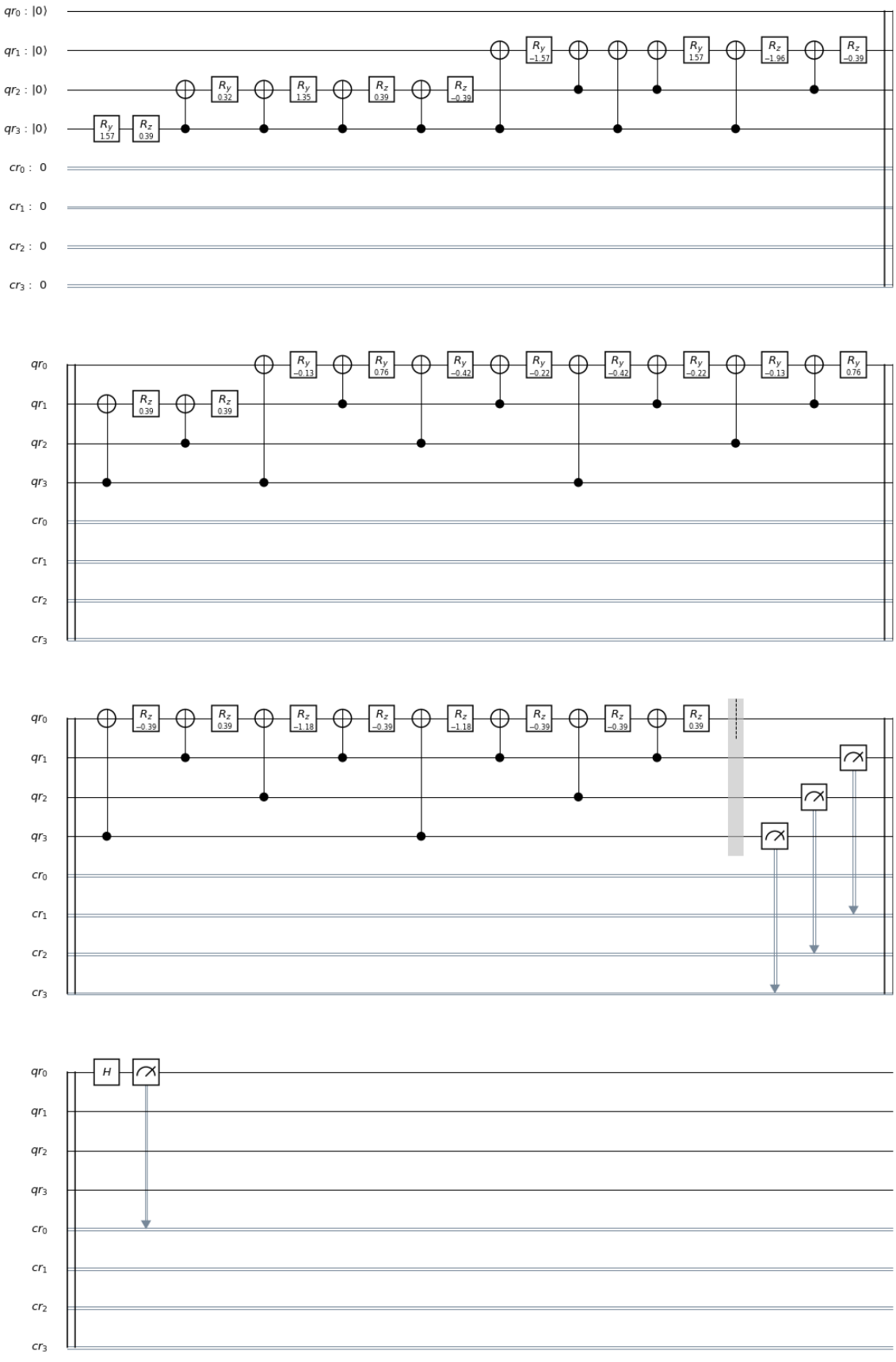
```
# *******************************************************
#
# Author: Nicolò Cangini
# Project: Quantum Computation with Qiskit (IBM)
# File: k-NN Routine on 4 qubit to Classify a vector
#       based on training set of 2 istances
# Sosftware: Python v3.7; Qiskit v0.7
# Date: March 2019
#
# *******************************************************

# Import packages
from math import pi
import numpy as np
from qiskit import BasicAer, execute
from qiskit import QuantumCircuit, ClassicalRegister, Quantu
mRegister
from qiskit.tools.visualization import plot_histogram, plot_
bloch_multivector, circuit_drawer

# Define registers and circuit
n = 4
qr = QuantumRegister(n, "qr")
cr = ClassicalRegister(n, 'cr')
circuit = QuantumCircuit(qr, cr, name="k-nn_circuit")

# This is how I want keep my qubits
# |ancilla> -----------
# |data> -----------------
# |data> -----------------
# |class> -----------------
```

```
# Define all input vectors you want to test.
x_in = [-0.948, 0.318]
#x_in = [-0.549, 0.836]  # x' in publication
#x_in = [0.053, 0.999]  # x'' in publication

# Training Set
xt_1 = [0.000, -1.000] # class 0
xt_2 = [-0.789, -0.615] # class 1

# Normalization Factor
M = 2*x_in[0]**2 + 2*x_in[1]**2 + xt_1[0]**2 + xt_1[1]**2 +
xt_2[0]**2 + xt_2[1]**2
print(M)
C= 1

# Amplitude value for each state
desired_vector = [
    1 / np.sqrt(M * C) * complex(x_in[0], 0),   # 0000
    1 / np.sqrt(M * C) * complex(xt_1[0], 0),   # 0001
    0,                                          # 0010
    0,                                          # 0011
    1 / np.sqrt(M * C) * complex(x_in[1], 0),   # 0100
    1 / np.sqrt(M * C) * complex(xt_1[1], 0),   # 0101
    0,                                          # 0110
    0,                                          # 0111
    0,                                          # 1000
    0,                                          # 1001
    1 / np.sqrt(M * C) * complex(x_in[0], 0),   # 1010
    1 / np.sqrt(M * C) * complex(xt_2[0], 0),   # 1011
    0,                                          # 1100
    0,                                          # 1101
    1 / np.sqrt(M * C) * complex(x_in[1], 0),   # 1110
    1 / np.sqrt(M * C) * complex(xt_2[1], 0),   # 1111
