Simulazione di Algoritmi Quantistici usando lo strumento Q#

Daniel Biasiotto

November 4, 2022

CONTENTS

1 INTRODUZIONE

Lo sviluppo di software quantistici si è confermato come un'area di grande interesse negli ultimi decenni, offrendo grandi possibilità di superare i limiti computazionali attualmente raggiunti in diverse aree di ricerca. È possibile che in futuro algoritmi quantistici possano sostituire controparti classiche in diverse applicazioni:

- crittografia;
- problemi di ricerca, l'algoritmo di Grover in questo campo è quadraticamente più veloce del analogo classico;
- utilizzo di computer quantistici per la simulazione di sistemi quantistici
- machine learning;
- computazione biologica;
- chimica generativa.

Ovviamente il calcolo classico non verrà abbandonato, l'approccio classico e quello quantistico differiscono nelle loro forze e debolezze. Mentre gli attuali computer diventano sempre più veloci e le tecniche industriali permettono miniaturizzazioni sempre maggiori gli hardware quantistici rimangono estremamente complicati da costruire e distribuire. Il modo attualmente più congeniale di utilizzare questi hardware per effettuare testing di software rimane quello della condivisione di risorse attraverso le tecnologie di *cloud computing*.

Ma questo non significa che non sia possibile o non sia utile studiare i problemi e le soluzioni algoritmiche che il calcolo quantistico offre a livello teorico e di sviluppo software. E' importante sottolineare che ci aspettiamo che i computer quantistici e le risorse di cui dispongono scalino diversamente nel contesto di alcune classi di problemi da risolvere, in particolare permettendo per i problemi giusti, più adatti, di superare di gran lunga la *performance* dei migliori algoritmi classici. Questo perché l'approccio algoritmico permesso da questa tecnologia è profondamente diverso da quello classico e porta a miglioramenti importanti in termini di complessità computazionale. D'altra parte non ci si deve aspettare che questa sia una soluzione perfetta, ci sono problemi che rimarranno difficili da risolvere nonostante le nuove possibilità della computazione quantistica.

Per molto tempo l'approccio a questo campo è rimasto accessibile solo a matematici e fisici con le conoscenze specifiche necessarie a comprendere appieno la meccanica quantistica e le sue sfumature tecniche.

Questo sta cambiando velocemente negli ultimi anni da quando l'industria ha cominciato a sviluppare strumenti e piattaforme che permettano a nuovi sviluppatori di interagire e imparare in quest'ambito astraendo dalla maggior parte della complessità della teoria matematica alla base di questo modello.

1.1 Definizione di un Computer Quantistico

Per prima cosa definiamo un *computer* come uno strumento che prendendo dei dati in input esegue delle operazioni su questi dati. I *computer* che conosciamo e utilizziamo attualmente possono e sono definiti nei confini teorici della fisica classica, ovvero attraverso le leggi di Newton e l'elettromagnetismo. Data questa definizione è semplice definire un *computer quantistico* come uno strumento che prendendo dei dati in input esegue delle operazioni su questi dati con processi descrivibili solo utilizzando i concetti della fisica quantistica. Detto questo la differenza tra un computer classico e un computer quantistico è la stessa che esiste tra la fisica classica e quella quantistica. La principale differenza è la scala su cui operano in termini di dimensioni e energia. I calcolatori quantistici sono in ogni caso controllati tramite interfacce e strumenti classici con cui è più semplice interagire e tramite cui i dati sono riconvertiti in dati di tipo classico utilizzabili dalle `CPU` che comunemente utilizziamo.

In altre parole un computer quantistico differisce da uno classico nella sua capacitá di non sottostare agli stessi limiti di complessitá temporale nella risoluzione di problemi computazionali. Importante quando serve risolvere problemi complessi per cui non sono accettabili semplificazioni e ci si avvicina sempre di piú ai limiti fisici nelle architetture hardware classiche. In questi casi infatti invece che creare supercomputer sempre piú potenti puó essere interessante un cambio di approccio, ritrovabile nel quantum computing.

1.2 Limiti Hardware

Attualmente i software sviluppati per hardware quantistici possono essere eseguiti su simulatori, sempre software, oppure sfruttando reali macchine quantistiche in remoto. Questo è necessario in quanto i requisiti e limiti tecnici della costruzione di un calcolatore quantistico sono estremamente complessi e richiedono ambienti fisici attualmente incompatibili con un utilizzo desktop. Questi limiti sono in particolare l'estrema sensibilità dei registri all'interno di un computer simile alle interferenze esterne, per ovviare a problemi di interferenza è necessario mantenere l'hardware in temperature vicine allo zero termico (0K o $-273.15°C$). Queste condizioni non sono ovviamente replicabili in un ambiente non strettamente controllato e isolato.

Dati questi limiti è facile dire che per molto tempo ancora questo tipo di hardware sarà limitato a un utilizzo da remoto tramite architetture cloud, molte aziende private in questi anni hanno sviluppato soluzioni in questo senso. Alcuni esempi sviluppati negli ultimi anni dai maggiori competitor sono `IBM Q`[1], Google Quantum AI[2], `XANADU` Quantum Cloud[3] con la sua interfaccia python **Strawberry Fields**[4] con supporto `Tensor Flow`

1 https://quantum-computing.ibm.com/
2 https://quantumai.google/
3 https://www.xanadu.ai/cloud
4 https://strawberryfields.ai/

integrato, Amazon Bracket[5] offerto da `AWS`. Queste sono solo le offerte piú commercialmente rilevanti, sono tante le alternative che sarebbe possibile considerare.

I simulatori e le macchine reali condividono interfacce condivise che permettono lo sviluppo di software che abbia la possibilità di essere testato efficacemente. Uno di questi simulatori é ad esempio `qsim`[6], sviluppato dal team di Google Quantum AI questo simulatore permette di emulare circuiti quantistici in modo efficace fino a 20 qbit interfacciandosi con il framework `Cirq`[7].

1.3 Utilizzi della tecnologia

I computer quantistici offrono nuove possibilità nella risoluzione di diverse classi di problemi, la ricerca in questo campo ha trovato alcuni esempi di algoritmi che utilizzando le proprietà particolari di questo approccio per sviluppare un vantaggio nei confronti dell'approccio classico.

Ad esempio:

- L'**algoritmo di Grover** effettua una ricerca in una lista di N elementi in tempo $O(\sqrt{N})$.
- L'**algoritmo di Shor** fattorizza velocemente grandi numeri, in particolare permette di fattorizzare con un grado di errore arbitrariamente piccolo e un numero polinomiale di passi rispetto alla lunghezza in bit dell'input.
- L'**algoritmo di Deutsch-Jozsa** verifica se una funzione è costante o bilanciata in tempo costante $O(1)$.
- L'**algoritmo di Simon**, ispirazione per il sopraccitato algoritmo di Shor, risolve in tempo esponenzialmente più veloce rispetto all'approccio classico il problema di determinare se una data funzione f *blackbox* sia **uno-a-uno** o **due-a-uno**.

