

UNIVERSITÀ DI PISA

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

Tesi di Laurea

QUANTUM GAME DESIGN: ESPLORAZIONE DI LABIRINTI CON L'ALGORITMO DI GROVER

Relatori:

Prof. ANNA BERNASCONI
Prof. GIANNA M. DEL CORSO
Dott. ALESSANDRO BERTI

Candidato:

ALESSIO LA GRECA

ANNO ACCADEMICO 2020 - 2021

*A Manu, un faro che non mi ha mai abbandonato in tutti
questi anni. “Sempre per sempre”.*

Alessio La Greca

Indice

Introduzione	iii
1 Fondamenti della computazione quantistica	1
1.1 Qubit	1
1.1.1 Qubit vs Bit	1
1.1.2 Sovrapposizione	2
1.1.3 Misurazione	3
1.2 Porte quantistiche a singolo qubit	3
1.2.1 Porta NOT o X	4
1.2.2 Porta R_x	4
1.2.3 Porta H	6
1.2.4 Porte Y e Z	6
1.3 Porte quantistiche a più qubit	6
1.3.1 Porta C-NOT	6
1.3.2 Porta Toffoli	7
1.3.3 Entanglement	8
1.3.4 Uncomputation	9
1.4 I qubit odierni	9
1.4.1 Sistemi fisici per la rappresentazione di qubit	9
1.4.2 Rumore nei computer quantistici	10
2 L'algoritmo di Grover	11
2.1 L'oracolo	11
2.1.1 Differenza tra conoscere una soluzione e saper riconoscere una soluzione	11
2.1.2 Costruire un oracolo	12
2.2 L'iterazione di Grover	14
2.3 Analisi dell'algoritmo	16
3 Stato dell'arte dei videogiochi quantistici	25
3.1 Il contributo di James Wootton	25
3.1.1 I primi videogiochi quantistici	25
3.1.2 Quantum Supremacy e Quantum Awesomeness	27
3.1.3 Il primo Quantum Game Jam	31
3.1.4 Generazione di mondi casuali	33
3.2 Altri esempi di quantum games	33
3.2.1 Quantum Game Chess	34
3.2.2 Quantum Pokémon Fight	35
3.3 L'universo quantistico di Outer Wilds	36
4 Esplorazione di dungeon mediante Grover	39
4.1 Esplorazione (quantistica) di un dungeon a quattro stanze	40
4.2 Esplorazione di un dungeon a otto stanze	43
4.2.1 Implementazione classica	43

4.2.2	Implementazione quantistica	46
4.2.3	Possibili generalizzazioni	49
5	Sviluppi futuri	51
A	Codice quantistico per l'esplorazione di un labirinto a quattro stanze	55
B	Codice classico per l'esplorazione di un labirinto a otto stanze	61
C	Codice quantistico per l'esplorazione di un labirinto a otto stanze	75

Introduzione

La necessità di computer più veloci e efficienti è sempre più richiesta oggigiorno per i più disparati motivi. La storia della crittografia ad esempio è stata fortemente influenzata dai miglioramenti tecnologici delle nostre macchine. Basti pensare che il cifrario di Vigenère, pubblicato nel 1586 e ritenuto estremamente sicuro per quasi trecento anni, oggi può essere forzato facilmente mediante un attacco di forza bruta svolto anche con un comune computer casalingo. Nel più recente periodo abbiamo invece la forzatura del cifrario DES, avvenuta nel 2008 con la macchina RIVYERA, un super-calcolatore costruito con il solo scopo di rompere il DES in meno di ventiquattr'ore (per maggiori informazioni si veda il Capitolo 7.3 di [3]). Alcuni scopi più nobili relativi ai miglioramenti dei nostri dispositivi sono ritrovabili nell'ambito del gaming videoludico, ove memorie più capienti, processori più performanti e schede grafiche migliori permettono la produzione di giochi sempre più capaci di stupire i giocatori. Tuttavia non possiamo più fare affidamento come una volta sulla sola legge di Moore [16], che afferma che il numero di transistor nei microprocessori va a raddoppiare ogni diciotto mesi circa, per ottenere calcolatori più potenti. Esistono infatti limiti fisici legati alle dimensioni dei transistor che non possono essere superati con mezzi classici. Ci sono quindi due strade percorribili: sfruttare la potenza di calcolo a più microprocessori offerta da servizi di cloud e calcolo parallelo, oppure iniziare a padroneggiare i principi della meccanica quantistica per la costruzione dei cosiddetti computer quantistici. Il *Quantum Computing* di fatto promette di velocizzare alcune operazioni di calcolo, basandosi su un modello di rappresentazione dei dati quantistico che ha come unità fondamentale i *qubit* al posto dei bit. Gli sviluppi che questa tecnologia ha visto negli ultimi anni sono più che promettenti: nel 2019 IBM ha presentato il primo quantum computer commerciale da 20 qubit, il *Q System One* [17], mentre un anno prima Intel ha sviluppato *Tangle Lake*, un processore quantistico da 49 qubit [7].

Dato questo panorama, la tesi si concentrerà sulla possibilità di utilizzare algoritmi quantistici al servizio dei videogiochi, ed è sviluppata nel modo seguente:

1. Nel Capitolo 1 introdurremo nozioni di base del Quantum Computing, come i concetti di *qubit* e *porte quantistiche*.
2. Nel Capitolo 2 spiegheremo nel dettaglio il funzionamento dell'*algoritmo di Grover*, un algoritmo quantistico che permette di cercare uno elemento che soddisfa certe proprietà all'interno di uno spazio di ricerca non ordinato, garantendo uno *speedup* quadratico rispetto alle controparti classiche.
3. Nel Capitolo 3 daremo uno sguardo al panorama dei videogiochi quantistici moderni, analizzando cosa è stato fatto e le direzioni future di questo mondo.
4. Nel Capitolo 4 proporremo una nostra implementazione quantistica di un algoritmo che permette di simulare l'esplorazione di un labirinto ostile, dove l'ambiente è parzialmente osservabile e non deterministico. Confronteremo inoltre questo algoritmo con la sua versione classica, mostrando vantaggi e svantaggi dell'uno e dell'altro approccio.

Capitolo 1

Fondamenti della computazione quantistica

In questo capitolo andremo ad introdurre alcune nozioni necessarie alla comprensione del *Quantum Computing*. Nella Sezione 1.1 parleremo dei qubit e delle loro proprietà. Nella Sezione 1.2 vedremo alcune porte quantistiche a singolo qubit. Nella Sezione 1.3 tratteremo di porte a più qubit, esaminando anche i concetti di *entanglement* e *uncomputation*. Concluderemo, nella Sezione 1.4, accennando i sistemi fisici che ad oggi permettono di implementare i qubit e gli effetti del rumore. Per i lettori più interessati all'argomento consigliamo la lettura di *Quantum Country* [10] per un'introduzione al Quantum Computing, all'algoritmo di Grover e al teletrasporto. Per chi invece volesse consultare testi più avanzati consigliamo [11] e [33].

1.1 Qubit

1.1.1 Qubit vs Bit

Nei computer classici il *bit* costituisce l'unità di informazione fondamentale. Questo vuol dire che qualunque programma, scritto in qualunque linguaggio, così come qualsiasi immagine, video, audio etc.. che conserviamo nella nostra memoria fisica può essere visto, a basso livello, come una sequenza di bit univoca. I bit possono trovarsi solo in uno di due stati: 0 o 1. Nella computazione quantistica tuttavia l'unità fondamentale di informazione cambia; dai bit si passa ai *qubit*, abbreviazione di *quantum bit*. Un qubit può essere visto come un vettore nello spazio vettoriale \mathbb{C}^2 , e può trovarsi in molti stati. Di questi, due corrispondono agli stati 0 o 1 del bit classico:

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

dove lo stato $|0\rangle$ di un qubit corrisponde allo stato 0 di un bit, e lo stato $|1\rangle$ di un qubit corrisponde allo stato 1 di un bit. Per differenziare un qubit da un bit abbiamo utilizzato la cosiddetta notazione *bra-ket*, o notazione di *Dirac*, dal nome del suo inventore. Il simbolo $|0\rangle$ può quindi essere letto come *ket-0*, e indica lo stato di un qubit. Allo stesso modo il vettore $\langle 0|$, che si legge *bra-0*, consiste nel trasposto coniugato di $|0\rangle$.

Gli stati $|0\rangle$ e $|1\rangle$ sono talmente importanti nel Quantum Computing che hanno un nome tutto loro: sono infatti chiamati *stati di base computazionale*. Questo perché ogni qubit $|q\rangle$, nella sua forma vettoriale, può essere riscritto come una combinazione lineare di questi due stati:

$$|q\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \alpha|0\rangle + \beta|1\rangle,$$

dove i termini α e β sono chiamati *ampiezze* di $|0\rangle$ e $|1\rangle$, e sono numeri complessi sottoposti al vincolo $|\alpha|^2 + |\beta|^2 = 1$ (le ragioni di questo vincolo verranno spiegate nella Sezione 1.1.3). Se ci fermiamo un

attimo a pensare, di fatto, ci rendiamo conto che essendo i qubit dei vettori bidimensionali in uno spazio complesso, questi possono sempre essere espressi come una combinazione lineare dei vettori della base canonica. Tuttavia nulla ci vieta di cambiare base per esprimere i nostri qubit in modo diverso. In particolare, la base composta dai vettori $|0\rangle$ e $|1\rangle$ viene chiamata base Z, ma ne esistono altre due molto famose:

- La base X è costituita dai vettori $|+\rangle$ e $|-\rangle$, ovvero:

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

- La base Y è costituita dai vettori $|R\rangle$ e $|L\rangle$, ovvero:

$$|R\rangle = \frac{|0\rangle + i|1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ i \end{bmatrix}, \quad |L\rangle = \frac{|0\rangle - i|1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -i \end{bmatrix}.$$

Usando coordinate sferiche polari è possibile visualizzare un singolo qubit come un vettore di lunghezza unitaria nella cosiddetta *sfera di Bloch*, visibile nella Figura 1.1. Nella figura è possibile inoltre vedere le tre basi X, Y e Z, dove i semiassi positivi indicano rispettivamente gli stati $|0\rangle$, $|+\rangle$ e $|R\rangle$, mentre quelli negativi indicano $|1\rangle$, $|-\rangle$ e $|L\rangle$.

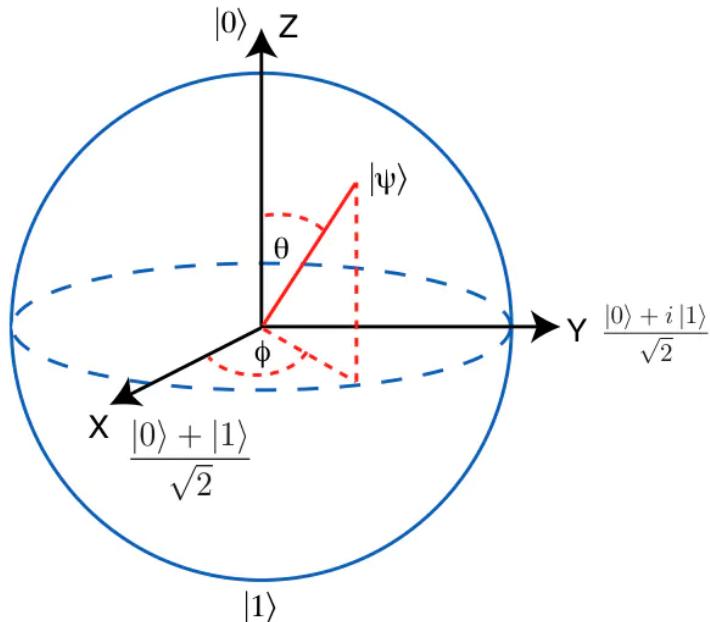


Figura 1.1: La sfera di Bloch

Se vogliamo rappresentare un byte, tutto ciò di cui abbiamo bisogno sono otto bit, ovvero otto cifre. Se invece vogliamo rappresentare un qubyte, abbiamo bisogno di un vettore appartenente allo spazio $\mathbb{C}^{2^8} = \mathbb{C}^{256}$. Quindi ci servirebbero 256 numeri complessi per rappresentare un qubyte. Questo aumento esponenziale delle cifre (e quindi della memoria) necessarie per la sola rappresentazione di un registro a più qubit rappresenta uno dei motivi per cui oggi si sta cercando di realizzare computer quantistici. È infatti possibile eseguire computazioni basate su qubit anche su computer classici, di fatto simulando l'esecuzione su computer quantistico, ma questo porta a una perdita esponenziale dell'efficienza.

1.1.2 Sovrapposizione

Abbiamo accennato al fatto che un qubit può trovarsi in molti più stati rispetto a un bit classico. In generale, un qubit $|q\rangle$ si troverà nello stato:

$$|q\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \alpha|0\rangle + \beta|1\rangle, \quad \alpha, \beta \in \mathbb{C}, \quad |\alpha|^2 + |\beta|^2 = 1.$$

In particolare possiamo dire che l'ampiezza di $|0\rangle$ è α , mentre quella di $|1\rangle$ è β . Quando α è pari a 1, β è pari a 0, e il qubit si trova nello stato di base computazionale $|0\rangle$, mentre quando questi valori delle ampiezze si invertono, ovvero quando $\alpha = 0$ e $\beta = 1$, il qubit si trova nello stato di base computazionale $|1\rangle$. Quando sia α che β hanno un valore diverso da 0, si dice che il qubit è in uno stato di *sovraposizione*. Per convenzione, le computazioni quantistiche partono con tutti i qubit nello stato $|0\rangle$, e passando attraverso specifiche porte, che vedremo nella prossima sezione di questo capitolo, possono entrare in uno stato di sovrapposizione. In tutto questo, vi è un vincolo che lo stato di un qubit deve rispettare: *il vettore che lo rappresenta deve essere unitario*, ovvero la sua norma (lunghezza) deve essere pari a uno. In formule, stiamo chiedendo che:

$$\| |q\rangle \| = \left\| \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \right\| = \sqrt{|\alpha|^2 + |\beta|^2} = 1,$$

quindi ci basta verificare che:

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

Il motivo di ciò verrà spiegato nella prossima sezione. È da notare inoltre che quindi gli stati di un qubit sono infiniti, ma non coincidono con gli infiniti vettori dello spazio \mathbb{C}^2 .

1.1.3 Misurazione

In generale, se qualcuno ci fornisce un qubit preparato in uno stato di sovrapposizione, non abbiamo modo di scoprire i valori di α e β . Ma possiamo comunque ottenere dell'informazione classica da un qubit mediante un processo conosciuto col nome di *misurazione*. Misurando un qubit possiamo ottenere come risultato solo 0 oppure 1, anche se questo si trova in uno stato di sovrapposizione. In particolare, dato il qubit $|q\rangle$ nello stato:

$$|q\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \alpha |0\rangle + \beta |1\rangle,$$

la probabilità di misurarla in un bit classico e ottenere come risultato 0 è pari a $|\alpha|^2$, mentre la probabilità di ottenere 1 come risultato della misurazione è pari a $|\beta|^2$. Dato che stiamo parlando di probabilità, la loro somma deve fare 1, ovvero:

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

che è il motivo per cui avevamo richiesto che il vettore rappresentativo di un qubit fosse unitario. Facciamo notare, come ultima cosa, che nella misurazione di un qubit contano non solo le ampiezze, ma anche la base computazionale scelta. Prendiamo per esempio il qubit $|q\rangle$, dove:

$$|q\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1 |0\rangle + 0 |1\rangle = 1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 0 \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Se misuriamo questo qubit rispetto alla base Z, sicuramente otteniamo come risultato 0. Lo stesso qubit lo possiamo però esprimere rispetto ai vettori $|+\rangle$ e $|-\rangle$ della base X, ovvero:

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

dove $\alpha = \beta = \frac{1}{\sqrt{2}}$. Quindi, se misuriamo il qubit $|q\rangle$ rispetto alla base X, la probabilità di leggere 0 è $\left| \frac{1}{\sqrt{2}} \right|^2 = \frac{1}{2}$, così come quella di leggere 1. Essendo consapevoli di questo, nel resto del capitolo supporremo che tutte le nostre misurazioni vengano fatte rispetto alla base Z.

1.2 Porte quantistiche a singolo qubit

Un qubit che si trova sempre nello stesso stato non è molto interessante, pertanto ci piacerebbe avere a disposizione degli strumenti per manipolarli. Questi strumenti vengono chiamati *porte*, e sono del tutto analoghe a quelle usate per i bit classici. Normalmente infatti un circuito classico è composto da una serie di bit che cambiano nel tempo attraversando porte NOT, AND, OR, XOR, etc.. Nel caso quantistico la

cosa non è tanto diversa: possiamo costruire un circuito quantistico a uno o più qubit, modificando poi questi ultimi con delle porte. Analizzeremo pertanto in questa sezione alcune porte a singolo qubit, per poi concentrarci nella prossima in quelle che coinvolgono due o più qubit contemporaneamente.

1.2.1 Porta NOT o X

La prima porta che esaminiamo è la porta NOT, chiamata anche porta X, che è parzialmente analoga a quella classica. Essa infatti porta un qubit che si trova nello stato $|0\rangle$ nello stato $|1\rangle$ e viceversa. Ma cosa succede ad un qubit che si trova in uno stato di sovrapposizione? Per capirlo, dobbiamo prima esaminare la *rappresentazione matriciale* della porta NOT:

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

Ogni porta quantistica infatti può essere rappresentata come una matrice quadrata. In generale, data una porta che accetta in ingresso n qubit, questa avrà dimensione $2^n \times 2^n$. Nel caso delle porte a singolo qubit quindi abbiamo a che fare con matrici 2×2 . Questa rappresentazione matriciale non dovrebbe stupirci più di tanto: dato che operiamo con dei vettori complessi, è naturale che per manipolarli ci sia bisogno di matrici a coefficienti complessi. Una cosa meno immediata da capire invece è che tutte le matrici che rappresentano porte quantistiche devono essere *unitarie*. Una generica matrice quadrata U si dice unitaria quando:

$$U^\dagger U = I,$$

dove con I intendiamo la matrice identità, mentre con U^\dagger intendiamo la matrice trasposta coniugata di U . Talvolta U^\dagger viene anche chiamata l'*aggiunta* di U . Il motivo per cui richiediamo che queste matrici siano unitarie è che, moltiplicandole per un qualunque vettore, la lunghezza di quest'ultimo rimane inalterata, ovvero:

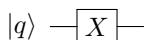
$$\|\psi\rangle\| = \|U|\psi\rangle\| = 1,$$

dove con $|\psi\rangle$ intendiamo un generico vettore che abbia la stessa dimensione della matrice U , che nel nostro caso potrebbe essere un qubit o un registro di questi. Il fatto che un qubit passi attraverso una porta rappresentata da una matrice unitaria è un bene, perché ciò vuol dire che la sua lunghezza (pari a uno) rimane inalterata, e quindi la somma delle ampiezze sotto modulo e al quadrato (ovvero la somma delle probabilità) rimane sempre uno.

Fissati questi concetti, possiamo rispondere alla domanda iniziale: come si comporta la porta X quando applicata a un qubit in uno stato di sovrapposizione? Immaginiamo di avere il nostro solito qubit $|q\rangle$ dove le ampiezze sono α e β . Allora, applicando la porta X, otteniamo:

$$X|q\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}.$$

Quindi l'effetto netto è che i valori delle ampiezze si invertono. L'ultima cosa che dobbiamo considerare per completezza è la rappresentazione circuitale della porta X:



In generale, in un circuito quantistico, rappresenteremo con un quadrato una porta quantistica a singolo qubit, dove la lettera all'interno identifica la particolare porta usata, mentre con una linea orizzontale un semplice cavo, che può essere anche visto come l'applicazione della matrice identità I.

1.2.2 Porta R_x

La porta X in realtà è un caso particolare di una porta più generale, la porta R_x . Abbiamo infatti visto, con la Figura 1.1, che un qubit può essere visualizzato come un vettore in una sfera unitaria, dove abbiamo anche evidenziato gli assi X, Y e Z. Ebbene, la porta $R_x(\phi)$, che vuole come parametro un angolo, da un

punto di vista geometrico non fa altro che prendere il vettore rappresentativo di un qubit e ruotarlo di ϕ gradi rispetto all'asse X. La sua rappresentazione matriciale in particolare è:

$$R_x(\phi) = \begin{bmatrix} \cos(\frac{\phi}{2}) & -i \sin(\frac{\phi}{2}) \\ -i \sin(\frac{\phi}{2}) & \cos(\frac{\phi}{2}) \end{bmatrix},$$

dove la porta X corrisponde alla porta $R_x(\phi)$ con $\phi = \pi$. Per esserne sicuri tuttavia svolgiamo un secondo i calcoli:

$$R_x(\pi) = \begin{bmatrix} \cos(\frac{\pi}{2}) & -i \sin(\frac{\pi}{2}) \\ -i \sin(\frac{\pi}{2}) & \cos(\frac{\pi}{2}) \end{bmatrix} = \begin{bmatrix} 0 & -i \\ -i & 0 \end{bmatrix} = -i \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Otteniamo dunque che la porta $R_x(\phi)$ è uguale alla porta X, a meno di una costante moltiplicativa $-i$. Potremmo allora concludere che non è vero che le due porte sono uguali, ma saremmo nel torto, perché dobbiamo considerare un altro elemento della computazione quantistica: il *fattore di fase globale*. Con questo termine andiamo ad indicare un numero complesso $e^{i\theta}$ che, se moltiplicato per il vettore rappresentativo di un qualunque qubit, non cambia le nostre probabilità di misura. Più formalmente dovremmo dire che, dato un generico qubit $|\psi\rangle$, il vettore $e^{i\theta}|\psi\rangle$ è equivalente al vettore $|\psi\rangle$ a meno di un fattore di fase globale. Questo vuol dire che la probabilità di misurare 0 nel primo caso è uguale a quella di misurare 0 nel secondo, e lo stesso dicasì per la probabilità di misurare 1. Infatti, applicando la porta X a un qubit $|\psi\rangle$, così come l'abbiamo conosciuta nella Sezione 1.2.1, otterremo il vettore $|\psi\rangle'$, dove:

$$|\psi\rangle' = X|\psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix},$$

mentre applicando la porta $R_x(\pi)$ otterremo il vettore $|\psi\rangle''$, definito come:

$$|\psi\rangle'' = R_x(\pi)|\psi\rangle = -i \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = -i \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \begin{bmatrix} -i\beta \\ \alpha \end{bmatrix}.$$

Ora, definendo $\alpha = a_1 + ia_2$, $\beta = b_1 + ib_2$ e facendo i calcoli, ci rendiamo conto che le probabilità di misura sono le stesse in entrambi i casi:

$$\begin{aligned} |\alpha|^2 &= |a_1 + ia_2|^2 = \left(\sqrt{a_1^2 + a_2^2} \right)^2 = a_1^2 + a_2^2, \\ |-i\alpha|^2 &= |-i(a_1 + ia_2)|^2 = |-ia_1 - i^2 a_2|^2 = |a_2 - ia_1|^2 = \left(\sqrt{a_1^2 + a_2^2} \right)^2 = a_1^2 + a_2^2, \\ |\beta|^2 &= |b_1 + ib_2|^2 = \left(\sqrt{b_1^2 + b_2^2} \right)^2 = b_1^2 + b_2^2, \\ |-i\beta|^2 &= |-i(b_1 + ib_2)|^2 = |-ib_1 - i^2 b_2|^2 = |b_2 - ib_1|^2 = \left(\sqrt{b_1^2 + b_2^2} \right)^2 = b_1^2 + b_2^2, \end{aligned}$$

pertanto le due porte provocano lo stesso effetto. In generale applicare un fattore di fase globale a un qubit non cambia i risultati che otterremo dalla computazione, ma esamineremo nel prossimo capitolo, dedicato all'algoritmo di Grover, un caso in cui, all'interno di un sistema a più qubit, questo fattore fa la differenza. Segue la rappresentazione circuitale della porta:

$$|q\rangle \xrightarrow{R_x(\phi)}$$

Come ultima cosa, ci teniamo a mettere in evidenza un aspetto interessante di questa porta. Partendo da un qubit iniziale che si trova nello stato $|0\rangle$ e applicandogli la porta $R_x(\phi)$ con $\phi \neq \pi$ e $\phi \neq 2\pi$, portiamo il nostro qubit in uno stato di sovrapposizione. Talvolta porte R_x come quella appena descritta vengono anche chiamate porte *Partial-NOT*. Proviamo allora ad applicare, a scopo dimostrativo, la porta $R_x(\frac{\pi}{2})$ a un qubit nello stato $|0\rangle$:

$$\begin{bmatrix} \cos(\frac{\pi}{4}) & -i \sin(\frac{\pi}{4}) \\ -i \sin(\frac{\pi}{4}) & \cos(\frac{\pi}{4}) \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -i \frac{\sqrt{2}}{2} \\ -i \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ -i \frac{\sqrt{2}}{2} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-i}{\sqrt{2}} \end{bmatrix}.$$

Quello che otteniamo è uno stato di sovrapposizione in cui abbiamo la stessa probabilità di misurare 0 o 1.

1.2.3 Porta H

Un'altra porta molto interessante è la porta H, abbreviazione di porta di *Hadamard*. La sua matrice associata è:

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

Se applichiamo questa porta a un qubit che si trova in uno degli stati di base computazionale ($|0\rangle$ o $|1\rangle$), otteniamo una sovrapposizione in cui si ha la stessa probabilità di misurare 0 o 1. Infatti:

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

Notiamo tra l'altro che questi stati di sovrapposizione coincidono con i vettori della base X:

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

Infine, la rappresentazione circuitale della porta di Hadamard:

$$|q\rangle \xrightarrow{[H]} \quad$$

1.2.4 Porte Y e Z

Abbiamo detto che la porta R_x ci permette di effettuare una rotazione del vettore unitario di un qubit rispetto all'asse X, e che la porta X non è altro che un caso particolare dell'applicazione della porta R_x , dove l'angolo di rotazione è pari a π radianti. Analogamente a ciò, esistono delle porte che ci permettono di ruotare il vettore rispetto agli assi Y e Z, che non molto sorprendentemente si chiamano R_y e R_z . Queste tuttavia non sono particolarmente interessanti per i nostri scopi, quindi non le approfondiremo. Citiamo però per completezza le porte Y e Z, anch'esse casi particolari delle rispettive porte di rotazione con angolo pari a π radianti:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

Le matrici X, Y e Z vengono inoltre raggruppate sotto il nome di *matrici di Pauli*.

1.3 Porte quantistiche a più qubit

Così come un bit da solo non ci permette di fare molto, allo stesso modo un singolo qubit, anche se manipolato da diverse porte, non ci condurrà a dei risultati troppo interessanti. In questa sezione pertanto esamineremo alcune porte che manipolano contemporaneamente più qubit alla volta, esaminando anche i concetti di *entanglement* e *uncomputation*.

1.3.1 Porta C-NOT

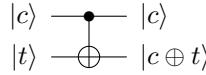
La prima porta a più qubit che vediamo è forse quella più semplice di tutte: la porta *C-NOT*, abbreviazione di *Controlled-NOT*, che agisce su due qubit alla volta. In particolare, nell'utilizzo di questa porta, si definisce un qubit di controllo e uno target. Quando il qubit di controllo si trova nello stato $|0\rangle$, non succede nulla (di fatto la porta non si attiva). Quando però il suo stato è pari a $|1\rangle$, quello del qubit target si inverte, passando da $|0\rangle$ a $|1\rangle$ e viceversa. Indicando con $|c\rangle$ il qubit di controllo e con $|t\rangle$ quello target, l'effetto complessivo che questa porta applica al sistema a due qubit, che indicheremo con $|ct\rangle = |c\rangle \otimes |t\rangle$, è riassumibile come:

$$\begin{aligned} |00\rangle &\rightarrow |00\rangle \\ |01\rangle &\rightarrow |01\rangle \\ |10\rangle &\rightarrow |11\rangle \\ |11\rangle &\rightarrow |10\rangle, \end{aligned}$$

dove con $|c\rangle \otimes |t\rangle$ intendiamo il prodotto tensore tra due vettori, ovvero:

$$|c\rangle \otimes |t\rangle = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} \otimes \begin{bmatrix} t_0 \\ t_1 \end{bmatrix} = \begin{bmatrix} c_0 t_0 \\ c_0 t_1 \\ c_1 t_0 \\ c_1 t_1 \end{bmatrix}.$$

La sua forma circuitale invece è:



Prima di procedere è bene fissare la rappresentazione vettoriale di un sistema a più qubit. Cominciamo da un esempio. In un caso semplice come quello appena descritto, dove abbiamo a che fare con soltanto due qubit, vale la seguente associazione stato-vettore:

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

In generale, dato un sistema a n qubit, lo stato di questo sistema verrà rappresentato da un vettore complesso appartenente allo spazio \mathbb{C}^{2^n} . Inoltre questo vettore è sempre sottoposto al vincolo unitario, ovvero la sua norma deve essere uno, in formule:

$$\|v\| = \sqrt{\sum_{i=1}^n |v_i|^2} = 1, \quad (1.1)$$

dove v è un vettore dello spazio \mathbb{C}^{2^n} . Sapendo questo, possiamo ora mostrare la matrice associata alla porta C-NOT:

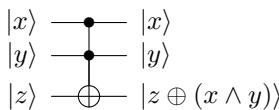
$$\text{C-NOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

che ci spiega anche cosa succede a un generico stato del nostro sistema a due qubit $|ct\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ \delta \\ \gamma \end{bmatrix}.$$

1.3.2 Porta Toffoli

Una porta che ci piacerebbe portare dall'ambito classico a quello quantistico è la porta AND, che normalmente prende in input due bit e ne restituisce uno che ha come valore 1 solo se i bit di ingresso erano a loro volta nello stato 1. Nel contesto quantistico purtroppo l'eleganza di questa porta non viene mantenuta, ed è necessario usare tre qubit anziché due per svolgere l'operazione di AND. Questo perché ogni circuito quantistico deve mantenere la proprietà di *reversibilità*, che esamineremo più a fondo nella Sezione 1.3.4. In particolare questa porta viene chiamata *porta di Toffoli*, e prende in ingresso tre qubit, che chiameremo $|x\rangle, |y\rangle$ e $|z\rangle$. L'operazione effettuata è quella di AND tra i qubit $|x\rangle$ e $|y\rangle$, ma il risultato viene conservato all'interno del qubit $|z\rangle$. In definitiva, quello che fa questa porta è applicare una porta NOT al qubit $|z\rangle$ quando gli altri due qubit si trovano nello stato $|1\rangle$. La sua forma circuitale forse rende meglio l'idea:



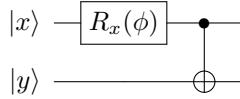
Come abbiamo però ormai imparato, non c'è porta senza matrice, e quella della porta di Toffoli è:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

È naturalmente possibile comporre più porte di questo tipo per ottenere una funzione di AND che prende in input un numero arbitrario di argomenti, proprio come nel caso classico. Ne vedremo un esempio nella Sezione 1.3.4.

1.3.3 Entanglement

Fino ad ora ci siamo concentrati su porte quantistiche a più qubit facendole però sembrare una trasposizione uno a uno della loro controparte classica. La loro vera potenza risiede tuttavia negli effetti che queste provocano su sistemi a più qubit quando abbiamo a che fare con stati di sovrapposizione. Può infatti verificarsi un fenomeno singolare tra di essi, che Einstein aveva chiamato *“spooky action at a distance”*: i qubit possono entrare in *entanglement* tra loro. Ciò vuol dire che i qubit coinvolti in questo fenomeno condividono in qualche modo lo stesso *“destino”*, ovvero un'azione compiuta su uno di essi avrà ripercussioni anche sugli altri, indipendentemente dalla distanza che li separa. Possiamo avere un'idea di cosa questo significhi dando uno sguardo a una porta costruibile a partire da due già viste: la *porta C-NOT parziale*. Essa è ottenibile dalla composizione di una porta R_x con una *C-NOT*:



Prima di studiarne gli effetti dobbiamo però capire che cosa succede quando applichiamo una porta a singolo qubit in un circuito multi-qubit. Abbiamo detto che il cavo quantistico può essere visto come un'applicazione della matrice identità I a un qubit. Pertanto in questo caso stiamo applicando parallelamente una porta R_x al primo qubit, e una I al secondo. La matrice associata a questa trasformazione è ottenuta effettuando il prodotto di Kronecker, conosciuto anche come prodotto tensore, tra le matrici relative alle due porte coinvolte. In generale date due matrici A e B di dimensione qualsiasi, il loro prodotto tensore è definito come:

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1m}B \\ a_{21}B & a_{22}B & \dots & a_{2m}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}B & a_{n2}B & \dots & a_{nm}B \end{bmatrix}.$$

Fissando a titolo dimostrativo l'angolo ϕ a $\frac{\pi}{2}$ radianti, pertanto, il prodotto tensore tra le matrici associate alle porte $R_x(\frac{\pi}{2})$ e I risulta:

$$R_x\left(\frac{\pi}{2}\right) \otimes I = \begin{bmatrix} \frac{\sqrt{2}}{2}I & -i\frac{\sqrt{2}}{2}I \\ -i\frac{\sqrt{2}}{2}I & \frac{\sqrt{2}}{2}I \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & -i\frac{\sqrt{2}}{2} & 0 \\ 0 & \frac{\sqrt{2}}{2} & 0 & -i\frac{\sqrt{2}}{2} \\ -i\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & -i\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix}.$$

Immaginando che entrambi i qubit $|x\rangle$ e $|y\rangle$ partano dallo stato $|0\rangle$, il vettore associato a questo sistema che otteniamo dopo l'applicazione della prima porta risulta:

$$\left(R_x\left(\frac{\pi}{2}\right) \otimes I\right) |00\rangle = \begin{bmatrix} \frac{\sqrt{2}}{2} & 0 & -i\frac{\sqrt{2}}{2} & 0 \\ 0 & \frac{\sqrt{2}}{2} & 0 & -i\frac{\sqrt{2}}{2} \\ -i\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & -i\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ 0 \\ -i\frac{\sqrt{2}}{2} \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{-i}{\sqrt{2}} \\ 0 \end{bmatrix}.$$

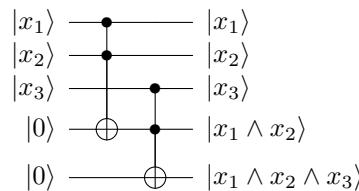
Applichiamo ora la porta C-NOT e vediamo come cambia il vettore:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{-i}{\sqrt{2}} \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{-i}{\sqrt{2}} \end{bmatrix} = \frac{1}{\sqrt{2}} |00\rangle - \frac{i}{\sqrt{2}} |11\rangle.$$

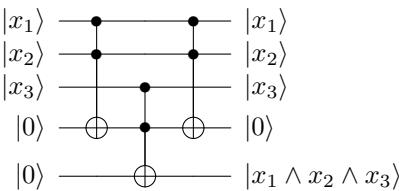
Uno stato del genere dimostra che cosa intendevamo poco fa, quando abbiamo asserito che dei qubit posti in entanglement sono soggetti allo stesso “destino”. In questo caso infatti abbiamo pari probabilità di misurare 0 o 1 per entrambi i qubit, ma non capiterà mai che uno venga misurato 0 e l’altro 1 (almeno in caso di assenza di rumore, ma questo è un argomento che tratteremo nella Sezione 1.4.2). Questo era solo uno dei tanti modi con cui dei qubit possono essere posti in entanglement, e ci sono circuiti capaci di legare un numero arbitrario di qubit in questo modo.