]
```

```
circuit.initialize(desired_vector, [qr[0], qr[1], qr[2], qr[
3]])
circuit.barrier()

# Hadamard on Ancilla qubit and measurement
circuit.h(qr[0])
circuit.measure(qr, cr)
print(circuit)

# Statevector Simulation
backend_sv = BasicAer.get_backend('statevector_simulator')
job_sv = execute(circuit, backend_sv)
result_sv = job_sv.result()
outputstate_sv = result_sv.get_statevector(circuit, decimals
=2)
print(outputstate_sv)

# QASM Simulation
sim_backend = BasicAer.get_backend('qasm_simulator')
job = execute(circuit, sim_backend, shots=1024)
results = job.result()
answer = results.get_counts(circuit)
print(answer)

# Plot Probabilities Results
plt = plot_histogram(answer)
plt.show()

total_samples = sum(answer.values())
# Subset creation of the results mesured with ancilla qubit
in state |0>
post_select = lambda sel_counts: [(state, occurences) for st
ate, occurences in sel_counts.items() if state[-1] == '0']
# Number of occurences with ancilla set to |0>
postselection = dict(post_select(answer))
```

```
print(postselection)
postselected_samples = sum(postselection.values())

print(f'Number of state with Ancilla bit |0> : {postselected
_samples}')
print(f'Ancilla post-selection probability was found to be {
postselected_samples / total_samples}')

# Separation of the results based on class mesured
retrieve_class = lambda binary_class: [occurences for state,
occurences in postselection.items() if
                                        state[0] == str(binar
y_class)]
# Printing results devided by class
print(f'Occurences of Class 0 :  {sum(retrieve_class(0))}')
print(f'Occurences of Class 1 :  {sum(retrieve_class(1))}')

# Probability evaluation
prob_class0 = sum(retrieve_class(0)) / postselected_samples
prob_class1 = sum(retrieve_class(1)) / postselected_samples
print(f'Probability for class 0 is {prob_class0}')
print(f'Probability for class 1 is {prob_class1}')

# printing final classification of input vector
if prob_class0 > prob_class1:
    print(f"Classifying input x as class 0 with noisy simula
tor backend")
elif prob_class0 < prob_class1:
    print(f"Classifying input x as class 1 with noisy simula
tor backend")
else:
    print(f"inconclusive. 50/50 results")
```