In aggiunta ai problemi di cui sopra, la cui soluzione è legata a un algoritmo in particolare, sono state trovate applicazioni per i computer quantistici in diversi altri ambiti:

- La stima di una somma di Gauss, un tipo di somma esponenziale, con precisione polinomiale e in tempo polinomiale contro il tempo esponenziale degli algoritmi classici.
- La valutazioni di formule booleane complesse può essere velocizzata tramite un approccio quantistico.
- Questi computer permettono di simulare sistemi quantistici permettendone uno studio più approfondito.
- La generazioni di numeri casuali è un'importante componente della crittologia e utilizzando tecniche quantistiche è possibile generarne che siano davvero casuali e non più pseudo-casuali come necessario in computer classici.

5 https://aws.amazon.com/braket/
6 https://quantumai.google/qsim
7 https://quantumai.google/cirq

Questi sono risultati importanti e in particolare l'algoritmo di Shor pone dei dubbi sulla sicurezza degli attuali protocolli crittografici che si basano sulla difficoltà computazionale della fattorizzazione di grandi numeri interi. Se tale algoritmo fosse facilmente eseguibile significherebbe che un attaccante potrebbe facilmente violare questi protocolli di sicurezza correntemente alla base della comunicazione via Internet.

Rimane difficile trovare possibili algoritmi quantistici che diano un vantaggio computazionale nel campo dell'apprendimento automatico dove è fondamentale l'accesso casuale a una grande quantità di dati.

In generale è più probabile che un problema che abbia una piccola mole di dati in entrata e in uscita ma una grande quantità di manipolazioni per arrivare all'output sia un buon candidato per l'utilizzo di computer quantistici.

2 AMBIENTE

Per lo sviluppo di software quantistici sono disponibili diversi ambienti e framework, tra i più conosciuti troviamo **Microsoft Azure** con il proprio Quantum Development Kit (QDK) o l'ambiente di sviluppo di IBM **Qiskit**. Altri *Software Development Kit* che possono essere utilizzati per eseguire circuiti quantistici su prototipi di device quantistici o simulatori sono:

- Ocean
- ProjectQ
- Forest
- t|ket>
- Strawberry Fields
- PennyLane

Molti di questi progetti sono open-source e sviluppati sulla base di Python.

Per questo lavoro abbiamo utilizzato gli strumenti offerti da Microsoft per l'ottima documentazione consultabile sulle loro pagine web e in quanto questo strumento era utilizzato dalla nostra fonte principale *Learn Quantum Computing with Python and Q#*.[8] Nella documentazione ufficiale di Microsoft Azure è presente una guida[9] all'installazione dell'ambiente di programmazione Q# di cui qui riportiamo dei passaggi.

Il lavoro per questa tesi è stato fatto in un ambiente Linux, il processo di installazione è del tutto equivalente nel caso si utilizzasse Windows utilizzando la powershell e conda o pip.

L'ambiente di esecuzione Q# può essere configurato sul editor Visual Studio Code tramite l'add-on proprietario Microsoft Quantum Development Kit[10]. Quest'ultimo è disponibile solo sulla versione non FOSS del software, che è possibile installare tramite le repository opensource linux.

In alternativa o anche parallelamente è possibile sviluppare codice Q# ed eseguirlo tramite Jupyter Notebook tramite Python. Questo con i kernel necessari installati, avendo quindi l'ultima versione di dotnet disponibile. Attualmente la versione LTS è la .NET Core 6.0 ed è quella che useremo. Si può trovare direttamente sul sito della microsoft[11] o più semplicemente tramite il *package manager* del proprio sistema operativo.

Altro step necessario per l'esecuzione dei Jupyter Notebook è l'installazione delle runtime aspnet [12].

Una volta installata l'ultima versione di dotnet è possibile eseguire:

```
$ dotnet tool install -g Microsoft.Quantum.IQSharp
$ dotnet iqsharp install
```

8 il libro si può trovare a https://www.manning.com/books/learn-quantum-computing-with-python-and-q-sharp
9 documentazione azure: https://learn.microsoft.com/en-us/azure/quantum/install-overview-qdk
10 https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing/
11 https://dotnet.microsoft.com/en-us/download
12 https://dotnet.microsoft.com/en-us/download/dotnet/6.0

Per alcune istallazioni linux sarà necessario eseguire in alternativa:

```
$ dotnet iqsharp install --user
```

Questo installa i kernel `IQ#` che useremo con i `Jupyter Notebook`.

Per l'installazione in locale di tutto ciò che è necessario per lo sviluppo di software in questo ambito e in altre applicazioni scientifiche risulta molto più semplice l'utilizzo di una distribuzione pre-impostata come quella di *Anaconda*[13]. Uno strumento simile aiuta nella gestione di `Python` e altri strumenti software di ambito scientifico. Se necessario lo si dovrà aggiungere al `PATH` [14]:

```
$ PATH=/opt/anaconda/bin:$PATH
```

Tramite `Anaconda` si crea un ambiente di esecuzione con tutto quello che ci serve per i nostri obiettivi tramite il *package manager* incluso, `conda`:

```
$ conda create -n qsharp-env -c microsoft qsharp notebook
$ conda activate qsharp-env
```

In qualsiasi momento si può attivare l'ambiente `conda` che abbiamo creato per avere il necessario all'esecuzione dei nostri programmi `Q#` / `Python`. Al momento della scrittura `Anaconda` supporta la versione di `Python` 3.9, per gli scopi di questa tesi si suppone di avere a disposizione almeno usa versione superiore alla 3 per garantire compatibilità.

L'esecuzione del software `Q#` può essere testato localmente predisponendo un ambiente di simulazione tramite il pacchetto `Python` chiamato `qsharp`.

```
1 import qsharp
2 from QsharpNamespace import Operation_One, Operation_Two
3 var1 = 10
4 print("Simulation started...")
5 Operation_One.simulate(par1=var1)
6 Operation_Two.simulate(par2=var1,par3=5)
```

Listing 1: host.py

Ad esempio come nel listato qui sopra utilizziamo uno script `host.py` per creare un ambiente di simulazione per poter eseguire le operazioni `Q#` definite in `Operation_One` e `Operation_Two`. Il pacchetto automaticamente va a cercare nella directory locale le definizioni.

13 Si trovano informazioni a riguardo di questa distribuzione software all'indirizzo https:www.anaconda.com

14 Supponiamo l'uso di un ambiente `unix`

```
import qsharp

prepare_qubit = qsharp.compile("""
    open Microsoft.Quantum.Diagnostics;

    operation PrepareQubit(): Unit {
        using (qubit = Qubit()) {
            DumpMachine();
        }
    }
""")

if __name__ == "__main__":
    prepare_qubit.simulate()
```

Listing 2: qsharp-interop.py

Un esempio più complesso può essere quello definito in `qsharp-inteop.py` dove definiamo direttamente *inline* il contenuto del codice `Q#` che il `package qsharp` compila e simula.
Con il necessario installato è possibile leggere ed eseguire il codice di esempio pubblicato dagli autori di *Learn Quantum Computing with Python and Q#* sulla loro repository github[15].