1.3.4 Uncomputation

Un altro aspetto interessante e per alcuni algoritmi quantistici fondamentale è quello dell’*uncomputation*. Per capirlo supponiamo di voler costruire un circuito che calcoli l’AND tra tre qubit $|x_1\rangle$, $|x_2\rangle$ e $|x_3\rangle$. Possiamo usare due volte la porta di Toffoli per raggiungere lo scopo:



Effettivamente il quinto qubit del circuito ci dà il risultato desiderato. Tuttavia abbiamo anche accumulato un risultato intermedio nel quarto qubit, ovvero l’AND tra i primi due, che non ci interessa. Ciò potrebbe sembrare un problema di poco conto, ma per algoritmi complessi in cui i qubit vengono posti in entanglement, avere risultati intermedi lasciati in giro può rappresentare un problema. Inoltre il qubit usato per conservare questo AND intermedio potrebbe essere usato per altri scopi se venisse “resettato”, ovvero riportato nello stato $|0\rangle$. È qui che entra in gioco l’*uncomputation*, che prevede, una volta calcolato il risultato cercato, di applicare le stesse porte usate per effettuare tale calcolo (ad eccezione di quella che modifica il qubit risultato) nell’ordine inverso, così da riportare i qubit di lavoro, a volta chiamati anche *ancilla qubit* (nel nostro esempio il quarto qubit ricade in questa categoria), nel loro stato originale. Il circuito mostrato sopra verrebbe pertanto modificato come segue:



1.4 I qubit odierni

1.4.1 Sistemi fisici per la rappresentazione di qubit

Fino ad ora abbiamo parlato dei qubit come oggetti puramente matematici, ma fintanto che questi rimangono oggetti astratti, nella pratica, non servono a molto, se non a capirne il funzionamento. Fisicamente, quindi, come viene implementato un qubit? In vari modi. Alcuni di questi possono essere:

- un elettrone in due differenti orbitali attorno al nucleo di un atomo.
- un fotone in uno tra due stati di polarizzazione.
- una particella subatomica che ha una tra le due direzioni di spin.

Ad oggi tuttavia è raro trovare implementazioni di qubit soddisfacenti, o almeno tali da permetterci di svolgere calcoli complessi. Perfino implementare un semplice cavo quantistico può essere difficile. Se ad esempio scegliamo di conservare un qubit in un neutrino, allora sarà facile preservare il suo stato nel tempo, dato che i neutrini interagiscono debolmente con il mondo esterno. Tuttavia, se questo è vero, allora sarà difficile manipolare il loro stato con delle porte. L'argomento è complesso e non rientra nei nostri scopi. Ciò che invece riteniamo interessante sottolineare è che ad oggi molti di questi sistemi non sono in grado di compiere calcoli per periodi di tempo troppo lunghi. Questo è dovuto a un problema conosciuto col nome di *rumore*.

1.4.2 Rumore nei computer quantistici

Il rumore è purtroppo un fenomeno che si verifica con costanza nei moderni device quantistici e che va a incidere negativamente sui risultati nelle nostre computazioni. Le sorgenti di rumore possono essere di vario tipo, come i segnali elettromagnetici emessi dalle reti Wi-Fi o perfino disturbi creati dal campo magnetico terrestre. L'effetto netto del rumore è di degradare la qualità dell'informazione che decidiamo di conservare in un sistema di qubit. Può perfino accadere, a causa del rumore, che un qubit inizializzato nello stato $|0\rangle$ e fatto passare attraverso due porte H in sequenza (operazione che coincide con l'applicazione di nessuna porta, dato che $H^\dagger H = HH = I$) dia come risultato della misurazione 1, cosa che in un computer quantistico perfetto non potrebbe mai accadere. Infatti le simulazioni svolte su dispositivi classici, che simulano un device quantistico perfetto, non sono soggette a questo fenomeno, ma sono anche molto più lente rispetto alle corrispettive computazioni eseguite su macchine quantistiche. Oggigiorno infatti lo scopo è quello di riuscire a eseguire circuiti quantistici per periodi di tempo sempre più lunghi prima che il rumore corrompa i risultati della computazione.

Capitolo 2

L'algoritmo di Grover

Nel calcolo classico, se dobbiamo cercare uno specifico oggetto all'interno di un insieme non strutturato che ne contiene N , siamo costretti ad esaminare uno a uno gli elementi dell'insieme e controllare se l'elemento corrente è quello cercato. In caso affermativo, la ricerca si interrompe, in caso negativo, la ricerca continua. Questo se diamo per scontato che gli elementi dell'insieme non seguano alcun tipo di ordinamento, altrimenti è possibile usare un algoritmo di ricerca binaria per ridurre la complessità della ricerca a $\mathcal{O}(\log_2 N)$. In caso di assenza di qualsivoglia ordinamento, l'unica cosa che possiamo fare è procedere per esclusione, con una complessità $\mathcal{O}(N)$. Con l'algoritmo di ricerca quantistico, conosciuto anche come *algoritmo di Grover*, è possibile tuttavia ridurre la complessità della ricerca di un fattore quadratico, avendo quindi una complessità $\mathcal{O}(\sqrt{N})$. Il capitolo è strutturato come segue:

1. Nella prima sezione tratteremo del primo elemento che compone l'algoritmo di Grover: l'*oracolo*, cercando di darne un'intuizione.
2. Nella seconda sezione andremo ad illustrare la procedura per intero.
3. Nella terza sezione cercheremo di capire la matematica che sta dietro a questa procedura, dando un'interpretazione geometrica ai passi dell'algoritmo così da “convincerci” che funziona.

Durante la lettura delle fonti che trattavano l'argomento, ci siamo resi conto che venivano date per scontate alcune nozioni di geometria e di algebra lineare. Ciò a volte può rallentare l'apprendimento, data la complessità del tema. Onde evitare che ciò accada anche ai lettori di questo testo, ci soffermeremo su alcuni punti che riteniamo meritevoli di maggiore approfondimento. Chi ritenesse di avere sufficiente confidenza con queste nozioni può saltare tranquillamente le parti ad esse dedicate.

2.1 L'oracolo

2.1.1 Differenza tra conoscere una soluzione e saper riconoscere una soluzione

Supponiamo che ci venga fornita una formula 3-SAT come quella della Figura 2.1, ovvero una formula booleana in forma normale congiuntiva dove tutte le clausole contengono esattamente tre simboli proposizionali.

$$(A \vee B \vee \neg C) \wedge (\neg A \vee D \vee C) \wedge (\neg B \vee \neg D \vee \neg A) \wedge (\neg C \vee \neg B \vee D).$$

Figura 2.1: Esempio di formula 3-SAT con 4 simboli proposizionali.

Un algoritmo che risolve questo problema è un algoritmo che prova tutte le 2^4 possibili interpretazioni per la formula, fino a quando non trova un suo modello. Quindi l'algoritmo *non conosce a priori* la soluzione del problema, ma quando ne vede una *sa riconoscerla*. Facciamo un esempio molto alla mano: immaginiamo che il problema di soddisfare la formula booleana della Figura 2.1 faccia parte di un compito d'esame di Logica Per la Programmazione. Ci sono solo due studenti che partecipano alla prova. Il

primo non ha studiato, ma la sera prima si è intrufolato nell'ufficio del Professore e ha fatto una copia della soluzione dell'esame, che è riuscito a portarsi in aula senza farsi scoprire. In questo caso, lo studente conosce *a priori* la soluzione del problema, e sta di fatto barando. Per sua sfortuna viene scoperto dal Docente, che gli ritira il testo d'esame e lo boccia sul posto. Il secondo studente invece, che ha seguito tutte le lezioni studiando in maniera costante, alla fine consegna la sua soluzione, che è in particolare $A = F, B = T, C = T, D = T$. Il Professore, per potergli assegnare un voto, deve *verificare* che la soluzione proposta sia effettivamente una soluzione al problema. Valuta quindi la formula proposizionale nell'interpretazione proposta:

$$(F \vee T \vee F) \wedge (T \vee T \vee T) \wedge (F \vee F \vee T) \wedge (F \vee F \vee T) \equiv T \wedge T \wedge T \wedge T \equiv T.$$

Figura 2.2: Valutazione della formula nell'interpretazione dello studente.

La soluzione è corretta, e lo studente passa con 30L. Il Docente ha potuto appurare la correttezza della soluzione perché sapeva *riconoscere* un suo modello quando ne vedeva uno. Quando ci si approccia per la prima volta all'algoritmo di Grover, può sembrare che l'oracolo sia in grado di riconoscere la soluzione del problema perché già la conosce: non è così! Sa riconoscere la soluzione perché sa quali vincoli deve rispettare. Da questo punto di vista, l'oracolo sarebbe il nostro Professore dell'esempio, e **non** lo studente disonesto.

2.1.2 Costruire un oracolo

Torniamo al nostro esempio generale. Vogliamo cercare una soluzione per un problema all'interno di uno spazio di ricerca contenente N elementi. Dato che non sempre è possibile convertire in modo semplice questi oggetti in sequenze binarie (riesce facile se stiamo parlando di interi, ma diventa più difficile se parliamo di oggetti complessi del mondo reale come animali, cose, persone etc..), definiamo una funzione bigettiva $g(x)$ che associa un elemento x dell'insieme ad un *indice* codificato in binario. Per fare ciò ci servono, nel calcolo classico, n bit, dove $N = 2^n$. Stiamo supponendo che N sia una potenza di due per semplificare la spiegazione, ma questa ipotesi non è necessaria. Infatti, qualora 2^n fosse maggiore di N , associeremo i $2^n - N$ indici che vanno da N a 2^n a un valore fittizio, e sapremo che non potranno mai essere indici di una soluzione. Nel calcolo quantistico l'unica differenza è che invece di n bit avremo n qubit. Così come per la formula 3-SAT della Figura 2.1 non è detto che ci sia un solo modello, allo stesso modo in un problema non è detto che la soluzione sia soltanto una. Indicheremo pertanto con la lettera M il numero di soluzioni all'interno dello spazio di ricerca, dove $0 \leq M \leq N$. Abbiamo bisogno di un ultimo ingrediente per poter costruire il nostro oracolo: una funzione $f(x)$ che restituisce 1 quando x è una soluzione del problema, 0 altrimenti. Più in generale, dovremmo dire che $f(x)$ restituisce 1 quando x è *indice* di una soluzione del problema e 0 altrimenti, ma per semplificare la spiegazione supporremo che la funzione $g(x)$ sia la funzione identità, ovvero che gli indici delle possibili soluzioni rappresentino le possibili soluzioni stesse. Con queste premesse, l'oracolo può essere visto come una scatola nera (*black box*) che, dati come input un registro degli indici $|x\rangle$, ovvero un registro quantistico a n qubit, e un ulteriore qubit $|q\rangle$, svolge la seguente operazione:

$$|x\rangle |q\rangle \xrightarrow{O} |x\rangle |q \oplus f(x)\rangle,$$

dove \oplus indica la somma modulo 2. L'operazione ha come obiettivo quello di applicare una porta *NOT* al qubit $|q\rangle$ quando $|x\rangle$ è una soluzione del problema. Viene infatti sfruttato in questo passo dell'algoritmo un effetto quantistico molto importante: il *kickback di fase*. Per capire al meglio la potenza di questo effetto, facciamo un esempio con 2 qubit.

Esempio. Si consideri il seguente circuito quantistico:

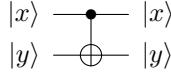


Figura 2.3: Circuito quantistico con una porta *C-NOT*.

Si tratta di un semplice circuito dove viene usata una sola porta, la *C-NOT*, che abbiamo visto nella Sezione 1.3.1. Algoritmamente il suo funzionamento può essere espresso da un programma del tipo:

```

1: if  $x = 1$  then
2:    $y \leftarrow 1 \oplus y$ 
3: else
4:   do nothing

```

Come però abbiamo già studiato, l'effetto della porta *C-NOT* è ben più generale di così, e consiste nell'applicare, a un generico circuito a due qubit del tipo $|xy\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$, la seguente trasformazione lineare:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ \delta \\ \gamma \end{bmatrix}.$$

Alla luce di questo fatto, vediamo ora un caso particolare. Dal codice sopra e dalla sua implementazione quantistica potrebbe sembrare che il qubit di controllo rimanga sempre inalterato, e faccia semplicemente da discriminante per il valore del secondo qubit. Tuttavia considerando due qubit negli stati computazionali $|+\rangle$ e $|-\rangle$, ottenuti applicando una porta di Hadamard a due qubit rispettivamente nello stato di base computazionale $|0\rangle$ e $|1\rangle$, osserviamo un effetto interessante, mostrato nella Figura 2.4:

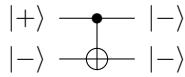


Figura 2.4: Circuito con porta *C-NOT* dove il qubit di controllo cambia, mentre quello obiettivo rimane inalterato.

Dato che il circuito da solo non rende bene l'idea, eseguiamo i calcoli a mano per convincerci che funziona:

$$|+-\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{|00\rangle}{2} - \frac{|01\rangle}{2} + \frac{|10\rangle}{2} - \frac{|11\rangle}{2} = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix},$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix},$$

$$|--\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} = \frac{|00\rangle}{2} - \frac{|01\rangle}{2} - \frac{|10\rangle}{2} + \frac{|11\rangle}{2} = \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix}.$$

Abbiamo quindi trovato un esempio di circuito a più qubit dove *il qubit di controllo viene modificato da quello obiettivo*. Questo fenomeno è conosciuto come kickback di fase.

I motivi per cui vogliamo applicare questo particolare effetto al qubit $|q\rangle$ dell'oracolo saranno chiari quando daremo un'interpretazione geometrica dell'algoritmo. Per il momento però è bene sottolineare un'ultima volta un concetto importante: *l'oracolo non conosce la soluzione a priori, ma è in grado di riconoscerne una quando la vede*. Riteniamo questo punto fondamentale in quanto l'oracolo è un pezzo importantissimo dell'algoritmo di Grover, eppure spesso viene liquidato dicendo che è una scatola nera capace di riconoscere la soluzione, ma non viene mai fatto vedere *come* fa a riconoscerla. Apriamo quindi questa scatola nera e vediamo come potrebbe essere implementato l'oracolo relativo al problema della Figura 2.1, mediante un circuito quantistico. Per fare ciò tuttavia dobbiamo prima modificare la formula applicando la regola di De Morgan per il calcolo proposizionale. Supponiamo inoltre che ogni qubit sia inizializzato nello stato di base computazionale $|0\rangle$:

$$\begin{aligned}
 & (A \vee B \vee \neg C) \wedge (\neg A \vee D \vee C) \wedge (\neg B \vee \neg D \vee \neg A) \wedge (\neg C \vee \neg B \vee D) \\
 \equiv & \{(DeMorgan)\} \\
 & (\neg A \wedge \neg B \wedge C) \wedge (A \wedge \neg D \wedge \neg C) \wedge (B \wedge D \wedge A) \wedge (C \wedge B \wedge \neg D).
 \end{aligned}$$

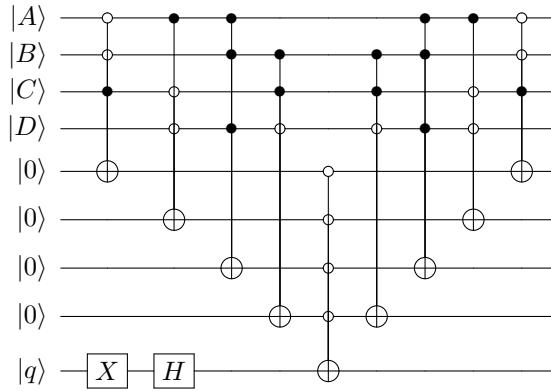


Figura 2.5: Oracolo relativo al problema 3-SAT della figura 2.1, capace di riconoscerne una soluzione.

I primi quattro qubit rappresentano i simboli proposizionali, dove $|0\rangle = F$ e $|1\rangle = T$, mentre i quattro successivi sono qubit di lavoro (*ancilla qubit*), utili solo a conservare i risultati intermedi della computazione. L'ultimo qubit invece è quello che applicherà il kickback di fase qualora $|ABCD\rangle$ rappresentasse un'interpretazione che soddisfa la formula logica. Il pallino bianco nelle porte di Toffoli indica, come quello nero, un qubit di controllo, con la differenza che richiede il valore $|0\rangle$ per attivarsi. Il motivo per cui $|q\rangle$ viene inizializzato nello stato $|-\rangle$, mediante l'applicazione delle porte X e H in sequenza, è che così facendo è possibile applicare il kickback di fase sui primi quattro qubit, lasciando inalterato l'ultimo. Si noti inoltre che si è sfruttato il meccanismo della *uncomputation* per ripristinare lo stato dei qubit ancilla. L'effetto complessivo dell'oracolo può quindi essere riassunto come:

$$|x\rangle |-\rangle \xrightarrow{O} (-1)^{f(x)} |x\rangle |-\rangle,$$

poiché i qubit di lavoro tornano al loro stato originale grazie alla *uncomputation* e il qubit q rimane sempre nello stato $|-\rangle$, applicando eventualmente un kickback di fase -1 al registro quantistico quando questo contiene una soluzione al problema. Nel nostro esempio, $|x\rangle = |ABCD\rangle$.

2.2 L'iterazione di Grover

L'algoritmo di per sé non è troppo complicato, e richiede veramente poche componenti. La prima cosa da fare è, dato un registro quantistico a n qubit nello stato $|0\rangle^{\otimes n}$, far passare tale registro attraverso una porta $H^{\otimes n}$. Detto in altre parole, si comincia avendo tutti gli n qubit nello stato di base computazionale $|0\rangle$, per poi far passare ciascuno di essi attraverso una porta di Hadamard così da porre in entanglement i

qubit e ottenere lo *stato di ugual sovrapposizione* $|\psi\rangle$, dove:

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

Ricordiamo che $N = 2^n$, dove N è la dimensione del dominio di ricerca e n è il numero di qubit che usiamo per rappresentare gli indici degli oggetti in tal dominio.

Esempio. Con 3 qubit, lo stato di ugual sovrapposizione sarebbe:

$$|\psi\rangle = \frac{1}{\sqrt{8}} |000\rangle + \frac{1}{\sqrt{8}} |001\rangle + \frac{1}{\sqrt{8}} |010\rangle + \frac{1}{\sqrt{8}} |011\rangle + \frac{1}{\sqrt{8}} |100\rangle + \frac{1}{\sqrt{8}} |101\rangle + \frac{1}{\sqrt{8}} |110\rangle + \frac{1}{\sqrt{8}} |111\rangle.$$

Dopo questa inizializzazione l'algoritmo di Grover si riduce ad un'applicazione reiterata di una funzione quantistica (funzione intesa come subroutine informatica) chiamata *iterazione di Grover* o *operatore di Grover*, che denotiamo con la lettera G . Questa procedura può essere divisa in quattro passi:

1. Si applica l'oracolo O .
2. Si applica la trasformazione di Hadamard $H^{\otimes n}$ al registro quantistico a n qubit.
3. Si applica un kickback di fase -1 ad ogni stato di base computazionale del registro eccetto che allo stato $|0\rangle$ (ovvero $|0_1 0_2 \dots 0_n\rangle$). Una porta di questo tipo applica la trasformazione lineare $2|0\rangle\langle 0| - I$ al vettore degli stati, che corrisponde a una riflessione attorno al vettore $|0\rangle$.
4. si applica di nuovo la trasformazioni di Hadamard $H^{\otimes n}$.

Ma quante volte va ripetuta quest'operazione? Al più \sqrt{N} volte, per trovare una soluzione.

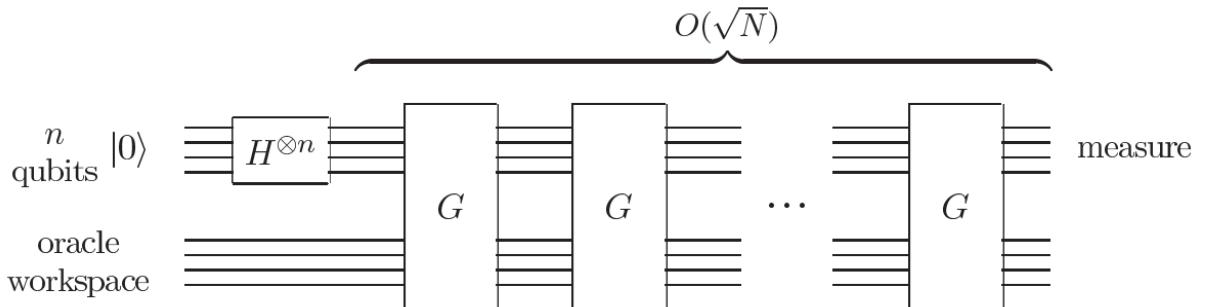


Figura 2.6: Rappresentazione schematica dell'algoritmo di ricerca quantistico. Anche se questi impiega dei qubit di lavoro, l'analisi della complessità coinvolge solo il registro a n qubit (si veda il Capitolo 6.1 di [11])

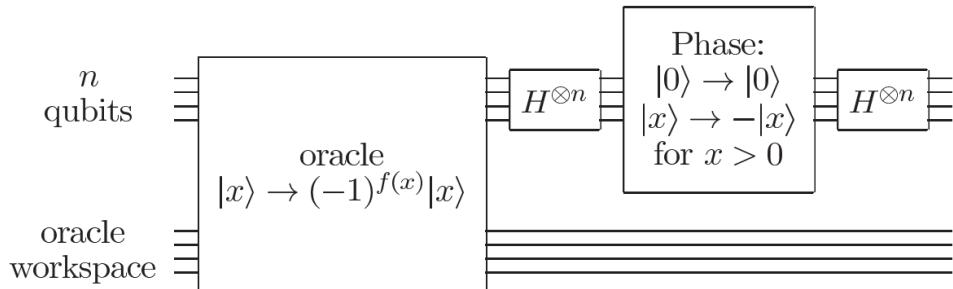


Figura 2.7: Circuito rappresentativo dell'iterazione di Grover G (si veda Capitolo 6.1 di [11])

Quando parliamo di algoritmi non possiamo esimerci dal parlare della loro complessità, e quelli quantistici non fanno eccezione. I passi 2 e 4, ovvero le trasformazioni di Hadamard, hanno un costo lineare nel

numero di qubit di $n = \log(N)$. Il passo 3 invece può essere implementato con $\mathcal{O}(N)$ porte. Tutta la complessità dell'algoritmo ricade dunque sull'oracolo e sul numero di volte che va invocato, in quanto gli altri passi rimangono identici e costanti indipendentemente dalla formulazione del problema di ricerca. L'implementazione dell'oracolo cambia da problema a problema, come abbiamo visto nella Figura 2.5, pertanto il suo costo, in linea di principio, può essere arbitrariamente alto. Tuttavia per costruirlo solitamente si parte da un circuito classico e lo si converte in forma quantistica, implementando una funzione $f : \{0, 1\}^n \rightarrow \{0, 1\}$. Dato che si tratta di una copia uno a uno di un circuito, non abbiamo modo di controllare la complessità di una singola esecuzione dell'oracolo, che potremmo migliorare solo cercando un algoritmo (e quindi un circuito) classico più efficiente da cui partire. La vera differenza risiede dunque nel numero di applicazioni dell'oracolo: $\mathcal{O}(N)$ volte in ambito classico, $\mathcal{O}(\sqrt{N})$ in quello quantistico. Quando daremo un'interpretazione geometrica all'algoritmo nella prossima sezione sarà chiaro questo miglioramento quadratico. Per il momento ci accontentiamo di una osservazione finale. I passi 2, 3 e 4 possono essere combinati per ottenere la trasformazione lineare:

$$H^{\otimes n} (2|0\rangle\langle 0| - I) H^{\otimes n} = 2|\psi\rangle\langle\psi| - I,$$

dove $|\psi\rangle$ è lo stato di ugual sovrapposizione. Un'iterazione di Grover G può quindi essere espressa come l'applicazione del seguente operatore unitario:

$$G = (2|\psi\rangle\langle\psi| - I)O,$$

dove O è l'operatore oracolo.

2.3 Analisi dell'algoritmo

Per convincersi della correttezza dell'algoritmo di Grover non è possibile esimersi da una sua visualizzazione geometrica. Ciò che faremo in questa sezione sarà pertanto affiancare alla spiegazione testuale degli esempi, procedendo su due binari paralleli ma inseparabili:

- Da un lato, useremo un sistema a due qubit per mostrare algebricamente lo svolgimento dell'iterazione di Grover. Il problema giocattolo che prenderemo in considerazione sarà più che banale: dato uno spazio di ricerca contenente i numeri che vanno a 0 a 3 (estremi inclusi), vogliamo che l'algoritmo ci restituisca come soluzione il numero 3, corrispondente allo stato computazionale $|11\rangle$. L'incredibile semplicità e, se vogliamo, stupidità di questo problema ci permetterà di effettuare calcoli chiari e semplici, senza tirare in ballo matrici e vettori troppo grandi.
- Dall'altro, per dare un'interpretazione geometrica a questi passi algebrici, mostreremo l'effetto che tali operazioni avrebbero su uno spazio tridimensionale. Notate che questa associazione è puramente a scopo illustrativo: non esiste un numero di qubit capace di formare uno spazio vettoriale in \mathbb{C}^3 . Forse ciò sarebbe possibile se potessimo manipolare un registro quantistico a $\log_2(3)$ qubit. *Peccato però che non esista un simile registro, in quanto non avrebbe senso.* Il motivo per cui non abbiamo optato per una rappresentazione geometrica usando 2 qubit è che non è possibile rappresentare efficientemente lo spazio \mathbb{C}^4 su due dimensioni. Allo stesso tempo, non abbiamo usato un sistema a un solo qubit sia per la parte algebrica che per quella geometrica poiché non avrebbe permesso di visualizzare in maniera incisiva il funzionamento dell'algoritmo. Anzi, avremmo rischiato di porre in contrasto tra loro spiegazione scritta e interpretazione geometrica: *un disastro insomma.*

L'algoritmo di Grover può essere visualizzato come una rotazione del vettore iniziale $|\psi\rangle$ su un certo piano. Per definire tale piano ci occorrono due ultimi ingredienti. Sappiamo già che N è la dimensione dello spazio di ricerca, mentre M è il numero di soluzioni al problema, dove $0 \leq M \leq N$. Sappiamo anche che $|\psi\rangle$ è il vettore che rappresenta lo stato di ugual sovrapposizione $\frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$. Definiamo adesso con \sum'_x la sommatoria dei vettori $|x\rangle$ che rappresentano una soluzione al problema, e con \sum''_x la sommatoria dei vettori $|x\rangle$ che NON sono soluzione del problema. Definiamo dunque due nuovi vettori $|\alpha\rangle$ e $|\beta\rangle$ come segue:

$$|\alpha\rangle = \frac{1}{\sqrt{N-M}} \sum'_x |x\rangle,$$

$$|\beta\rangle = \frac{1}{\sqrt{M}} \sum_x' |x\rangle .$$

Notate che tutti questi vettori sono sempre unitari, ovvero $\|\psi\| = \|\alpha\| = \|\beta\| = 1$, dove, ricordiamo, con $\|\cdot\|$ indichiamo la norma unitaria o norma 2 di un vettore, calcolabile mediante la formula:

$$\|v\| = \sqrt{\sum_{i=1}^n |v_i|^2}. \quad (2.1)$$

Esempio. (Algebra) Nello spazio vettoriale \mathbb{C}^4 abbiamo:

$$|\psi\rangle = \frac{1}{\sqrt{4}} \sum_{x=0}^3 |x\rangle = \frac{1}{\sqrt{4}} |00\rangle + \frac{1}{\sqrt{4}} |01\rangle + \frac{1}{\sqrt{4}} |10\rangle + \frac{1}{\sqrt{4}} |11\rangle = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix},$$

dove, ricordiamo, stiamo cercando come soluzione 3, ovvero vorremmo misurare lo stato di base computazionale $|11\rangle$. Quindi $N = 4$ e $M = 1$. Non semplificheremo $\sqrt{4}$ con 2 in quanto vogliamo che sia sempre chiaro che quel $\sqrt{4}$ è un'istanza di \sqrt{N} . Inoltre:

$$\sum_x' |x\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \sum_x'' |x\rangle = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix},$$

$$|\beta\rangle = \frac{1}{\sqrt{1}} \sum_x' |x\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad |\alpha\rangle = \frac{1}{\sqrt{3}} \sum_x'' |x\rangle = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

(Geometria) Immaginiamo di avere, nello spazio vettoriale \mathbb{R}^3 :

$$|\psi\rangle = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix},$$

dove definiamo le due sommatorie e i vettori $|\alpha\rangle$ e $|\beta\rangle$ come:

$$\sum_x' |x\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad \sum_x'' |x\rangle = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix},$$

$$|\beta\rangle = \frac{1}{\sqrt{1}} \sum_x' |x\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad |\alpha\rangle = \frac{1}{\sqrt{2}} \sum_x'' |x\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

Nello spazio tridimensionale ci troviamo dunque nella situazione della figura 2.8:

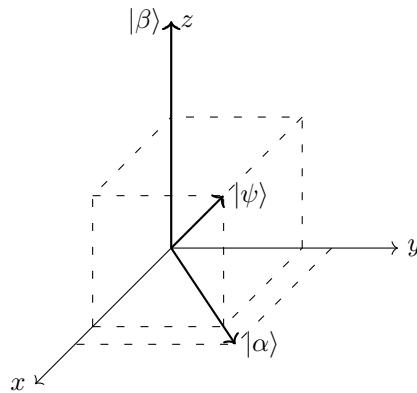


Figura 2.8: Rappresentazione di $|\alpha\rangle$, $|\beta\rangle$ e $|\psi\rangle$ nello spazio \mathbb{R}^3 .

Prima di procedere, notiamo una cosa che può darci un'intuizione circa la natura dei vettori $|\alpha\rangle$ e $|\beta\rangle$. Questi vettori, così come tutti i vettori che possiamo trovare in ambito quantistico, sono unitari, ovvero hanno lunghezza uno. Ciò è in linea con il fatto che la somma delle probabilità deve fare uno. Infatti sappiamo dal Capitolo 1 che nella misurazione di un qubit (o di un sistema a più qubit posti in entanglement tra loro) la probabilità di misurare un certo stato corrisponde all'ampiezza di quello stato in valore assoluto e elevata alla seconda. La somma di queste probabilità è proprio l'Equazione (2.1), a meno della radice quadrata. Ma dato che $\sqrt{1} = 1$, la radice ce la possiamo anche scordare. I vettori $|\alpha\rangle$ e $|\beta\rangle$ in particolare hanno, per ogni loro componente con un'ampiezza diversa da 0, la stessa ampiezza: $\frac{1}{\sqrt{K}}$, dove K è uguale proprio al numero di componenti diverse da 0. Solo che:

- $|\alpha\rangle$ ha come uniche componenti diverse da 0 quelle che NON rappresentano una soluzione al problema. Ciò vuol dire che non importa quante volte misuriamo questo vettore, otterremo sempre un fallimento per il problema. Potremmo anche chiamarlo, vista la equidistribuzione di probabilità, *vettore di ugual sovrapposizione dei fallimenti*. Per $|\alpha\rangle$, abbiamo $K = N - M$, poiché $N - M$ sono i fallimenti (totale - successi).
- $|\beta\rangle$ è esattamente l'opposto: ha come uniche componenti diverse da 0 quelle che rappresentano una soluzione al problema, così che ogni sua misura ci restituisca un successo per il problema. Di nuovo, queste componenti hanno la stessa ampiezza, pertanto possiamo chiamarlo anche *vettore di ugual sovrapposizione dei successi*. Per $|\beta\rangle$, $K = M$, poiché M sono le soluzioni.

È quindi come se $|\alpha\rangle$ prendesse tutti e soli i fallimenti di $|\psi\rangle$ distribuendo le ampiezze equamente, e lo stesso facesse $|\beta\rangle$, ma con le soluzioni. Questa intuizione trova un corrispettivo algebrico nel fatto che $|\psi\rangle$ appartiene allo span di $|\alpha\rangle$ e $|\beta\rangle$. Infatti:

$$\begin{aligned} |\psi\rangle &= \sqrt{\frac{N-M}{N}} |\alpha\rangle + \sqrt{\frac{M}{N}} |\beta\rangle = \sqrt{\frac{N-M}{N}} \frac{1}{\sqrt{N-M}} \sum_x'' |x\rangle + \sqrt{\frac{M}{N}} \frac{1}{\sqrt{M}} \sum_x' |x\rangle = \\ &= \frac{1}{\sqrt{N}} \left(\sum_x'' |x\rangle + \sum_x' |x\rangle \right) = |\psi\rangle. \end{aligned}$$

Le rotazioni che l'algoritmo di Grover esegue in successione servono, algebricamente, ad “allargare” le ampiezze delle soluzioni e “schiacciare” quelle degli insuccessi, e geometricamente ad avvicinare il più possibile il vettore iniziale $|\psi\rangle$ al vettore di *soluzione assicurata* $|\beta\rangle$, idealmente sovrapponendo il primo al secondo.