VI.I. Circuito per il Classificatore k-nearest neighbour



VI.II. Stampa del risultato

```
Number of state with Ancilla bit |0> : 573
Ancilla post-selection probability
                  was found to be 0.5595703125
Occurences of Class 0 :  167
Occurences of Class 1 :  406
Probability for class 0 is 0.2914485165794066
Probability for class 1 is 0.7085514834205934
Classifying input x as class 1 with noisy simulator backend
```

VII. Algoritmo HHL per sistema 2x2

```
# ********************************************************
#
# Author: Nicolò Cangini
# Project: Quantum Computation with Qiskit (IBM)
# File: HHL algorithm to resolve linear system
#       with matrix 2x2
#       with eigenvalues 1 and 3
# Sosftware: Python v3.7; Qiskit v0.7
# Date: March 2019
#
# ********************************************************

# Import packages
import numpy as np
from math import pi
from qiskit import BasicAer, execute
from qiskit import QuantumCircuit, ClassicalRegister,
                   QuantumRegister
from qiskit.tools.visualization import plot_histogram,
                   plot_state_city, plot_state_paulivec
```

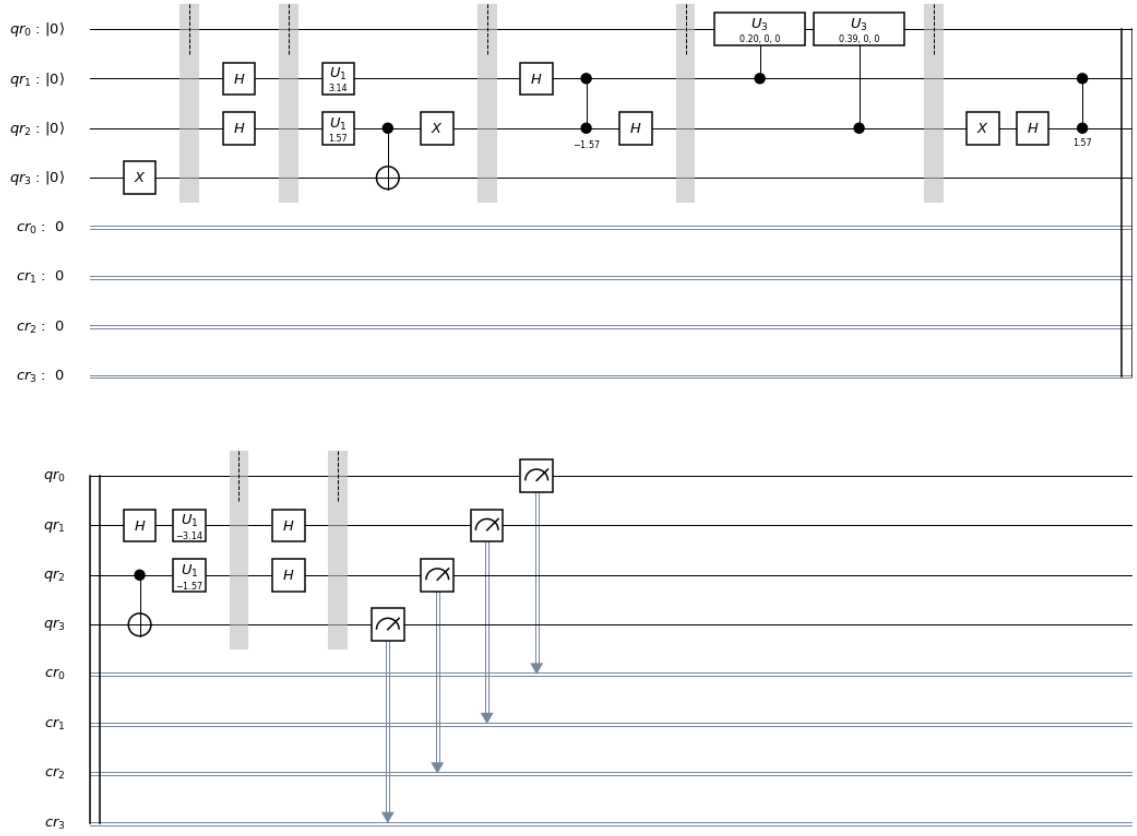
```
from qiskit.tools.qi.qi import outer, purity
import qiskit.tools.qcvv.tomography as tomo
from qiskit.quantum_info import state_fidelity

# Define registers and circuit
backend_qasm = BasicAer.get_backend('qasm_simulator')
qr = QuantumRegister(4, name="qr")
cr = ClassicalRegister(4, name="cr")
circuit = QuantumCircuit(qr, cr, name="HLL_2x2")

# This is how I want keep my qubits
# |ancilla> -----------
# |C> -----------------
# |C> -----------------
# |b> -----------------

# By commenting next commands decide what you want to use
# Nothing to have |b> = (1 0)
# |b> =(0 1)
circuit.x(qr[3])
#|b> = 1/sqrt(2)(1 1)
#circuit.h(qr[3])
circuit.barrier()

# Create Superposition of C
circuit.h(qr[1])
circuit.h(qr[2])
circuit.barrier()

# Phase Estimation and controlled-rotation to store the eige
nvalue
circuit.u1(pi, qr[1])
circuit.u1(pi/2, qr[2])
circuit.cx(qr[2], qr[3])
circuit.barrier()
```

```
# Quantum Inverse Fourier Transform
circuit.h(qr[1])
circuit.cu1(-pi / 2, qr[1], qr[2])
circuit.h(qr[2])
circuit.x(qr[2])
circuit.barrier()

# R（lamda^-1） Rotation
circuit.cu3(pi / 16, 0, 0, qr[1], qr[0])