15 https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp

3 Q#

Nei prossimi capitoli utilizzeremo `Q#` per implementare alcuni algoritmi quantistici. Per questo ci sarà utile introdurre delle basi in questo linguaggio per facilitare la lettura dei listati che saranno presentati successivamente.

`Q#` è il linguaggio di programmazione di algoritmi quantistici open-source[16] sviluppato da Microsoft, fa parte del Quantum Development Kit di quest'ultima. Come linguaggio eredita caratteristiche classiche di linguaggi imperativi ad oggetti come Python, C# supportando loop, blocchi if/then e strutture dati di base. Altre queste introduce in aggiunta costrutti specifici per le applicazioni nell'ambito della programmazione di algoritmi quantistici come ad esempio il *repeat until success*[17] e la *phase estimation*[18]. Il linguaggio è ad alto livello e agnostico riguardo l'hardware su cui verrà eseguito.

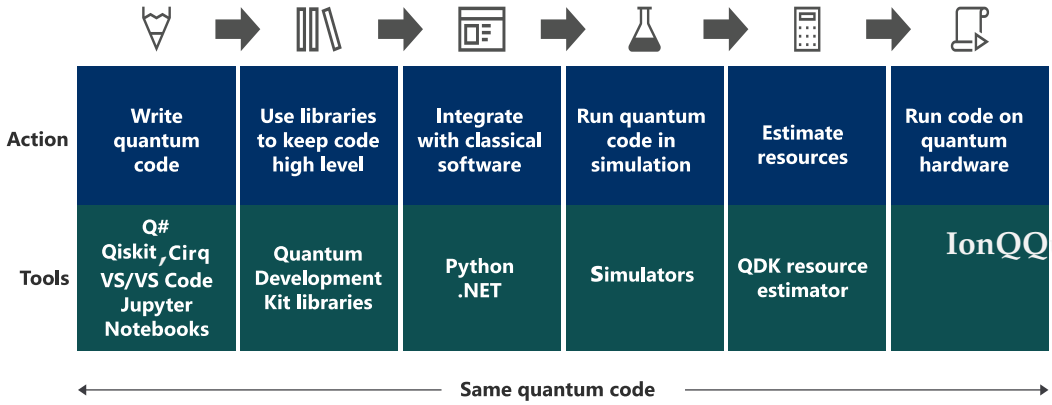

Figure 1: Diagramma che mostra i passaggi da idea a implementazione di un programma nel framework QDK, tratto dalla documentazione Microsoft QDK.

16 https://github.com/microsoft/qsharp-language

17 https://learn.microsoft.com/en-us/azure/quantum/user-guide/language/statements/conditionalloops

18 Per approfondire a riguardo si può leggere la documentazione di Microsoft Azure: https://learn.microsoft.com/en-us/azure/quantum/user-guide/libraries/standard/algorithms#quantum-phase-estimation. L'algoritmo di stima della fase quantistica o stima dell'autovalore quantistico è utilizzato per stimare con alta probabilità dato un errore le operazioni di operatori unitari U e m qubit. La *phase estimation* è spesso una subroutine di altri algoritmi quantistici, per esempio l'algoritmo di Shor, ed è un'altra applicazione della trasformata di Fourier che nominiamo nel capitolo 5 parlando della trasformata di Hadamard.

Un semplice programma in Q# può essere:

```
namespace HelloQuantum {

    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;


    @EntryPoint()
    operation SayHelloQ() : Unit {
        Message("Hello quantum world!");
    }
}
```

Questo stampa la stringa "Hello quantum world!", `EntryPoint` indica al compilatore dove inizia l'esecuzione del programma. Tra i tipi offerti dal linguaggio ci sono quelli classici: `Int`, `Double`, `Bool`, `String`. Inoltre esistono dei tipi specifici al quantum computing: `Result` rappresenta il risultato di una misurazione di qubit e può evere solamente uno di due valori - `One` o `Zero`. Il linguaggio permette di specificare nuovi tipi per un proprio programma ma non offre feature di linguaggi come C# o Java come interfacce o classi.
I qubit vengono allocati tramite la keyword `use`. Se ne possono allocare uno o diversi alla volta.

```
use q = Qubit();
```

I principali attori di un programma che manipola qubit sono le cosidette `Operations`, queste sono routine chiamabili di un programma che contengono operazioni quantistiche che manipolano lo stato del registro di qubit.

```
operation SayHelloQ() : Unit {
    Message("Hello quantum world!");
}
```

Una parte fondamentale di un qualsiasi algoritmo quantistico è la misurazione dei qubit e la loro manipolazione. Per questo vengono utilizzate le misure di Pauli per misurazioni di singoli qubit secondo una data base.

```
operation MeasureOneQubit() : Result {
    // Alloca un qubit, di default nello stato zero
    use q = Qubit();
    // Applichiamo Hadamard allo stato
    // A seguito di questa operazione la misurazione
    // potrebbe risultare 0 o 1 con uguale probabilità
    H(q);
    // Misuriamo in base Z il qubit
    let result = M(qubit);
    // Resettiamo il qubit prima di rilasciarlo
    if result == One { X(qubit); }
    return result;
}
```

Nel listato vediamo un esempio di misurazione in base Z di un qubit. Il qubit viene allocato, gli viene applicata la trasformata di Hadamard tramite la procedura `H` e poi viene misurato utilizzando `M`. `M` effettua una misura di un singolo qubit in base Z di Pauli. Questa è del tutto equivalente a `Measure([PauliZ], [qubit])`.

L'operazione di misura è spesso seguita dal reset, quindi spesso è comodo l'utilizzo dell'operazione `MResetX`[19], che si assicura che il qubit sia riportato allo stato $|0\rangle$.

```
operation MResetX (target : Qubit) : Result
```

19 https://learn.microsoft.com/en-us/qsharp/api/qsharp/microsoft.quantum.measurement.mresetx

4 ORACOLI

Per poter applicare l'algoritmo che andremo a descrivere e implementare in seguito è necessario creare dei cosiddetti **oracoli** delle funzioni che utilizzeremo come input. Prima definiamo cos'è un oracolo in questo contesto:

> Un oracolo U_f è una matrice unitaria definita applicando f condizionatamente rispetto alle etichette assegnate agli stati dei `qubit`. L'applicazione di un oracolo per due volte risulta nella matrice identità $\mathbb{1}$.