Possiamo compiere una rotazione effettuando due riflessioni in successione: la prima volta rispetto a $|\alpha\rangle$, la seconda rispetto a $|\psi\rangle$. Dato che usare lo stesso termine per due concetti diversi può portare confusione, indicheremo d'ora in poi con $|\phi\rangle$ il vettore che vogliamo ruotare prima rispetto a $|\alpha\rangle$ e poi rispetto a $|\psi\rangle$ per avvicinarlo il più possibile a $|\beta\rangle$. Notate che questo vettore all'inizio coincide con $|\psi\rangle$, ma ciò non vale più dopo la prima iterazione di Grover. Se indichiamo con $\frac{\theta}{2}$ l'angolo compreso tra $|\psi\rangle$ e $|\alpha\rangle$, ogni rotazione ci permette di avvicinarci di θ radianti al vettore di ugual sovrapposizione dei successi. Vediamo come. Innanzitutto, riflettiamo il vettore $|\phi\rangle$ rispetto a $|\alpha\rangle$. Dato che stiamo parlando di vettori unitari farlo è molto semplice, basta applicare l'operatore di riflessione $2|\alpha\rangle\langle\alpha| - I$ al vettore che vogliamo ruotare, $|\phi\rangle$. Chiamando $|\phi'\rangle$ il vettore risultante, otteniamo:

$$(2|\alpha\rangle\langle\alpha| - I)|\phi\rangle = |\phi'\rangle.$$

Esempio. (Algebra) L'operatore di riflessione rispetto al vettore $|\alpha\rangle$ risulta:

$$2|\alpha\rangle\langle\alpha| - I = 2\frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \frac{1}{\sqrt{3}} \begin{bmatrix} 1 & 1 & 1 & 0 \end{bmatrix} - I = 2\frac{1}{3} \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} & 0 \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} & 0 \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

Otteniamo pertanto:

$$|\phi'\rangle = (2|\alpha\rangle\langle\alpha| - I)|\phi\rangle = \begin{bmatrix} -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} & 0 \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} & 0 \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix}.$$

(Geometria) In questo caso l'operatore di riflessione risulta:

$$2|\alpha\rangle\langle\alpha| - I = 2\frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 & 0 \end{bmatrix} - I = \frac{2}{2}\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}.$$

Dunque:

$$|\phi'\rangle = (2|\alpha\rangle\langle\alpha| - I)|\phi\rangle = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \frac{1}{\sqrt{3}}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{3}}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}.$$

Abbiamo così ruotato $|\phi\rangle$ attorno a $|\alpha\rangle$, che tridimensionalmente corrisponde alla seguente situazione:

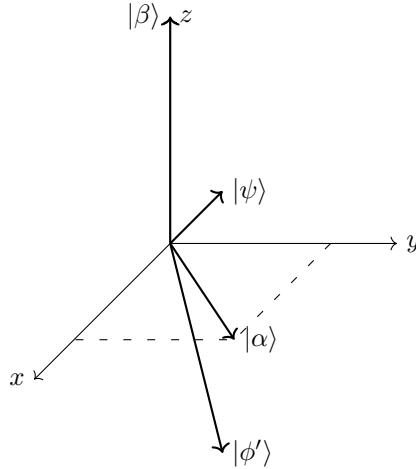


Figura 2.9: Rotazione del vettore $|\phi\rangle$ nello spazio \mathbb{R}^3 . Notare che, anche se può non sembrare a causa del fatto che stiamo rappresentando uno spazio tridimensionale in uno bidimensionale, anche il vettore $|\phi'\rangle$ è normalizzato, proprio come gli altri. Infatti in questa prima rotazione si ha che $|\psi\rangle = |\phi\rangle = [\phi_1, \phi_2, \phi_3]$, e $|\phi'\rangle = [\phi_1, \phi_2, -\phi_3]$.

Il prossimo passo è compiere la seconda riflessione, stavolta rispetto al vettore di ugual sovrapposizione $|\psi\rangle$. Di nuovo, possiamo usare l'operatore di riflessione $2|\psi\rangle\langle\psi| - I$, così da ruotare il vettore $|\phi'\rangle$ e ottenere il vettore $|\phi''\rangle$, ovvero:

$$(2|\psi\rangle\langle\psi| - I)|\phi'\rangle = |\phi''\rangle.$$

Esempio. (Algebra) L'operatore di riflessione rispetto al vettore $|\psi\rangle$ risulta:

$$2|\psi\rangle\langle\psi| - I = 2\frac{1}{\sqrt{4}}\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \frac{1}{\sqrt{4}}\begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} - I = \frac{2}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix}.$$

Pertanto:

$$|\phi''\rangle = (2|\alpha\rangle\langle\alpha| - I)|\phi'\rangle = \begin{bmatrix} -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{bmatrix} \frac{1}{\sqrt{4}}\begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix},$$

che in questo caso corrisponde proprio al vettore $|\beta\rangle$. Ci è bastata quindi una sola rotazione per ottenere proprio il risultato che volevamo. Occhio però che questo avviene perché siamo in un sistema a due qubit con una soluzione, in generale ci vogliono più rotazioni, e non sempre si finisce esattamente su $|\beta\rangle$, ma ci si arriva molto vicino.

(Geometria) L'operatore di riflessione rispetto a $|\psi\rangle$ risulta:

$$2|\psi\rangle\langle\psi| - I = 2\frac{1}{\sqrt{3}}\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \frac{1}{\sqrt{3}}\begin{bmatrix} 1 & 1 & 1 \end{bmatrix} - I = \frac{2}{3}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \end{bmatrix}.$$

Così otteniamo:

$$|\phi''\rangle = (2|\psi\rangle\langle\psi| - I)|\phi'\rangle = \begin{bmatrix} -\frac{1}{3} & \frac{2}{3} & \frac{2}{3} \\ \frac{2}{3} & -\frac{1}{3} & \frac{2}{3} \\ \frac{2}{3} & \frac{2}{3} & -\frac{1}{3} \end{bmatrix} \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{3}} \begin{bmatrix} -\frac{1}{3} \\ -\frac{1}{3} \\ \frac{5}{3} \end{bmatrix}.$$

Geometricamente:

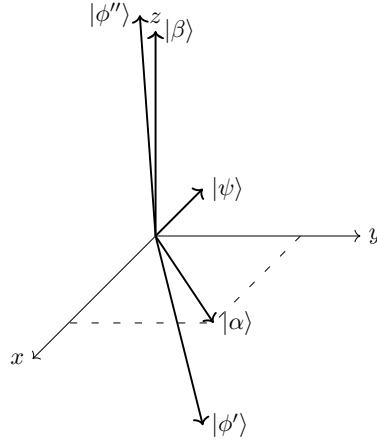


Figura 2.10: Rotazione del vettore $|\phi'\rangle$ rispetto a $|\psi\rangle$. Anche in questo caso il vettore rimane normalizzato, anche se può non sembrare.

Notate che in questo caso, anche se ci siamo avvicinati alla soluzione, *l'abbiamo addirittura superata!* Come già detto questo esempio geometrico serve solo a visualizzare l'effetto della rotazione e non corrisponde a uno scenario reale, ma ci mette in guardia dal fatto che se ruotiamo indiscriminatamente potremmo anche superare la soluzione.

Abbiamo così compiuto la prima rotazione, avvicinandoci di θ radianti al vettore $|\beta\rangle$. Per capire il perché ragioniamo sugli angoli tra i vettori. L'angolo che separa $|\phi\rangle$ da $|\alpha\rangle$ è di $\frac{\theta}{2}$ radianti. Riflettendo il primo vettore rispetto al secondo, ci siamo allontanati da $|\psi\rangle$ di θ radianti, ottenendo $|\phi'\rangle$. Infine, riflettendo quest'ultimo rispetto a $|\psi\rangle$, abbiamo ottenuto $|\phi''\rangle$, che è sempre distante θ radianti da $|\psi\rangle$, ma “dalla parte opposta”. L'angolo complessivo tra $|\phi''\rangle$ e $|\alpha\rangle$ è quindi di $\frac{3}{2}\theta$ radianti. Nelle successive iterazioni di Grover il ragionamento rimane lo stesso, solo che il vettore $|\phi''\rangle$ della i -esima rotazione diventa il vettore $|\phi\rangle$ della $(i+1)$ -esima. Per questo abbiamo distinto il vettore $|\phi\rangle$ dal vettore $|\psi\rangle$: alla prima iterazione corrispondono, ma dalla seconda in poi differiscono. Ma quante volte dobbiamo ruotare per arrivare alla soluzione? Per capirlo, proviamo a vedere il problema da un altro punto di vista.

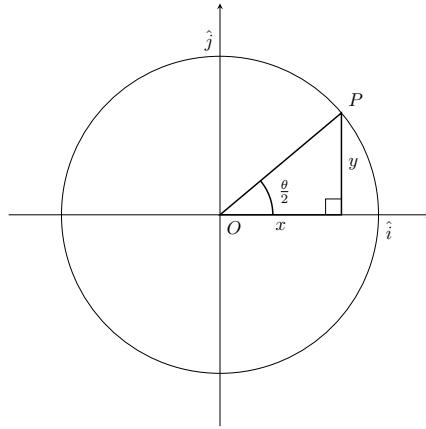
Sia dato un piano cartesiano con una circonferenza di raggio 1 (unitaria) e di centro 0, chiamata anche *circonferenza goniometrica*, come quella della Figura 2.11. Costruiamo un triangolo rettangolo su di essa, con ipotenusa il segmento \overline{OP} e cateti le proiezioni di questo sui due assi. Dato l'angolo $\frac{\theta}{2}$ compreso tra l'asse delle ascisse e l'ipotenusa del triangolo, si definisce con $\sin(\frac{\theta}{2})$ il rapporto tra il cateto opposto all'angolo e l'ipotenusa, e con $\cos(\frac{\theta}{2})$ il rapporto tra il cateto adiacente e l'ipotenusa (ad essere precisi, il rapporto tra le lunghezze).

Tuttavia, dato che stiamo parlando di una circonferenza unitaria, l'ipotenusa è lunga 1:

$$\sin\left(\frac{\theta}{2}\right) = \frac{y}{\|\overline{OP}\|} = \frac{y}{1} = y, \quad \cos\left(\frac{\theta}{2}\right) = \frac{x}{\|\overline{OP}\|} = \frac{x}{1} = x.$$

Il seno e il coseno ci danno quindi una misura della lunghezza della proiezione di \overline{OP} sull'asse delle ascisse e quello delle ordinate. Dato che il piano cartesiano è uno spazio vettoriale a due dimensioni (la cui base canonica è composta dai vettori $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ e $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$), se consideriamo l'ipotenusa \overline{OP} come un vettore, questo può anche essere scritto come:

$$\overline{OP} = \cos\left(\frac{\theta}{2}\right) \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \sin\left(\frac{\theta}{2}\right) \begin{bmatrix} 0 \\ 1 \end{bmatrix} = x \begin{bmatrix} 1 \\ 0 \end{bmatrix} + y \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Figura 2.11: Circonferenza unitaria avente \overline{OP} come ipotenusa e x e y come cateti.

Tornando al nostro problema iniziale, sappiamo che $|\alpha\rangle$ e $|\beta\rangle$ sono ortogonali tra loro, in quanto ogni componente a 0 nel primo è diversa da 0 nel secondo e viceversa. Questo implica che ogni vettore dello spazio \mathbb{C}^n può essere espresso in termini di $|\alpha\rangle$ e $|\beta\rangle$. In particolare, $|\psi\rangle$, vettore di lunghezza unitaria, apparterrà al loro span. Nulla ci vieta di vedere questi tre vettori come appartenenti ad una circonferenza goniometrica come quella nella Figura 2.11, e applicare lo stesso ragionamento visto nel piano cartesiano (i lettori più attenti si saranno sicuramente insospettiti quando abbiamo chiamato $\frac{\theta}{2}$ l'angolo nella circonferenza). Infatti l'ipotenusa sta al vettore $|\psi\rangle$ come i due vettori della base canonica stanno a $|\alpha\rangle$ e $|\beta\rangle$. Proviamo allora a calcolare la proiezione del vettore $|\psi\rangle$ sui due vettori ortogonali. Per farlo possiamo usare l'operatore di proiezione. Dato un vettore colonna u di lunghezza unitaria, l'operatore che permette di proiettare un qualunque vettore v su di esso è definito come:

$$\Pi = uu^\dagger.$$

Calcolare la lunghezza di questo vettore proiettato è molto semplice, basta calcolare la norma di Πv :

$$\|\Pi v\| = \|(uu^\dagger)v\| = \|u(u^\dagger v)\| = |u^\dagger v| \|u\|.$$

Esempio. (Algebra) fissato $\frac{\theta}{2}$ come l'angolo in radianti compreso tra $|\psi\rangle$ e $|\alpha\rangle$, risulta:

$$\text{Proiezione di } |\psi\rangle \text{ su } |\alpha\rangle = \left(\frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \frac{1}{\sqrt{3}} \begin{bmatrix} 1 & 1 & 1 & 0 \end{bmatrix} \right) \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

$$\cos\left(\frac{\theta}{2}\right) = \|\text{Proiezione di } |\psi\rangle \text{ su } |\alpha\rangle\| = \left\| \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \right\| = \sqrt{\frac{1}{4} + \frac{1}{4} + \frac{1}{4}} = \sqrt{\frac{3}{4}} = \sqrt{\frac{4-1}{4}}.$$

$$\text{Proiezione di } |\psi\rangle \text{ su } |\beta\rangle = \left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} [0 & 0 & 0 & 1] \right) \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

$$\sin\left(\frac{\theta}{2}\right) = \|\text{Proiezione di } |\psi\rangle \text{ su } |\beta\rangle\| = \left\| \frac{1}{\sqrt{4}} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \right\| = \sqrt{\frac{1}{4}}.$$

(Geometria) anche se in questo caso non servirebbe la controparte geometrica in quanto non mostreremo lo spazio tridimensionale, è utile fare questi calcoli una seconda volta per “convincersi” che c’è un pattern:

$$\text{Proiezione di } |\psi\rangle \text{ su } |\alpha\rangle = \left(\frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \right) \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}.$$

$$\cos\left(\frac{\theta}{2}\right) = \|\text{Proiezione di } |\psi\rangle \text{ su } |\alpha\rangle\| = \left\| \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \right\| = \sqrt{\frac{1}{3} + \frac{1}{3}} = \sqrt{\frac{2}{3}} = \sqrt{\frac{3-1}{3}}.$$

$$\text{Proiezione di } |\psi\rangle \text{ su } |\beta\rangle = \left(\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \right) \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \frac{1}{\sqrt{3}} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{3}} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

$$\sin\left(\frac{\theta}{2}\right) = \|\text{Proiezione di } |\psi\rangle \text{ su } |\beta\rangle\| = \left\| \frac{1}{\sqrt{3}} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\| = \sqrt{\frac{1}{3}}.$$

In entrambi i casi, otteniamo che:

$$\cos\left(\frac{\theta}{2}\right) = \sqrt{\frac{N-M}{N}}, \quad \sin\left(\frac{\theta}{2}\right) = \sqrt{\frac{M}{N}}.$$

che sono proprio i coefficienti che avevamo usato quando avevamo fatto vedere che $|\psi\rangle$ appartiene allo span di $|\alpha\rangle$ e $|\beta\rangle$:

$$|\psi\rangle = \sqrt{\frac{N-M}{N}} |\alpha\rangle + \sqrt{\frac{M}{N}} |\beta\rangle = \cos\left(\frac{\theta}{2}\right) |\alpha\rangle + \sin\left(\frac{\theta}{2}\right) |\beta\rangle.$$

Ragioniamo un’ultima volta per analogia per capire l’incredibile ruolo che hanno seno e coseno in questo contesto. Sappiamo che la potenza dei qubit sta nel fatto che questi possono essere in stati di sovrapposizione, che non sono né $|0\rangle$ né $|1\rangle$. L’esempio più famoso è lo stato $|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$, che corrisponde tra l’altro allo stato di ugual sovrapposizione per un sistema a un solo qubit. Tuttavia, quando effettuiamo una misura di un (sistema di) qubit, otteniamo un risultato classico, corrispondente a uno degli stati di base computazionale possibili. La probabilità di ottenere un certo stato di base dalla misurazione è pari alla sua ampiezza, sotto valore assoluto e al quadrato. Consideriamo allora la Figura 2.11 come rappresentazione bidimensionale di un sistema a un solo qubit (quella vera vorrebbe tre dimensioni, nella sfera di Bloch, ma dato che nella figura stiamo immaginando di trovarci su un campo reale non abbiamo bisogno della terza dimensione), dove lo stato è descritto dal vettore \overline{OP} , e i versori \hat{i} e \hat{j} sono gli stati di base computazionale $|0\rangle$ e $|1\rangle$. Allora:

$$\overline{OP} = x|0\rangle + y|1\rangle = \cos\left(\frac{\theta}{2}\right) |0\rangle + \sin\left(\frac{\theta}{2}\right) |1\rangle,$$

ovvero, la lunghezza delle proiezioni del vettore \overline{OP} sugli stati di base computazionale corrisponde all’ampiezza degli stati di base stessi. Se quindi prendiamo queste ampiezze, le mettiamo sotto modulo e le eleviamo alla seconda, otteniamo la probabilità di misurare $|0\rangle$ o $|1\rangle$ (lo stesso vale per il caso generale a n qubit):

$$|\phi\rangle = \cos\left(\frac{\theta}{2}\right) |\alpha\rangle + \sin\left(\frac{\theta}{2}\right) |\beta\rangle.$$

Dove, se prendiamo $|\cos(\frac{\theta}{2})|^2$, otteniamo la probabilità di misurare uno degli stati di base corrispondenti a un fallimento, mentre $|\sin(\frac{\theta}{2})|^2$ equivale alla probabilità di misurare uno degli stati di base corrispondenti a una soluzione. Questa è un’ottima notizia: vuol dire che finché conosciamo l’angolo tra $|\phi\rangle$ e $|\alpha\rangle$, siamo sempre in grado di calcolare la probabilità di misurare una soluzione o un fallimento. Per questo ci interessa ruotare il vettore iniziale vicino a $|\beta\rangle$. Minimizzando il valore del coseno e massimizzando quello del seno (essendo $|\alpha\rangle$ e $|\beta\rangle$ ortogonali c’è un angolo di esattamente $\frac{\pi}{2}$ tra loro, e $\cos(\frac{\pi}{2}) = 0$, $\sin(\frac{\pi}{2}) = 1$) riduciamo la nostra probabilità di fallimento e aumentiamo quella di successo.

Possiamo ora rispondere alla nostra domanda iniziale: quante volte dobbiamo applicare l'operatore di Grover per raggiungere la soluzione? Un numero di volte tale da permetterci di compiere una rotazione il più vicina possibile a $\rho = \frac{\pi}{2} - \frac{\theta}{2}$ radianti (l'idea è quella di trovare l'angolo $\rho' = \theta k + \frac{\theta}{2}$ tale da minimizzare il valore $|\rho' - \rho|$, dove k è il numero di rotazioni). Detta così però sembra che non abbiamo fatto progressi rispetto a prima. Cerchiamo quindi di rendere più “tangibili” questi angoli. Essendo $\frac{\theta}{2}$ e ρ angoli complementari, vale che:

$$\cos(\rho) = \sin\left(\frac{\theta}{2}\right) = \sqrt{\frac{M}{N}} \quad \Rightarrow \quad \rho = \arccos\left(\sqrt{\frac{M}{N}}\right).$$

Ricapitolando, sappiamo che:

- Tra lo stato iniziale e la soluzione $|\beta\rangle$ c'è un angolo di ρ radianti, con $\rho = \arccos(\sqrt{M/N})$.
- Possiamo spostarci verso la soluzione di θ radianti alla volta.

Quindi il numero di rotazioni R che dobbiamo effettuare in totale è:

$$R = \left\lceil \frac{\arccos(\sqrt{M/N})}{\theta} \right\rceil.$$

dove con $[n]$ indichiamo il numero naturale più vicino a n , arrotondando le metà per difetto (ovvero, ad esempio, $[2.7] = 2$, $[4.1] = 4$ e $[3.5] = 3$). Questo farà sì che alla fine la distanza in radianti tra il vettore ruotato $|\phi\rangle$ e il vettore soluzione $|\beta\rangle$ sia al più di $\frac{\theta}{2}$ radianti, dove vale sempre che $\frac{\theta}{2} \leq \frac{\pi}{4}$ (per convincersene, i lettori possono provare a eseguire una rotazione in un sistema a un solo qubit, dove ad esempio $|\alpha\rangle = |0\rangle$ e $|\beta\rangle = |1\rangle$). Questo ci assicura che in ogni caso la probabilità di successo sia maggiore o uguale a $\frac{1}{2}$. Fatte R rotazioni, però, di quanto abbiamo migliorato la situazione? Ovvero, a quanto si è ridotta la probabilità di fallimento, che all'inizio era $(N - M)/N$? Per capirlo dobbiamo basarci sull'angolo di massimo $\frac{\theta}{2}$ radianti che ci separa da $|\beta\rangle$. Per lo sviluppo in serie di Taylor, dato un angolo θ molto piccolo, vale che $\sin(\theta) \approx \theta$, e quando $M \ll N$ l'angolo $\frac{\theta}{2}$ sarà molto piccolo. Dunque:

$$\frac{\theta}{2} \approx \sin\left(\frac{\theta}{2}\right) = \sqrt{\frac{M}{N}} \quad \Rightarrow \quad \frac{\theta}{2} \approx \sqrt{\frac{M}{N}} \quad \Rightarrow \quad \arcsin\left(\frac{\theta}{2}\right) \approx \frac{\theta}{2} \approx \arcsin\left(\sqrt{\frac{M}{N}}\right) \approx \sqrt{\frac{M}{N}}.$$

La probabilità di fallimento si è quindi ridotta a M/N , mentre quella di successo è aumentata a $1 - M/N = (N - M)/N$. Diamo adesso un limite superiore al numero di rotazioni, dimostrando che di fatto la complessità dell'algoritmo di Grover è $\mathcal{O}(\sqrt{N})$. Essendo $|\alpha\rangle$ e $|\beta\rangle$ ortogonali, la nostra rotazione non sarà mai superiore a $\frac{\pi}{2}$ radianti, pertanto:

$$R \leq \left\lceil \frac{\pi}{2\theta} \right\rceil.$$

Supponiamo ora che $M \leq N/2$ (giustificheremo tra poco il perché di questa ipotesi). Algebricamente è verificato che:

$$\frac{\theta}{2} \geq \sin\left(\frac{\theta}{2}\right) = \sqrt{\frac{M}{N}},$$

quindi:

$$\theta \geq 2\sqrt{\frac{M}{N}} \quad \Rightarrow \quad R \leq \left\lceil \frac{\pi}{2} \frac{1}{2\sqrt{\frac{M}{N}}} \right\rceil = \left\lceil \frac{\pi}{4} \sqrt{\frac{N}{M}} \right\rceil \quad \Rightarrow \quad R \leq \left\lceil \frac{\pi}{4} \sqrt{\frac{N}{M}} \right\rceil.$$

Dato che il valore dominante in questa diseguaglianza è N , poiché $\frac{\pi}{4}$ è una costante e $M \leq N/2$, abbiamo che:

$$R = \mathcal{O}(\sqrt{N}),$$

che è proprio la misura della complessità dell'intero algoritmo di Grover. Il motivo per cui abbiamo supposto che M fosse minore o uguale a $N/2$ è che, se più della metà degli elementi nello spazio di ricerca sono soluzioni per il problema, già dopo la prima rotazione ci si allontana dalla soluzione. Infatti, in casi del genere, tanto vale prendere un elemento a caso dal dominio di ricerca e verificare se è soluzione per il problema o meno. È come lanciare una moneta finché non esce testa. Tuttavia non sempre potremmo

sapere in anticipo quante (non quali!) sono le soluzioni del problema, e potremmo eseguire l'algoritmo lo stesso anche quando $M > N/2$. In casi del genere possiamo cavarsela raddoppiando il numero di elementi del dominio di ricerca, dove sappiamo fin dal principio che gli elementi aggiunti non potranno essere soluzioni. Questo richiede semplicemente l'aggiunta di un qubit al nostro registro quantistico, pertanto non va a influire sulle prestazioni dell'algoritmo. Aggiungiamo un'ultima osservazione: negli esempi svolti in questa sezione abbiamo supposto che M fosse sempre uguale a 1, così da mettere in evidenza le parti fondamentali dell'algoritmo, ovvero l'interpretazione geometrica della rotazione e l'effetto della stessa anche da un punto di vista algebrico. Tuttavia, quando $M > 1$, la complessità si riduce, dato che aumenta l'angolo $\frac{\theta}{2}$, e il ragionamento non cambia. Ciò che però cambierà saranno le matrici di proiezione e rotazione, oltre ai vettori $|\alpha\rangle$ e $|\beta\rangle$.

Capitolo 3

Stato dell'arte dei videogiochi quantistici

Gli studi sul Quantum Computing trovano l'inizio della loro storia nel recentissimo periodo, considerato che già negli anni ottanta del secolo scorso le geniali menti di *Yuri Manin* e *Richard Feynman* proposero un nuovo modo di elaborare l'informazione basato sui principi della meccanica quantistica [9]. Allo stesso modo, negli ultimi cinque anni, aziende come l'IBM e Google si sono prodigate nello sviluppo di computer quantistici capaci di gestire dai 50 agli oltre 70 qubit. Ciò nonostante, sono sempre di più le aziende e programmati indipendenti a mostrare non poco interesse nelle applicazioni di questa tecnologia all'ambito videoludico. E non c'è da stupirsene, considerato che, sebbene la loro storia inizi nel 1947 con il *Cathode-ray tube amusement device* e in larga scala nel 1972 con l'avvento di *Pong* [23], oggi il mercato dei videogiochi è uno dei più grandi al mondo, capace di fatturare ogni anno miliardi di dollari in tutto il globo [6]. Tutto questo senza citare la valenza culturale e artistica che contraddistingue molti di essi. Ma che vantaggi potremmo ottenere coniugando il mondo del gaming e quello della computazione quantistica?

3.1 Il contributo di James Wootton

3.1.1 I primi videogiochi quantistici

Una prima risposta ci viene data dall'IBM, in particolare dal ricercatore *James Wootton*. Egli è autore del primo videogame quantistico mai esistito. Nel marzo del 2017 [28] infatti, quasi per gioco, scrisse *Cat-Box-Scissors* [30], una versione quantistica di Sasso-Carta-Forbici con un obiettivo tanto semplice quanto per nulla scontato: riuscire a dargli connotati strettamente quantistici. Per far ciò, è stato sviluppato un circuito quantistico a cinque qubit, basato sul funzionamento delle porte $R_x(\frac{\pi}{2})$ e $R_x(\frac{\pi}{2})^\dagger$. Applicando infatti per due volte di fila una di queste porte si ottiene una porta *NOT*, che invertirà il valore del qubit che ci passa attraverso. Se invece viene applicata prima una porta e poi l'altra, tale qubit non cambierà di valore (ricordiamo dal Capitolo 1 che ogni matrice associata a una porta quantistica è unitaria, pertanto $R_x(\frac{\pi}{2})R_x(\frac{\pi}{2})^\dagger = I$). Sapendo questo, ogni giocatore è chiamato a scegliere quale porta tra queste due applicare al qubit di gioco. Laddove il giocatore umano può scegliere mediante una *Command Line Interface* quale porta applicare, l'avversario, ovvero il computer quantistico su cui si svolge il gioco, porrà un qubit in uno stato di ugual sovrapposizione, poi misurato per scegliere se applicare l'una o l'altra porta. Vince il giocatore umano se il qubit alla fine si inverte (ovvero entrambi hanno scelto la stessa porta), il computer se invece il qubit non cambia (scegliendo la porta opposta rispetto a quella dell'avversario). Come il gioco da cui prende ispirazione, si tratta di un processo puramente stocastico, anche se, a causa del rumore quantistico presente oggigiorno su queste macchine, l'umano può tentare di prevedere quale sarà la prossima misurazione del computer quantistico. Anche il secondo videogame quantistico nasce dalla mano di James Wootton, e si tratta stavolta di una versione quantistica (semplificata) di Battaglia Navale [31]. La prima differenza rispetto al suo predecessore la si riscontra nel fatto che stavolta entrambi i giocatori sono umani, in competizione tra loro. Il primo

deve scegliere dove piazzare la sua nave tra sei luoghi possibili, mentre il secondo ha a disposizione tre bombe da poter lanciare, dove ogni bomba può prendere o un lato della nave o non prenderla affatto. La componente quantistica si vede nel fatto che le navi sono rappresentate da coppie di qubit poste in entanglement tra di loro. Tanto più una coppia si trova nello stato di ugual sovrapposizione, tanto più la nave associata è intatta. Ma una bomba lanciata su un lato della nave fa collassare questo stato, portando uno dei qubit più vicino a $|0\rangle$ o $|1\rangle$ mediante l'applicazione di alcune porte. Alla fine del gioco si misurano queste coppie per 1024 volte ciascuna, facendo una media dei risultati e verificando quanto è intatta la nave. Da notare come entrambi questi giochi, estremamente semplici, mettano in risalto un aspetto della computazione quantistica, ovvero il rumore e la possibilità di avere dei veri generatori di numeri casuali o l'entanglement tra qubit. Quest'idea di sfruttare i videogiochi (quantistici ma anche non) come mezzo non tanto per intrattenere quanto per insegnare i principi della meccanica quantistica sarà una costante in quasi tutti gli esempi che riporteremo. Questo non dovrebbe stupire più di tanto considerato che il termine “videogiochi” ormai non rende più giustizia a questo medium, che può proporsi in varie salse, da semplice svago per riempire il tempo libero a vero e proprio mezzo divulgativo, educativo e istruttivo, per non parlare delle varie arti (visive, sonore e narrative) che spesso amalgama saggiamente per creare qualcosa di innovativo e altrimenti impossibile da realizzare.

Dopo appena due mesi di tempo non solo Wootton scrisse una versione aggiornata di Battaglia Navale, ma implementò il primo programma quantistico che prendeva spunto da un vero videogioco, *Hunt the Wumpus*, titolo pioniere nel campo dei videogiochi di avventura, dove lo scopo era quello di vagare all'interno di una grotta alla ricerca del pericoloso mostro Wumpus, per ucciderlo. Nella sua versione quantistica, *Hunt the Quantupus* [29], le meccaniche principali del gioco non cambiano. La differenza sta nel fatto che i due Quantupus presenti nel labirinto, Alice e Bob, sono inizialmente in entanglement tra di loro, e tale informazione può essere sfruttata a vantaggio del giocatore. Infatti ogni Quantupus può essere, in un dato istante, colpito solamente o dalle frecce o dalle bombe, di cui l'avventuriero è abbondantemente munito. Tuttavia, se si colpisce un Quantupus con una freccia scoprendo che questi ne è immune, si può stare certi che l'altro ne sarà vulnerabile, e lo stesso dicasi per le bombe. Munito perfino di interfaccia grafica, questo è il primo videogioco quantistico che non solo ha uno scopo divulgativo, ma utilizza un principio della meccanica quantistica come meccanica di gioco.

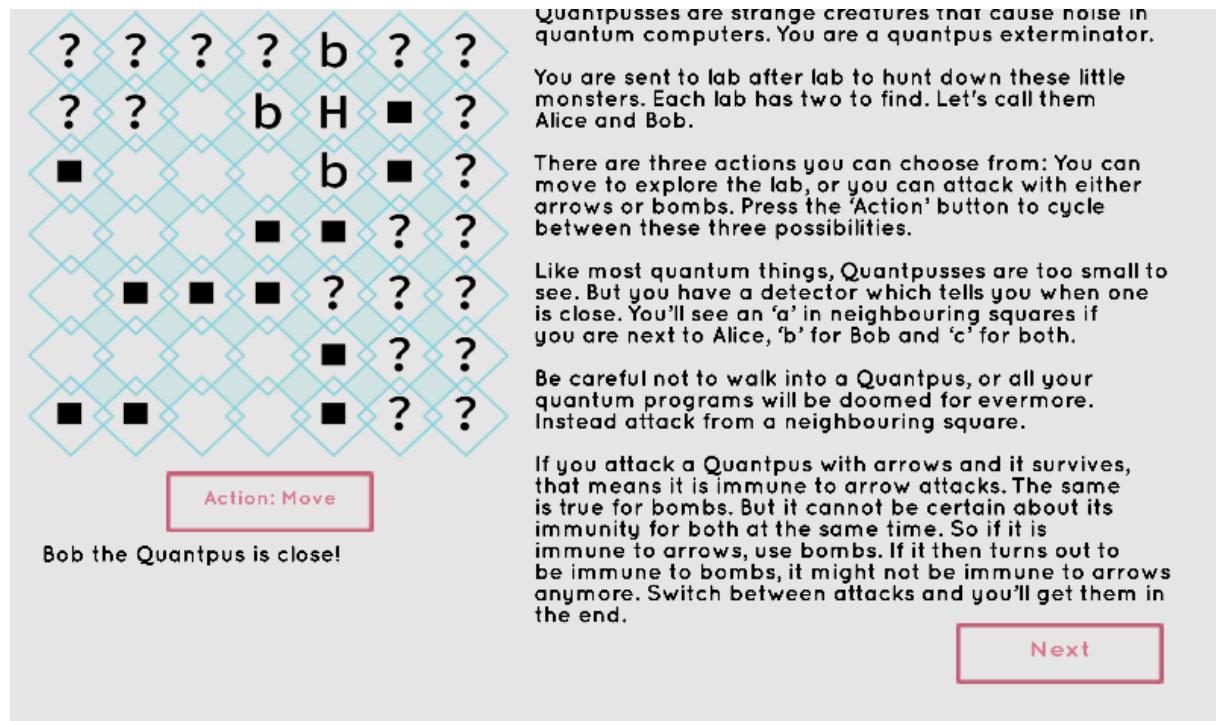


Figura 3.1: Screenshot di una partita a Hunt the Quantupus [29]

3.1.2 Quantum Supremacy e Quantum Awesomeness

Un tema oggi attualissimo nell'ambito del Quantum Computing è la cosiddetta *Quantum Supremacy*, ovvero la dimostrazione che i computer quantistici sono in grado di fare calcoli che nessun altro computer classico sarebbe in grado di fare in tempi utili [21]. L'argomento, estremamente affascinante quanto complicato, richiede tuttavia sia strumenti astratti che fisici per poter essere studiato ed elaborato, pertanto non c'è da stupirsi nel sapere che i principali studi in questo ambito sono condotti da IBM e Google. Per poter risolvere questa diatriba basterebbe pertanto trovare un problema che può essere affrontato facilmente con un computer quantistico e computazionalmente intrattabile per uno classico. Sebbene ci siano state più proposte di problemi con queste caratteristiche nel tempo, quello più interessante risulta essere il *Random Circuit Sampling*, in breve *RCS*. Un problema infatti ancora irrisolto è quello di trovare un generatore di numeri casuali. Esistono infatti generatori di numeri **pseudo**-casuali, che partendo da un valore chiamato *seme* generano una sequenza arbitrariamente lunga di numeri. La generazione di questi numeri segue tuttavia una procedura matematica ben definita e deterministica, che se scoperta, assieme al seme, permette di risalire all'intera sequenza pseudo-casuale generata (per maggiori informazioni si veda il Capitolo 4 di [3]). Tutto questo però è un problema dei computer classici. Quelli quantistici, di contro, posta l'assenza di rumore che è probabile raggiungeremo in un non troppo lontano futuro, e la natura dei qubit, che con l'applicazione di determinate porte possono trovarsi in stati di sovrapposizione $|\phi\rangle$ a metà tra lo stato $|0\rangle$ e $|1\rangle$, possono permetterci di creare veri e propri *generatori di numeri casuali*. Da qui nasce il problema RCS: lo scopo è quello di costruire un circuito quantistico in maniera casuale, ovvero applicando porte quantistiche a caso a qubit scelti altrettanto a caso (non importa quanto sia casuale questa procedura), e poi misurare i vari output dal programma. Se il circuito è veramente un circuito quantistico, alla fine ogni output ottenuto dovrebbe avere circa la stessa probabilità di essere misurato. Di fatto risolvere questo problema in tempo utile equivarrebbe a costruire un generatore di numeri casuali efficiente. Nulla vieta di simulare un circuito quantistico su una macchina classica per raggiungere questo scopo, ma quanto sarebbe efficiente tale simulazione? Molto poco, dato che la simulazione di un circuito quantistico su una macchina classica comporta un peggioramento esponenziale della sua complessità sia temporale che spaziale. Se riuscissimo quindi a implementare un circuito RCS su una macchina quantistica ottenendo velocemente misurazioni dell'output, avremmo risolto il problema della Quantum Supremacy. Google aveva già dichiarato, nel 2016, di voler dimostrare la Quantum Supremacy risolvendo il problema RCS usando un chip a 49 qubit capace di campionare distribuzioni di output inaccessibili ai computer classici in un tempo ragionevole. Nell'ottobre del 2019 pubblicò i suoi risultati su un articolo di *Nature* [2], avendo sviluppato un processore a 53 qubit chiamato "Sycamore" capace di risolvere il problema di meno di quattro minuti, laddove, a loro dire, con un supercomputer classico sarebbero stati necessari 10.000 anni. Quest'ultima affermazione sarebbe poi stata contestata dall'IBM, affermando che un algoritmo classico efficiente sarebbe stato capace di risolvere il problema in due giorni e mezzo sullo stesso dispositivo classico. In mezzo a tutto questo troviamo James Wootton, che ad agosto del 2017 decise di sviluppare Quantum Awesomeness [32] [27], un videogioco con lo scopo sia di facilitare e rendere più stimolante la comprensione del problema della Quantum Supremacy, sia di fungere da campo di prova per i device quantistici, così da testarne le capacità. Prendiamolo alla lontana per capirne la natura. Dato un computer quantistico, Quantum Awesomeness punta a verificare quanti qubit sono presenti in tale macchina, qual è la sua connettività, ovvero quante porte controllate posso applicare in un circuito arbitrario prima che il rumore renda i risultati insensati, e quanto questa riesce a tenere sotto controllo il rumore stesso. Ricordiamo che una porta controllata è una porta che collega due qubit, dove uno funge da controllo e l'altro da target. Allo stesso tempo, Quantum Awesomeness ci permetterà di creare circuiti RCS e verificarne la qualità, ovvero se ci sono qubit il cui esito è sempre prevedibile. Quindi, dato un computer quantistico a n qubit:

1. Vengono scelte alcune coppie di qubit casuali, e a ciascuna di queste viene applicata una *porta C-NOT parziale*, che ricordiamo dal Capitolo 1 essere porte dove al qubit di controllo viene applicata prima una porta $R_x(\phi)$, con $\phi \neq \frac{\pi}{2}$, che lo pone in uno stato di sovrapposizione tra $|0\rangle$ e $|1\rangle$, e poi una porta *C-NOT* che ha come target l'altro qubit della coppia. Ricordiamo inoltre che in questo modo, per entrambi i qubit, la probabilità di misurare 1 può essere più o meno elevata rispetto a quella di misurare 0 (dipende dall'angolo ϕ scelto), ma tutte le volte che il qubit di controllo viene

misurato 0 (parimenti per 1), lo stesso accade per quello target, e viceversa. L'unico vincolo qui sta nel numero di coppie che possiamo avere: deve essere strettamente minore di $\frac{n}{2}$.