circuit.cu3(pi / 8, 0, 0, qr[2], qr[0])
circuit.barrier()

# Uncompute
circuit.x(qr[2])
circuit.h(qr[2])
circuit.cu1(pi / 2, qr[1], qr[2])
circuit.h(qr[1])
circuit.cx(qr[2], qr[3])
circuit.u1(-pi/2, qr[2])
circuit.u1(-pi, qr[1])
circuit.barrier()
circuit.h(qr[2])
circuit.h(qr[1])
circuit.barrier()
print(circuit)

# Statevector Simulation
backend_sv = BasicAer.get_backend('statevector_simulator')
job_sv = execute(circuit, backend_sv)
result_sv = job_sv.result()
outputstate_psi = result_sv.get_statevector(circuit, decimal
s=5)
outputstate_rho = outer(outputstate_psi) # construct the den
sity matrix from the state vector
```

```
plt = plot_state_city(outputstate_rho)
plt.show()

# Construct state tomography set for measurement of qubits
bell_tomo_set = tomo.state_tomography_set([0, 1, 2, 3])
bell_tomo_circuits = tomo.create_tomography_circuits(circuit
, qr, cr, bell_tomo_set)
backend = BasicAer.get_backend('qasm_simulator')
shots = 5000
# Run the simulation
bell_tomo_job = execute(bell_tomo_circuits, backend=backend,
shots=shots)
bell_tomo_result = bell_tomo_job.result()
bell_tomo_data = tomo.tomography_data(bell_tomo_result, circ
uit.name, bell_tomo_set)
rho_fit = tomo.fit_tomography_data(bell_tomo_data)

# calculate fidelity, concurrence and purity of fitted state
F_fit = state_fidelity(rho_fit, outputstate_psi)
pur = purity(rho_fit)

print('Fidelity =', F_fit)
print('purity = ', str(pur))
# Plot Results
plt = plot_state_city(outputstate_rho, title='Matrice di Den
sità')
plt.show()
plt = plot_state_city(rho_fit, title='Tomografia
                                        Quantistica')
plt.show()

# QASM Simulation
circuit.measure(qr, cr)
job = execute(circuit, backend=backend_qasm, shots=10000)
results = job.result()
```

```
answer = results.get_counts()
n_0 = answer['0001']
n_1 = answer['1001']
print('Occurrences in 0001: %d' % n_0)
print('Occurrences in 1001: %d' % n_1)
p_scale = float(n_0 / n_1)
print('Ratio: %f' % p_scale)
# Plot Results
plt = plot_histogram(answer, title='Valori della
                    misurazione del circuito HHL')
plt.show()
```

VII.I. Circuito dell'Algoritmo HHL per sistema 2x2 per $|b\rangle = [0\ 1]$



VII.II. Istogrammi del HHL per differenti valori di $|b\rangle$ su diversi backend