Per ottenere questo è necessaria una manipolazione per convertire funzioni *irreversibili* in oracoli *reversibili* utilizzabili in ambito quantistico. Questa manipolazione va fatta utilizzando le operazioni su `qubit` proprie di un simulatore o device quantistico come:

- $x(t)$
  - questa operazione è l'equivalente del classico `NOT`
  - $x|0\rangle = |1\rangle$
  - $x|1\rangle = |0\rangle$
- $\textsc{cnot}(c, t)$
  - questa operazione è definibile come un `NOT` controllato secondo l'input c
  - $\textsc{cnot}|00\rangle = |00\rangle$
  - $\textsc{cnot}|01\rangle = |01\rangle$
  - $\textsc{cnot}|10\rangle = |11\rangle$
  - $\textsc{cnot}|11\rangle = |10\rangle$

Dove t è il `qubit` target e c è il `qubit` di controllo per il `Controlled-NOT`.

- $\textsc{swap}(t_1, t_2)$
  - come si può intuire dal nome scambia i valori dei `qubit`
  - $\textsc{swap}|10\rangle = |01\rangle$ e $\textsc{swap}|01\rangle = |10\rangle$

Le difficoltà maggiori nella definizione di oracoli per le funzioni che ci interessano le abbiamo con quelle *costanti*, questo in quanto passando da input a output si perde l'informazione dell'input utilizzato. Rendendo tali funzioni irreversibili.

Fortunatamente esiste una tecnica generale per rendere una funzione classica irreversibile $f : \mathtt{Bool} \to \mathtt{Bool}$ in una funzione classica reversibile g.

$$h(x, y) = (x, y \oplus f(x))$$

Questa nuova funzione h aggiunge al input originario di f x un nuovo input y che non è altro che il valore di output che andrà a modificare tramite l'operazione $\oplus$[20].

Questa stessa tecnica è trasponibile per definire un oracolo U_f:

$$U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$$

In questo modo manteniamo traccia dell'input x che altrimenti andrebbe perso dopo l'applicazione di f.

20 dove $\oplus$ è l'addizione modulo 2 o `XOR`

Gli oracoli che utilizziamo per testare gli algoritmi definiti nelle prossime sezioni sono riportati nel listato `oracles.qs` (`Listing 3`).

```
operation ApplyZeroOracle(control : Qubit, target : Qubit) : Unit {
  }

  operation ApplyOneOracle(control : Qubit, target : Qubit) : Unit {
    X(target);
  }

  operation ApplyZeroOracleN(control : Qubit[], target : Qubit) : Unit {
  }

  operation ApplyOneOracleN(control : Qubit[], target : Qubit) : Unit {
    X(target);
  }

  operation ApplyIdOracle(control : Qubit, target : Qubit) : Unit {
    CNOT(control,target);
  }

  operation ApplyXOROracleN(control : Qubit[], target : Qubit) : Unit {
    for qubit in control {
        CNOT(qubit,target);
    }
  }

  operation ApplyNotOracle(control : Qubit, target : Qubit) : Unit {
    X(control);
    CNOT(control,target);
    X(control);
  }
```

Listing 3: oracles.qs

Qui sopra sono definite le versioni a singolo qbit e a n-qbit degli oracoli quantistici di alcune funzioni booleane costanti e bilanciate, definiamo cosa siano funzioni di questo tipo nel prossimo capitolo. Tutte queste funzioni hanno tipo

$$f : \mathrm{Bool}^n \to \mathrm{Bool}$$

In particolare abbiamo definito oracoli per le seguenti funzioni:

- $f_1(x) = 0$
- $f_2(x) = 1$
- $f_3(x) = x$
- $f_4(x) = \neg x$ ovvero $f_4(x) = 1 - x$
- $f_5(x) = \oplus_{i=0}^{n-1} x_i$
  - dove x è l'input lungo n `qubit`

In questi casi le prime due funzioni sono costanti e le restanti sono bilanciate. È facile verificare che gli oracoli definiti in `Q#` corrispondono alle funzioni sopra definite, in particolare:

- `ApplyZeroOracle` e la sua versione a n `qubit` equivalgono a f_1
- `ApplyOneOracle` e la sua versione a n `qubit` equivalgono a f_2
- `ApplyIdOracle` equivale all'identità f_3
- `ApplyNotOracle` equivale a f_4
- `ApplyXOROracleN` equivale a f_5

In `figura 1` vediamo un altro esempio di oracolo bilanciato che applica 3 porte `CNOT` all'ultimo qubit:

- $q_3 = q_3 \oplus q_0 \oplus q_1 \oplus q_2$

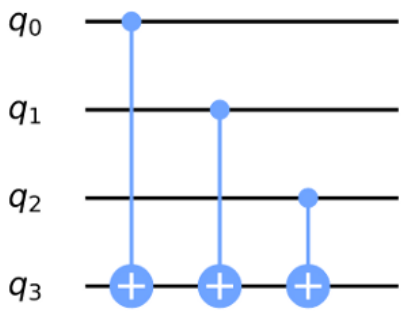

Figure 2: esempio di oracolo bilanciato utilizzando porte CNOT

La precedente definizione single `qubit` di U_f può essere estesa per il caso di f con n `qubit`

$$f(x_0, x_1, \cdots, x_{n-1})$$

in questa maniera:

$$U_f|x_0x_1\cdots x_{n-1}y\rangle = |x_0x_1\cdots x_{n-1}\rangle \otimes |f(x_0, x_1, \cdots, x_{n-1}) \oplus y\rangle$$

Il nome **oracolo** deriva da una convenzione di nomenclatura nell'ambito della Teoria della Complessità. In particolare è stata definita in quanto una classe di complessità A può essere convertita in una nuova classe di problemi A^B, che permettono ad A di risolvere problemi di tipo B in un singolo passo, proprio come se stesse consultando un oracolo.

Una *macchina oracolo* si può immaginare come una macchina di Turing connessa a un **oracolo**, in questo contesto si intende con oracolo una entità *blackbox* in grado di risolvere un qualche problema. Questo problema non deve per forza essere computabile in quanto l'oracolo non è una reale macchina o programma ma semplicemente una scatola oscura che produce una soluzione corretta per ogni istanza del problema computazionale in un singolo passo.[21]

21 https://en.wikipedia.org/wiki/Oracle_machine

5 ALGORITMO DI DEUTSCH–JOZSA

L'algoritmo di **Deutsch-Jozsa** ha interesse storico in quanto primo algoritmo quantistico in grado di superare in performance il miglior algoritmo classico corrispondente, mostrando che possono esistere vantaggi nel calcolo quantistico. Questo algoritmo ha spinto la ricerca in questa direzione per determinati problemi.

L'algoritmo risponde a una domanda su una funzione f booleana con n bit in input

$$f : \mathrm{Bool}^n \to \mathrm{Bool}$$

$$f(\{x_0, x_1, \cdots, x_n\}) \to 0 \text{ o } 1$$

Questa funzione su cui agisce l'algoritmo ha la proprietà di essere una di due forme:

- costante
- bilanciata

Definite come:

- Una funzione è **costante** se restituisce per tutti gli input $\{x_0, x_1, \cdots, x_n\}$ lo stesso risultato
- Una funzione è **bilanciata** se restituisce 0 esattamente per metà degli input, e 1 esattamente per metà degli input

Il problema di Deutsch-Jozsa è stato ideato per essere facile da risolvere con una soluzione algoritmica quantistica ed essere difficile per qualsiasi algoritmo classico.