2. Costruiamo un grafo a partire dal circuito nel modo seguente:

- Ad ogni qubit facciamo corrispondere un nodo.
 - Dopo aver eseguito il circuito per un certo numero di volte, ad esempio 1024, sapremo con che probabilità ciascun qubit ha dato 1 al momento della misurazione. Riportiamo, nel nodo corrispondente, questa esatta probabilità, come un numero reale che può assumere un valore da 0 a 100 (potevamo anche riportare direttamente il numero ottenuto, la scelta di convertirlo in percentuale è arbitraria). A questo punto, per maggiore leggibilità dell'utente, possiamo anche far corrispondere un colore a ciascun nodo dipendente da questa probabilità. Per esempio, un nodo sarà tanto più rosso quanto più la sua probabilità è vicina allo 0, mentre diventerà bluastro quando questa probabilità si avvicina al 100%.
 - Possiamo, per rendere la figura più leggibile all'utente, fissare anzitempo le *possibili* coppie di qubit a cui applicare una porta controllata. Per esempio, potremmo fissare il vincolo che al terzo qubit può essere applicata, in un qualunque momento, una porta controllata dove il secondo qubit coinvolto può essere solo uno tra il secondo, il quarto e il sesto. Ripetiamo, questo vincolo aggiuntivo è da applicare solo se vogliamo costruire un grafo più leggibile per l'utente, ma non è necessario per giocare. Basta solo, nel caso lo si volesse applicare, decidere all'inizio tutte le possibili coppie di qubit, così da non introdurre nei livelli successivi nuovi archi.
 - Per ogni coppia di nodi costruiamo un arco che li collega, identificato dai suoi estremi e dal suo peso. Su ogni arco infatti va riportata la probabilità che i due qubit relativi ai nodi che collega abbiano dato valori diversi dopo la misurazione. Riprendiamo l'esempio delle 1024 esecuzioni. Supponiamo che il secondo qubit del nostro circuito sia stato misurato come 1 per 256 volte, mentre il quinto addirittura 1024. Ciò vuol dire che il numero di volte in cui si sono trovati in disaccordo è $1024 - 256 = 768$, che è il 75% di 1024. Dunque nell'arco che collega i nodi relativi al secondo e quinto qubit va riportato il valore 75.
3. Mostriamo il grafo all'utente, chiedendogli di trovare le coppie di qubit che sono state poste in entanglement tra loro. Due qubit in questo stato infatti saranno tali per cui tutte le volte che la misurazione di uno ci ha restituito 1, lo ha fatto anche l'altro. Per trovarle pertanto bisogna cercare i nodi (collegati da un arco) il cui numero (o equivalentemente colore) è uguale, o alternativamente le coppie collegate da un arco il cui valore è 0 (perché ciò vorrebbe dire che i due qubit non si sono mai trovati in disaccordo).
4. L'utente, dopo aver osservato il grafo e comunicato alla macchina le coppie che secondo lui erano in entanglement, provoca una modifica del circuito. Infatti, per ogni coppia che era effettivamente in entanglement a causa dell'applicazione di una porta *C-NOT* parziale e che l'utente ha segnalato come tale, viene riapplicata la stessa porta, che per il fenomeno della uncomputation fa sì che i due qubit tornino allo stato iniziale, che tipicamente è $|0\rangle$. Se però una coppia era in entanglement e l'utente non se n'è accorto, allora non viene fatta, per quella porta, l'uncomputation, e i due qubit rimangono legati per il resto della partita. Se allo stesso modo il giocatore indica due qubit come entangled quando invece non lo erano, verrà applicata una porta *C-NOT* parziale a questa coppia, rendendoli quindi entangled.
5. A questo punto il livello corrente è completato, e si passa al successivo ricominciando dal passo 1, con la differenza però che, a causa di un possibile errore dell'utente, qualche coppia di qubit potrebbe essere ancora legata. Ciò andrà a complicare la partita del giocatore, perché se uno di questi qubit viene posto nuovamente in entanglement con un altro, non sarà così semplice accorgersene. Una volta che il giocatore non riesce più a trovare neanche una coppia di qubit in entanglement, il gioco termina. Lo scopo è pertanto quello di superare più livelli possibile prima di raggiungere l'inevitabile *Game Over*.

Abbiamo dunque spiegato le regole del gioco. Vediamo ora come questo si lega all'RCS e a una valutazione qualitativa di un computer quantistico.

- Lo scopo dell'RCS, come già detto, è quello di creare un circuito quantistico tale per cui ogni output abbia **circa** la stessa probabilità di uscire rispetto a qualunque altro possibile output. La parola “circa” qui gioca un ruolo fondamentale, perché nel momento in cui riusciamo a eliminarla del tutto vuol dire che abbiamo ottenuto un circuito con una distribuzione di probabilità uniforme degli output, dove nessuna stringa di bit misurata alla fine è favorita rispetto alle altre: hanno tutte la stessa identica probabilità di essere misurate. Per capirci meglio, se abbiamo ad esempio un circuito quantistico a 4 qubit, le possibili stringhe di bit diverse tra loro che possiamo misurare sono 2^4 . Nel momento in cui eseguiamo il circuito, per esempio, per 1024 volte, e otteniamo ciascuna di queste 16 stringhe diverse esattamente 64 volte, possiamo dire di aver costruito un piccolo generatore di numeri casuali. Ora, forse con 4 qubit è anche facile costruire un circuito che, almeno teoricamente, produce una distribuzione di probabilità perfetta come quella che abbiamo descritto. Ma sarebbe più interessante avere stringhe di bit molto più lunghe di così. Come facciamo quindi a costruire un circuito quantistico a tanti qubit che rispetti queste proprietà? Insomma, come facciamo ad essere sicuri che gli output prodotti da esso seguano una distribuzione di probabilità come quella desiderata?

È proprio qui che entra in gioco Quantum Awesomeness, che ci permette di costruire un circuito del genere se facciamo una partita non con lo scopo di vincere, *ma con quello di perdere con assoluta certezza*. L’idea è la seguente: iniziamo una partita, dove al primo round avremo una serie di coppie di qubit in entanglement tra loro. Il nostro scopo è sempre quello di trovare tali coppie, ma una volta fatto ciò faremo finta di non averle viste, e procederemo al livello successivo. Questo farà sì che non venga fatta l’uncomputation di alcuna porta. Ripetiamo questo procedimento fino a quando non arriviamo a un livello avanzato in cui:

1. Nessun nodo riporta il valore 0 o 100. Questo vuol dire che tutti i qubit hanno una probabilità diversa da 0 sia di misurare 0 che 1, ovvero, ogni qubit è in entanglement con almeno un altro.
2. Non riusciamo più a identificare con certezza coppie di qubit in entanglement tra loro. Se si verifica questa situazione vuol dire che abbiamo accumulato così tanti errori durante la partita da rendere il gioco impossibile da proseguire, raggiungendo quindi il Game Over. E questo ci piace molto, perché vuol dire che abbiamo creato un circuito dove non è prevedibile l’esito della misurazione di alcun qubit.

Quindi con Quantum Awesomeness la creazione di circuiti RCS davvero casuali è misurabile a occhio nudo: basta controllare se è possibile riconoscere qubit legati tra loro, e nel caso ciò sia vero vuol dire che il circuito non è abbastanza casuale, e vanno applicate ulteriori porte.

- Perdere di proposito però non sembra molto stimolante. Fortunatamente anche impegnarsi per cercare di portare avanti una partita e battere il record del mondo comporta dei vantaggi che non riguardano solo il mero orgoglio personale. Uno dei problemi più sentiti ad oggi riguardo i computer quantistici, come detto nel Capitolo 1, è la inevitabile presenza di rumore, che diventa sempre più importante durante l’esecuzione di un programma quantistico col passare del tempo. È proprio per colpa del rumore infatti se ad oggi non riusciamo a mantenere in esecuzione un circuito quantistico per periodi di tempo troppo lunghi, ottenendo risultati paradossali come un qubit che quando misurato, ogni tanto, restituisce lo stato 1 pur essendo, almeno in teoria, sicuramente nello stato $|0\rangle$. Fattori come il numero di qubit coinvolti nella computazione o la quantità di porte a due o più qubit applicate nel circuito non fanno altro che amplificare gli effetti nocivi del rumore. Quantum Awesomeness interviene in questo panorama fornendoci un modo divertente e intuitivo di misurare la capacità di un arbitrario dispositivo quantistico di gestire il rumore. Nel caso di assenza di quest’ultimo, giocare numerosi round di fila sarà estremamente facile, perché sarà sempre possibile, per un giocatore esperto e competitivo (ovvero interessato a vincere più livelli possibile) trovare le coppie di qubit giuste in un dato round. Tanto più invece questo è presente, tanto più il grafo prodotto ad ogni turno ci sembrerà costruito a caso, poiché qubit posti in entanglement non avranno la stessa identica probabilità di misurare 1 né tanto meno andranno d’accordo nel 100% dei casi,

avendo quindi sull'arco che li collega un valore diverso da 0. Per avere un'idea grafica di quello che stiamo dicendo si veda la Figura 3.2. Pertanto, per verificare la qualità di un dispositivo quantistico, tutto ciò che dobbiamo fare è provare a farci girare sopra Quantum Awesomeness e fare una partita, vedendo quanto a lungo riusciamo a giocare prima che il rumore renda il grafo costruito durante un certo round illeggibile (sottolineiamolo ancora: illeggibile non a causa di errori del giocatore, ma del rumore prodotto dalla macchina). Se vogliamo, possiamo dire di essere nel caso opposto rispetto a quello dei circuiti RCS: lì avevamo come obiettivo quello di far giocare un incompetente, qui invece vorremmo avere il campione del mondo!). Tanti più nodi avrà il grafo, tanto più saranno sofisticati i suoi archi e tanto più sarà il tempo di gioco che avremo a disposizione, tanto più avremo davanti un computer quantistico di qualità. *Google e IBM, fatevi sentire!*

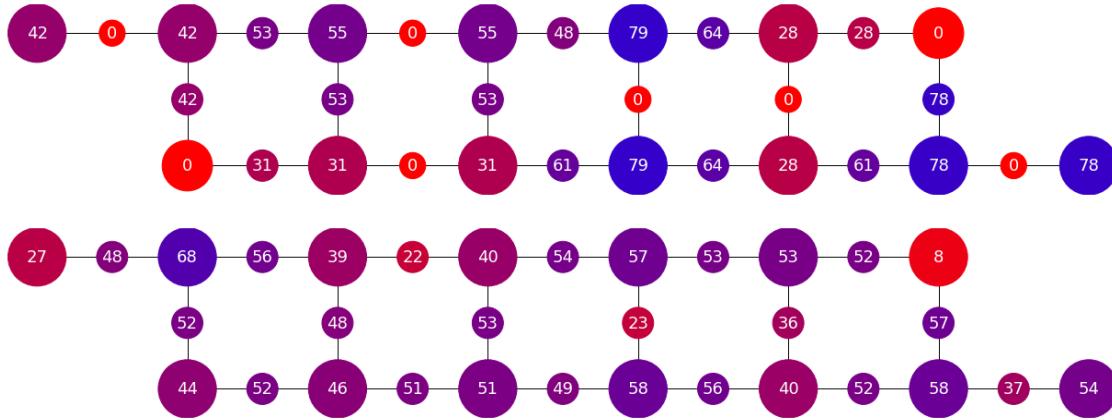


Figura 3.2: Esempio di grafi costruiti a partire da un circuito quantistico a 14 qubit. Sopra, il grafo ottenuto dopo aver eseguito per 1024 volte su un simulatore (esente quindi da rumore) il circuito quantistico. Sotto, il grafo costruito a partire dallo stesso circuito eseguito per 1024 volte su un vero computer quantistico, dove il rumore ha cambiato, a volte anche pesantemente, i valori dei nodi e degli archi. [32] [27]

C'è una piccola nota a margine che dobbiamo fare per quanto riguarda il problema RCS. Sicuramente Quantum Awesomeness è capace di generare circuiti quantistici dove l'esito della misurazione è completamente casuale. Ma se il nostro obiettivo è avere una sorgente di numeri casuali, perché non usare semplicemente delle porte Hadamard? Sappiamo infatti che queste porte, prendendo un qubit in uno stato di base computazionale, lo portano in uno stato di ugual sovrapposizione, dove cioè abbiamo la stessa probabilità di misurare 0 o 1. Quindi perché non usare un circuito a n qubit dove semplicemente applichiamo a ciascuno di loro una porta di Hadamard e poi misuriamo il risultato? O meglio ancora, misuriamo per n volte lo stesso qubit preparato nello stato $|+\rangle$? Questa è una strada sicuramente percorribile, ma come ogni cosa ha i suoi pro e i suoi contro.

Sicuramente l'ultimo approccio proposto è quello più economico in termini di risorse: ci bastano un qubit e una porta di Hadamard, e possiamo ottenere stringhe di numeri casuali lunghe quanto ci pare. Tuttavia l'applicazione di una porta di questo tipo richiede che il qubit, supponiamo inizializzato nello stato $|0\rangle$, venga ruotato di $\frac{\pi}{2}$ radianti rispetto all'asse X. Qualunque sia l'implementazione della porta su una macchina quantistica, è probabile che a un certo punto venga richiesto dal dispositivo l'angolo di rotazione, che sarà espresso in maniera classica, ad esempio come numero in virgola mobile. Essendo però questi angoli spesso una frazione di π , che è già un numero irrazionale, i numeri che li rappresentano saranno per forza di cose una loro approssimazione [8]. Quello che potrebbe accadere insomma è che, pur applicando una porta di Hadamard, potremmo, a causa del fatto che non siamo in grado di rappresentare in maniera esatta numeri irrazionali, effettuare una rotazione non di 90° , ma di un valore ad esso molto vicino, per esempio $89,9^\circ$ o $90,1^\circ$. Anche se su carta queste cifre sembrano irrisorie, non dobbiamo dimenticarci che lo scopo finale del problema RCS è proprio quello di generare numeri nella maniera più casuale possibile, per poi applicarli in ambiti in cui anche differenze minuscole come queste contano, ad esempio la crittografia. Infatti un ipotetico crittoanalista che conoscesse l'hardware quantistico usato per generare bit casuali nella maniera appena descritta, potrebbe sfruttare questa imprecisione a suo

vantaggio per decifrare degli eventuali crittogrammi. Inoltre gli stessi qubit potrebbero essere imperfetti, a seconda della loro implementazione fisica, e pertanto perfino il processo di inizializzazione nello stato $|0\rangle$ potrebbe essere soggetto a imprecisioni. Sommando tutti questi fattori ci rendiamo conto che, sebbene questo approccio sia il più semplice da adottare e su buoni computer quantistici produca dei risultati più che soddisfacenti, in generale non è una soluzione definitiva al problema RCS.

La strada percorsa da Quantum Awesomeness, allo stesso modo, ha pregi e difetti. Innanzitutto non è di così facile attuazione come quella precedente. Richiede device quantistici capaci di gestire numerosi qubit, porte e in grado di svolgere calcoli per periodi di tempo relativamente lunghi, abbastanza da permettere a un giocatore di concludere una partita creando il maggiore caos possibile. Ad oggi quindi questo approccio è indubbiamente più costoso di quello basato sulla porta di Hadamard per molti computer quantistici. Tuttavia il livello di casualità di cui è capace è, in linea di principio, maggiore rispetto a quello del suo concorrente. Ci sono vari elementi di aleatorietà che intercorrono in una partita di Quantum Awesomeness: la scelta delle coppie possibili, gli angoli di rotazione scelti per ciascuna porta R_x applicata e le misurazioni stesse dei qubit una volta giunti al Game Over. Tutti questi elementi possono portare a una maggiore imprevedibilità dei risultati. Usiamo il condizionale perché tutto sta nel come si decidono le possibili coppie e gli angoli delle porte. Se usassimo un generatore di numeri pseudo-casuali classico a questo scopo correremmo qualche rischio, in quanto un crittoanalista potrebbe risalire a questi generatori e avere informazioni parziali sulla struttura del circuito. È tuttavia vero che se, per esempio, avessimo bisogno di generare 1024 bit casualmente, potremmo fare k partite a Quantum Awesomeness dove ogni volta otteniamo $\frac{1024}{k}$ bit, impiegando per ogni esecuzione un seme diverso per il nostro generatore. Oppure potremmo, anziché cercare subito il Game Over, prolungare la partita azzeccando, nel corso dei round, alcune coppie, così da annullare porte precedentemente applicate e lasciare spazio a nuove combinazioni: un crittoanalista a quel punto non avrebbe modo di sapere come abbiamo giocato. Oppure potremmo applicare generatori di coppie e angoli “fisici”, ad esempio mettendo dei banalissimi pezzi di carta in un sacchetto, dove su ciascuno di essi è trascritta una possibile coppia di qubit o un angolo in radianti, e pescare alla cieca elementi da esso. In conclusione, Quantum Awesomeness propone una soluzione al problema RCS più complicata da implementare con i mezzi odierni, ma che ha svariati assi nella manica da giocarsi per assicurare una generazione casuale di bit.

3.1.3 Il primo Quantum Game Jam

A febbraio del 2019 troviamo un ulteriore evento storico per i videogiochi quantistici, al quale ovviamente non è mancato James Wootton, anche se stavolta più in veste di tutore: il *Quantum Game Jam* di Helsinki [26]. Con *Game Jam* si intende una competizione tra programmatori, game designers e simili, dove ogni gruppo di sviluppatori ha il compito di partorire un videogioco in un lasso di tempo che va dalle 24 alle 72 ore. Occasioni del genere servono a mettere alla prova la creatività, bravura e conoscenza dei partecipanti, stimolandoli a migliorare e, se il gioco è di buona fattura, anche a farsi pubblicità. La particolarità di questo Game Jam sta nel fatto che tutti i videogiochi prodotti avrebbero dovuto in qualche modo incorporare nelle loro meccaniche di gioco un aspetto della fisica quantistica. Come campo di prova, i partecipanti avrebbero avuto dei simulatori e dei veri computer quantistici forniti dall'IBM su cui far girare le proprie creazioni. Visto che uno dei principali studi nell'ambito del quantum computing riguarda la riduzione del rumore, è stato chiesto ai presenti di pensare come incorporare questo particolare effetto nelle meccaniche di gioco. E quello che più di tutti ha preso alla lettera questo consiglio è stato $Q|Cards\rangle$, un gioco di carte atto ad insegnare ai partecipanti le basi del quantum computing. L'idea era quella di simulare nel gioco l'esecuzione di un circuito che poi sarebbe stato mandato in esecuzione su una vera macchina. Ogni giocatore avrebbe avuto, a inizio partita, un personalissimo qubit e una mano di carte. Lo scopo era quello di giocare le giuste carte nel giusto ordine nel corso della partita, dove ogni carta corrisponde ad una trasformazione applicata al proprio qubit. Segnandosi l'ordine e il tipo delle giocate, sarebbe stato possibile costruire un circuito quantistico da mandare in esecuzione su un simulatore o su una macchina quantistica. Se l'esecuzione nel simulatore non fa altro che rispecchiare l'esito della partita, nel caso di esecuzione su computer quantistico abbiamo una dimensione aggiuntiva nel gameplay. Infatti, a causa del rumore, fino all'ultimo non è possibile stabilire quale sarà il qubit

vincitore. In tal caso, la strategia dei giocatori avrebbe dovuto tenere di conto di questa variabile, che li avrebbe portati a fare talvolta giocate più sicure (meno rumorose) ma meno vantaggiose, talvolta giocate più audaci (più rumorose) ma che avrebbero potuto portare più vicini alla vittoria. Ancora una volta, notiamo come la casualità fornita dai dispositivi quantistici sia estremamente invitante per i game designers che progettano un elemento di aleatorietà nel loro gioco. Ed è infatti proprio sulla capacità di questi dispositivi di fungere da random number generators che si basano altri due giochi del Game Jam, *Qubit Gardener* e *SneaQysnake*. Nel caso di *Qubit Gardener* abbiamo un rilassante passatempo, dove il nostro unico scopo è quello di annaffiare un giardino per vederne nascere i fiori. La particolarità sta nella vera generazione casuale delle piante, dove sia l'immagine sulla corolla che il colore e la forma dei petali sono generati casualmente. Molti giochi odierni, come *Minecraft* [18], attuano un processo di generazione procedurale del mondo, dove è praticamente impossibile avere due volte la stessa mappa da esplorare. Tuttavia gli ambienti che vengono generati diventano riconoscibili dopo una decina di partite: si tratta sempre delle stesse pianure, delle stesse miniere abbandonate e degli stessi villaggi con le stesse case, dalle quali si impara cosa aspettarsi. Con una generazione realmente casuale del mondo di gioco, però, ogni partita avrebbe sempre un elemento non riscontrabile altrimenti. *SneaQysnake*, una versione quantistica del famoso *Snake*, allo stesso modo, sfrutta questa casualità per generare il prossimo frutto che il serpente dovrà mangiare in un luogo imprevedibile. L'ultimo gioco prodotto durante questo Game Jam che citiamo è *Qubit the Barbarian*, gioco in stile roguelike [22] basato sulla misurazione di più qubit. Vestendo i panni di un barbaro esploratore, il nostro obiettivo è quello di esplorare una caverna sotterranea a più piani popolata da nemici, alla ricerca della chiave che permette di sbloccare le scale e scendere al piano successivo, il tutto cercando cibo che ci impedirà di morire di fame. La particolarità di questo prodotto videoludico sta nel fatto che ogni casella della grotta è associata a un qubit preparato in uno stato di base computazionale scelto a caso tra la base X, Y o Z, e il barbaro, in un qualunque momento della sua esplorazione, può decidere di effettuare la misurazione delle caselle (qubit) adiacenti in una di queste tre basi. Facendo una misurazione di una casella, questa può far sparire il muro che aveva sopra di sé, può farne comparire uno se prima non c'era o può non cambiare affatto, permettendo al giocatore di creare un percorso che lo porti alla destinazione desiderata. Pur essendo a livello di concept e di grafica molto rudimentale, *Qubit the Barbarian* ci dà un'idea di come un gioco quantistico del futuro potrebbe basare alcune delle sue meccaniche su peculiarità del quantum computing. Da quest'idea potrebbero nascere nuovi puzzle-game dove il giocatore deve trovare un pattern tra più elementi di gioco associati a qubit, scovando la base computazionale con cui sono stati preparati così da fare le giuste misure nel giusto ordine e raggiungere lo stato desiderato. Non solo, ma con questo gioco ci viene anche mostrato come il calcolo quantistico possa essere usato anche solo per compiere particolari operazioni in background, lontano dagli occhi del giocatore. Pur non conoscendo le basi computazionali, infatti, un qualunque giocatore può imparare con l'esperienza il funzionamento dei vari comandi, arrivando a muovere i muri circostanti a proprio piacimento.

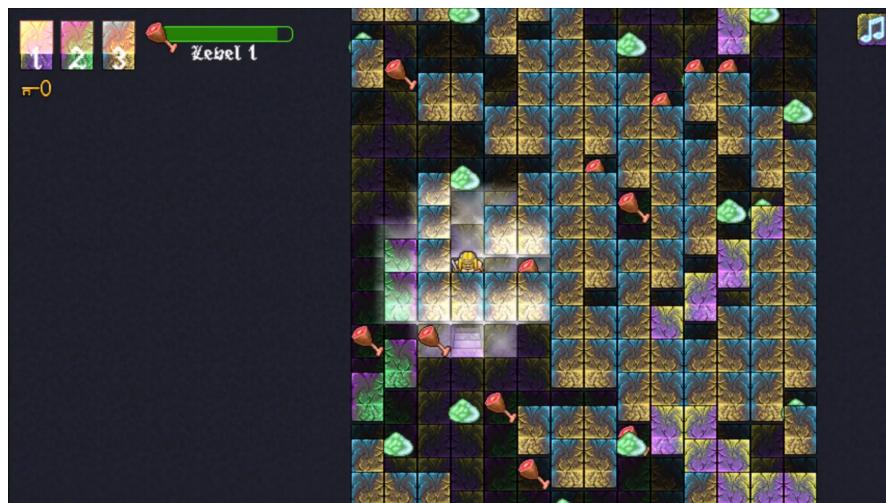


Figura 3.3: Screenshot di una partita a *Qubit the Barbarian* [32]

3.1.4 Generazione di mondi casuali

Abbiamo già sottolineato più volte come uno dei principali vantaggi che possiamo ottenere utilizzando macchine quantistiche al posto di quelle classiche è la creazione di un vero random number generator, dove i valori prodotti sono realmente casuali. Questo apre le porte, nel mondo del gaming, a una enorme quantità di possibilità che oggi ci sono proibite. Di fatto sono moltissimi i videogiochi moderni che basano la propria difficoltà e rigiocabilità su processi stocastici. Il famoso roguelike *The Binding of Isaac* [24] garantisce infatti migliaia di ore di gioco riempite di partite sempre differenti grazie a una generazione pseudo-casuale di mappe. L'idea fondamentale è che, dato un insieme di possibili stanze, oggetti, vincoli sulla dimensione della mappa e un *seed* iniziale, tipicamente una stringa a otto caratteri, il gioco è in grado di generare ogni volta un mondo da esplorare diverso da tutti quelli precedenti, garantendo un'esperienza sempre diversa al giocatore. Ciò nonostante, dopo svariate ore di gioco, si comincia ad intravedere un pattern: le stanze sono sempre le solite, popolate dagli stessi nemici che si trovano sempre nella medesima posizione e che compiono un numero predefinito di mosse. Lo stesso principio si applica a tutti i giochi del genere anche se in salsa leggermente diversa. Il più recente *No Man's Sky* [19] aveva fatto molto parlare di sé al momento del suo lancio, anche se per i motivi sbagliati. Il gioco prometteva infatti una generazione completamente procedurale dei suoi contenuti, a partire dai pianeti esplorabili fino alle creature che li abitavano. Di fatto però dopo qualche decina di ore di gioco si cominciavano a riconoscere le stesse strutture poste in ambienti diversi, le stesse specie, gli stessi consumabili. Insomma, la generazione casuale era sì presente, ma non in modo innovativo come invece sembrava dalle presentazioni antecedenti al lancio. La domanda che ci poniamo è pertanto: è possibile usare i computer quantistici per avere una generazione casuale di contenuti in un gioco? E perché non possiamo raggiungerla con mezzi classici? Tutte le volte che vogliamo produrre in modo casuale un puzzle, un livello o una mappa, dobbiamo scontrarci con una serie di vincoli difficili da soddisfare. Ad esempio vogliamo che i puzzle generati siano risolvibili, i livelli battibili e le mappe prive di trappole, ovvero zone che, se accedute, non permettono di tornare indietro. Un altro vincolo meno matematico che dobbiamo soddisfare è che il contenuto generato sia stimolante per il giocatore. Infatti nessuno vuole mettersi a risolvere un puzzle troppo facile o difficile, sia questo anche generato casualmente e munito di una soluzione. In linea di principio un gioco dovrebbe quindi generare casualmente un livello e verificarne in tempo reale la soddisfabilità e ludicità. Ma un'operazione del genere può richiedere tempi esosi ai dispositivi classici. Si pensi ad un semplice gioco dove, dato un insieme di città collegate da strade di costo variabile, si chiede al giocatore di trovare il percorso minimo che le attraversi tutte una volta soltanto. Il gioco, per poter verificare la qualità della soluzione proposta, dovrebbe prima trovare da sé quella ottima. Trattandosi però del problema del commesso viaggiatore, famoso per essere un problema computazionalmente difficile da risolvere con mezzi classici, non possiamo aspettarci che il software di gioco riesca a trovarne la soluzione in tempi ragionevoli. Qui entra in gioco la computazione quantistica, che ci permetterebbe di trovare il percorso minimo con uno speedup polinomiale. Allo stesso tempo, i computer quantistici potrebbero essere usati non solo per verificare velocemente la soddisfabilità di problemi generati casualmente, ma anche di generarli loro stessi. A dare una primo algoritmo per la generazione casuale di terreni di gioco con queste caratteristiche ci pensa come sempre James Wootton in un suo articolo [25], dimostrandoci come il mondo dei contenuti generati casualmente sia non vicino, ma già disponibile. Dovremo tuttavia attendere ulteriori sviluppi degli hardware quantistici prima che questi risultati possano essere sfruttati a pieno dai giocatori.

3.2 Altri esempi di quantum games

Con l'avvento di pacchetti software come Qiskit dell'IBM e Cirq di Google che hanno permesso a tutti di costruire e simulare circuiti quantistici usando un semplice linguaggio di programmazione, avendo inoltre la possibilità di mandare in esecuzione i propri programmi su veri computer quantistici, diversi sviluppatori si sono interessati sempre di più al mondo del quantum computing. Con ciò, alcuni game designers indipendenti non hanno tardato a sperimentare nuove tipologie di giochi che potessero sfruttare le peculiarità della meccanica quantistica. Vediamone pertanto alcuni esempi.

3.2.1 Quantum Game Chess

Un caso di videogioco quantistico che ha ottenuto molta popolarità negli ultimi anni è quello degli Scacchi Quantistici, o *Quantum Chess* [5], ideati da Chris Cantwell nel 2014. All'epoca il neo-laureato studente di fisica e game designer cominciò a pensare a un videogioco basato sui principi della meccanica quantistica, capace di insegnare questi ai giocatori in modo indiretto e pratico, così che anche i meno esperti potessero approcciarsi all'argomento in modo intuitivo. Il progetto venne notato dall'*Institute for Quantum Information and Matter (IQIM, California Institute of Technology)*, in particolare da *Spiros Michalakis*, che decise di aiutare Chris nel suo progetto. Dopo solo un anno Quantum Chess era pronto, e per pubblicizzarlo venne girato un video, “*Anyone Can Quantum*”, avente come protagonisti Stephen Hawking e Paul Rudd (attore che ha vestito i panni di *Ant-Man* nell'omonimo film), narrato da Keanu Reeves [4]. Il video, dai toni molto ironici, aveva lo scopo di mostrare agli spettatori le meccaniche base del gioco e come, con i mezzi odierni, non sia necessario essere dei geni della fisica per cominciare a lavorare con macchine quantistiche. Il gioco, oggi disponibile sulla nota piattaforma per la vendita di videogiochi *Steam*, venne perfino eseguito su una vera macchina quantistica durante il *Quantum Summer Symposium* di Google del 2020 [15]. Ma come funziona esattamente?