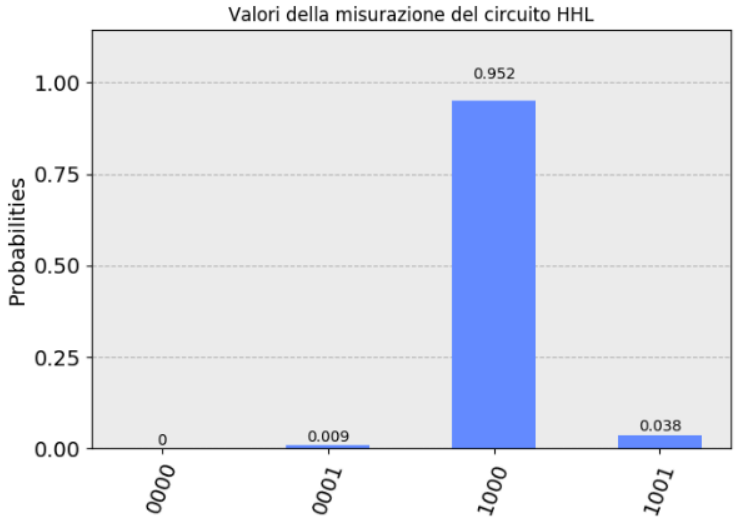
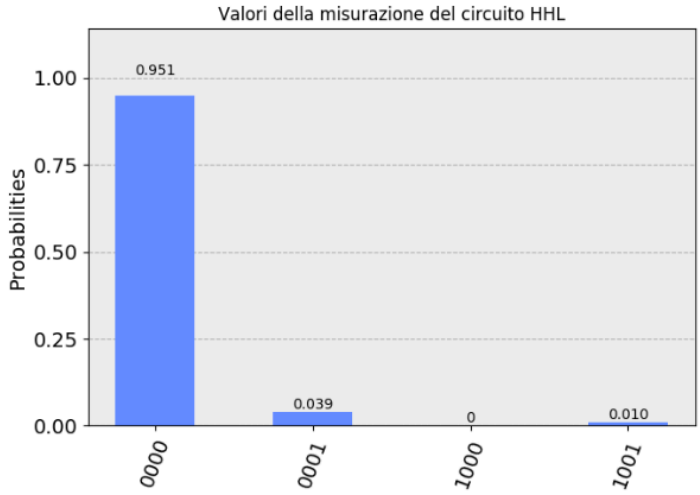
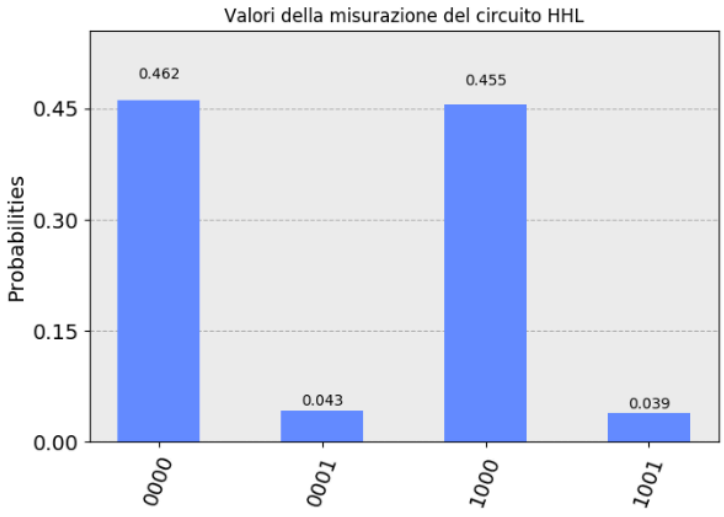
$\|b\rangle$	Esecuzione a shots = 10000 su backend QASM
$\|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$	
$\|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$	
$\|+\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\end{bmatrix}$	

Tabella 3-Riassunto degli istogrammi ottenuti sul simulatore (backend=QASM) al variare dei termini noti dell'algoritmo HHL