Questo per dimostrare che un problema cosiddetto *blackbox* può essere risolto efficientemente e senza errore da un computer quantistico, risultato non possibile tramite un computer classico.

In particolare questo risultato mostra che la classe computazionale EQP (a volte chiamata QP) **Exact Quantum Polynomial Time** è distinta da P ovvero la classe dei problemi risolvibili classicamente in tempo polinomiale.

5.1 La Soluzione Classica

Nella soluzione classica nel **caso migliore** due *query* all'oracolo sono sufficienti per riconoscere la funzione f come bilanciata. Per esempio supponiamo di avere due chiamate con i seguenti risultati:

$$f(0,0,\cdots) \to 0$$

$$f(1,0,\cdots) \to 1$$

Dato che è assunto che f sia *garantita* essere costante oppure bilanciata questi risultati ci dimostrano f come bilanciata.

Per quanto riguarda il caso peggiore tutte le nostre interrogazioni daranno lo stesso output, decidere in modo certo che f sia costante necessita di metà più uno interrogazioni. Dato che il numero di input possibili è 2^n questo significa che, nel caso peggiore, saranno necessarie $2^{n-1}+1$ interrogazioni per essere certi che $f(x)$ sia costante.

È possibile una soluzione probabilistica tramite un algoritmo randomizzato, con un numero costante di valutazioni k è possibile produrre un risultato con alta probabilità corretto.

Dato $k \geqslant 1$, un algoritmo di questo tipo fallisce con probabilità

$$\epsilon \leqslant \frac{1}{2^k}$$

In ogni caso l'unico modo per avere un risultato certo rimane avere $k = 2^{n-1}+1$.

La complessità di questi algoritmi rimane $\textsc{time} = O(2^n)$ e quindi difficili da trattare al crescere della lunghezza dell'input.

5.2 La Soluzione Quantistica

La soluzione di David Deutsch e Richard Jozsa del 1992, poi migliorata nel 1998 è molto più efficace delle alternative classiche.

Tramite la computazione quantistica è possibile risolvere questo problema con un'unica chiamata della funzione $f(x)$. Questo a patto che la funzione f sia implementata come un oracolo quantistico U_f, che mappi: $|x\rangle|y\rangle$ a $|x\rangle|y \oplus f(x)\rangle$ [20]

I passi dell'algoritmo in particolare sono:

1. prepara 2 registri di `qubit`, il primo di `n` `qubit` inizializzato a $|0\rangle$ e il secondo di un singolo `qubit` inizializzato a $|1\rangle$
2. applica `Hadamard` a entrambi i registri
3. applica l'oracolo quantistico U_f definito per f
4. a questo punto il secondo registro può essere ignorato, riapplica `Hadamard` al primo registro
5. misura il primo registro, questo risulta 1 per $f(x)$ costante e 0 altrimenti nel caso bilanciato

Nei listati successivi riportiamo l'implementazione `Q#` della versione a singolo `qubit` e la generalizzazione nel caso di `n-qubit`.

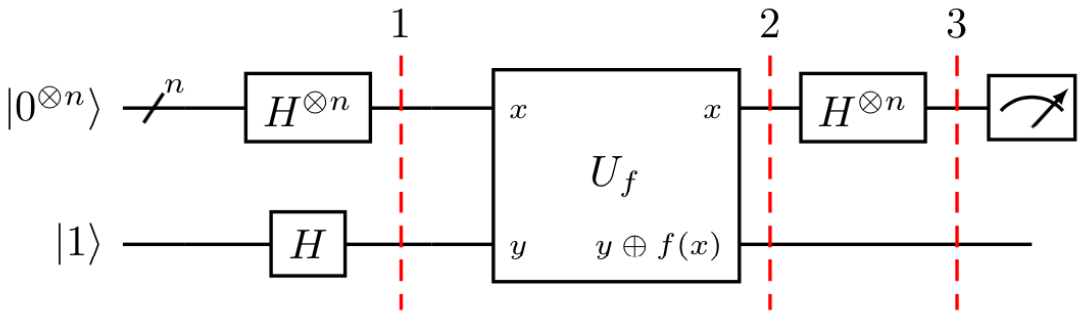

Figure 3: i passi dell'algoritmo n-qubit in forma di circuito

Un punto fondamentale dell'algoritmo è l'utilizzo della porta `Hadamard`, chiamata anche trasformata di `Hadamard`. Questa è una generalizzazione delle trasformate di Fourier definita dalla matrice $H_m = 2^m \times 2^m$. Questa è definibile ricorsivamente a partire dall'identità $H_0 = 1$ e, per $m > 0$:

$$H_m = \frac{1}{\sqrt{2}} \begin{pmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{pmatrix}$$

e quindi alcuni esempi di porte di `Hadamard` sono:

$$H_0 = +(1)$$

$$H_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

$$H_2 = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}$$

Il trasformato di `Hadamard` H_1 è la porta logica quantistica conosciuta come porta `Hadamard`, l'applicazione di questa porta a ciascun qubit di un registro a n-qubit parallelamente è equivalente alla trasformata H_n.

Applicando un circuito di `Hadamard` a un qubit nello stato $|0\rangle$ si crei uno stato sovrapposto tra gli stati $|0\rangle$ e $|1\rangle$ denominato $|+\rangle$. A livello matematico sono definite:

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

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

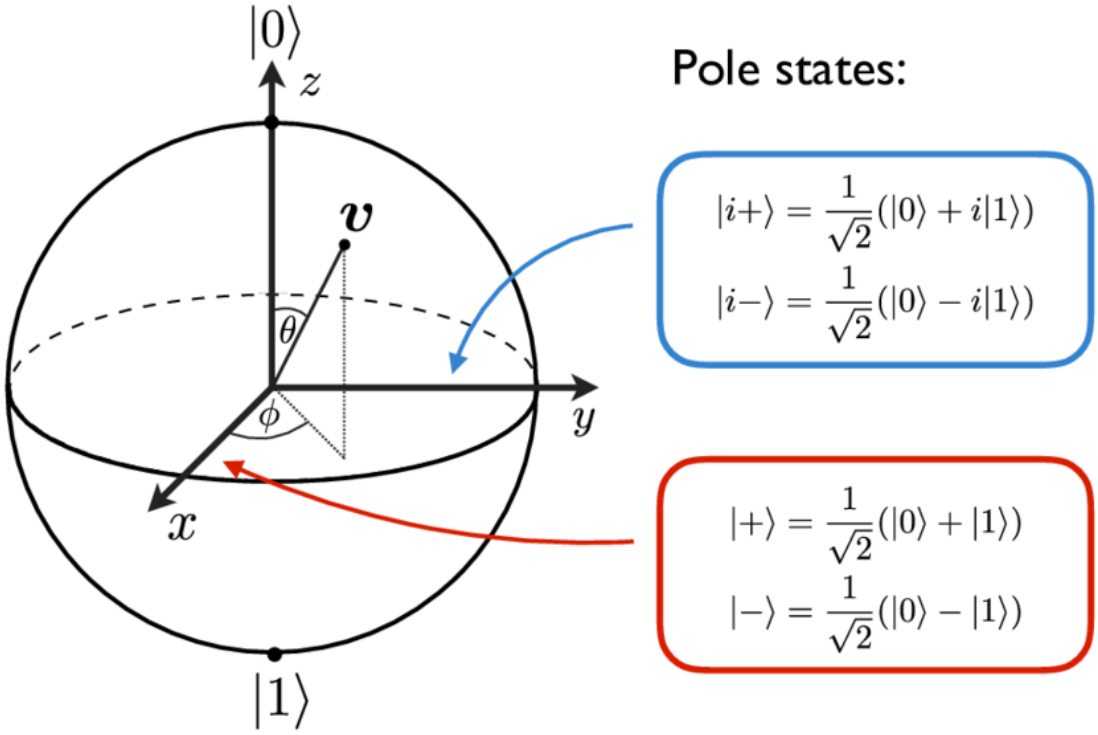

Figure 4: Rappresentazione geometrica di un qubit con la sfera di Bloch. Sono rappresentati come poli sull'asse z gli stati equivalenti allo 0 e 1 di un bit classico, sull'asse x invece i poli sono gli stati sopraccitati $|+\rangle$ e $|-\rangle$. Con questa rappresentazione è possibile notare come H non sia altro che una rotazione in questo spazio tridimensionale.

Inoltre con una funzione f applicata a questa sovrapposizione si ottiene, nel caso $n = 1$, uno stato sovrapposto tra $f(0)$ e $f(1)$. Questo effetto è utilizzato dall'algoritmo in quanto riapplicando `Hadamard` si controlla in un solo passo se si ottiene la sovrapposizione di due stati uguali o di due stati diversi, o meglio se $f(0) = f(1)$ o meno. La riapplicazione di H restituirà 1 nel primo caso, 0 nel secondo.

Seguono i calcoli per il caso a 1 qubit: L'obiettivo è controllare la condizione $f(0) = f(1)$, equivalente a controllare $f(0) \oplus f(1)$.

In questo caso lo `XOR` è implementato come una Controlled NOT gate `CNOT`.

Lo stato iniziale aggiungendo un `qubit` di controllo è $|0\rangle|1\rangle$, si applica `Hadamard` a entrambi:

$$\frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle$$

Data l'implementazione quantistica in forma di **oracolo** U_f della funzione iniziale f, che ricordiamo è definita come una mappa tra $|x\rangle|y\rangle$ e $|x\rangle|f(x) \oplus y\rangle$ La applichiamo allo stato ottenuto:

$$\begin{aligned}
&\frac{1}{2}(|0\rangle(|f(0) \oplus 0\rangle - |f(0) \oplus 1\rangle) + |1\rangle(|f(1) \oplus 0\rangle - |f(1) \oplus 1\rangle)) \\
&\qquad = \frac{1}{2}((-1)^{f(0)}|0\rangle(|0\rangle - |1\rangle) + (-1)^{f(1)}|1\rangle(|0\rangle - |1\rangle)) \\
&\qquad = (-1)^{f(0)}\frac{1}{2}(|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle)(|0\rangle - |1\rangle)
\end{aligned}$$

La fase globale -1 e il secondo `qubit` finale vengono ignorati, a questo punto della computazione si ha lo stato:

$$\frac{1}{\sqrt{2}}(|0\rangle + (-1)^{f(0)\oplus f(1)}|1\rangle)$$

Applicando nuovamente `Hadamard` otteniamo:

$$\begin{aligned}
&\frac{1}{2}(|0\rangle + |1\rangle + (-1)^{f(0)\oplus f(1)}|0\rangle - (-1)^{f(0)\oplus f(1)}|1\rangle) \\
&\qquad = \frac{1}{2}((1 + (-1)^{f(0)\oplus f(1)})|0\rangle + (1 - (-1)^{f(0)\oplus f(1)}|1\rangle)
\end{aligned}$$

A questo punto viene misurato il `qubit`:

- $f(0) \oplus f(1) = 0$ se e solo se misuriamo $|0\rangle$
- $f(0) \oplus f(1) = 1$ se e solo se misuriamo $|1\rangle$

Concludiamo che sappiamo con certezza se $f(x)$ è costante o bilanciata in un singolo uso della **black box** U_g.

```
operation DeutschJozsaSingleBit(oracle : (( Qubit, Qubit ) => Unit)) :
    ↪ Bool {
    use control = Qubit();
    use target = Qubit();

    H(control);
    X(target);
    H(target);

    oracle(control, target);

    H(target);
    X(target);

    return MResetX(control) == One;
}
```

Listing 4: single-qubit Deutsch-Jozsa

Il caso $n = 2$ `qubit` non è diverso:
L'algoritmo inizia nello stato di $n + 1$ `qubit` $|00\rangle|1\rangle$. Applicando `Hadamard` si ottiene lo stato

$$\frac{1}{\sqrt{2^3}} \sum_{x=0}^{2^2-1} |x\rangle(|0\rangle - |1\rangle)$$

Dove la sommatoria esprime le configurazioni x in `qubit` da 0 a 3

$$|00\rangle, |01\rangle, |10\rangle, |11\rangle$$

Per ciascuna di queste x, $f(x)$ vale 0 oppure 1, dato ciò la formula precedente equivale a

$$\frac{1}{\sqrt{2^3}} \sum_{x=0}^{3} (-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle)$$

A questo punto l'ultimo `qubit` $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ possiamo ignorarlo, ottenendo

$$\frac{1}{\sqrt{2^2}} \sum_{x=0}^{3} (-1)^{f(x)}|x\rangle$$

Riapplichiamo `Hadamard` a tutti gli $n = 2$ `qubit`

$$\begin{aligned} &\frac{1}{\sqrt{2^2}} \sum_{x=0}^{3} (-1)^{f(x)} \left[\frac{1}{\sqrt{2^2}} \sum_{y=0}^{3} (-1)^{x \cdot y}|y\rangle \right] \\ =&\frac{1}{2^2} \sum_{y=0}^{3} \left[\sum_{x=0}^{3} (-1)^{f(x)}(-1)^{x \cdot y} \right] |y\rangle \end{aligned}$$

Dove $x \cdot y = x_0y_0 \oplus x_1y_1 \oplus x_2y_2 \oplus x_3y_3$, somma modulo 2 del prodotto bit a bit.
Data questo risultato la probabilità di misurare $|00\rangle$ è

$$\left| \frac{1}{2^2} \sum_{x=0}^{3} (-1)^{f(x)} \right|^2$$

Che risulta 1 se $f(x)$ è costante e 0 se altrimenti $f(x)$ è bilanciata.
In altre parole, la misura finale sarà $|00\rangle$ se $f(x)$ è costante e un qualche altro stato nel caso in cui $f(x)$ sia bilanciata.

```
operation DeutschJozsa(size : Int, oracle : ((Qubit[], Qubit ) => Unit) )
  ↪  : Bool {
    use control = Qubit[size];
    use target = Qubit();

    ApplyToEachA(H, control);
    X(target);
    H(target);

    oracle(control, target);

    H(target);
    X(target);

    let result = MResetX(control[0]) == One;
    ResetAll(control);
    return result;
}
```

Listing 5: n-qubit Deutsch-Jozsa

6 TELETRASPORTO QUANTISTICO

Concludiamo la nostra trattazione mostrando un interessante fenomeno quantistico che riguarda la sovrapposizione di più stati, la **correlazione quantistica** o **entanglement quantistico**.