Ebbene, cominciamo col dire che tutte le regole degli scacchi classici continuano a valere in questa variante quantistica. L'unica eccezione sta nel fatto che non esiste il concetto di “scacco matto”. Per concludere una partita bisogna per forza catturare il re avversario. Il motivo di ciò risiede nelle meccaniche quantistiche del gioco, basate sui fenomeni di sovrapposizione, entanglement e misurazione. Ci sono infatti una serie di regole quantistiche aggiuntive che i giocatori possono sfruttare a loro vantaggio:

1. Durante il proprio turno, si può effettuare una *mossa quantistica* al posto di quella classica. Per fare ciò, basta scegliere un pezzo diverso dai pedoni, e fargli compiere *due mosse* di fila. Ad esempio, si potrebbe spostare la regina avanti di due caselle e poi a sinistra di tre. Se la regola fosse soltanto questa però ogni giocatore la applicherebbe durante ogni turno. Ma è qui che entra in ballo la prima meccanica di gioco quantistica: *un pezzo mosso in questo modo entra in uno stato di sovrapposizione, dove la probabilità che si sia effettivamente spostato nella casella desiderata diventa la metà della probabilità che si trovasse nella casella di partenza*. Riprendiamo l'esempio della regina di prima per capirci meglio. All'inizio della partita, la regina si troverà con una probabilità pari al 100% nella sua casella iniziale. Dopo che però è stata eseguita la mossa quantistica, questa si troverà con il 50% di probabilità nella nuova casella, e con l'altro 50% di probabilità ancora al punto di partenza. Per visualizzare meglio questo evento possiamo immaginarci che la scacchiera si sia sdoppiata in due: nella prima di queste la regina si è effettivamente mossa avanti di due e a sinistra di tre, nella seconda la mossa non è mai avvenuta, e il giocatore ha di fatto sprecato un turno. Si tratta quindi di una giocata rischiosa ma che può portare a diversi vantaggi, quantificabili solo quando uno di questi pezzi viene *misurato*.
2. Ma quand'è che effettivamente un pezzo viene misurato? Anche se in questa variante degli scacchi è ammesso che un pezzo si trovi in due caselle distinte in un dato momento grazie al fenomeno della sovrapposizione, continua ad essere proibito, come nella versione normale, che ci siano due pezzi diversi nella stessa casella in contemporanea. Questo vuol dire che se l'avversario decide ad un certo punto di attaccare la regina, prendendo di mira una delle due possibili caselle che potrebbe star occupando, deve essere stabilita la reale posizione di questo pezzo. È qui che avviene la misurazione: nel momento dell'attacco viene stabilita la casella in cui si trova la regina. Se l'avversario ha avuto fortuna e la misurazione ha stabilito che questa si trovava effettivamente nella posizione attaccata, allora verrà catturata. Altrimenti, sarà come se l'avversario avesse mosso il suo pezzo in una casella vuota, e la regina non si fosse mai spostata dalla sua posizione iniziale. Lo stesso accade se un pezzo dello stesso colore della regina decide di muoversi in una casella dove potrebbe trovarsi quest'ultima. C'è tuttavia ancora un fenomeno da considerare, quello dell'*entanglement*.
3. Immaginiamo di trovarci ancora nella situazione iniziale in cui la regina ha effettuato una mossa quantistica, e che sia il turno del giocatore che l'ha mossa. Negli scacchi normali un pedone non potrebbe superare un pezzo che gli sta di fronte. In questo caso però c'è solo il 50% di probabilità che la regina si trovi effettivamente davanti al pedone, impedendogli di muoversi. Pertanto, come

bisogna comportarsi? Per capirlo, dobbiamo introdurre l'ultima meccanica di gioco quantistica presente: la capacità dei pezzi di entrare in *entanglement*. In questo caso infatti il pedone potrà muoversi avanti di due caselle, “saltando” oltre la regina che *potrebbe* trovarsi davanti a lui. Per farlo però deve entrare in entanglement con il pezzo saltato. Questo vuol dire che i “destini” dei due pezzi sono ora legati indissolubilmente. Quando infatti dovrà essere misurata la regina o il pedone, lo stesso dovrà avvenire anche per l'altro pezzo. In tal caso avremo una scacchiera in cui o il pedone si è mosso e la regina no, o il pedone è rimasto fermo e la regina si è mossa.

Il motivo per cui quindi è necessario catturare il re avversario per vincere piuttosto che porlo sotto scacco è presto detto: il re potrebbe trovarsi in uno stato di sovrapposizione, e finché non lo si prova a catturare non si può sapere se effettivamente era sotto scacco oppure no. Ci sarebbero poi tutta una serie di scenari da considerare, nel caso si volesse fare una vera partita, come ad esempio capire cosa succede se un pezzo posto in sovrapposizione tenta di catturarne un altro, sia questo a sua volta in sovrapposizione o meno, ma ciò esula dai nostri scopi. I lettori più interessati ad approfondire possono provare il gioco da Steam o cercare su Youtube vari video di partite registrate. Quello che però ci interessa sottolineare è come Quantum Chess riesca a riassumere in maniera elegante un po' tutti gli aspetti dei videogiochi quantistici analizzati fino ad ora: il gioco è capace non solo di presentare i principi della meccanica quantistica in un modo più stimolante e comprensibile anche ai meno preparati, ma li integra al suo interno al punto da:

1. Farne le sue meccaniche chiave di gioco
2. Essere in grado di rendere ancora più realistica l'esperienza facendo decidere a un vero computer quantistico l'esito di una giocata. Ogni pezzo è infatti rappresentato da un qubit, a cui vengono applicate le porte necessarie e che viene misurato nel momento del bisogno.

Ovviamente non chiediamo che un videogioco quantistico del futuro debba per forza far andare a braccetto il lato didattico con quello ludico. Anche se riuscisse “solo” a velocizzare calcoli ad oggi lenti come quelli di alcune schermate di caricamento avremmo raggiunto un traguardo di non poco conto. Ciò nonostante, Quantum Chess rimane un esempio di lustro nel panorama dei quantum videogames moderni.

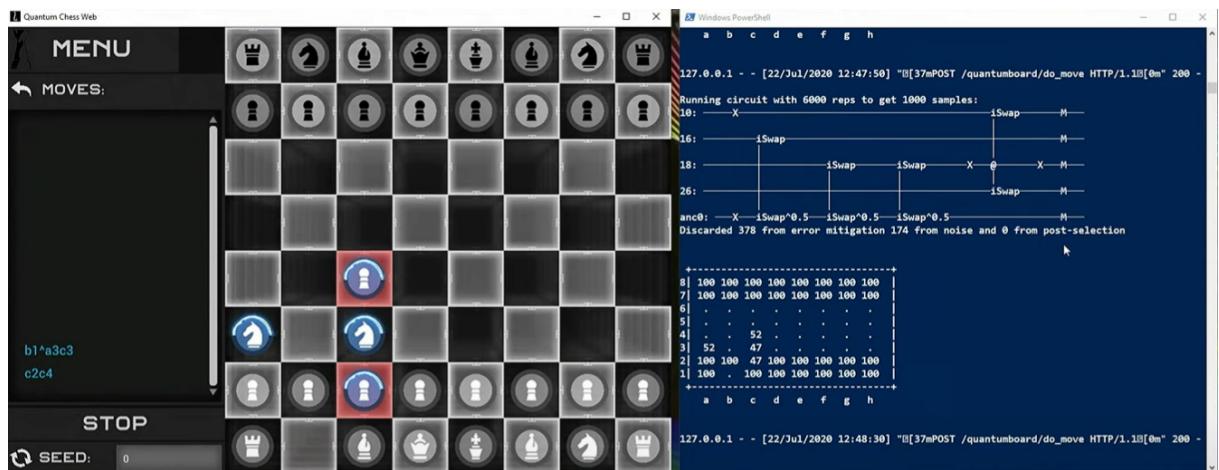


Figura 3.4: Esempio di esecuzione di una partita a Quantum Chess su un computer quantistico di Google [15]

3.2.2 Quantum Pokémon Fight

Parlando di come con i mezzi quantistici potremmo riuscire a velocizzare calcoli ad oggi costosi, è stato sviluppato in maniera indipendente un prototipo di intelligenza artificiale quantistica per i videogiochi della saga *Pokémon* [20], a cura di Michaël Rollin [13]. Tale codice cerca di simulare il comportamento di un allenatore nemico durante uno scontro tra allenatori Pokémon, basato sulle regole fondamentali del gioco, come i rapporti tra i tipi delle mosse e dei mostri atti coinvolti. Il primo aspetto quantistico lo troviamo nella generazione casuale dei cosiddetti “effetti secondari” che una mossa può generare. Nel

mondo del competitivo Pokémon moderno infatti uno dei principali elementi di gioco che rende talvolta le partite impari è la frequenza dei cosiddetti “*brutti colpi*”, ovvero attacchi critici che si verificano con probabilità del 4.17% [14] e che permettono ad una mossa di causare un danno 1.5 volte maggiore del normale. Dato che le strategie dei giocatori più bravi si basano sul calcolo preventivo dei danni che un pokémon può aspettarsi di ricevere quando attaccato, è chiaro che questi colpi critici possono risultare un elemento di aleatorietà piuttosto importante, a volte tale da decidere le sorti di un match. Non è troppo raro vedere partite in cui lo stesso giocatore subisce anche per due o tre volte di fila un brutto colpo, venendo sconfitto. Questo è dovuto anche al fatto che la decisione di applicare questo effetto secondario ad un attacco deriva da un generatore di numeri pseudo-casuali, che nel corso di una partita può creare risultati troppo impari. Se però questo processo di decisione fosse completamente quantistico, come accade nell'articolo di Rollin, allora potremmo aspettarci una distribuzione di brutti colpi più regolare, riducendo situazioni assurde dove l'esito della partita è determinato unicamente dai colpi critici.

L'altro aspetto quantistico trattato riguarda la scelta della mossa da compiere da parte dell'avversario. Per prendere questa decisione viene utilizzato l'algoritmo di ricerca quantistico di Grover. Ricevendo come input le mosse conosciute dal proprio pokémon in campo e le resistenze e debolezze della creatura avversaria, il circuito, composto come al solito da un oracolo e dal diffuser, è in grado di dire se conviene sostituire il mostriattolo schierato con uno della riserva oppure attaccare con una specifica mossa, quella che risulterà super-efficace sul nemico. Pur essendo solo un prototipo, si potrebbe partire da questa implementazione per sviluppare ulteriori meccanismi di intelligenza artificiale atti a velocizzare processi di scelta all'interno di questi giochi e non solo. Riprendendo l'esempio del competitivo pokémon, un altro problema con cui vari giocatori si scontrano quotidianamente è la scelta delle creature da portare in un match. Quando infatti ha inizio una partita, vengono dati novanta secondi di tempo ai due contendenti per decidere tre o quattro dei sei pokémon della propria squadra da far partecipare all'incontro. Se però il giocatore non effettua questa scelta nel tempo prestabilito, il software renderà automaticamente partecipi i primi quattro pokémon della squadra. Questo meccanismo ha il vantaggio di essere molto veloce, condizione necessaria quando si tratta di lotte on-line, ma altrettanto inefficiente, considerato che spesso le prime quattro creature potrebbero non essere le più performanti per il match corrente. Un circuito quantistico basato su Grover che riceve come input informazioni sui pokémon propri e avversari, tuttavia, potrebbe scegliere velocemente la squadra migliore per la partita corrente. Tali informazioni potrebbero essere elementari, come le tipologie dei vari pokémon coinvolti, portando quindi ad un circuito relativamente semplice da costruire al prezzo di una scelta sub-ottimale, oppure ricche di dettagli complessi, come le statistiche, le abilità, gli oggetti portati e le mosse, costringendoci a costruire un circuito più complicato ma che conduce all'ottimo.

3.3 L'universo quantistico di Outer Wilds

Fino ad ora abbiamo esaminato vari videogiochi che erano quantistici sia di nome che di fatto. Il loro scopo non era solo quello di insegnare le meccaniche della fisica quantistica in modo accessibile, ma di incorporarle direttamente al loro interno, con la possibilità di essere eseguiti su un vero quantum device. Questa scelta di design ha il vantaggio, se usata in maniera creativa, di essere molto efficace nell'insegnare l'argomento a chi sta muovendo i suoi primi passi in questo mondo tanto complesso quanto affascinante. Tuttavia, è innegabile che un individuo con scarse conoscenze di base sul tema faticherà molto di più a coglierne l'essenza, anche se messo davanti a una scacchiera di Quantum Chess. Questi giochi, legandosi inevitabilmente a un contesto strettamente matematico, potrebbero dunque non catturare l'attenzione del pubblico meno preparato, che però in potenza potrebbe essere interessato ad approfondire tale topic. Purtroppo anche l'occhio vuole la sua parte, e in questo, almeno ad oggi, i quantum games sono un passo indietro rispetto ai videogiochi sviluppati per dispositivi classici, dato che la grafica migliore l'abbiamo riscontrata con la scacchiera 2D di Quantum Chess e con la *pixel art* di Qubit the Barbarian, che impallidiscono di fronte a titoli classici degli ultimi anni. La domanda diventa pertanto: sacrificando l'elemento di rigorosità, è possibile creare videogiochi che girino su macchine classiche ma che diano un'intuizione ai loro giocatori di come funzioni il mondo della fisica quantistica? La risposta è ovviamente sì, e ci sono vari esempi al riguardo. L'esponente che però riteniamo incarni al meglio questa volontà è un titolo uscito nel 2019 dalle geniali menti dello studio *Mobius Digital* [1]: stiamo parlando di *Outer Wilds*. Segue

adesso una breve introduzione al mondo di gioco, continuata da una trattazione su come questo riesca ad incarnare molto bene gli elementi principali della meccanica quantistica. È chiaro che per poter far ciò bisogna anticipare alcuni dei suoi contenuti, se pertanto siete interessati a dargli uno sguardo “alla cieca” vi consigliamo di saltare per il momento questo paragrafo e andare direttamente al prossimo capitolo. In Outer Wilds vestiremo i panni di un *teporiano*, trattasi di una razza aliena appartenente ad un sistema solare diverso dal nostro. L'avventura ha inizio nel villaggio di “Cuore Legnoso”, l'equivalente della nostra Terra, dove sono partite ormai da anni le Esplorazioni Outer Wilds, ovvero spedizioni nello spazio atte a raccogliere maggiori informazioni sul nostro sistema solare e sull'antica civiltà che vi ha abitato a lungo per poi sparire lasciando poche tracce: quella dei *nomai*. Visitando il museo del nostro villaggio che ci introdurrà alle meccaniche principali di gioco, non potremo fare a meno di passare di fronte a una statua costruita dai nomai stessi che ne raffigura un esemplare, la quale si animerà per qualche secondo di fronte ai nostri occhi, compiendo un gesto che ci cambierà il futuro per sempre. Di lì a poco avrà inizio la nostra prima spedizione spaziale, che però purtroppo terminerà dopo appena ventidue minuti: allo scadere di questo periodo di tempo il sole esploderà, trasformandosi in una supernova che cancellerà noi e l'intera galassia. Per un motivo a noi sconosciuto, ma che sappiamo essere legato in qualche modo a quella strana statua, verremo catapultati ventidue minuti indietro nel tempo, potendo di fatto ripetere il nostro viaggio tutte le volte che vogliamo... o quasi. Non c'è un obiettivo specifico nel gioco, siamo noi a decidere dove muoverci e quali risposte cercare alle numerose domande che Outer Wilds ci insinuerà nel corso delle nostre spedizioni. Chi erano i nomai? Perché si sono estinti? Perché siamo intrappolati in un loop temporale? Non si può fare nulla per evitare che il sole collassi, cancellando la nostra specie, cultura e tutti coloro che ci sono cari? E qual è il nostro ruolo nell'universo? Solo esplorando i vari pianeti del sistema solare e cercando tracce della civiltà nomai, composte da manoscritti, edifici in rovina e astronavi abbandonate potremo dare una risposta a tutti questi quesiti.

Dopo aver svolto un certo quantitativo di esplorazioni ci renderemo conto che i nomai provavano particolare interesse per un'entità, apparentemente un corpo celeste lontano anni luce da noi, chiamata *l'Occhio dell'universo*, che però non sono mai riusciti a raggiungere. Ulteriori studi hanno dimostrato che c'è una correlazione tra l'Occhio e un altro astro che è invece visibile, e teoricamente raggiungibile, dal nostro sistema solare, trattasi della *Luna Quantica*. Tale luna presenta delle proprietà molto particolari. Quando non è osservata da un *osservatore consapevole*, questa è contemporaneamente in orbita sui cinque pianeti che gravitano attorno al nostro sole e sull'Occhio. Quando invece viene osservata, in particolare da noi, questa “decide” la sua posizione, e comincia ad orbitare solo attorno ad un pianeta. Qui troviamo la prima meccanica quantistica applicata al mondo di Outer Wilds, ovvero la sovrapposizione: fintanto che la luna non viene misurata da un osservatore consapevole che si mette a guardarla, questa si trova in uno stato di sovrapposizione, che fa sì che sia contemporaneamente in orbita attorno a sei pianeti. Come anticipato l'interpretazione di questo principio (e non solo) è piuttosto romanzata nel gioco, ma è proprio questo il suo elemento di forza, ovvero non fare affidamento su una spiegazione rigorosa del fenomeno ma lasciare che sia l'utente a intuirlo e interiorizzarlo con parole sue. Detto questo, un nuovo giocatore difficilmente riuscirà ad addentrarsi nell'atmosfera di questo corpo celeste, che svanirà davanti ai suoi occhi nel momento in cui proverà ad atterrарvi. E se anche dovesse essere abbastanza fortunato da entrarci provando strategie casuali fino a trovare quella corretta, non riuscirebbe comunque a scoprirne i segreti più profondi. Questo perché egli ancora non conosce tutte le leggi necessarie per l'esplorazione di tale satellite, che però fortunatamente i nomai ci hanno lasciato in luoghi specifici del nostro sistema solare. In particolare, bisognerà visitare la *Torre quantica di sperimentazione*, la *Torre della conoscenza quantica* e la *Caverna Fondolago* per esplorare l'astro a fondo. Le due torri non sono di nostro particolare interesse, dato che introducono regole incentrate più su meccaniche esplorative del gioco che sulla fisica quantistica stessa. La Caverna Fondolago tuttavia introduce il concetto dell'entanglement quantistico con la cosiddetta *Regola della correlazione quantica*. In questo luogo troveremo infatti un frammento di Luna Quantica (ce ne sono vari sparsi per il mondo di gioco), con annessi manoscritti dei nomai che raccontano di come grazie ad esso abbiano scoperto una proprietà quantistica piuttosto interessante. Tale caverna è situata nelle profondità di uno dei due pianeti più vicini al sole, *Gemello Brace*, pertanto sarebbe in uno stato di costante oscurità se non fosse per la presenza della nostra torcia. Proprio come accade per la luna, questo oggetto si trova in uno stato di sovrapposizione che lo fa essere contemporaneamente in più luoghi del pianeta, almeno finché non viene osservato attivamente. Allo stesso modo, anche

l'assenza di luce costringe ad uno stato di sovrapposizione il frammento, che però può essere misurato se illuminato e posto sotto i nostri occhi. Può dunque balenarci in testa un dubbio: cosa accade se tocchiamo quest'oggetto quando è buio? La risposta è presto detta: entriamo in entanglement con lui. Al momento della riaccensione della torcia il frammento deciderà la sua posizione nel pianeta, facendoci trovare talvolta “teletrasportati” all'interno di un'altra caverna. Questa regola ci permetterà, una volta atterrati sulla Luna Quantica, di capirne meglio le proprietà e di esplorarla a fondo. Come abbiamo già detto la luna può essere il satellite di sei corpi celesti differenti, e una volta superata la sua atmosfera (e quindi misurata, perché stiamo letteralmente toccando e osservando la sua superficie) noteremo come questa si trovi in entanglement con il pianeta attorno al quale sta orbitando, come ci fa notare anche la nomai *Solanum* nella Figura 3.5. Se infatti vi atterriamo quando orbita sul *Profondo Gigantesco*, un pianeta composto prevalentemente da acqua e uragani, la troveremo allagata con un enorme vortice su uno dei poli, diversamente da come la troveremmo se stesse orbitando su *Rovo Oscuro*, ovvero ghiacciata e piena di enormi rovi. Fintanto che però vaghiamo stupiti sulla sua superficie felici di essere finalmente giunti su questo luogo sacro e occulto, non otterremo risposte alle domande che cerchiamo. Per poter proseguire dovremo trovare il *Tempio della Luna Quantica* eretto dai nomai, un edificio capace di celarci nell'oscurità così da porci in entanglement con l'astro. Il nostro obiettivo sarà quello di “saltare” di orbita in orbita, sfruttando le caratteristiche che la luna eredita dal pianeta più vicino, per giungere al suo polo nord. Qui infatti, accedendo ancora una volta al tempio, saremo capaci di misurare il corpo celeste quando si trova nelle prossimità dell'Occhio dell'universo. Esplorando adesso la luna ci troveremo davanti a *Solanum*, l'ultima nomai rimasta in vita, che ci racconterà di come il suo popolo sia stato attirato su questo sistema solare proprio dall'Occhio, e di come abbia cercato in tutti i modi di raggiungerlo per svelarne la natura, ma riuscendo “solo”, con il suo contributo, a mettere piede sulla Luna Quantica, senza spingersi oltre. Outer Wilds quindi non punta a una descrizione rigorosa dei fenomeni quantistici, ma fa appello all'intuizione e, perché no, ai sentimenti del giocatore per introdurlo ai concetti fondamentali di questo mondo complesso, facendolo contemporaneamente affezionare a un'ambientazione e una specie che, seppur esistano solo dietro uno schermo, diventano, man mano che se ne svelano i segreti e la storia, sempre più tangibili.

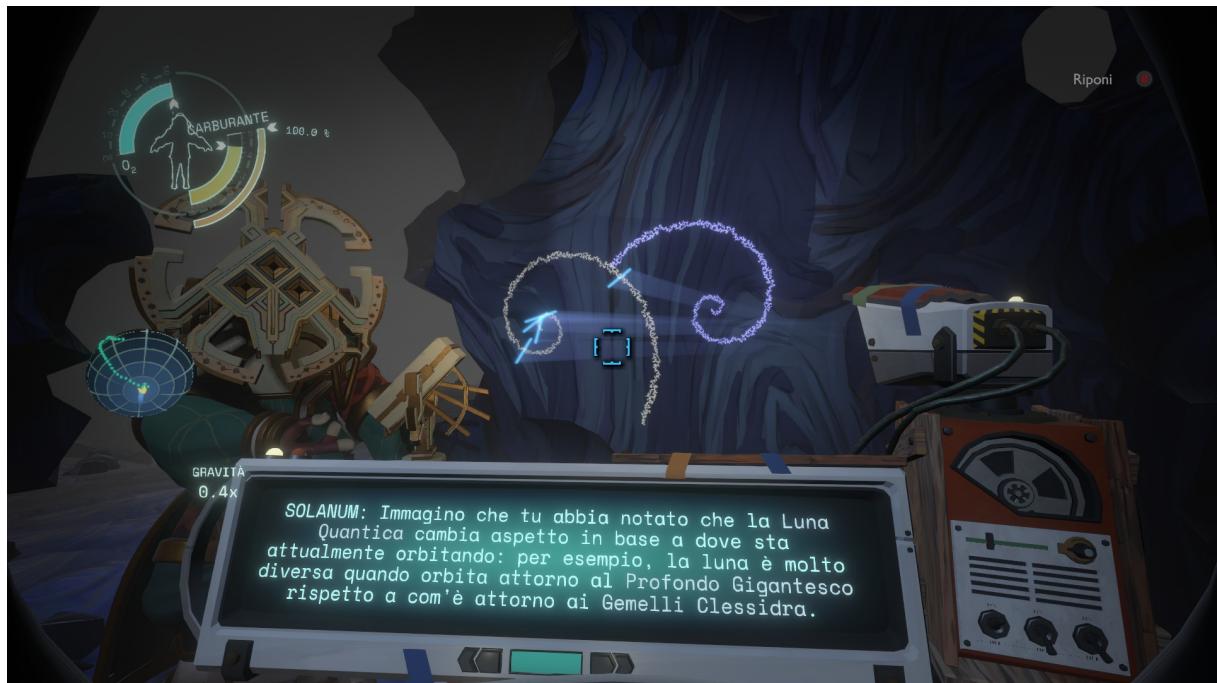


Figura 3.5: “Io e te non sembriamo legare molto. Eppure questo incontro sembra essere speciale. Spero che tu sia d'accordo se ci considero amici.” -Solanum

Capitolo 4

Esplorazione di dungeon mediante Grover

Nello scorso capitolo abbiamo analizzato una serie di esempi che hanno avuto l’obiettivo comune di avvicinare il mondo del Quantum Computing a quello dei videogiochi. Tuttavia la maggior parte di questi ultimi costruivano alcune delle loro caratteristiche ludiche basandosi su principi della meccanica quantistica, sia per divulgare l’argomento sia per creare esperienze virtuali inedite. Sebbene questo sia ottimo e estremamente efficace in un momento in cui il medium dei videogiochi sta ottenendo sempre più successo e il mondo del Quantum Computing fatica ad essere compreso in modo semplice, pur avendone bisogno per esprimere appieno le proprie potenzialità attraverso le menti e le mani di giovani sviluppatori, esistono anche altri modi di coniugare questi due universi. Ne abbiamo avuto un esempio quando abbiamo parlato di Quantum Pokémon Fight [13]: l’obiettivo non era quello di ideare nuove meccaniche di gioco, bensì sfruttare i circuiti quantistici per velocizzare operazioni inefficienti o comunque lente. In questo capitolo, seguendo tale filosofia, cercheremo di costruire un circuito che utilizza le potenzialità dell’algoritmo di Grover per l’esplorazione di dungeon generati casualmente all’interno di una particolare categoria di giochi che sta avendo sempre più successo nell’ultimo periodo: i *roguelike* [22].

Con il termine *roguelike* (traducibile letteralmente come “*stile Rogue*”) si intende una categoria di giochi che si ispirano al capostipite del genere: Rogue appunto, videogioco nato nel 1980 sulla base del famoso gioco di ruolo *Dungeons & Dragons*. L’idea fondamentale era quella di vestire i panni di un esploratore avventuratosi all’interno di un dungeon (labirinto sotterraneo) composto da varie stanze connesse tra di loro mediante corridoi, all’interno delle quali si potevano trovare nemici, potenziamenti o il boss di fine livello. Il gioco originale prevedeva un’interfaccia utente composta da soli caratteri ASCII, ma ad oggi i giochi roguelike sono diventati talmente popolari e di successo da essere sia gradevoli da vedere che da udire, con grafiche e colonne sonore di tutto rispetto. Alcuni elementi comuni di questi giochi, e che anzi li definiscono, sono:

- Ad ogni partita il dungeon deve essere generato casualmente, mutando i tipi di stanze, il modo in cui sono connesse e i nemici e potenziamenti che è possibile trovare al loro interno.
- Alcuni oggetti all’apparenza identici devono avere effetti diversi in partite diverse. Un’ampolla verde ad esempio potrebbe essere una pozione che potenzia la forza del personaggio in una partita, mentre in quella successiva potrebbe essere un pericoloso veleno.
- Quando il personaggio muore, tutti i progressi fatti vengono perduti, e nella partita successiva bisogna ricominciare dal primissimo livello senza alcun tipo di potenziamento. Il senso di progresso in questi giochi viene comunque garantito in modo laterale, ad esempio lo sconfiggere un determinato boss ad un livello avanzato può garantire al giocatore la possibilità di trovare un nuovo tipo di potenziamento nelle sue partite successive.

Questa categoria di giochi si distingue nell’avere diversi esponenti che presentano una difficoltà estremamente elevata, richiedendo al giocatore di fallire più e più volte e apprendere dai suoi errori prima di riuscire a concludere una partita. I giocatori più navigati infatti ponderano ogni più piccola

decisione, dalle stanze da visitare prima di passare al livello successivo ai potenziamenti da prendere, dato che questi potrebbero avere degli effetti secondari poco gradevoli. Da questo punto di vista si potrebbe pensare al problema dell'esplorazione del dungeon come un problema di ottimizzazione: bisogna attraversare solo alcune stanze prima di combattere il boss e passare oltre, ovvero trovare un giusto *trade-off* tra rischi e guadagni (questa meccanica è meglio nota come *high risk - high reward*), tenendo conto del fatto che alcuni eventi dipenderanno dal caso. Un bravo giocatore è infatti anche uno che sa sfruttare al meglio le risorse che il gioco genera casualmente. Sarebbe utile quindi avere un agente capace di simulare l'esplorazione del dungeon e dirci quale percorso è meglio imboccare date le informazioni di cui si dispone, ovvero le proprie statistiche, la stanza corrente e il tipo delle stanze adiacenti. Nell'ambito classico questo può essere visto come un problema di intelligenza artificiale, dove lo scopo è quello di sviluppare un agente che operi in un ambiente parzialmente osservabile e non deterministico. Si tratta quindi di un problema di ricerca che può essere affrontato in due modi: con un approccio *offline* e uno *online*.

Nell'approccio *offline* l'agente produrrà un *albero AND-OR* che rappresenta un piano di contingenza, ovvero sequenze alternative di passi tra cui scegliere a seconda degli stimoli che l'ambiente offre. Questo albero presenta due diversi tipi di nodi: nodi OR, in cui la scelta del nodo figlio dipende esclusivamente dall'agente, e nodi AND, dove invece la scelta del successore dipende dal caso, cioè dalle contingenze dell'ambiente. Ogni percorso che va dal nodo radice a un nodo foglia è una possibile sequenza di eventi a cui può andare incontro l'agente quando deve esplorare l'ambiente circostante. Una volta che tutti i nodi foglia sono stati determinati l'agente sarà sempre in grado di effettuare “la scelta giusta”. Il problema principale (comunque affrontato da varianti di questo approccio *offline*) sta nel fatto che talvolta possono crearsi cammini lungo l'albero di profondità infinita, formando di fatto dei loop. Legato a ciò vi è inoltre un secondo problema, tipico dell'intelligenza artificiale: la quantità di memoria richiesta per mantenere un albero che potenzialmente può avere profondità infinita.

L'approccio *online*, di contro, preferisce alternare computazione e azione, soprattutto quando l'ambiente è non deterministico, consentendo «*all'agente di concentrare le attività di calcolo sulle contingenze che si verificano effettivamente, anziché su quelle che potrebbero verificarsi ma probabilmente non lo faranno*» (per maggiori informazioni si veda il Capitolo 4.5 di [12]). Questo tipo di approccio può essere preferibile nell'ambito dell'esplorazione di un dungeon ignoto, poiché gli effetti di alcune azioni potrebbero essere sconosciuti all'agente fino a che tali azioni non siano state messe in atto (un giocatore non può sapere se prenderà danno o meno entrando in una specifica stanza: deve entrarci per scoprirla).

Seguendo quest'ultimo approccio abbiamo deciso di sviluppare un algoritmo di ricerca online per l'esplorazione di un dungeon generato casualmente. Tale dungeon presenta una serie di semplificazioni, rispetto a un caso reale, dovute sia al fatto che vuole essere rappresentativo dei roguelike in generale, sia al fatto che dati i limiti della tecnologia quantistica odierna non è possibile lavorare efficientemente con troppi qubit. presenteremo dunque due esempi di esplorazione, da parte di un agente, di un dungeon non deterministico e parzialmente osservabile, dove utilizzeremo l'algoritmo di Grover per scegliere lo stato successivo migliore tra quelli disponibili.

- Nel primo esempio il nostro agente dovrà esplorare un dungeon composto soltanto da quattro stanze, dove l'algoritmo di Grover opererà su un solo qubit. Pur essendo un caso di applicazione in cui l'algoritmo non è in grado di mostrare il suo vero potenziale, lo tratteremo ugualmente per prendere familiarità con le semplificazioni utilizzate anche nel caso successivo.
- Nel secondo esempio complicheremo la struttura del dungeon portando il numero di stanze da quattro a otto e aggiungendone una nuova tipologia. Introdurremo inoltre delle statistiche (salute e attacco) di cui l'agente dovrà tenere di conto al momento della scelta dell'azione successiva.

4.1 Esplorazione (quantistica) di un dungeon a quattro stanze

In questo esempio il nostro agente si troverà a doversi muovere all'interno di un dungeon composto da quattro stanze. Ad ogni stanza è assegnato un indice univoco a due bit, e ciascuna di queste è connessa, mediante dei corridoi, ad altre due stanze, in particolare quelle il cui indice è ottenuto effettuando lo swap di un solo bit dell'indice corrente (ad esempio la stanza 00 sarà connessa alle stanze 01 e 10). Possiamo quindi vedere il labirinto come un grafo non orientato e senza pesi sugli archi, poiché il movimento da una

stanza all'altra ha sempre costo costante e comunque non ci interessa. Ciascuna stanza può contenere uno di questi tre elementi al suo interno:

1. Dei nemici
2. Un tesoro
3. Il boss del dungeon

All'avvio di ogni partita ad ogni stanza viene assegnato un contenuto casuale, con alcuni vincoli:

- La stanza 00, che sarà sempre la prima stanza da esplorare nel labirinto, può contenere solamente dei nemici.
- Il resto del labirinto (ovvero le altre tre stanze) deve essere sempre composto da esattamente una stanza contenente il tesoro, una contenente il boss e una popolata da nemici. L'associazione *indice-contenuto* di queste stanze è generata casualmente all'inizio di una partita.
- L'effetto di una stanza si attiva nel momento in cui l'agente vi entra. A questo livello di semplificazione è difficile vedere gli effetti di questa regola, ma l'idea è che se ad esempio il nostro protagonista entra in una stanza piena di nemici, vi combatte, e se sopravvive può fermarsi un attimo per riposare e ponderare dove sia meglio spostarsi adesso.
- Quando l'agente esce dalla stanza del tesoro (ovvero dopo aver ottenuto il potenziamento ivi contenuto) questa viene popolata immediatamente da nemici. Il motivo per cui la riempiamo di nemici e non la lasciamo vuota è sempre legato alla nostra semplificazione, ma serve anche a disincentivare l'agente dal percorrere stanze già visitate in favore dell'ignoto.

Con questa definizione del problema lo scopo del nostro agente è molto semplice: una volta entrato nel dungeon questi deve trovare la stanza del tesoro, per poi dirigersi verso il boss. In tutto questo deve visitare meno stanze contenenti nemici possibile. Come ogni agente che si rispetti, questi avrà delle percezioni, in particolare saprà sempre il contenuto delle stanze adiacenti a lui (non ha bisogno di sapere gli indici di queste stanze in quanto, questi, sono dati implicitamente dall'indice della stanza corrente, che invece l'agente conosce). La funzione di transizione per il nostro avventuriero, che corrisponde al nostro oracolo nell'algoritmo di Grover, è quindi molto semplice, e può essere schematizzata nel seguente pseudocodice:

```

1: procedure QUANTUM_CHOICE(current_room, treasure_seen, adjacent_rooms)
2:   if adjacent_rooms.contains(treasure_room) then
3:     current_room  $\leftarrow$  treasure_room
4:     treasure_seen  $\leftarrow$  True
5:   else if not treasure_seen and adjacent_rooms = [enemies, boss] then
6:     current_room  $\leftarrow$  enemies
7:   else if treasure_seen and adjacent_rooms = [enemies, enemies] then
8:     current_room  $\leftarrow$  adjacent_rooms.chooseRandom()
9:   else if treasure_seen and adjacent_rooms.contains(boss) then
10:    current_room  $\leftarrow$  boss

```

Per implementare dunque l'esplorazione online del nostro agente nel labirinto possono bastarci 7 qubit:

- Un qubit $|\psi\rangle$ posto nello stato di ugual sovrapposizione a cui dovrà essere applicato il kickback di fase -1 quando si è deciso che direzione prendere. In particolare a seconda dello stato che verrà misurato da questo qubit si sceglierà se andare nella stanza a sinistra (ovvero quella ottenuta effettuando lo swap del primo bit di indice della stanza corrente) o a destra (swappando invece il secondo bit).
- Due qubit che descrivano il contenuto della stanza a sinistra (data un'associazione arbitraria, per esempio 00 = tesoro, 01 = nemico e 11 = boss).

- Due qubit che descrivano il contenuto della stanza a destra.
- Un qubit che indichi se si è già esplorata la stanza del tesoro.
- Un qubit, quello dell'oracolo, inizializzato nello stato $|-\rangle$ che applichi il kickback di fase al qubit $|\psi\rangle$ quando necessario.