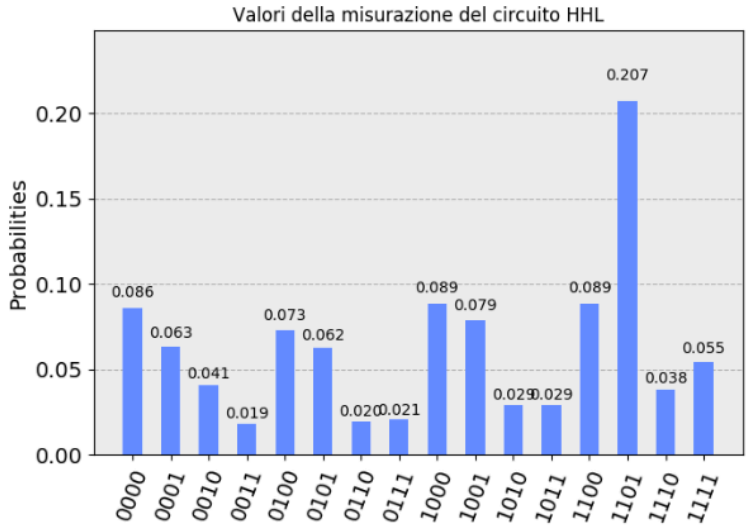
$\|b\rangle$	Esecuzione a shots = 10000 su quantum real device
$\|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	

Tabella 4-Istogramma dell'esecuzione su quantum real device dell'algoritmo HHL

VIII. Codice per eseguire algoritmi e circuiti su IBM real quantum device

```
# ************************************************************
#
# Author: Nicolò Cangini
# Project: Quantum Computation with Qiskit (IBM)
# File: IBM enviroment to execute on real quantum device
# Sosftware: Python v3.7; Qiskit v0.7
# Date: March 2019
#
# ************************************************************

# Import packages
from math import pi
import numpy as np
from qiskit import QuantumCircuit, ClassicalRegister,
            QuantumRegister
from qiskit import BasicAer, IBMQ, execute, compile
```

```
from qiskit.providers.ibmq import least_busy
from qiskit.tools.monitor import job_monitor
from qiskit.tools.visualization import circuit_drawer,
            plot_histogram, plot_bloch_multivector

# Connection to My IBM Account
APItoken = '### Your account token ###'
IBMQ.enable_account(APItoken)

# Show if connection is correctly done
print(IBMQ.active_accounts())

# Show available backend
print(IBMQ.backends(operational=True, simulator=False))

# Select less busy real backend available
large_enough_devices = IBMQ.backends(
      filters=lambda x: x.configuration().n_qubits > 3
      and not x.configuration().simulator)
backend = least_busy(large_enough_devices)
print("The best backend is " + backend.name())

#-----------------------------------------------
# Define your circuit or your algorithm
#-----------------------------------------------

# Execution on real device
job_exp = execute(circuit, backend=backend, shots=1024, max_
credits=3)
job_monitor(job_exp)
result_exp = job_exp.result()
counts_exp = result_exp.get_counts(circuit)

# Plot Results
plt = plot_histogram(counts_exp)
```

```
plt.show()

# Disable Account connection
IBMQ.disable_accounts(token=APItoken)
# Show available backend
print(IBMQ.backends())
```

Bibliografia

[1] Kopczyk, D. (2018). Quantum machine learning for data scientists. *arXiv preprint arXiv:1804.10068*.

[2] Ekert, A., Hayden, P. M., & Inamori, H. (2001). Basic concepts in quantum computation. In *Coherent atomic matter waves* (pp. 661-701). Springer, Berlin, Heidelberg.

[3] Zeh, H. D. (1970). On the interpretation of measurement in quantum theory. *Foundations of Physics*, *1*(1), 69-76.

[4] Di Pierro, A. (2010). Quantum Computing. *University of Verona, Tech. Rep*.

[5] Shor, P. W. (1994, November). Polynomial-time algorithms for prime factorization and discrete logarithms. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science* (pp. 124-134).

[6] Barenco, A., Bennett, C. H., Cleve, R., DiVincenzo, D. P., Margolus, N., Shor, P., ... & Weinfurter, H. (1995). Elementary gates for quantum computation. *Physical review A*, *52*(5), 3457.

[7] Wiebe, N., Kapoor, A., & Svore, K. (2014). Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning. *arXiv preprint arXiv:1401.2142*.

[8] Aïmeur, E., Brassard, G., & Gambs, S. (2007, June). Quantum clustering algorithms. In *Proceedings of the 24th international conference on machine learning*(pp. 1-8). ACM.

[9] Rebentrost, P., Mohseni, M., & Lloyd, S. (2014). Quantum support vector machine for big data classification. *Physical review letters*, *113*(13), 130503.

[10] Schuld, M., Sinayskiy, I., & Petruccione, F. (2014, December). Quantum computing for pattern classification. In *Pacific Rim International Conference on Artificial Intelligence* (pp. 208-220). Springer, Cham.