Il termine *entanglement*, traducibile come groviglio in italiano, fu introdotto dal nobel per la fisica Erwin Schrödinger, i cui contributi alla meccanica quantistica furono fondamentali e indica la forte relazione che due particelle *entangled* in un sistema quantistico mantengono secondo la **legge di conservazione**.

Questa legge continua a valere senza alcun limite spaziale, permettendo che la misura di una singola particella influenzi istantaneamente il corrispondente valore dell'altra.

Un altro risultato fondamentale della meccanica quantistica sono il **teorema di no-cloning** e quello di **non discriminazione**. Il primo vieta la creazione di un duplicato esatto di uno stato quantistico sconosciuto, il secondo afferma che dati due stati quantistici non ortogonali di un sistema non sia possibile distinguerli con certezza.

Non siamo in grado di distinguere due particelle elementari: se queste fossero scambiate sarebbe impossibile accorgersene. Non ha senso affermare che le particelle elementari abbiano una individualità.

È più corretto affermare che le due posizioni nello spazio hanno la proprietà di avere campi quantistici nello stesso stato.

Partendo da uno stato *entangled* è però possibile *teletrasportare* lo stato di una particella A in una particella B.

Per effetto del teletrasporto lo stato di B sarà esattamente quello che aveva A precedentemente all'operazione. In letteratura questi attori sono spesso soprannominati `Alice` e `Bob`.

Lo stesso risultato si potrebbe avere trasportando fisicamente A al posto di B. Per le proprietà dell'*entanglement* non c'è un limite spaziale al teletrasporto quantico.

Gli schemi per effettuare questa operazione sono diversi, noi riportiamo quello più semplice: il teletrasporto di un `qubit`.

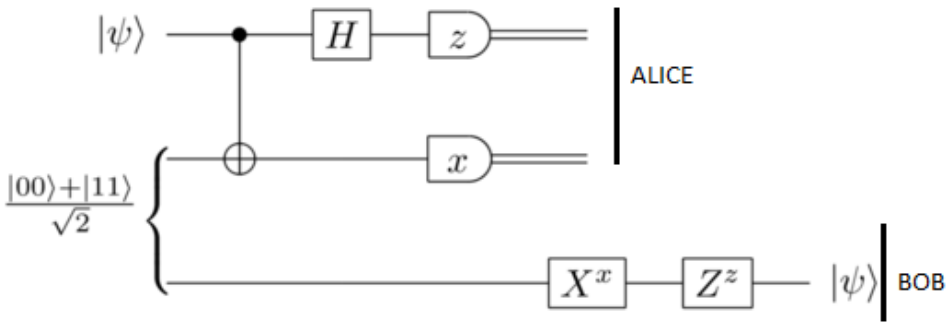

Figure 5: Trasposizione in circuito delle operazioni necessarie al teletrasporto quantistico.

Definiamo:

- $|\psi\rangle_{A1} = \alpha|0\rangle_{A1} + \beta|1\rangle_{A1}$ è il generico stato da teletrasportare
- A2 `qubit` *entangled* di `Alice`
- B `qubit` *entangled* di `Bob`

Lo stato si inizializza in

$$\frac{1}{\sqrt{2}}|\psi\rangle_{A1}(|0\rangle_{A2}|1\rangle_B - |1\rangle_{A2}|0\rangle_B)$$

Si riscrive lo stato complessivo in

$$\begin{aligned}
&-\frac{1}{2}(|0\rangle_{A1}|1\rangle_{A2} - |1\rangle_{A1}|0\rangle_{A2})(\alpha|0\rangle_B + \beta|1\rangle_B)\\
&-\frac{1}{2}(|0\rangle_{A1}|1\rangle_{A2} - |1\rangle_{A1}|0\rangle_{A2})(\alpha|0\rangle_B - \beta|1\rangle_B)\\
&+\frac{1}{2}(|0\rangle_{A1}|0\rangle_{A2} - |1\rangle_{A1}|1\rangle_{A2})(\beta|0\rangle_B + \alpha|1\rangle_B)\\
&-\frac{1}{2}(|0\rangle_{A1}|0\rangle_{A2} - |1\rangle_{A1}|1\rangle_{A2})(\beta|0\rangle_B - \alpha|1\rangle_B)
\end{aligned}$$

`Alice` può ridurre attraverso una misura di Bell lo stato di `Bob` a uno dei quattro stati con coefficienti α e β. `Bob` non può comunque ancora distinguere in quale dei quattro stati il proprio `qubit` si trovi, per questo è necessario un ulteriore passo.

La trasmissione dell'informazione avviene quando `Alice` comunica a `Bob` il risultato della misura. A questo punto `Bob` può effettuare una trasformazione unitaria opportuna che trasformi il proprio stato in quello stato $|\psi\rangle$ originario.

Questo ultimo passaggio è fondamentale: per poter ricostruire lo stato iniziale il destinatario deve conoscere il risultato di una misurazione del mittente, questa informazione viene trasmessa attraverso un mezzo di trasmissione classico. La trasmissione di questa misura limita la velocità del teletrasporto che non è quindi istantaneo, ma limitato dalla velocità della luce in accordo con la relatività speciale.

La misurazione da parte di `Alice` porta alla perdita dello stato iniziale rispettando quindi il **teorema di no-cloning** citato precedentemente.

Molti esperimenti sono stati effettuati nell'ambito del teletrasporto quantistico, l'attuale record di distanza per un esperimento di questo tipo è stato registrato in un esperimento all'aperto che ha avuto luogo nelle isole Canarie e teletrasportò particelle tra due osservatori astronomici dell'*Instituto de Astrofísica de Canarias* ad una distanza di 143Km.