Ci serve poi, oltre a questi qubit, un bit classico per conservare il risultato della misura del qubit $|\psi\rangle$. A questo punto possiamo eseguire l'intero algoritmo di Grover più e più volte utilizzando l'output di una fase per quella successiva, fino a quando l'agente non rileva di essere giunto al boss. L'intero codice scritto in Qiskit che implementa questo algoritmo di esplorazione è riportato nell'appendice A. Mostriamo qua solamente un'immagine del circuito ottenuto durante una delle esecuzioni dell'algoritmo di Grover. Diciamo “una delle esecuzioni” in quanto, essendo l'output di ogni circuito l'input del successivo, può essere necessario modificare alcune porte di ingresso tra una stanza esplorata e l'altra. Infatti una volta misurato il bit classico ed esplorata la stanza successiva cambieranno l'indice della stanza corrente, i contenuti delle stanze adiacenti e forse avremo trovato il tesoro. Nella successiva esecuzione del circuito bisognerà tenere di conto di questi fatti, applicando porte NOT ai qubit che li descrivono laddove necessario.

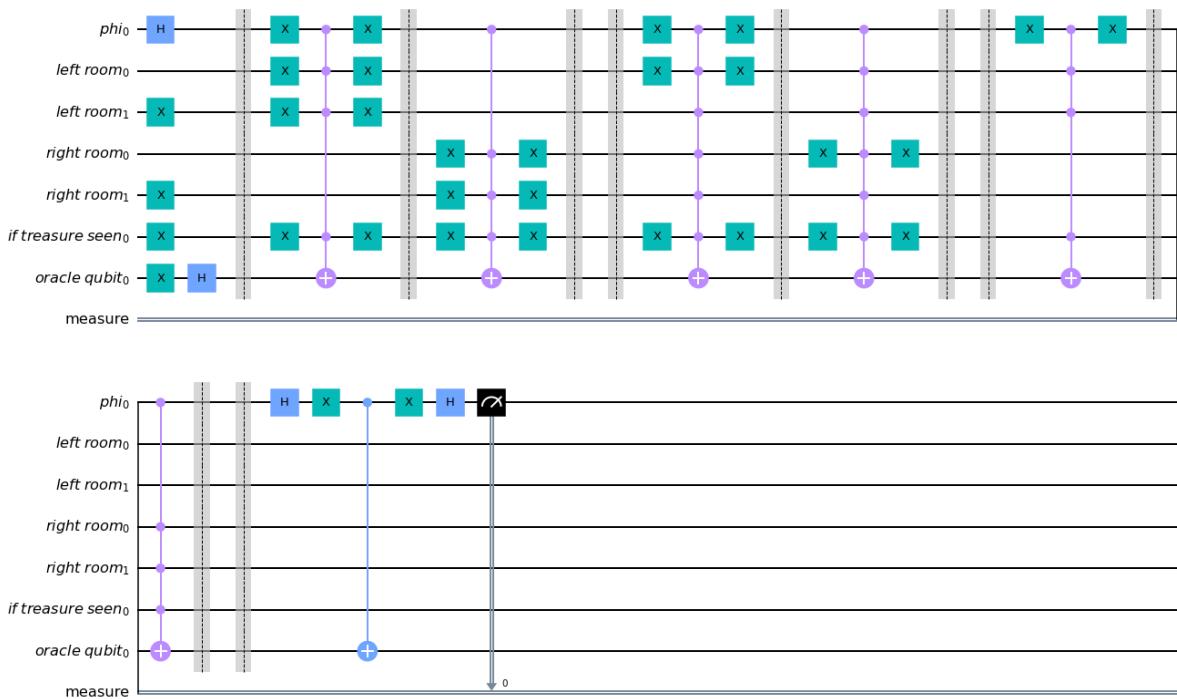


Figura 4.1: Esempio di circuito ottenuto per la scelta della stanza successiva in un labirinto a quattro stanze.

Come anticipato questo esempio ha solo uno scopo illustrativo e preparatorio a quello che segue, in quanto qui l'uso dell'algoritmo di Grover non è necessario per effettuare la scelta: anzi, è completamente inutile, dato che, lavorando noi su un solo qubit per scegliere che direzione prendere, finiamo col riflettere semplicemente il vettore $|\psi\rangle$ rispetto a $|\beta\rangle$ (applichiamo l'operatore di Grover una sola volta), non migliorando minimamente la situazione e avendo sempre la stessa probabilità di misurare 0 o 1 (ricordiamo inoltre che $|\beta\rangle$ era il vettore di ugual sovrapposizione dei successi, dove giacciono le soluzioni). L'esecuzione relativa a questo esempio riportata nell'appendice A infatti è un caso fortunato in cui l'agente ha scelto un percorso ottimale, data la mappa iniziale. Vediamo ora di complicare leggermente le cose così da renderle più interessanti, sia dal punto di vista della formulazione del problema, sia da quello dell'utilizzo di Grover.

4.2 Esplorazione di un dungeon a otto stanze

4.2.1 Implementazione classica

Per rendere le cose più interessanti, oltre all'ampliamento del dungeon, introduciamo le statistiche di salute e di attacco, che l'agente dovrà considerare al momento della scelta della stanza successiva. Anche in questo caso ogni stanza è identificata univocamente da un indice a tre bit, che implicitamente indica anche le tre stanze adiacenti raggiungibili. Le regole fondamentali del gioco sono le stesse di prima, con qualche nota aggiuntiva:

- L'agente, che comincerà sempre la sua esplorazione dalla stanza di indice 000, partirà avendo quattro punti salute, il massimo possibile. Sarà importante mantenere il valore di questa statistica strettamente sopra lo zero, sia per evitare di essere uccisi dai nemici, sia per aumentare le chance di battere il boss.
- Anche la statistica di attacco gioca un ruolo fondamentale. Tanto più è alta, tanto più diviene facile evitare di essere colpiti dai nemici, e tanto più sarà facile sconfiggere il boss. Inizialmente il nostro agente sarà molto debole e avrà solo un punto attacco, ma potrà migliorare questa statistica sia visitando il tesoro che il negozio.

Approfondiamo maggiormente queste ultime due stanze.

Il tesoro, che prima fungeva semplicemente da stanza necessaria a rendere il problema dell'esplorazione appena più complicato, qui gioca un ruolo fondamentale. Entrandoci, l'agente potrà trovare una qualunque tra queste quattro ricompense:

- Un punto salute.
- Due punti salute.
- Un punto attacco.
- Due punti attacco.

Se i punti attacco garantiscono sempre un potenziamento netto all'esploratore, dall'altro esistono situazioni in cui quelli salute risultano completamente inefficaci. Ad esempio, se l'agente entra nella stanza del tesoro con tre punti salute e trova due punti salute, potrà ottenerne solo uno.

Il negozio invece, se visitato, sottrae un punto salute all'agente, ma ne aggiunge uno di attacco. Questo scambio risulterà essere a volte vantaggioso, a volte indifferente e a volte persino dannoso: dipende da ciò che l'agente ha già visto, dal valore corrente delle sue statistiche e dalla configurazione del dungeon. Ma perché crucciarsi tanto per tenere alte queste statistiche? Per due motivi, fondamentalmente. Il primo è che, entrando in una stanza piena di nemici, si può venire danneggiati, e ciò comporta la perdita di un punto salute. Se questi scendono a zero, si muore, e l'esplorazione termina con un fallimento. Se da un lato quindi l'idea di potersi curare trovando punti salute nella stanza del tesoro sembra una prospettiva niente male, dall'altro è comunque una buona notizia (se non addirittura migliore) trovare dei punti attacco, dato che questi modificano le probabilità di essere colpiti dai nemici in una stanza che li contiene. In particolare, si ha che, entrando in una stanza contenente nemici avendo:

- un solo punto attacco, si ha il 75% di probabilità di essere danneggiati.
- due punti attacco, si ha il 50% di probabilità di essere danneggiati.
- tre punti attacco, si ha il 25% di probabilità di essere danneggiati.
- quattro punti attacco, non si verrà danneggiati.

Le statistiche di salute e attacco sono però fondamentali soprattutto per il fatto che determinano la percentuale di vittoria sul boss una volta raggiunto, seguendo la Tabella 4.1.

	attacco = 1	attacco = 2	attacco = 3	attacco = 4
salute = 1	10%	20%	30%	40%
salute = 2	15%	30%	45%	60%
salute = 3	20%	40%	60%	80%
salute = 4	25%	50%	75%	100%

Tabella 4.1: Percentuali di vittoria contro il boss sulla base dei punti salute e attacco.

È possibile definire questa percentuale in forma più compatta mediante l'equazione:

$$\text{probabilità di vittoria} = 5 \times (1 + \text{salute}) \times \text{attacco}.$$

Anche in questo caso ci sono alcuni vincoli sulla struttura del dungeon (sempre generato casualmente) che seguono logicamente da quelli già visti nel caso delle quattro stanze, con qualche aggiunta:

- La stanza 000 sarà sempre quella iniziale, e potrà contenere solo nemici.
- Il resto del labirinto si compone di esattamente:
 - Una stanza del tesoro.
 - Un negozio.
 - Una stanza del boss.
 - Quattro stanze contenenti nemici.
- Quando l'agente entra in una stanza, se ne attiva l'effetto, ovvero:
 - Entrando in una stanza popolata da nemici viene deciso in modo stocastico se l'agente ha preso danno (cioè ha perso un punto salute) o meno, sulla base dei suoi punti attacco e salute.
 - Entrando nella stanza del tesoro viene scelto casualmente un potenziamento e applicato alle statistiche, se possibile.
 - Entrando nel negozio viene automaticamente sottratto un punto salute all'agente, che in cambio riceve un punto attacco aggiuntivo.
- L'agente non può avere più di quattro punti per statistica. Per come stiamo formulando il problema del dungeon a otto stanze non c'è pericolo che si possano ottenere più di quattro punti attacco, ma lo stesso non può essere detto per la salute. Se ad esempio l'agente entrasse nella stanza del tesoro con tutti e quattro i punti salute e trovasse come potenziamento uno o due punti di questa statistica, non potrebbe ottenerli, e tale potenziamento andrebbe perso per la partita corrente.
- Nel caso della stanza del tesoro e del negozio, subito dopo che l'agente vi è entrato e i loro effetti sono stati applicati, queste si trasformano in stanze contenenti nemici. Ciò non implica che l'agente debba subito combattere con loro, in quanto abbiamo detto che l'effetto di una stanza si applica solo al momento dell'entrata. Se tuttavia, per qualche motivo, l'agente dovesse tornare in queste stanze in futuro, non solo non troverà alcun tipo di potenziamento, ma dovrà scontrarsi con dei nemici.
- L'esplorazione dell'agente può terminare prematuramente se, entrando in una stanza contenente nemici, i suoi punti salute scendono a zero.
- Quando l'agente entra nella stanza del boss viene deciso, sempre casualmente e in accordo con le probabilità riportate nella Tabella 4.1, se questi è riuscito a vincere contro il boss oppure ha perso, facendosi uccidere da esso.

L'esplorazione che il nostro protagonista deve condurre questa volta non è banale. Sicuramente dovrà visitare il tesoro, dato che nel migliore dei casi aumenterà una delle sue statistiche e nel peggiore rimarrà com'era prima. Il negozio, d'altra parte, non è sempre una tappa obbligatoria. Togliendo un punto salute e aggiungendo un punto attacco, a volte può migliorare significativamente la probabilità di vittoria, a volte può lasciarla inalterata e a volte può addirittura peggiorarla. Basta confrontare ogni cella della tabella 4.1 con quella che le sta in alto a destra (rappresentativa della probabilità di sconfiggere il boss con un punto salute in meno e uno di attacco in più) per vedere se vale la pena entrare in questa stanza. È chiaro dunque che l'avventuriero dovrà cambiare la propria strategia esplorativa sulla base delle statistiche correnti, delle stanze adiacenti e degli effetti che queste avrebbero su di esso se ci entrasse, tenendo di conto anche alcune stanze già esplorate. Ragionando dunque sulle varie situazioni in cui questi può trovarsi, abbiamo scritto un programma Python che implementa l'esplorazione dell'agente all'interno del dungeon, con una funzione di transizione che assegna un valore numerico ad ogni stanza adiacente a seconda di vari fattori. Dopo questo assegnamento, viene selezionata la stanza di valore massimo, e l'esplorazione procede in essa. L'agente conoscerà sempre i suoi punti salute, punti attacco, la stanza corrente, il contenuto delle stanze adiacenti, se ha già visitato la stanza del tesoro e se ha già visto il negozio, tutti fattori determinanti per la decisione di quale sia la prossima stanza in cui mettere piede. I valori riportati nel codice, apparentemente casuali, sono stati scelti in seguito a ragionamenti sulla struttura del problema e prove ripetute, dimostrandosi adatti ai nostri scopi su un numero di esecuzioni che supera le centinaia di migliaia. Riportiamo in pseudocodice la funzione di transizione (l'intero codice sorgente è disponibile nell'appendice):

```

1: procedure CHANCE_VICTORY(health, attack)
2:   return  $5 \times (1 + \text{health}) \times \text{attack}$ 
3: procedure CLASSICAL_CHOICE(current, treasure_explored, shop_seen, adjacents, health, attack)
4:    $S \leftarrow \text{health}$ 
5:    $A \leftarrow \text{attack}$ 
6:   if current = treasure_room then
7:     treasure_explored  $\leftarrow \text{True}$ 
8:   if current = shop_room then
9:     shop_seen  $\leftarrow \text{True}$ 
10:  for room in adjacents do
11:    if room = treasure and not treasure_explored then
12:      room.value  $\leftarrow 10$ 
13:    if room = shop and health = 1 then
14:      room.value  $\leftarrow -1$ 
15:    if room = shop and chance_victory(S - 1, A + 1) > chance_victory(S, A) then
16:      room.value  $\leftarrow 8$ 
17:    if room = shop and chance_victory(S - 1, A + 1) = chance_victory(S, A) then
18:      room.value  $\leftarrow 6$ 
19:    if room = shop and chance_victory(S - 1, A + 1) < chance_victory(S, A) then
20:      room.value  $\leftarrow 0$ 
21:    if room = boss and not treasure_explored then
22:      room.value  $\leftarrow 0$ 
23:    if room = boss and treasure_explored and shop_seen then
24:      room.value  $\leftarrow 7$ 
25:    if room = boss and treasure_explored and not shop_seen and
26:      (S = 1 or chance_victory(S - 1, A + 1) < chance_victory(S, A)) then
27:      room.value  $\leftarrow 3$ 
28:    if room = boss and treasure_explored and not shop_seen and
29:      chance_victory(S - 1, A + 1) > chance_victory(S, A) then
30:      room.value  $\leftarrow 1$ 
31:    if room = enemies then
32:      room.value  $\leftarrow 2$ 
33:    if adjacents.contains(shop) then
34:      shop_seen  $\leftarrow \text{True}$ 
35:    current  $\leftarrow \text{max\_value}(\text{adjacents})$ 

```

4.2.2 Implementazione quantistica

Per l'implementazione quantistica il ragionamento è analogo, e il codice rimane identico salvo per la parte in cui l'agente deve scegliere la stanza successiva. La funzione di transizione è infatti rappresentata da un circuito quantistico che implementa l'algoritmo di Grover per la scelta della prossima stanza. La differenza fondamentale è che, se prima si assegnava un valore numerico a ogni stanza adiacente e poi si prendeva quella che lo massimizzava, qui l'obiettivo dell'algoritmo di Grover è, date le informazioni correnti dell'agente, determinare quale dei tre bit che descrivono l'indice della stanza corrente è meglio invertire, così da ottenere l'indice della migliore stanza successiva. La scelta della prossima stanza si può riassumere nel seguente pseudocodice:

```

1: procedure QUANTUM_CHOICE(current_room,  $|\phi\rangle$ , work_qubits)
2:   new_room_chosen  $\leftarrow$  False
3:   while not new_room_chosen do
4:     circuit  $\leftarrow$  new_quantum_circuit(current_room,  $|\phi\rangle$ , work_qubits)
5:     grover_oracle(circuit)
6:     diffuser(circuit)
7:     result  $\leftarrow$  execute(circuit)
8:     direction  $\leftarrow$  measure_  $|\phi\rangle$  (result)
9:     if direction = 00 then
10:      tmp  $\leftarrow$  current_room.flip_index(0)                                 $\triangleright$  stanza a sinistra
11:     if direction = 01 then
12:       tmp  $\leftarrow$  current_room.flip_index(1)                                 $\triangleright$  stanza al centro
13:     if direction = 10 then
14:       tmp  $\leftarrow$  current_room.flip_index(2)                                 $\triangleright$  stanza a destra
15:     if direction = 00 or direction = 01 or direction = 10 then
16:       new_room_chosen  $\leftarrow$  True
17:       current_room  $\leftarrow$  tmp

```

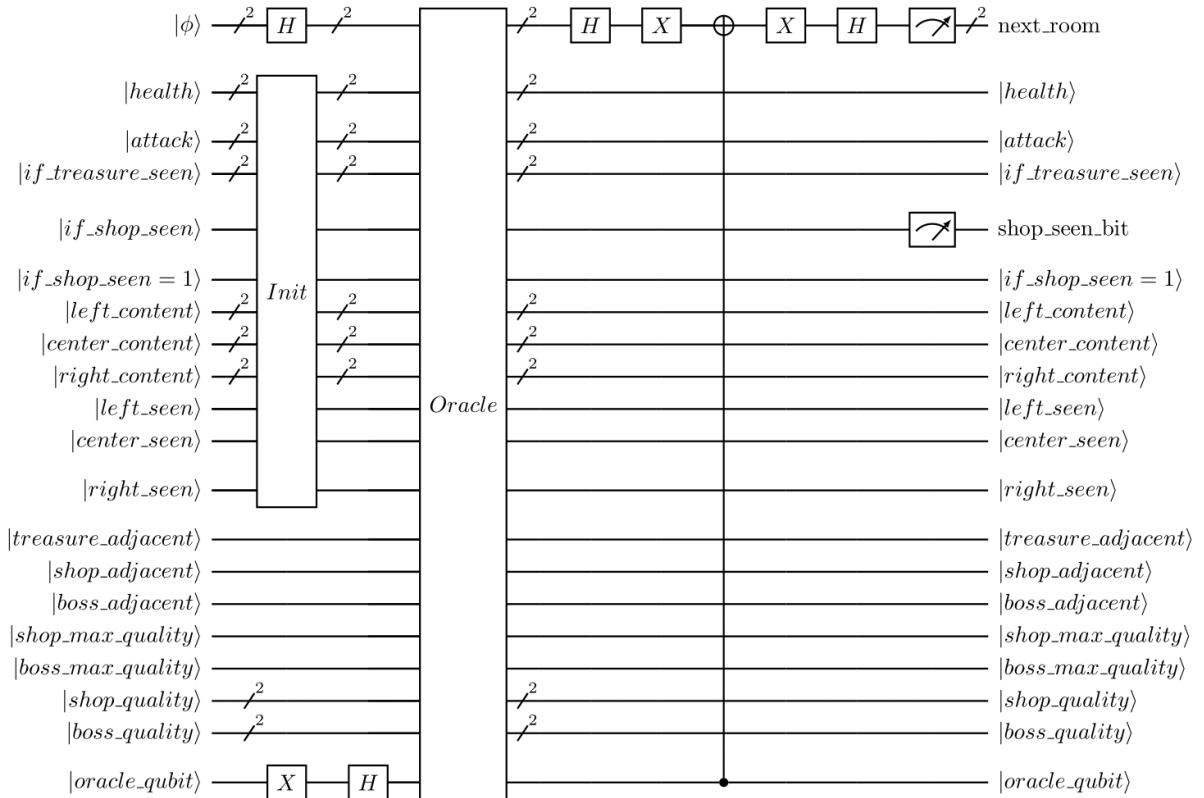


Figura 4.2: Esempio di circuito ottenuto per la scelta della stanza successiva in un labirinto a otto stanze

Per come abbiamo costruito il circuito, lo spazio di ricerca (ovvero le tre stanze adiacenti) conterrà sempre al più una sola soluzione. Dato che però stiamo usando due qubit ($|\phi\rangle$) per la ricerca, può accadere, raramente, che venga misurato 11, e in tal caso si chiede all'agente di ripetere la scelta. Potrebbe di fatto non esserci una stanza preferita, e in tal caso la misura dei due qubit nello stato $|\phi\rangle$ restituirà un valore casuale, che corrisponde al comportamento che ci aspettiamo dall'agente in caso di mancanza di una preferenza. Proprio perché lo spazio di ricerca conterrà sempre al massimo una soluzione, il numero di rotazioni da eseguire, e quindi di applicazioni dell'operatore di Grover, è al più uno, come già visto nel

Capitolo 2. Potevamo anche aggiungere un terzo qubit al registro $|\phi\rangle$ così da applicare sempre l'operatore di Grover il numero di volte necessario, sapendo a priori che cinque delle otto possibili soluzioni sarebbero state sbagliate. Tuttavia, dato che il circuito costruito si compone già di ventotto qubit, non abbiamo ritenuto saggio introdurne un altro, considerando che ciò avrebbe potuto anche portare all'aggiunta di ulteriori qubit di lavoro. Infatti, a seconda della stanza corrente, di quelle adiacenti e dello stato dell'agente, il numero di soluzioni può variare, e ciò implica maggiore logica per il circuito. Abbiamo preferito costruirne uno *ad hoc* che sfruttasse a pieno i qubit disponibili, riciclandoli quando necessario. A scanso di equivoci, specifichiamo quindi che il circuito così costruito può essere personalizzato nella scelta dei contenuti delle varie stanze, nel **rispetto dei vincoli del problema citati sopra**: in particolare, la prima stanza (di indice 000) dovrà sempre contenere nemici ed essere il punto di ingresso per il dungeon, ci dovrà essere sempre esattamente un tesoro, un negozio, un boss, e altre quattro stanze contenenti nemici.

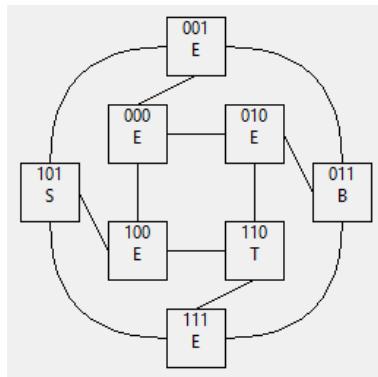


Figura 4.3: Esempio di dungeon a otto stanze che rispetta i vincoli del problema. Ogni lettera rappresenta il maniera sintetica il contenuto della stanza, dove S = Shop, E = Enemies, T = Treasure e B = Boss.

Come però si decide di mappare questo insieme di stanze agli otto indici di stanza, è a nostra discrezione, e il circuito quantistico (così come il codice classico) fornirà sempre il percorso che ha la maggiore probabilità di far vincere l'agente. Non entreremo ulteriormente nei dettagli implementativi del circuito, limitandoci a confrontarlo con la sua versione classica e fare un'analisi costi-benefici di queste due implementazioni. Anche in questo caso, per il lettore interessato, il codice sorgente del circuito scritto in Qiskit è disponibile nell'appendice C, adeguatamente commentato così da permettere a chiunque di seguire passo passo il ragionamento dietro la sua costruzione. Ci teniamo inoltre a specificare che l'algoritmo quantistico permette solo di velocizzare il passo di scelta, e non influisce in alcun modo sulla probabilità di vittoria dell'agente. A riprova di ciò riportiamo l'esito dell'esplorazione del medesimo dungeon, effettuata sia col codice classico che con quello quantistico, nella Figura 4.4.

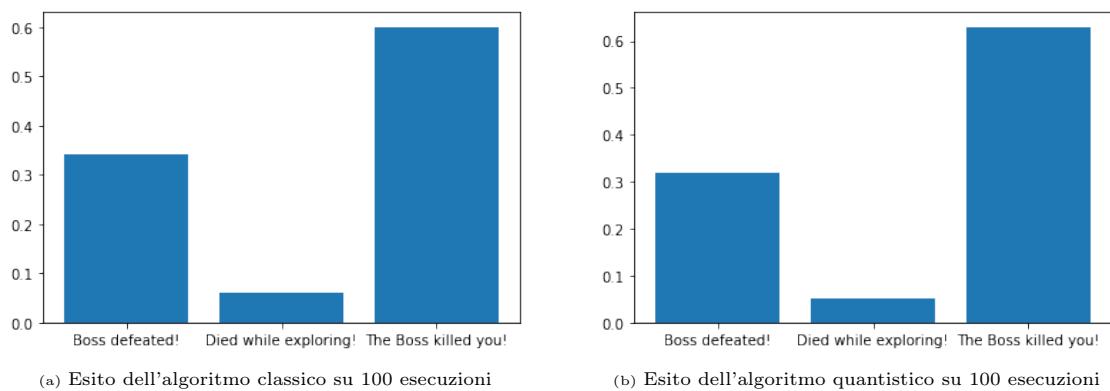


Figura 4.4: Confronto delle esecuzioni classiche con quelle quantistiche. L'algoritmo di esplorazione classico e quello quantistico sono stati eseguiti partendo dallo stesso dungeon (in particolare quello della Figura 4.3), e per entrambi sono state effettuate 100 esecuzioni (esplorazioni). I risultati sono molto simili, come ci aspettavamo, in quanto la strategia di esplorazione è sempre la stessa.

La complessità dell'algoritmo di esplorazione risiede nella scelta della prossima stanza in cui mettere piede e nel numero di volte che questa operazione viene effettuata. Per poter decidere infatti lo stato successivo l'algoritmo classico deve prima invocare la funzione `assign_quality()` su ogni stanza adiacente, che assegna a ciascuna di queste un valore qualitativo dipendente da fattori come la salute, l'attacco, le stanze precedentemente esplorate e via discorrendo. Dopo di che va trovato il massimo tra questi valori, così da ottenere la stanza vincitrice di questo processo di scelta. Indipendentemente dalla struttura dati utilizzata per conservare i valori delle stanze, nel caso peggiore la ricerca del massimo avrà complessità $\mathcal{O}(N)$, se immaginiamo che ogni stanza ne abbia N adiacenti. Con un circuito quantistico che implementa la funzione di transizione, tuttavia, la complessità scende a $\mathcal{O}(\sqrt{N})$ (a patto che l'operatore di Grover sia stato scritto correttamente). Su esempi piccoli come quelli esaminati fino ad ora il vantaggio quantistico non solo non lo si vede, ma non lo si ha nemmeno, data la pochezza di computer quantistici capaci di gestire un numero così elevato di qubit e del rumore quantistico. Infatti tutte le simulazioni su Qiskit del labirinto ad otto stanze sono state eseguite su un computer classico che ne simula uno quantistico. Tuttavia, augurandoci che negli anni a venire questa tecnologia riesca a permetterci di gestire il rumore, costruire computer quantistici a molti qubit, fare arrivare questi nelle nostre case e, chissà, permetterci di avere un linguaggio di programmazione di più alto livello per costruire programmi quantistici, il miglioramento della performance si paleserebbe nel caso di labirinti molto grandi.

4.2.3 Possibili generalizzazioni

Fino ad ora abbiamo presentato programmi quantistici atti all'esplorazione di labirinti a quattro e otto stanze. Ma se volessimo gestire dimensioni arbitrarie per il dungeon, saremmo capaci di farlo?

Ebbene, quando abbiamo raddoppiato la dimensione del labirinto, abbiamo anche modificato la formulazione del problema, introducendo nuove statistiche, stanze e variabili, e ciò ha comportato una completa rivisitazione del circuito. È chiaro che, se volessimo procedere per questa strada introducendo cose come punti difesa, mini-boss e caratteristiche simili, dovremmo ancora una volta mettere mano ai qubit e alle porte. D'altro canto, semplici modifiche alla dimensione del dungeon non richiederebbero cambiamenti troppo drastici. Fintanto che il nostro labirinto continua ad avere N stanze (dove N è una potenza di 2), la struttura ad ipercubo (ovvero ogni stanza è collegata ad esattamente altre $n = \log_2(N)$ stanze) e si hanno precisamente una stanza del boss, una del tesoro, un negozio, tutte le altre ricolme di nemici, un numero massimo di punti salute e attacco pari a quattro e la stanza iniziale di indice $|0_1 0_2 \dots 0_n\rangle$, basta aggiungere:

- $\mathcal{O}(n)$ qubit al registro $|\phi\rangle$.
- $\mathcal{O}(n)$ qubit che dicano se si è esplorata una certa stanza adiacente.
- $\mathcal{O}(n)$ coppie di qubit che indichino il contenuto di ogni stanza adiacente.

Andrebbe poi collegato il tutto aggiungendo qubit di controllo e target alle porte multi-qubit. L'ultima modifica necessaria sarebbe cambiare il numero di applicazioni dell'operatore di Grover ogni volta che bisogna scegliere la stanza successiva, seguendo il limite superiore riportato nel Capitolo 2 e di ordine $\mathcal{O}(\sqrt{N})$.

Capitolo 5

Sviluppi futuri

In questa tesi ci siamo occupati delle potenzialità del calcolo quantistico rapportato all'ambito dei videogiochi, concentrando i nostri sforzi su una comprensione teorica e pratica dell'algoritmo di Grover. In particolare, nel Capitolo 1 abbiamo esaminato i principi del Quantum Computing, studiando le proprietà matematiche dei qubit, presentando alcune delle porte più comuni e esaminando concetti fondamentali come l'entanglement e l'uncomputation. Nel Capitolo 2 abbiamo analizzato in profondità l'algoritmo di Grover, capace di darci uno speedup quadratico nella ricerca all'interno di insiemi non strutturati. Nel Capitolo 3 abbiamo analizzato il panorama odierno dei videogiochi quantistici, sia che questi avessero lo scopo di introdurre i concetti della fisica quantistica ai giocatori sia che implementassero nelle loro meccaniche di giochi dei veri circuiti quantistici. Nel Capitolo 4 abbiamo presentato un nostro circuito quantistico che utilizza l'algoritmo di Grover e capace di esplorare efficientemente un labirinto a otto stanze.

Nella costruzione di questo circuito abbiamo imposto alcuni vincoli sulla struttura del dungeon, come la necessaria presenza di un solo boss, un solo tesoro, un solo negozio, un numero fisso di stanze adiacenti e caratteristiche specifiche quali la salute e l'attacco. Un'alternativa a quest'approccio potrebbe consistere nel definire, in ordine:

1. Una serie di caratteristiche, per l'agente, a nostra discrezione.
2. Una formula che, date queste caratteristiche, esprime la qualità di una stanza.
3. Una funzione che, dato il nome di un nuovo tipo di stanza e l'effetto che questa ha sull'esploratore, produce una porta personalizzata, costruita a partire da quelle già esistenti, da inserire nel circuito.

Tuttavia dovremo ancora attendere probabilmente qualche anno prima di avere un computer quantistico a centinaia o migliaia di qubit capace di gestire labirinti con un numero arbitrariamente alto di stanze. Un'altra strada percorribile consisterebbe nel semplificare ulteriormente la formulazione del problema supponendo un dungeon deterministico. In tal caso l'agente non avrebbe più a che fare con stanze dall'esito incerto e potrebbe limitarsi a cercare il percorso minimo che va dal punto di partenza al boss passando per le stanze che ritiene interessanti, come quella del tesoro e il negozio nel nostro esempio. Questo può avere senso, in un caso d'uso realistico, quando il gioco simulato non prevede nemici o simili, oppure quando il giocatore che sta usando il programma si ritiene esperto a sufficienza da poter correre dei rischi, assumendo quindi di non essere mai colpito durante il tragitto.

Ringraziamenti

Mi sembra ancora incredibile di essere già arrivato alla stesura della tesi. Questi tre anni a Pisa sono durati un battito di ciglia, e ciò lo devo a molti fattori diversi. Il primo è stato l'essere riuscito a trovare fin da subito un ambito, quello dell'Informatica, che mi appassionasse davvero. Il secondo è stato l'ambiente inclusivo e professionale che fin da subito mi si è presentato davanti. E il terzo sono le preziose persone che mi hanno accompagnato durante questo meraviglioso percorso di studi. Ritengo quindi doveroso ringraziarle, perché senza di loro oggi non sarei qui, o almeno, non con questo sorriso e gioia nel cuore.

Ringrazio innanzitutto i miei relatori, ovvero le Professoressa A. Bernasconi e G. Del Corso e il Dottor A. Berti. Grazie del costante feedback sulla tesi e di aver sempre avuto tempo e pazienza di rispondere ai miei dubbi, anche durante i corsi che ci hanno avvicinato e mi hanno fatto capire il vostro valore.

Un necessario ringraziamento va poi a tutta la mia famiglia, che mi ha supportato dal primo istante e fatto le feste dopo ogni esame. Mamma, Babbo, grazie di avermi permesso di studiare a Pisa. Mi avete fatto un regalo più grande di quanto possiate pensare.

Ci tengo poi a ringraziare tutti gli amici che mi sono fatto durante questo percorso, dato che mi hanno permesso di vivere meglio quest'università. Ringrazio Carlo per avermi dato fiducia quando ancora non l'avevo sviluppata, e per le serate di DnD in compagnia che ricordo con affetto. Ringrazio Enrico per aver sempre sopportato e chiarito i miei dubbi matematici, nonché per avermi dato una spinta quando ne avevo bisogno. Ringrazio Anna e Simone per aver condiviso con me risate e sessioni di studio che nemmeno la distanza ha potuto interrompere. Ringrazio Andrea e Giovanni per le innumerevoli battute scambiate con me e per l'aiuto reciproco che ci siamo dati nel corso di questi anni. E ringrazio di cuore Samuele e Filippo per avermi accettato nel loro gruppo fin dal primo istante, per aver condiviso con me gioie, dolori, ansie e risate, e per avermi accompagnato nel mio primo e indimenticabile viaggio quantistico compiuto tra gli affascinanti astri dell'universo di Outer Wilds.

Un sentito ringraziamento va anche a Paolo, Mattia e Marco, amici che, nonostante la distanza dovuta a scelte diverse circa il percorso di studi da intraprendere, sono sempre rimasti al mio fianco.

Non posso poi esimermi dal ringraziare il Professor C. R. Grisanti, che ha cambiato completamente la visione che avevo della matematica, insegnandomi in modo stimolante e appassionante, in soli tre mesi, quello che non ero riuscito a capire in un intero anno di liceo, e il Professor V. Gervasi, non solo per la sua professionalità e simpatia, ma anche per avermi mostrato come dovrebbe essere un Informatico a trecentosessanta gradi. Ringrazio inoltre i Professori G. Ghelli, G. Prencipe, P. Degano, M. Passacantando, A. Corradini, M. Danelutto, A. Micheli, il Dottor A. Michienzi e le Professoressa F. Paganelli, L. Ricci, R. Gori e M. Simi. Grazie di cuore per la vostra serietà e professionalità, mi ritengo fortunato ad avervi avuto come insegnanti.