[11] Lloyd, S., Mohseni, M., & Rebentrost, P. (2013). Quantum algorithms for supervisedand unsupervised machine learning. *arXiv preprint arXiv:1307.0411*.

[12] Nielsen, M. A., & Chuang, I. L. (2000). Quantum computation and quantum information.

[13] Wiebe, N., Kapoor, A., & Svore, K. (2014). Quantum algorithms for nearest-neighbor methods for supervised and unsupervised learning. *arXiv preprint arXiv:1401.2142*.

[14] Macaluso, A. (May, 2017). Computer Science and Engineering PhD Project – Arificial Intelligence.

[15] Aïmeur, E., Brassard, G., & Gambs, S. (2006, June). Machine learning in a quantum world. In *Conference of the Canadian Society for Computational Studies of Intelligence* (pp. 431-442). Springer, Berlin, Heidelberg.

[16] Giovannetti, V., Lloyd, S., & Maccone, L. (2008). Quantum random access memory. *Physical review letters*, *100*(16), 160501.

[17] Alpaydin, E. (2004). Introduction to machine learning, chapter 7.

[18] Lu, S., & Braunstein, S. L. (2014). Quantum decision tree classifier. *Quantum information processing*, *13*(3), 757-770.

[19] Harrow, A. W., Hassidim, A., & Lloyd, S. (2009). Quantum algorithm for linear systems of equations. *Physical review letters*, *103*(15), 150502.

[20] Dervovic, D., Herbster, M., Mountney, P., Severini, S., Usher, N., & Wossnig, L. (2018). Quantum linear systems algorithms: a primer. *arXiv preprint arXiv:1802.08227*.

[21] Coles, P. J., Eidenbenz, S., Pakin, S., Adedoyin, A., Ambrosiano, J., Anisimov, P., ... & Gunter, D. (2018). Quantum algorithm implementations for beginners. *arXiv preprint arXiv:1804.03719*.

[22] Cai, X. D., Weedbrook, C., Su, Z. E., Chen, M. C., Gu, M., Zhu, M. J., ... & Pan, J. W. (2013). Experimental quantum computing to solve systems of linear equations. *Physical review letters*, *110*(23), 230501.

[23] Pan, J., Cao, Y., Yao, X., Li, Z., Ju, C., Chen, H., ... & Du, J. (2014). Experimental realization of quantum algorithm for solving linear systems of equations. *Physical Review A*, *89*(2), 022313.

[24] https://qiskit.org/documentation/

[25] https://quantumexperience.ng.bluemix.net/qx/editor

Ringraziamenti

Ringrazio infinitamente il professor Claudio Sartori per avermi dato l'opportunità di studiare ed esplorare questo nuovo modello computazionale, facendo del mio percorso di tesi il viaggio più appassionante di tutto il mio percorso di studi.

Ringrazio di cuore il dottorando Antonio Macaluso per avermi aiutato nell'affrontare tutti gli aspetti del quantum computing, per il suo supporto, per il tempo dedicatomi e per aver condiviso con me la sua passione e la sua determinazione nello studio di questo ramo del Computer Science; a lui auguro di completare il suo percorso di dottorato con splendidi risultati e di proseguire in questo ambito del quantum computing, sperando siano alte le probabilità di rincontrarlo.

Grazie agli amici di sempre, per esserci a prescindere dalla distanza che a volte ci separa e dal tempo che passiamo senza notizie l'uno dell'altro.

Grazie a Claudia, per la tua bontà, per credere in me come nessuno aveva mai fatto prima, per le esperienze fatte assieme e il tempo vissuto in quella Isla che tanto ha fatto per noi, e per la vicinanza che mi fai sentire nonostante 1872 chilometri di distanza, dimostrare l'entanglement non è poi così difficile; grazie alla tua famiglia, per avermi fatto sempre sentire a casa e avermi reso i periodi di studio i più produttivi di sempre.

Grazie infine alla mia famiglia, soprattutto ai miei genitori, per il sostegno che in ogni secondo ho ricevuto in questi anni, per l'appoggio in ogni mia libera decisione, giusta o sbagliata che fosse; grazie a Baloo, per la tua infinita compagnia nelle giornate passate a scrivere la tesi e grazie ai miei nonni, per tutto, senza di voi non sarei mai stato quello che sono.