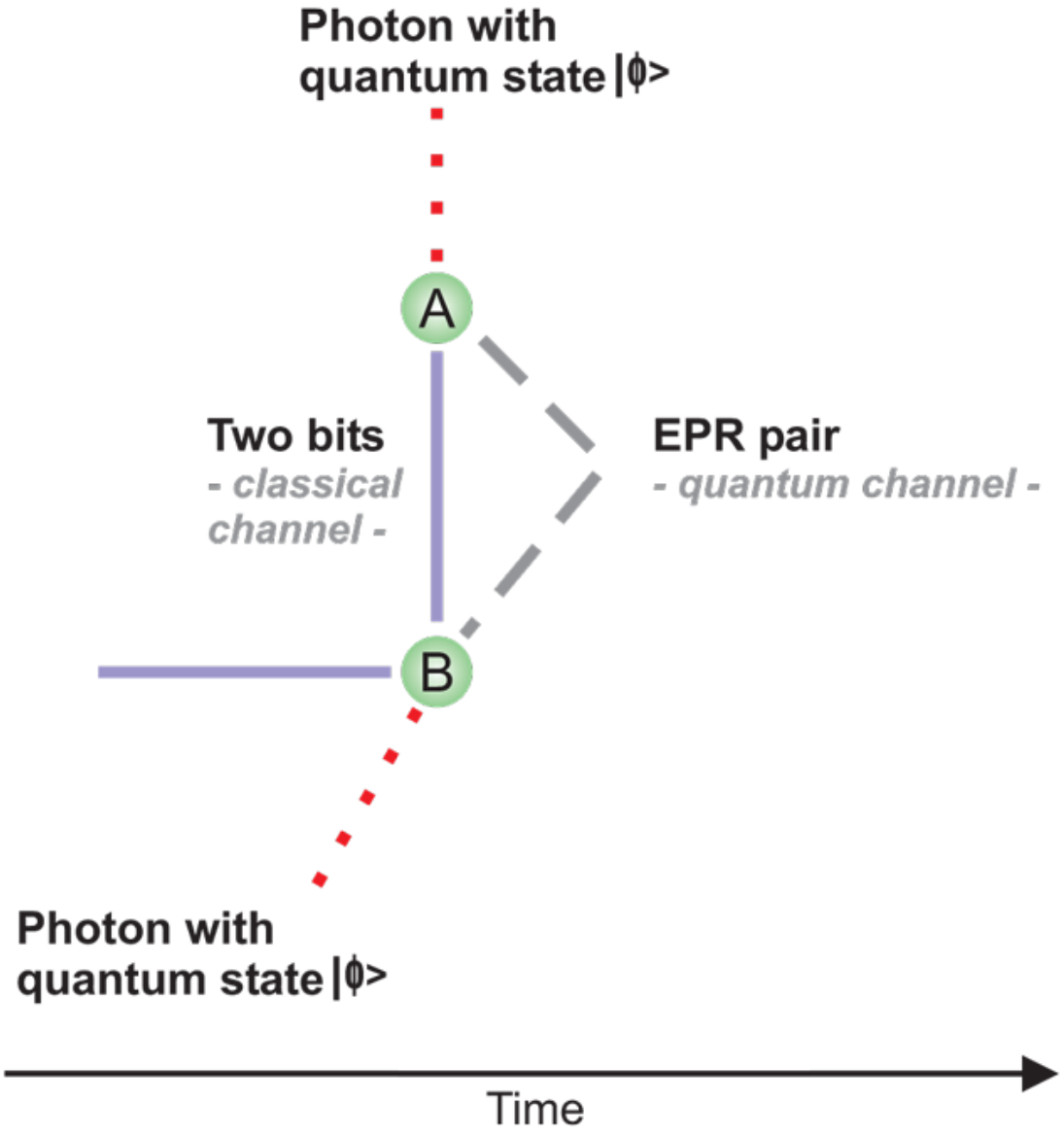

Figure 6: Il teletrasporto quantistico agisce in accordo alla relatività speciale con l'utilizzo di due canali di comunicazione, uno classico e una quantistico.

```python
from interface import QuantumDevice, Qubit
from simulator import Simulator

# parametri:
# msg qubit che vogliamo muovere
# here qubit temporaneo
# there qubit di destinazione
# here, there sono inizializzati nello stato |0⟩
def teleport(msg: Qubit, here: Qubit, there: Qubit) -> None:
    here.h()
    here.cnot(there)

    msg.cnot(here)
    msg.h()

    # Il risultato della misura è informazione classica
    # che verrà trasmessa al ricevente attraverso un
    # mezzo di comunicazione classico
    if msg.measure(): there.z()
    if here.measure(): there.x()

    msg.reset()
    here.reset()
```

Listing 6: Programma di teletrasporto in python

```
1 //
  ↪ https://github.com/microsoft/quantum/tree/main/samples/getting-started/teleportatio
2 namespace Microsoft.Quantum.Samples.Teleportation {
3     open Microsoft.Quantum.Intrinsic;
4     open Microsoft.Quantum.Canon;
5     open Microsoft.Quantum.Measurement;
6
7     operation Teleport (msg : Qubit, target : Qubit) : Unit {
8         use register = Qubit();
9
10        H(register);
11        CNOT(register, target);
12
13        CNOT(msg, register);
14        H(msg);
15
16        // misurando con MResetZ resettiamo nello
17        // stesso passo i qubit misurati rendendoli
18        // utilizzabili nuovamente se necessario
19        if (MResetZ(msg) == One) { Z(target); }
20        if (IsResultOne(MResetZ(register))) { X(target); }
21    }
22 }
```

Listing 7: Programma di teletrasporto in Q#, tratto dai samples nella documentazione Microsoft

7 CONCLUSIONI

L'area di ricerca sui fenomeni quantistici e le sue possibilità a livello computazionale è in crescita e offre grandi possibilità e spunti in quanto relativamente giovane, nascendo negli anni 80 con il primo modello quantistico della macchina di Turing[22] creato dal fisico Paul Benioff. L'Unione Europea ha inserito lo sviluppo delle tecnologia di quantum computing tra gli obiettivi strategici del Decennio Digitale Europeo, in cui la Commissione Europea presenta una strategia per lo sviluppo e la digitalizzazione da raggiungere dagli stati membri nel entro il 2030. Inoltre sempre su questo tema gli stati membri hanno firmato la Dichiarazione Europea sull'Infrastruttura per la Comunicazione Quantistica - `EuroQCI` - per costruire una rete di comunicazione condivisa tra le macchine quantistiche sul territorio. Tutto questo per garantire competitività tecnologica sul piano mondiale, anche a livello di cybersicurezza nel cui ambito sono tanti gli sforzi di ricerca in campo quantistico.

Dopo 50 anni in cui la velocità di calcolo dei calcolatori classici ha continuato a crescere raddoppiando circa ogni due anni - in accordo con la legge di Moore - i componenti dei computer stanno raggiungendo i limiti fisici del progresso tecnologico e ingegneristico in quella direzione, con componentistiche nelle dimensioni dell'atomo.

Per questo negli ultimi anni è cresciuta la necessità di sviluppare nuovi modelli computazionali che permettano di andare oltre il modello classico.

È da questa necessità che negli ultimi anni l'interesse per le tecnologie quantistiche è cresciuto rapidamente e certamente continuerà a evolversi intanto che il computer classico potrebbe essere vicino a raggiungere i propri limiti fisici.

22 Paul Benioff - *The computer as a physical system* (https://link.springer.com/article/10.1007/BF01011339)