Grazie Pisa, grazie Unipi. Grazie di avermi dato modo di esprimere il mio valore.

Alessio La Greca

Appendice A

Codice quantistico per l'esplorazione di un labirinto a quattro stanze

```
1 import math
2 from random import *
3 from qiskit import *
4 from qiskit.tools.visualization import plot_histogram
5 import random
6
7 #Cominciamo col creare un dizionario, che mappa valori binari delle stanze
8 #ai loro contenuti.
9 #OCCHIO: questo dizionario e' quello che cambiera' ad ogni partita.
10 #In generale dovrei creare una funzione che genera una lista che
11 #contiene esattamente un boss, una treasure e due enemies, randomizzarla
12 #e associare ogni elemento ad ogni coppia di bit. Ma per ora faccio
13 #un caso semplice. Se dovessi disegnare la mappa su un foglio di carta
14 #sarebbe una cosa tipo:
15 #alto a sinistra: 00
16 #alto a destra: 01
17 #basso a sinistra: 10
18 #basso a destra: 11
19 labyrinth_map = {"00": "enemies", "01": "boss", "10": "treasure", "11": "enemies"}
20
21 #ora definisco un altro dizionario che mappa tipo di stanza a valore
22 #numerico a due bit. Mi serve per
23 #poi sapere nel circuito cosa c'e' a destra e cosa c'e' a sinistra
24 room_types_dict = {"treasure": "00", "enemies": "01", "boss": "11"}
25
26 #voglio pero' anche sapere dove mi trovo
27 current_room = "00";
28 #ho bisogno di un bit per sapere se ho visto il tesoro o no
29 treasure_seen_bit = "0";
30
31 #preparo i registri di cui ho bisogno per il circuito (in particolare per l'oracolo).
32 #ho bisogno del bit che andra' nello stato di ugual sovrapposizione,
33 #tanto per cominciare.
34 #IMPORTANTE: dopo la misura, 0 = cambio il primo qubit, 1 = cambio il secondo qubit
35 phi = QuantumRegister(1, "phi")
36
37 #poi ho bisogno di due qubit che mi dicono qual e' il contenuto della stanza a sinistra
38 left_room = QuantumRegister(2, "left room")
39
40 #stessa cosa per la stanza a destra
41 right_room = QuantumRegister(2, "right room")
42
43 #ora ho bisogno di un qubit che mi dica se ho gia' visto la stanza del tesoro o no
44 treasure_seen = QuantumRegister(1, "if treasure seen")
45
46 #poi ho chiaramente bisogno del qubit dell'oracolo, quello che deve applicare
```

```

47 #un kickback di fase pari a -1 quando passa la soluzione
48 oracle_q = QuantumRegister(1, "oracle qubit")
49
50 #Infine mi serve un bit classico per fare la misura
51 bit = ClassicalRegister(1, "measure")
52
53 #costruisco un circuito quantistico partendo da questi qubit
54 starting_qc = QuantumCircuit(phi, left_room, right_room, treasure_seen, oracle_q, bit)
55
56 starting_qc.draw('mpl')
57
58 #funzione per inizializzare il qubit dell'oracolo nello stato |->
59 def initialize_oracle_q qc:
60     qc.x(oracle_q)
61     qc.h(oracle_q)
62
63 #definisco una funzione che dato il bit (qubit) 0 mi restituisce 1 e viceversa
64 def opposite(bit):
65     res = "-1"
66     if bit == "0": res = "1"
67     if bit == "1": res = "0"
68     return res
69
70 #definisco una funzione che, data la stanza corrente del giocatore
71 #e la mappa del labirinto, inizializza i qubit che descrivono
72 #cosa c'e' a sinistra e cosa a destra
73 def initialize_left_right(qc, current_room, labyrinth_map, room_types_dict):
74
75     print("Current room = " + current_room)
76
77     #prima prendo gli indici delle stanze adiacenti a quella in cui
78     #il giocatore si trova
79     left_room_index = opposite(current_room[0]) + current_room[1]
80     print("Left room index = " + left_room_index)
81     right_room_index = current_room[0] + opposite(current_room[1])
82     print("Right room index = " + right_room_index)
83
84     #poi devo vedere cosa contengono queste stanze
85     left_room_content = labyrinth_map.get(left_room_index)
86     right_room_content = labyrinth_map.get(right_room_index)
87     print("Left room content = " + left_room_content)
88     print("Right room content = " + right_room_content)
89
90     #ora posso decodificare il contenuto delle stanze in forma di due bit (qubit)
91     left_room_content_bits = room_types_dict.get(left_room_content)
92     right_room_content_bits = room_types_dict.get(right_room_content)
93     print("Left room content (bits) = " + left_room_content_bits)
94     print("Right room content (bits) = " + right_room_content_bits)
95
96     #ora posso usare i valori di questi bit per inizializzare i qubit
97     #della stanza a sinistra e quella a destra
98     if left_room_content_bits[0] == "1":
99         qc.x(left_room[0])
100    if left_room_content_bits[1] == "1":
101        qc.x(left_room[1])
102    if right_room_content_bits[0] == "1":
103        qc.x(right_room[0])
104    if right_room_content_bits[1] == "1":
105        qc.x(right_room[1])
106
107 #funzione per inizializzare il qubit treasure_seen
108 def initialize_treasure_seen():
109     if treasure_seen_bit == "1":
110         qc.x(treasure_seen)
111
112 #definisco il diffuser (quello che serve a effettuare
113 #una riflessione rispetto a |phi>

```

```

114
115 def diffuser(qc):
116     qc.h(phi)
117     qc.x(phi)
118     qc.cx(phi, oracle_q)
119     qc.x(phi)
120     qc.h(phi)
121
122 #definisco una funzione che iteri nel dizionario e mi restituisca la chiave di valore
123 #piu' alto (serve per capire dove dovrei andare secondo l'algoritmo di Grover)
124 def choice_quantum_movement():
125     choice = ""
126     max = -1
127     for k in counts.keys():
128         if counts[k] > max:
129             max = counts[k]
130             choice = k
131     return choice
132
133 #funzione che modifica il valore della stanza corrente data la stanza corrente
134 #e il movimento. Inoltre, se mi sono appena mosso nella stanza del tesoro
135 #e non la avevo ancora esplorata, faccio due cose:
136 #1) mi segno che l'ho esplorata
137 #2) la sovrascrivo con una stanza con nemici
138 def update_current_room(direction):
139     global current_room
140     global treasure_seen_bit
141     global labyrinth_map
142     if direction == "0":
143         current_room = opposite(current_room[0]) + current_room[1]
144     if direction == "1":
145         current_room = current_room[0] + opposite(current_room[1])
146
147     if treasure_seen_bit == "0" and labyrinth_map[current_room] == "treasure":
148         treasure_seen_bit = "1"
149         labyrinth_map[current_room] = "enemies"
150
151
152
153 #variabili aggiuntive che uso per tenere traccia della situazione
154 qc_list = []
155 counts_list = []
156
157
158
159 #Da qui comincia la ciccia
160
161 #come prima cosa ho bisogno di sapere in che stanza si trova il boss.
162 #Infatti, se la conosco, so anche quando fermarmi, ovvero
163 #quando il personaggio e' entrato nella stanza del boss.
164 boss_index = ""
165 for key_elem in labyrinth_map.keys():
166     if labyrinth_map[key_elem] == "boss":
167         boss_index = key_elem
168
169 #qui inizia il gioco: fintanto che il personaggio non e' giunto
170 #nella stanza del boss, continua a esplorare
171 while (current_room != boss_index):
172
173     #bisogna innanzitutto costruire il circuito
174     qc = QuantumCircuit(phi, left_room, right_room, treasure_seen, oracle_q, bit)
175
176     #adesso dobbiamo inizializzare il circuito con le informazioni
177     #che abbiamo a disposizione
178
179     #come prima cosa, metto nello stato di ugual sovrapposizione phi
180     qc.h(phi)

```

```

181 #poi devo inizializzare il qubit dell'oracolo, oracle_q, nello stato |->
182 initialize_oracle_q(qc)
183
184
185 #poi devo inizializzare i qubit che descrivono il contenuto
186 #della stanza sinistra e della stanza destra. I concetti di "sinistra" e "destra"
187 #sono difficili da formalizzare, pertanto assumiamo cio' che segue:
188 #se il giocatore si trova nella stanza xx, la stanza a sinistra corrispondera'
189 #alla stanza yx, dove y e' l'opposto di x. Se il giocatore si trova
190 #nella stanza xx, la stanza a destra corrispondera' alla stanza xy,
191 #dove y e' l'opposto di x. La nozione di opposto e' da intendersi come
192 #opposto(0)=1 e opposto(1)=0 (cvd).
193 #Ergo, per sapere cosa c'e' a destra e a sinistra del giocatore
194 #mi servono solo due cose:
195 #1) la stanza in cui si trova il giocatore adesso
196 #2) la mappa del labirinto
197 #occhio che la mappa del gioco e' conosciuta dal GIOCO, non dal GIOCATORE.
198 #Questo sa solo le stanze che gli stanno adiacenti
199
200 initialize_left_right(qc, current_room, labyrinth_map, room_types_dict)
201
202 #dobbiamo ora inizializzare il qubit che dice se il giocatore ha esplorato
203 #la treasure room. Il bit relativo a questa informazione, che e' poi cio'
204 #su cui ci basiamo per l'inizializzazione di questo qubit, viene aggiornato
205 #dopo che il giocatore si e' spostato sulla mappa. Di fatto questo metodo applica
206 #semplicemente una porta x a questo qubit se la stanza del tesoro e' gia'
207 #stata vista
208 initialize_treasure_seen()
209
210 qc.barrier()
211
212
213
214 #perfetto, adesso la logica vera e propria del labirinto. Lo scopo del
215 #nostro giocatore e' passare prima per la treasure room e poi andare
216 #subito dal boss.
217
218 #se non ho visto il tesoro e ce l'ho di fianco, ci vado
219 qc.x(phi)
220 qc.x(left_room)
221 qc.x(treasure_seen)
222 qc.mcx([phi[0], left_room[0], left_room[1], treasure_seen[0]], oracle_q[0])
223 qc.x(treasure_seen)
224 qc.x(left_room)
225 qc.x(phi)
226
227 qc.barrier()
228
229 qc.x(right_room)
230 qc.x(treasure_seen)
231 qc.mcx([phi[0], right_room[0], right_room[1], treasure_seen[0]], oracle_q)
232 qc.x(treasure_seen)
233 qc.x(right_room)
234
235 qc.barrier()
236 qc.barrier()
237
238 #se non ho visto il tesoro e non ce l'ho di fianco, sconfiggo i nemici
239
240 qc.x(phi)
241 qc.x(left_room[0])
242 qc.x(treasure_seen)
243 qc.mcx([phi[0], left_room[0], left_room[1], right_room[0], right_room[1],
244             treasure_seen[0]], oracle_q)
245 qc.x(treasure_seen)
246 qc.x(left_room[0])
247 qc.x(phi)

```

```

248 qc.barrier()
249
250
251 qc.x(right_room[0])
252 qc.x(treasure_seen)
253 qc.mcx([phi[0], left_room[0], left_room[1], right_room[0], right_room[1],
254         treasure_seen[0]], oracle_q)
255 qc.x(treasure_seen)
256 qc.x(right_room[0])
257
258 qc.barrier()
259 qc.barrier()
260
261 #se ho visto il tesoro e ho il boss di fianco, vado dal boss
262 qc.x(phi)
263 qc.mcx([phi[0], left_room[0], left_room[1], treasure_seen[0]], oracle_q)
264 qc.x(phi)
265
266 qc.barrier()
267
268 qc.mcx([phi[0], right_room[0], right_room[1], treasure_seen[0]], oracle_q)
269
270 qc.barrier()
271 qc.barrier()
272
273 #come ultima cosa effettuo la rotazione rispetto a |phi>
274 diffuser(qc)
275
276 #poi associo il primo qubit al bit classico per la misurazione
277 qc.measure(phi, bit)
278
279
280 #aggiungo il circuito all'array di circuiti
281 qc_list.append(qc)
282
283
284
285 #ora che ho costruito il circuito, posso misurare. Dovrei ripete questa operazione
286 #per un tot di volte, ma quante esattamente? sqrt(2), che quindi mi porta
287 #a una sola esecuzione. Dunque passo alla simulazione su computer classico
288
289 simulator = Aer.get_backend('qasm_simulator')
290 result = execute(qc, backend = simulator).result()
291 counts = result.get_counts(qc)
292
293
294 #aggiungo i counts di questa esecuzione all'array di counts
295 counts_list.append(counts)
296
297
298 #prendo il movimento che il personaggio ha deciso di eseguire
299 movement_chosen = choice_quantum_movement()
300
301 #in base a dove ha deciso di muoversi (0 = cambio il primo qubit,
302 #1 = cambio il secondo qubit), modiflico il valore della stanza corrente
303
304 print("REMEMBER: Current room = " + current_room)
305 update_current_room(movement_chosen)
306 print("Direction chosen = " + movement_chosen)
307 print("New current room = " + current_room)
308 print("New map configuration = " + str(labyrinth_map))
309
310 print("Boss reached")
311
312
313 qc_list[0].draw('mpl')
314

```

```

315 plot_histogram(counts_list[0])
316
317 qc_list[1].draw('mpl')
318
319 plot_histogram(counts_list[1])
320
321 qc_list[2].draw('mpl')
322
323 plot_histogram(counts_list[2])

```

Riportiamo qui di seguito l'output di un'esecuzione del circuito:

```

Current room = 00
Left room index = 10
Right room index = 01
Left room content = treasure
Right room content = boss
Left room content (bits) = 00
Right room content (bits) = 11
'0': 516, '1': 508
REMEMBER: Current room = 00
Direction chosen = 0
New current room = 10
New map configuration = '00': 'enemies', '01': 'boss', '10': 'enemies', '11': 'enemies'
Current room = 10
Left room index = 00
Right room index = 11
Left room content = enemies
Right room content = enemies
Left room content (bits) = 01
Right room content (bits) = 01
'0': 521, '1': 503
REMEMBER: Current room = 10
Direction chosen = 0
New current room = 00
New map configuration = '00': 'enemies', '01': 'boss', '10': 'enemies', '11': 'enemies'
Current room = 00
Left room index = 10
Right room index = 01
Left room content = enemies
Right room content = boss
Left room content (bits) = 01
Right room content (bits) = 11
'0': 510, '1': 514
REMEMBER: Current room = 00
Direction chosen = 1
New current room = 01
New map configuration = '00': 'enemies', '01': 'boss', '10': 'enemies', '11': 'enemies'
Boss reached

```

Appendice B

Codice classico per l'esplorazione di un labirinto a otto stanze

```
1 #-----STRUMENTI PER IL DISEGNO-----
2 import threading
3 from tkinter import Tk, Canvas, Frame, BOTH
4 from typing import Annotated
5
6 class Example(Frame):
7
8     def __init__(self, dungeon):
9         self.dungeon = dungeon
10        super().__init__()
11
12        self.initUI()
13
14    def my_create_rectangle(self, canvas, x, y):
15        xlen = 40
16        ylen = 40
17        canvas.create_rectangle(x, y, x+xlen, y+ylen)
18
19    def my_create_text(self, canvas, x, y, t):
20        xoffset = 20
21        yoffset = 7
22        canvas.create_text(x+xoffset, y+yoffset, text=t)
23
24    def get_letter(self, index):
25        c = self.dungeon[index].content
26        if c == RoomContent.ENEMIES:
27            return "E"
28        if c == RoomContent.TREASURE:
29            return "T"
30        if c == RoomContent.SHOP:
31            return "S"
32        if c == RoomContent.BOSS:
33            return "B"
34
35
36
37    def initUI(self):
38
39        self.master.title("Lines")
40        self.pack(fill=BOTH, expand=1)
41
42        canvas = Canvas(self)
43
44        line_len = 40
45        rectangle_dim = 40
```

```

47     x000 = 100
48     y000 = 100
49
50     x010 = x000 + rectangle_dim + line_len
51     y010 = y000
52
53     x100 = x000
54     y100 = y000 + rectangle_dim + line_len
55
56     x110 = x000 + rectangle_dim + line_len
57     y110 = y000 + rectangle_dim + line_len
58
59     x001 = x000 + rectangle_dim
60     y001 = y000 - rectangle_dim - line_len/2
61
62     x011 = x010 + rectangle_dim + line_len/2
63     y011 = y010 + rectangle_dim
64
65     x111 = x100 + rectangle_dim
66     y111 = y100 + rectangle_dim + line_len/2
67
68     x101 = x100 - rectangle_dim - line_len/2
69     y101 = y100 - rectangle_dim
70
71     self.my_create_rectangle(canvas, x000, y000)
72     self.my_create_rectangle(canvas, x010, y010)
73     self.my_create_rectangle(canvas, x100, y100)
74     self.my_create_rectangle(canvas, x110, y110)
75     self.my_create_rectangle(canvas, x001, y001)
76     self.my_create_rectangle(canvas, x011, y011)
77     self.my_create_rectangle(canvas, x111, y111)
78     self.my_create_rectangle(canvas, x101, y101)
79
80     self.my_create_text(canvas, x000, y000, "000")
81     self.my_create_text(canvas, x010, y010, "010")
82     self.my_create_text(canvas, x100, y100, "100")
83     self.my_create_text(canvas, x110, y110, "110")
84     self.my_create_text(canvas, x001, y001, "001")
85     self.my_create_text(canvas, x011, y011, "011")
86     self.my_create_text(canvas, x111, y111, "111")
87     self.my_create_text(canvas, x101, y101, "101")
88
89     self.my_create_text(canvas, x000, y000 + 15, self.get_letter("000"))
90     self.my_create_text(canvas, x010, y010 + 15, self.get_letter("010"))
91     self.my_create_text(canvas, x100, y100 + 15, self.get_letter("100"))
92     self.my_create_text(canvas, x110, y110 + 15, self.get_letter("110"))
93     self.my_create_text(canvas, x001, y001 + 15, self.get_letter("001"))
94     self.my_create_text(canvas, x011, y011 + 15, self.get_letter("011"))
95     self.my_create_text(canvas, x111, y111 + 15, self.get_letter("111"))
96     self.my_create_text(canvas, x101, y101 + 15, self.get_letter("101"))
97
98
99     canvas.create_line(x000 + rectangle_dim, y000 + rectangle_dim/2,
100     x000 + rectangle_dim + line_len, y000 + rectangle_dim/2)
101     canvas.create_line(x100 + rectangle_dim, y100 + rectangle_dim/2,
102     x100 + rectangle_dim + line_len, y100 + rectangle_dim/2)
103     canvas.create_line(x000 + rectangle_dim/2, y000 + rectangle_dim,
104     x000 + rectangle_dim/2, y000 + + rectangle_dim + line_len)
105     canvas.create_line(x010 + rectangle_dim/2, y010 + rectangle_dim,
106     x010 + rectangle_dim/2, y010 + + rectangle_dim + line_len)
107
108     canvas.create_line(x101 + rectangle_dim/2, y101,
109     x101 + rectangle_dim/2, y001 + rectangle_dim/2,
110     x001, y001 + rectangle_dim/2, smooth = 1)
111
112     canvas.create_line(x001 + rectangle_dim, y001 + rectangle_dim/2,
113     x011 + rectangle_dim/2, y001 + rectangle_dim/2,

```

```

114     x011 + rectangle_dim/2, y011, smooth = 1)
115
116     canvas.create_line(x011 + rectangle_dim/2, y011 + rectangle_dim,
117     x011 + rectangle_dim/2, y111 + rectangle_dim/2,
118     x111 + rectangle_dim, y111 + rectangle_dim/2, smooth = 1)
119
120     canvas.create_line(x111, y111 + rectangle_dim/2,
121     x101 + rectangle_dim/2, y111 + rectangle_dim/2,
122     x101 + rectangle_dim/2, y101 + rectangle_dim, smooth = 1)
123
124     canvas.create_line(x000 + rectangle_dim/2, y000,
125     x001 + rectangle_dim/2, y001 + rectangle_dim)
126     canvas.create_line(x010 + rectangle_dim, y010 + rectangle_dim/2,
127     x011, y011 + rectangle_dim/2)
128     canvas.create_line(x110 + rectangle_dim/2, y110 + rectangle_dim,
129     x111 + rectangle_dim/2, y111)
130     canvas.create_line(x100, y100 + rectangle_dim/2,
131     x101 + rectangle_dim, y101 + rectangle_dim/2)
132
133     canvas.pack(fill=BOTH, expand=1)
134
135 def draw_dungeon(dungeon):
136
137     root = Tk()
138     ex = Example(dungeon)
139     root.geometry("700x500+600+100")
140     a = Agent(dungeon["000"], dungeon)
141     root.after(0, a.explore())
142     root.mainloop()
143
144
145
146 #-----VARIABILI PER CONSERVARE L'ESECUZIONE DI UNA ESPLORAZIONE-----
147 all_explorations_path = []
148
149 class Exploration:
150     def __init__(self):
151         self.rooms_explored = []
152         self.health = -1
153         self.attack = -1
154         self.probability_of_victory = ""
155         self.outcome = None
156
157 class ExplorationOutcome():
158     BOSSDEFEATED = "Boss defeated!"
159     BOSSKILLEDYOU = "The Boss killed you!"
160     YOUDIED = "You died while exploring the dungeon!"
161
162
163 #-----LOGICA DEL DUNGEON-----
164
165 import math
166 import random
167
168 class RoomContent():
169     ENEMIES = "Enemies"
170     TREASURE = "Treasure"
171     SHOP = "Shop"
172     BOSS = "Boss"
173
174
175
176 class Room:
177
178     #ogni stanza avra' un indice (tre bit rappresentati come stringa) e un
179     #contenuto (rappresentato anch'esso da una stringa). Altre cose che mi
180     #interessa sapere di una stanza sono se l'ho gia' visitata e qual e' la

```

```

181 #sua percentuale qualitativa (quest'ultimo attributo sara'
182 #l'avventuriero a settarlo)
183 def __init__(self, index):
184     self.index = index
185     self.adjacent_rooms = getAdjacentRooms(self.index)
186     self.content = None
187     self.visited = False
188     self.quality = -10
189
190 def room_to_string(self):
191     print("Stanza = " + self.index + ", content = " + self.content)
192
193
194
195 class Agent:
196
197     DEBUG = False
198
199     #definisco il mio agente in termini di:
200     #-la stanza in cui si trova
201     #-se ha visto il tesoro
202     #-se ha visto lo shop
203     #-i suoi punti salute
204     #-i suoi punti attacco
205
206     #la qualita' della stanza puo' essere intesa come la probabilita' di
207     #non prendere danno quando si parla di stanze coi nemici, probabilita'
208     #di vincere contro il boss se si tratta della stanza del boss, sara'
209     #sempre 100% quando si tratta del tesoro e puo' variare per il negozio.
210     #Sia il negozio che il boss usano la stessa formula per indicare la
211     #loro qualita', che ha come parametri i punti vita e i punti attacco
212     def __init__(self, current_room, dungeon):
213         self.current_room = current_room
214         self.dungeon = dungeon
215         self.treasure_seen = False
216         self.shop_seen = False
217         self.health_points = 4
218         self.attack_points = 1
219
220
221
222
223     #funzione per calcolare la probabilita' di battere il boss in base
224     #ai punti salute e i punti attacco
225     def get_probability_boss_victory(self, health, attack):
226         return 5 * (1 + health) * attack
227
228
229
230     #funzione che implementa l'intera esplorazione del mio agente
231     def explore(self):
232
233         #eo" sara' l'oggetto che maniene il riepilogo di questa esplorazione
234         eo = Exploration()
235
236         #finche' non ho trovato il boss
237         while not self.current_room.content == RoomContent.BOSS:
238
239             eo.rooms_explored.append(self.current_room.index)
240
241             if(Agent.DEBUG):
242                 print("+++++++\n")
243                 print("Stanza = " + str(self.current_room.index))
244                 print("Contenuto = " + str(self.current_room.content))
245                 print("Vista = " + str(self.current_room.visited))
246
247             #-----PARTE IN CUI APPLICO GLI EFFETTI DELLA STANZA

```

```

248 #IN CUI SONO APPENA ENTRATO -----
249
250 #se sono entrato in una stanza con nemici, lancio una monetina
251 #e forse prendo danno. Questo non vale all'inizio della partita,
252 #perche' e' vero che sono in una stanza popolata da nemici, ma
253 #di fatto non ci sono entrato, ci sono "nato".
254 if (self.current_room.content == RoomContent.ENEMIES and
255     not (self.current_room.index == "000" and
256           self.current_room.visited == False)):
257     #capisco se ho preso danno o no sulla base dell'attacco
258     l = []
259     r = 1
260     if self.attack_points == 1:
261         for i in range(25):
262             l.append(1)
263             for i in range(75):
264                 l.append(0)
265                 r = random.choice(l)
266     if self.attack_points == 2:
267         for i in range(50):
268             l.append(1)
269             for i in range(50):
270                 l.append(0)
271                 r = random.choice(l)
272     if self.attack_points == 3:
273         for i in range(75):
274             l.append(1)
275             for i in range(25):
276                 l.append(0)
277                 r = random.choice(l)
278     if self.attack_points == 4:
279         r = 1
280
281     if r == 0:
282         self.health_points -= 1
283         if Agent.DEBUG:
284             print("Damage taken! Argh!")
285         elif Agent.DEBUG:
286             print("No damage taken! Phew!")
287
288 #se sono entrato nella stanza del tesoro o nel negozio, me lo
289 #segno, e aggiorno il contenuto di queste stanze con dei nemici
290 if self.current_room.content == RoomContent.TREASURE:
291     self.treasure_seen = True
292
293     #ottengo il potenziamento casuale
294     values = [1, 2]
295     power_up_types = ["health", "attack"]
296     v = random.choice(values)
297     p = random.choice(power_up_types)
298     if Agent.DEBUG:
299         print("Power up found = " + str(v) + p)
300     if v == 1 and p == "health" and self.health_points <= 3:
301         self.health_points += 1
302         if Agent.DEBUG:
303             print("+1 h")
304     if v == 2 and p == "health" and self.health_points <= 3:
305         if self.health_points <= 2:
306             if Agent.DEBUG:
307                 print("+2 h")
308             self.health_points += 2
309         else:
310             if Agent.DEBUG:
311                 print("+2 h, but really +1 h")
312             self.health_points += 1
313     if v == 1 and p == "attack":
314         if Agent.DEBUG:

```

```

315         print("+1 a")
316         self.attack_points += 1
317     if v == 2 and p == "attack":
318         if Agent.DEBUG:
319             print("+2 a")
320         self.attack_points += 2
321
322     self.current_room.content = RoomContent.ENEMIES
323
324     if self.current_room.content == RoomContent.SHOP:
325         self.shop_seen = True
326
327         #tolgo un punto vita e ne aggiungo uno di danno
328         self.health_points -= 1
329         self.attack_points += 1
330
331     self.current_room.content = RoomContent.ENEMIES
332
333     if Agent.DEBUG:
334         print("Health = " + str(self.health_points))
335         print("Attack = " + str(self.attack_points))
336
337     #----STANZA ESPLORATA, ORA DECIDO SE SONO MORTO O MENO.
338     #SE NON SONO MORTO MI SEGNO CHE HO ESPLORATO QUESTA STANZA-----
339
340     #se un nemico mi ha ucciso, esco dal ciclo
341     if self.health_points == 0:
342         break
343
344     #segno che mi sono visto la stanza corrente
345     self.current_room.visited = True
346
347     #-----ADESSO C'e' LA FUNZIONE DI TRANSIZIONE, OVVERO
348     #LA LOGICA CHE PORTA A SCEGLIERE LA PROSSIMA STANZA-----
349
350     #considero le stanze adiacenti
351     adj_r_index = self.current_room.adjacent_rooms
352     adjacent_rooms = []
353     for i in adj_r_index:
354         adjacent_rooms.append(dungeon[i])
355
356     adjacent_rooms = self.assign_quality(adjacent_rooms)
357
358     if Agent.DEBUG:
359         for room in adjacent_rooms:
360             print("Adjacent room " + str(room.index) + " has " +
361                 str(room.quality) + " quality")
362
363     #ora che ho assegnato una qualita' a tutte le stanze adiacenti,
364     #devo scegliere dove andare. Potrebbero tuttavia esserci piu'
365     #stanze con la qualita' massima, e io devo segnarmele tutte.
366     max_quality = 0
367     best_quality_rooms = []
368     for room in adjacent_rooms:
369
370         if room.quality == max_quality:
371             best_quality_rooms.append(room)
372
373         elif room.quality >= max_quality:
374             max_quality = room.quality
375             best_quality_rooms.clear()
376             best_quality_rooms.append(room)
377
378     if Agent.DEBUG:
379         print("Best rooms = ", end = "")
380         for elem in best_quality_rooms:
381             print(str(elem.index) + ", ", end = "")

```

```

382     print()
383
384     #ora controllo: se c'e' una sola stanza con la qualita' migliore,
385     #ho finito. Altrimenti preferiro' quella non ancora esplorata
386     next_room = None
387     if len(best_quality_rooms) > 1:
388         not_visited_rooms = []
389         for room in best_quality_rooms:
390             if room.visited == False:
391                 not_visited_rooms.append(room)
392
393     if Agent.DEBUG:
394         print("Not visited rooms = ", end = "")
395         for room in not_visited_rooms:
396             print(str(room.index) + ", ", end="")
397         print()
398
399         #se c'e' una sola stanza che non ho ancora esplorato,
400         #vado in quella
401         if len(not_visited_rooms) == 1:
402             next_room = not_visited_rooms[0]
403         #se invece ci sono piu' stanze che non ho ancora esplorato,
404         #ne scelgo una a caso
405         if len(not_visited_rooms) > 1:
406             next_room = random.choice(not_visited_rooms)
407         #se invece le ho esplorate tutte, ne scelgo comunque
408         #una a caso
409         if len(not_visited_rooms) == 0:
410             next_room = random.choice(best_quality_rooms)
411
412     else:
413         next_room = best_quality_rooms[0]
414
415     if Agent.DEBUG:
416         print("Next Room = " + next_room.index)
417
418     #-----
419
420     self.current_room = next_room
421
422     outcome = None
423     if self.health_points == 0:
424         outcome = ExplorationOutcome.YOUDIED
425         if Agent.DEBUG:
426             print("Sei morto!")
427     else:
428         health = self.health_points
429         attack = self.attack_points
430         probability_of_victory = 5 * (1 + health) * attack
431         if Agent.DEBUG:
432             print("La tua probabilita' di vittoria, con " +
433                 str(self.health_points) + " punti vita e " +
434                 str(self.attack_points) + " punti attacco, e' del " +
435                 str(probability_of_victory) + " %")
436         l = []
437         for i in range(probability_of_victory):
438             l.append(1)
439         for i in range(100 - probability_of_victory):
440             l.append(0)
441         r = random.choice(l)
442         if r == 1:
443             outcome = ExplorationOutcome.BOSSDEFATED
444             if Agent.DEBUG:
445                 print("Vittoria!")
446         else:
447             outcome = ExplorationOutcome.BOSSKILLEDYOU
448             if Agent.DEBUG:

```

```

449     print("Il Boss ti ha ucciso!")
450
451     #cosi' che tutte le partite che finiscono dal boss abbiano
452     #come ultima stanza proprio quella del boss
453     eo.rooms_explored.append(self.current_room.index)
454     eo.probability_of_victory = str(probability_of_victory) + "%"
455
456
457     eo.health = self.health_points
458     eo.attack = self.attack_points
459     eo.outcome = outcome
460
461     all_explorations_path.append(eo)
462
463
464
465     #funzione per assegnare la giusta qualita' alle stanze
466     def assign_quality(self, adjacent_rooms):
467
468         #definisco un ordinamento delle stanze. Questo serve per decidere
469         #al meglio la prossima stanza da visitare
470         tmp = []
471         for i in range(len(adjacent_rooms)):
472             tmp.append(None)
473
474         j = 0
475         #fintanto che non ho tolto tutte le stanze da adjacent_rooms
476         while len(adjacent_rooms) > 0:
477             #controllo il contenuto di ciascuna stanza. I tipi inseriti
478             #nell'array room_types sono in ordine decrescente di
479             #importanza (nel senso che voglio prima valutare le stanze
480             #del tesoro, poi lo shop, poi il boss e poi i nemici. Questo
481             #perche', a seconda della situazione, la "probabilita' di
482             #vittoria" di una stanza non rispecchia la qualita' effettiva
483             #della stanza. Se ad esempio ho health = 2 e attack = 3,
484             #e' vero che ho il 75% di probabilita' di vincere contro
485             #un nemico e solo il 45% di vincere contro il boss, ma se
486             #ho gia' visitato negozio e tesoro, non me ne importa nulla
487             #dei nemici, voglio affrontare il boss!)
488             room_types = [RoomContent.TREASURE, RoomContent.SHOP,
489                           RoomContent.BOSS, RoomContent.ENEMIES]
490             for c in room_types:
491                 i = 0
492                 while i < len(adjacent_rooms):
493                     if adjacent_rooms[i].content == c:
494                         tmp[j] = adjacent_rooms.pop(i)
495                         j += 1
496                         break
497                     else:
498                         i += 1
499
500             #ora vado ad assegnare la qualita' vera e propria alle stanze
501             for room in tmp:
502                 #ora assegno una qualita' alle stanze. La qualita' di una
503                 #stanza dipendera' dalle mie statistiche, dal fatto che
504                 #ho visto o meno il tesoro e il negozio, e da cosa
505                 #potrebbe succedere se entrassi in una stanza o meno.
506
507                 #se la stanza adiacente contiene il tesoro (e non l'ho
508                 #ancora visto, ma in teoria non ci sara' mai uno scenario
509                 #in cui ho gia' visto la stanza del tesoro ma sara' presente
510                 #tale stanza), ci vado. Il tesoro ha sempre una
511                 #priorita' elevata
512                 if (self.treasure_seen == False and
513                     room.content == RoomContent.TREASURE):
514                     room.quality = 10
515

```

```

516     #negozi
517     h = self.health_points
518     a = self.attack_points
519     if room.content == RoomContent.SHOP:
520         #se ho uno di vita per me non ha la minima importanza,
521         #dato che morirei se ci entrassi.
522         if h == 1:
523             room.quality = -1
524         else:
525             #se entrando nel negozio le mie probabilita',
526             #di vittoria aumentano nettamente, ci vorro' andare,
527             #dando pero' sempre priorita' al tesoro
528             if (self.get_probability_boss_victory(h -1, a +1)
529                 - self.get_probability_boss_victory(h, a) > 0):
530                 room.quality = 8
531             #se invece le mie probabilita' rimanessero le stesse
532             #negozi o meno, a volta puo' interessarmi altre no,
533             #dipende se ho il boss vicino
534             elif (self.get_probability_boss_victory(h -1, a +1)
535                 - self.get_probability_boss_victory(h, a) == 0):
536                 room.quality = 6
537             else:
538                 room.quality = 0
539
540             #devo trattare il negozio in modo diverso a seconda
541             #del fatto che ho gia' visto o meno il tesoro. Se sono
542             #gia' passato per la stanza del tesoro, devo decidere
543             #ADESSO se il negozio mi interessa (dato che la mia
544             #vita puo' solo diminuire a questo punto), altrimenti,
545             #se non ho ancora visto il tesoro, magari adesso non
546             #mi interessa, ma potrebbe in futuro. In questo esempio
547             #con soltanto 8 stanze in realta' questo controllo non
548             #servirebbe, ma sarebbe necessario in uno scenario
549             #piu' complicato
550             if self.treasure_seen == True:
551                 #se ho gia' visto il tesoro, a prescindere dal fatto
552                 #che poi entrero' nel negozio o meno, mi segno che
553                 #l'ho gia' visto, cosi' che, se anche non dovesse
554                 #entrarci adesso, in futuro non lo terro' nemmeno
555                 #in considerazione
556                 self.shop_seen = True
557
558             #boss
559             if room.content == RoomContent.BOSS:
560                 #se non ho visto il tesoro il boss non mi interessa
561                 if self.treasure_seen == False:
562                     room.quality = 0
563                 #se ho visto sia il negozio che il tesoro dipende:
564                 #in generale vorro' andare dal boss. Se pero' il negozio
565                 #e' adiacente, ci sono volte in cui vorrei poterci andare,
566                 #altre in cui invece non voglio
567                 elif self.shop_seen == True:
568                     room.quality = 7
569                 #se ho visto il tesoro ma non il negozio, mi interessera'
570                 #andare prima al negozio solo se andarci mi porta
571                 #effettivamente un vantaggio stretto (ovvero > e non >=)
572             else:
573                 #se ho uno di vita e' chiaro che il negozio non
574                 #mi interessa. Ma non mi interessa nemmeno se non
575                 #ci guadagnerei nulla nell'andare nel negozio
576                 #((in realta' e' possibile dimostrare, nell'esempio
577                 #del labirinto a 8 stanze cosi' definito, che la seconda
578                 #condizione della guardia non puo' MAI essere vera)
579                 if (h == 1 or
580                     (self.get_probability_boss_victory(h -1, a +1)
581                     - self.get_probability_boss_victory(h, a) <= 0)):
582                     room.quality = 3

```

```

583     #altrimenti valuto il guadagno che avrei andando
584     #nel negozio. Se mi conviene andare nel negozio, vuol
585     #dire che il boss ha meno importanza anche dei nemici,
586     #dato che il negozio potrebbe essere non immediatamente
587     #disponibile
588     else:
589         if (self.get_probability_boss_victory(h -1, a +1 )
590             - self.get_probability_boss_victory(h, a) > 0):
591             #in realta' questo if non servirebbe dato che,
592             #se siamo arrivati a questo punto, la condizione
593             #nell'if e' sicuramente vera (e' possibile
594             #dimostrarlo, sarebbe il duale di quanto
595             #detto sopra)
596             room.quality = 1
597
598
599         #in fondo alla catena alimentare ci sono i nemici, che mi
600         #interessano solo quando tutte le altre alternative
601         #sono peggiori
602         if room.content == RoomContent.ENEMIES:
603             room.quality = 2
604
605     return tmp
606
607
608 #definisco un metodo che, dato l'indice di una stanza, restituisce tutte
609 #le stanze adiacenti
610 def getAdjacentRooms(index):
611
612     res = []
613
614     for i in range(len(index)):
615
616         #prendo l'indice originale e swappo ogni suo qubit, ottenendo
617         #tre nuove stringhe
618         current_bit = index[i]
619         if current_bit == "1":
620             current_bit = "0"
621         else:
622             current_bit = "1"
623
624         list_index = list(index)
625         list_index[i] = current_bit
626
627
628         new_index = ""
629         for i in list_index:
630             new_index += i
631
632         res.append(new_index)
633
634     return res
635
636
637
638 #funzione che, dato un numero naturale e un numero di bits,
639 #codifica tale numero in una stringa di bit lunga number_of_bits
640 def intToBinary(num, number_of_bits):
641
642     if num > pow(2, number_of_bits) - 1:
643         raise Exception("Error: number " + str(num) +
644             " cannot be represented with only " + str(number_of_bits) +
645             " bits")
646
647     list_bits = []
648
649     for i in reversed(range(number_of_bits)):

```

```

650     current_pow_of_two = pow(2, i)
651     if current_pow_of_two <= num:
652         list_bits.append(1)
653         num -= current_pow_of_two
654     else:
655         list_bits.append(0)
656
657     res = ""
658     for i in list_bits:
659         res += str(i)
660
661     return res
662
663
664
665
666
667 dungeon = {}
668 def full_exploration(draw):
669     rooms = []
670     for i in range(8):
671         rooms.append(Room(intToBinary(i, 3)))
672
673     #la prima stanza contiene sempre nemici
674     rooms[0].content = RoomContent.ENEMIES
675     rooms[0].visited = False
676
677     #il contenuto di tutte le altre lo scelgo io. Per personalizzare
678     #il dungeon basta cambiare il valore di ogni room[i].content.
679     #basta tradurre il valore di i (i = 1...7) in binario per ottenere
680     #l'indice a tre bit corrispondente.
681     rooms[1].content = RoomContent.ENEMIES
682     rooms[2].content = RoomContent.ENEMIES
683     rooms[3].content = RoomContent.BOSS
684     rooms[4].content = RoomContent.ENEMIES
685     rooms[5].content = RoomContent.SHOP
686     rooms[6].content = RoomContent.TREASURE
687     rooms[7].content = RoomContent.ENEMIES
688     rooms[1].visited = False
689     rooms[2].visited = False
690     rooms[3].visited = False
691     rooms[4].visited = False
692     rooms[5].visited = False
693     rooms[6].visited = False
694     rooms[7].visited = False
695
696
697
698     if Agent.DEBUG:
699         for i in rooms:
700             print("contenuto della stanza " + str(i.index) + " = " +
701                 str(i.content) + ", le sue stanze adiacenti sono " +
702                 str(i.adjacent_rooms))
703
704     #metto adesso le stanze in un dizionario cosi' che per il mio agente
705     #sia piu' semplice vederne il contenuto
706
707     for i in rooms:
708         dungeon[i.index] = i
709
710     if Agent.DEBUG:
711         for i in rooms:
712             i.room_to_string()
713
714     if not draw:
715         a = Agent(dungeon["000"], dungeon)
716         a.explore()

```

```

717     else:
718         #per visualizzare il dungeon a schermo, chiamare un'ultima
719         #volta full_exploration(True). Questo tipo di chiamata
720         #a questo metodo non effettua una visita del dungeon,
721         #ma stampa semplicemente a schermo la struttura del dungeon
722         #in modo comprensibile
723         draw_dungeon(dungeon)
724
725
726 #qui 100 e' un numero scelto arbitrariamente. Tanto piu' e' elevato,
727 #tante piu' saranno le esplorazioni che il mio agente fara', dandomi
728 #una soluzione piu' affidabile.
729 for k in range(100):
730     full_exploration(False)
731
732
733
734
735
736 #adesso, in all_explorations_path, ho tutti i percorsi che il mio
737 #agente ha compiuto. Cominciamo a scremarli.
738 #prima di tutto, rimuoviamo tutte le esecuzioni che hanno portato
739 #a una sconfitta, poi possiamo pensare di rimuovere anche
740 #le esecuzioni dove il boss ci ha sconfitti.
741 explorations_path_boss_reached = []
742 for elem in all_explorations_path:
743     if not (elem.outcome == ExplorationOutcome.YOUDIED or
744             elem.outcome == ExplorationOutcome.BOSSKILLEDOYOU):
745         explorations_path_boss_reached.append(elem)
746
747 #ora ho solo i casi in cui ho vinto. Da questi, mi piacerebbe
748 #tenermi le esecuzioni in cui ho massimizzato le possibilita'
749 #di vittoria.
750 explorations_path_high_stats = []
751 current_highest_probability = 100
752 while len(explorations_path_high_stats) == 0:
753
754     for elem in explorations_path_boss_reached:
755         if ((5 * (1 + elem.health) * elem.attack) ==
756             current_highest_probability):
757             explorations_path_high_stats.append(elem)
758
759     current_highest_probability -= 5
760
761
762
763 #ora dobbiamo rimuovere, tra i percorsi vincenti, quelli piu' lunghi.
764 #e' infatti piu' probabile prendere danno affrontando tante stanze
765 #che non affrontandone poche.
766 explorations_path_shortest = []
767 shortest_path_dim = 10000
768 for elem in explorations_path_high_stats:
769     if len(elem.rooms_explored) < shortest_path_dim:
770         explorations_path_shortest.clear()
771         explorations_path_shortest.append(elem)
772         shortest_path_dim = len(elem.rooms_explored)
773     elif len(elem.rooms_explored) == shortest_path_dim:
774         explorations_path_shortest.append(elem)
775
776
777 #e ora, tra tutti i percorsi di ugual minima lunghezza ottenuti,
778 #possiamo tenerci quello che ha un'occorrenza maggiore.
779 dict_explorations_path_shortest = {}
780 for i in range(len(explorations_path_shortest)):
781     curr = str(explorations_path_shortest[i].rooms_explored)
782     if curr in dict_explorations_path_shortest:
783         dict_explorations_path_shortest[curr] += 1

```

```

784     else:
785         dict_explorations_path_shortest[curr] = 1
786
787 max_occurrence = 0
788 solutions = []
789 for key in dict_explorations_path_shortest:
790     if dict_explorations_path_shortest[key] > max_occurrence:
791         solutions.clear()
792         solutions.append(key)
793         max_occurrence = dict_explorations_path_shortest[key]
794     elif dict_explorations_path_shortest[key] == max_occurrence:
795         solutions.append(key)
796
797 #per visualizzare i migliori percorsi disponibili
798 #(l'output del programma)
799 for elem in solutions:
800     print("Possible solution = " + elem)
801
802
803
804
805
806 #settando a true questa variabile e' possibile visualizzare
807 #il contenuto di una delle liste di percorsi con cui
808 #abbiamo lavorato finora.
809 visualize_a_list = True
810 if visualize_a_list:
811     #e' possibile modificare la lista da assegnare a f_list
812     #per vedere un diverso tipo di output. Le liste possibili includono:
813     #-all_explorations_path
814     #-explorations_path_boss_reached
815     #-explorations_path_high_stats
816     #-explorations_path_shortest
817
818 f_list = all_explorations_path
819 f = open("dungeon_summary.txt", "w")
820 for i in range(len(f_list)):
821     if i < 10:
822         f.write("00")
823     elif i>=10 and i<100:
824         f.write("0")
825     f.write(str(i) + " exploration = " +
826             str(f_list[i].rooms_explored))
827
828     #assumo che un agente non possa esplorare piu' di 20 stanze
829     blank_spaces = 20 - len(f_list[i].rooms_explored)
830     for j in range(blank_spaces):
831         f.write(" ")
832
833     f.write(", h = " + str(f_list[i].health) + ", a = " +
834         str(f_list[i].attack) + ", outcome = " +
835         str(f_list[i].outcome) + "\n")
836 f.close()
837
838
839 f = open("dungeon_summary.txt", "r")
840 print(f.read())
841
842 #per disegnare il dungeon
843 full_exploration(True)

```


Appendice C

Codice quantistico per l'esplorazione di un labirinto a otto stanze

```
1 import math
2 from random import *
3 from qiskit import *
4 from qiskit.tools.visualization import plot_histogram
5 import random
6
7 #Molte cose saranno simili al codice classico, cambia solo la rappresentazione della
8 #funzione di transizione, ovvero come l'agente sceglie la prossima stanza in cui andare
9
10 #variabili aggiuntive che uso per tenere traccia della situazione
11 qc_list = []
12 counts_list = []
13
14 #dizionario di coppie <contenuto di una stanza - relativo valore in bit>
15 room_types_dict = {"Enemies":"00", "Treasure":"01", "Shop":"10", "Boss":"11"}
16
17 #dichiaro i qubit di cui avro' bisogno nel mio circuito
18
19 #dato che la scelta e' tra tre stanze, ho bisogno di due qubit per salvarmi
20 #il movimento scelto dall'agente. In particolare:
21 #00 = vado a sinistra
22 #01 = vado al centro
23 #10 = vado a destra
24 phi = QuantumRegister(2, "phi")
25
26 #due qubit per la salute, dove 00 = 1, 01 = 2, 10 = 3 e 11 = 4
27 health = QuantumRegister(2, "health")
28
29 #due qubit anche per l'attacco, stesso discorso di prima
30 attack = QuantumRegister(2, "attack")
31
32 #un qubit per sapere se ho gia' visto la treasure, uno per sapere se ho gia'
33 #visto il negozio e un ulteriore qubit per sapere se il qubit che mi dice
34 #se ho gia' visto il negozio e' a 1 (e' un qubit di lavoro)
35 treasure_seen = QuantumRegister(1, "if treasure seen")
36 shop_seen = QuantumRegister(1, "if shop seen")
37 shop_seen_true = QuantumRegister(1, "if shop seen = 1")
38
39 #due qubit che descrivono il contenuto della stanza a sinistra
40 left_content = QuantumRegister(2, "left content")
41
42 #due qubit che descrivono il contenuto della stanza al centro
43 center_content = QuantumRegister(2, "center content")
44
45 #due qubit che descrivono il contenuto della stanza a destra
46 right_content = QuantumRegister(2, "right content")
```

```

47
48 #tre qubit, ciascuno dei quali mi dice se ho visto di gia' la stanza a sinistra,
49 #al centro e a destra
50 left_seen = QuantumRegister(1, "left seen")
51 center_seen = QuantumRegister(1, "center seen")
52 right_seen = QuantumRegister(1, "right seen")
53
54 #tre qubit, ciascuno per sapere se ho adiacente il tesoro, il negozio o il boss
55 is_treasure_adjacent = QuantumRegister(1, "treasure adjacent")
56 is_shop_adjacent = QuantumRegister(1, "shop adjacent")
57 is_boss_adjacent = QuantumRegister(1, "boss adjacent")
58
59 #due qubit di lavoro aggiuntivi che mi serviranno per scegliere tra negozio e boss quando
60 #ce li ho entrambi adiacenti
61 shop_max_quality = QuantumRegister(1, "shop max quality")
62 boss_max_quality = QuantumRegister(1, "boss max quality")
63
64 #nella implementazione classica associo un valore ad ogni stanza adiacente. In particolare sia
65 #il negozio che il boss possono assumere, a seconda delle stanze adiacenti e quelle viste
66 #in precedenza, uno tra quattro valori distinti. Bene, qui facciamo lo stesso: usiamo due qubit
67 #per mantenerci il valore del negozio e altri due per il valore del boss.
68 #Occhio alle associazioni (da confrontare col codice classico):
69 #NEGOZIO
70 #00 = -1
71 #01 = 0
72 #10 = 6
73 #11 = 8
74 #BOSS
75 #00 = 0
76 #01 = 1
77 #10 = 3
78 #11 = 7
79 shop_quality = QuantumRegister(2, "shop quality")
80 boss_quality = QuantumRegister(2, "boss quality")
81
82 #non dimentichiamoci il qubit dell'oracolo che deve effettuare il kickback di fase
83 oracle_qubit = QuantumRegister(1, "oracle qubit")
84
85 #cosi' come di due bit classici per leggere quale sara' la prossima stanza che il nostro agente
86 #vorra' esplorare
87 measure_bits = ClassicalRegister(2, "next_room")
88
89 #ci sara' un'altra cosa che vorremo misurare: se abbiamo visto il negozio o meno. Questo
90 #non e' necessario nel caso del tesoro, in quanto l'agente se vede la stanza del tesoro
91 #ci entra di sicuro. Per il negozio invece abbiamo visto nel codice classico che la situazione
92 #e' un po' diversa: se l'agente ha visto il tesoro e ha di fianco il negozio, si segna
93 #che comunque l'ha visto, in quanto, se non ci entra adesso, non ci entra piu' (dato che,
94 #una volta ottenuto il tesoro, la vita puo' solo scendere)
95 shop_bit = ClassicalRegister(1, "shop_seen_bit")
96
97
98 #costruiamo l'inizio del circuito
99 qc_1 = QuantumCircuit(phi, health, attack, treasure_seen, shop_seen, shop_seen_true,
100                         left_content, center_content, right_content, left_seen, center_seen,
101                         right_seen, is_treasure_adjacent, is_shop_adjacent, is_boss_adjacent,
102                         shop_max_quality, boss_max_quality, shop_quality, boss_quality,
103                         oracle_qubit, measure_bits, shop_bit)
104
105 qc_1.draw('mpl')
106
107
108
109 #ora ci serve una funzione che costruisca il circuito vero e proprio. O meglio, ci servono
110 #tre funzioni principali:
111 #-una che inizializzi i qubit del circuito
112 #-l'oracolo, ovvero la funzione di transizione
113 #-il diffuser

```

```

114 #a sua volta, la funzione dell'oracolo e' divisa in piu' parti per semplificare la lettura:
115 #-una funzione relativa alla scelta del tesoro
116 #-una funzione relativa alla scelta del negozio
117 #-una funzione relativa alla scelta del boss
118 #-una funzione che decide, quando necessario, se sia meglio andare al negozio o dal boss
119
120 def initialize_dungeon_circuit(qc, agent, adjacent_rooms):
121
122     initialize_dungeon_circuit_debug = False
123
124     #come prima cosa porto nello stato di ugual sovrapposizione i qubit phi
125     qc.h(phi)
126
127     #poi inizializzo i qubit che descrivono la salute e l'attacco del personaggio
128     if(agent.health_points == 2):
129         qc.x(health[0])
130     elif(agent.health_points == 3):
131         qc.x(health[1])
132     elif(agent.health_points == 4):
133         qc.x(health[0])
134         qc.x(health[1])
135
136     if(agent.attack_points == 2):
137         qc.x(attack[0])
138     elif(agent.attack_points == 3):
139         qc.x(attack[1])
140     elif(agent.attack_points == 4):
141         qc.x(attack[0])
142         qc.x(attack[1])
143
144     #poi i qubit che descrivono se ho visto o meno il tesoro e il negozio
145     if(agent.treasure_seen == True):
146         qc.x(treasure_seen[0])
147     if(agent.shop_seen == True):
148         qc.x(shop_seen[0])
149     qc.cx(shop_seen[0], shop_seen_true[0])
150
151
152
153     #ora i qubit che descrivono il contenuto delle stanze adiacenti
154     sx_cx_dx_content = [left_content, center_content, right_content]
155     i = 0
156
157     for room in adjacent_rooms:
158         value = room_types_dict[room.content]
159         if value == "01":
160             qc.x(sx_cx_dx_content[i][0])
161             if initialize_dungeon_circuit_debug:
162                 print("room number " + str(i) + ": applied content qubits 01")
163         elif value == "10":
164             qc.x(sx_cx_dx_content[i][1])
165             if initialize_dungeon_circuit_debug:
166                 print("room number " + str(i) + ": applied content qubits 10")
167         elif value == "11":
168             qc.x(sx_cx_dx_content[i][0])
169             qc.x(sx_cx_dx_content[i][1])
170             if initialize_dungeon_circuit_debug:
171                 print("room number " + str(i) + ": applied content qubits 11")
172         i += 1
173
174     #e anche i qubit che mi dicono se ho gia' visto una delle stanze adiacenti
175     dx_cx_sx_seen = [left_seen, center_seen, right_seen]
176     i = 0
177
178     for room in adjacent_rooms:
179         if room.visited == True:
180             qc.x(dx_cx_sx_seen[i])
181         i += 1

```

```

181 #inizializzo infine il qubit dell'oracolo
182 qc.x(oracle_qubit)
183 qc.h(oracle_qubit)
184
185
186
187
188 def treasure_logic():
189     qc = QuantumCircuit(phi, health, attack, treasure_seen, shop_seen, shop_seen_true,
190                         left_content, center_content, right_content, left_seen, center_seen,
191                         right_seen, is_treasure_adjacent, is_shop_adjacent, is_boss_adjacent,
192                         shop_max_quality, boss_max_quality, shop_quality, boss_quality,
193                         oracle_qubit)
194
195 #TREASURE: se ho la stanza del tesoro vicina, ci vado, e mi segno che mi e' adiacente
196
197 #tesoro a sinistra
198 qc.x(phi)
199 qc.x(left_content[1])
200 qc.x(treasure_seen[0])
201 qc.mcx([phi[0], phi[1], left_content[0], left_content[1], treasure_seen[0]], oracle_qubit)
202 qc.mcx([left_content[0], left_content[1], treasure_seen[0]], is_treasure_adjacent[0])
203 qc.x(treasure_seen[0])
204 qc.x(left_content[1])
205 qc.x(phi)
206
207
208
209 #tesoro al centro
210 qc.x(phi[1])
211 qc.x(center_content[1])
212 qc.x(treasure_seen[0])
213 qc.mcx([phi[0], phi[1], center_content[0], center_content[1], treasure_seen[0]],
214         oracle_qubit)
215 qc.mcx([center_content[0], center_content[1], treasure_seen[0]], is_treasure_adjacent[0])
216 qc.x(treasure_seen[0])
217 qc.x(center_content[1])
218 qc.x(phi[1])
219
220
221
222 #tesoro a destra
223 qc.x(phi[0])
224 qc.x(right_content[1])
225 qc.x(treasure_seen[0])
226 qc.mcx([phi[0], phi[1], right_content[0], right_content[1], treasure_seen[0]], oracle_qubit)
227 qc.mcx([right_content[0], right_content[1], treasure_seen[0]], is_treasure_adjacent[0])
228 qc.x(treasure_seen[0])
229 qc.x(right_content[1])
230 qc.x(phi[0])
231
232
233 gate = qc.to_gate()
234 gate.name = "Treasure logic"
235 return gate
236
237
238
239 def shop_logic():
240     qc = QuantumCircuit(phi, health, attack, treasure_seen, shop_seen, shop_seen_true,
241                         left_content, center_content, right_content, left_seen, center_seen,
242                         right_seen, is_treasure_adjacent, is_shop_adjacent, is_boss_adjacent,
243                         shop_max_quality, boss_max_quality, shop_quality, boss_quality,
244                         oracle_qubit)
245
246 #SHOP: devo capire il valore del negozio
247

```

```

248 #...ma mi segno anche se effettivamente ce l'ho il negozio adiacente
249 qc.x(left_content[0])
250 qc.mcx([left_content[0], left_content[1]], is_shop_adjacent[0])
251 qc.x(left_content[0])
252
253 qc.x(center_content[0])
254 qc.mcx([center_content[0], center_content[1]], is_shop_adjacent[0])
255 qc.x(center_content[0])
256
257 qc.x(right_content[0])
258 qc.mcx([right_content[0], right_content[1]], is_shop_adjacent[0])
259 qc.x(right_content[0])
260
261
262
263
264 #dicevamo: VALORE DEL NEGOZIO
265 #di default, se non ce l'ho adiacente, varra' -1. Questo semplifica il circuito
266 #quando arriviamo alla parte delle stanze contenenti nemici
267 #-1 anche se ho 1 di salute (default, non devo applicare porte)
268
269 #0 se, andando nel negozio, peggiorerei la mia chance di vittoria. Succede solo
270 #quando ho 2 salute e 3 attacco
271 qc.x(health[1])
272 qc.x(attack[0])
273 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[0])
274 qc.x(attack[0])
275 qc.x(health[1])
276
277
278
279 #6 se, andando nel negozio, le mie probabilita' di vittoria non cambiano. Succede quando:
280 #health = 2 attack = 2
281 #health = 3 attack = 3
282 qc.x(health[1])
283 qc.x(attack[1])
284 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[1])
285 qc.x(attack[1])
286 qc.x(health[1])
287
288
289
290 qc.x(health[0])
291 qc.x(attack[0])
292 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[1])
293 qc.x(attack[0])
294 qc.x(health[0])
295
296
297
298 #8 se, andando nel negozio, le mie probabilita' di vittoria aumentano strettamente.
299 #Succede quando:
300 #health = 2 and attack = 1
301 #health = 3 and (attack=1 or attack=2)
302 #health = 4 and (attack=1 or attack=2 or attack=3)
303 qc.x(health[1])
304 qc.x(attack)
305 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[0])
306 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[1])
307 qc.x(attack)
308 qc.x(health[1])
309
310
311
312 qc.x(health[0])
313 qc.x(attack)
314 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[0])

```

```

315 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[1])
316 qc.x(attack)
317 qc.x(health[0])
318
319
320
321 qc.x(health[0])
322 qc.x(attack[1])
323 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[0])
324 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[1])
325 qc.x(attack[1])
326 qc.x(health[0])
327
328
329
330 qc.x(attack)
331 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[0])
332 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[1])
333 qc.x(attack)
334
335
336
337 qc.x(attack[1])
338 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[0])
339 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[1])
340 qc.x(attack[1])
341
342
343
344 qc.x(attack[0])
345 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[0])
346 qc.mcx([health[0], health[1], attack[0], attack[1], is_shop_adjacent[0]], shop_quality[1])
347 qc.x(attack[0])
348
349
350
351
352 #prima di passare al boss, devo vedere se ho il negozio adiacente e ho gia' visto il tesoro.
353 #Questo serve per decidere poi se andare dal boss, in quanto, se ho visto il tesoro e ho
354 #il negozio adiacente, o ci vado ora o non ci vado mai piu'
355 #quindi: se ho visto il tesoro, non ho ancora segnato shop_seen = 1 e il negozio ce l'ho
356 #di fianco, mi segno di averlo visto
357
358 qc.x(left_content[0])
359 qc.x/shop_seen_true[0]
360 qc.mcx([treasure_seen[0], shop_seen_true[0], left_content[0], left_content[1]],
361     shop_seen[0])
362 qc.x/shop_seen_true[0]
363 qc.x(left_content[0])
364
365
366
367 qc.x(center_content[0])
368 qc.x/shop_seen_true[0]
369 qc.mcx([treasure_seen[0], shop_seen_true[0], center_content[0], center_content[1]],
370     shop_seen[0])
371 qc.x/shop_seen_true[0]
372 qc.x(center_content[0])
373
374
375
376 qc.x(right_content[0])
377 qc.x/shop_seen_true[0]
378 qc.mcx([treasure_seen[0], shop_seen_true[0], right_content[0], right_content[1]],
379     shop_seen[0])
380 qc.x/shop_seen_true[0]
381 qc.x(right_content[0])

```

```

382
383
384 #aggiungo questa riga per rimettere a zero il qubit shop_seen_true, cosi' da poterlo
385 #riciclare" piu' tardi. L'uso che ne faccio e' spiagato meglio in fondo a questo circuito,
386 #nella parte dedicata ai nemici
387 qc.cx(shop_seen[0], shop_seen_true[0])
388
389 gate = qc.to_gate()
390 gate.name = "Shop logic"
391 return gate
392
393
394
395 def boss_logic():
396
397     qc = QuantumCircuit(phi, health, attack, treasure_seen, shop_seen, shop_seen_true,
398                         left_content, center_content, right_content, left_seen, center_seen,
399                         right_seen, is_treasure_adjacent, is_shop_adjacent, is_boss_adjacent,
400                         shop_max_quality, boss_max_quality, shop_quality, boss_quality,
401                         oracle_qubit)
402
403     #BOSS: devo capire il valore del boss
404     #vale lo stesso discorso di prima: se non ho il boss adiacente, il suo valore
405     #e' il minimo possibile (0)
406
407     #controlliamo quindi se ce l'abbiamo adiacente
408     qc.mcx([left_content[0], left_content[1]], is_boss_adjacent[0])
409
410     qc.mcx([center_content[0], center_content[1]], is_boss_adjacent[0])
411
412     qc.mcx([right_content[0], right_content[1]], is_boss_adjacent[0])
413
414
415
416
417     #dunque: VALORE DEL BOSS
418     #0 se non ho visto il tesoro (default)
419
420     #7 se ho visto il tesoro E il negozio
421     qc.mcx([treasure_seen[0], shop_seen[0], is_boss_adjacent[0]], boss_quality[0])
422     qc.mcx([treasure_seen[0], shop_seen[0], is_boss_adjacent[0]], boss_quality[1])
423
424
425
426     #3 se ho visto il tesoro, non il negozio e non ho interesse ad andare nel negozio
427     #perche' non mi aumenterebbe le chance di vittoria)
428     qc.x(health)
429     qc.x(shop_seen)
430     qc.mcx([treasure_seen[0], shop_seen[0], health[0], health[1], is_boss_adjacent[0]],
431             boss_quality[1])
432     qc.x(shop_seen)
433     qc.x(health)
434
435
436
437     #1 se ho visto il tesoro, ma prima di andare dal boss, mi conviene passare per il
438     #negozi, dato che aumenterebbe strettamente le mie chance di vittoria.
439     #Succede quando (come prima):
440     #health = 2 and attack = 1
441     #health = 3 and (attack=1 or attack=2)
442     #health = 4 and (attack=1 or attack=2 or attack=3)
443
444     qc.x(health[1])
445     qc.x(attack)
446     qc.x(shop_seen)
447     qc.mcx([treasure_seen[0], shop_seen[0], health[0], health[1], attack[0], attack[1],
448             is_boss_adjacent[0]], boss_quality[0])

```

```

449 qc.x(shop_seen)
450 qc.x(attack)
451 qc.x(health[1])
452
453
454
455 qc.x(health[0])
456 qc.x(attack)
457 qc.x(shop_seen)
458 qc.mcx([treasure_seen[0], shop_seen[0], health[0], health[1], attack[0], attack[1],
459           is_boss_adjacent[0]], boss_quality[0])
460 qc.x(shop_seen)
461 qc.x(attack)
462 qc.x(health[0])
463
464
465
466 qc.x(health[0])
467 qc.x(attack[1])
468 qc.x(shop_seen)
469 qc.mcx([treasure_seen[0], shop_seen[0], health[0], health[1], attack[0], attack[1],
470           is_boss_adjacent[0]], boss_quality[0])
471 qc.x(shop_seen)
472 qc.x(attack[1])
473 qc.x(health[0])
474
475
476
477 qc.x(attack)
478 qc.x(shop_seen)
479 qc.mcx([treasure_seen[0], shop_seen[0], health[0], health[1], attack[0], attack[1],
480           is_boss_adjacent[0]], boss_quality[0])
481 qc.x(shop_seen)
482 qc.x(attack)
483
484
485
486 qc.x(attack[1])
487 qc.x(shop_seen)
488 qc.mcx([treasure_seen[0], shop_seen[0], health[0], health[1], attack[0], attack[1],
489           is_boss_adjacent[0]], boss_quality[0])
490 qc.x(shop_seen)
491 qc.x(attack[1])
492
493
494
495 qc.x(attack[0])
496 qc.x(shop_seen)
497 qc.mcx([treasure_seen[0], shop_seen[0], health[0], health[1], attack[0], attack[1],
498           is_boss_adjacent[0]], boss_quality[0])
499 qc.x(shop_seen)
500 qc.x(attack[0])
501
502
503 gate = qc.to_gate()
504 gate.name = "Boss logic"
505 return gate
506
507
508
509 def shop_vs_boss_logic():
510
511     qc = QuantumCircuit(phi, health, attack, treasure_seen, shop_seen, shop_seen_true,
512                          left_content, center_content, right_content, left_seen, center_seen,
513                          right_seen, is_treasure_adjacent, is_shop_adjacent, is_boss_adjacent,
514                          shop_max_quality, boss_max_quality, shop_quality, boss_quality,
515                          oracle_qubit)

```

```

516
517 #la scelta di andare nel negozio o dal boss dipende dal valore che ho assegnato a queste
518 #stanze e dal fatto che potrei averne di fianco una, l'altra o entrambe (ma in ogni caso
519 #NON avro' di fianco il tesoro, altrimenti andrei la'). Allora distinguo tre casi:
520 #‐ho adiacente solo il negozio
521 #‐ho adiacente solo il boss
522 #‐ho adiacenti entrambi
523 #a seconda di quale caso si verifica e della qualita' che ho assegnato a queste due stanze,
524 #decidero' dove andare
525
526 #caso 1) ho adiacente solo il negozio
527
528 #se non ho il tesoro adiacente...
529 qc.x(is_treasure_adjacent[0])
530 #...e nemmeno il boss...
531 qc.x(is_boss_adjacent[0])
532 #...e ho il negozio a sinistra...
533 qc.x(left_content[0])
534 #...e il negozio vale piu' dei nemici (ovvero vale 6 o 8, che si traduce
535 #in shop_quality[1] = 1)...
536 #...allora ci vado
537 qc.x(phi)
538 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_boss_adjacent[0], left_content[0],
539           left_content[1], shop_quality[1]], oracle_qubit)
540 qc.x(phi)
541 qc.x(left_content[0])
542 qc.x(is_boss_adjacent[0])
543 qc.x(is_treasure_adjacent[0])
544
545
546 #ripetere per il centro e per la destra
547 qc.x(is_treasure_adjacent[0])
548 qc.x(is_boss_adjacent[0])
549 qc.x(center_content[0])
550 qc.x(phi[1])
551 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_boss_adjacent[0], center_content[0],
552           center_content[1], shop_quality[1]], oracle_qubit)
553 qc.x(phi[1])
554 qc.x(center_content[0])
555 qc.x(is_boss_adjacent[0])
556 qc.x(is_treasure_adjacent[0])
557
558
559
560 qc.x(is_treasure_adjacent[0])
561 qc.x(is_boss_adjacent[0])
562 qc.x(right_content[0])
563 qc.x(phi[0])
564 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_boss_adjacent[0], right_content[0],
565           right_content[1], shop_quality[1]], oracle_qubit)
566 qc.x(phi[0])
567 qc.x(right_content[0])
568 qc.x(is_boss_adjacent[0])
569 qc.x(is_treasure_adjacent[0])
570
571
572
573
574 #caso 2) ho adiacente solo il boss
575
576 #se non ho adiacente la stanza del tesoro...
577 qc.x(is_treasure_adjacent[0])
578 #...e nemmeno il negozio...
579 qc.x(is_shop_adjacent[0])
580 #...e ho il boss a sinistra (left_content = 11)...
581 #...e il boss vale piu' dei nemici (vero quando vale 3 o 7, ovvero boss_quality[1] = 1)...
582 #...allora ci vado

```

```

583 qc.x(phi)
584 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_shop_adjacent[0], left_content[0],
585           left_content[1], boss_quality[1]], oracle_qubit)
586 qc.x(phi)
587 qc.x(is_shop_adjacent[0])
588 qc.x(is_treasure_adjacent[0])
589
590
591
592 #ripetere per il centro e per la destra
593 qc.x(is_treasure_adjacent[0])
594 qc.x(is_shop_adjacent[0])
595 qc.x(phi[1])
596 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_shop_adjacent[0], center_content[0],
597           center_content[1], boss_quality[1]], oracle_qubit)
598 qc.x(phi[1])
599 qc.x(is_shop_adjacent[0])
600 qc.x(is_treasure_adjacent[0])
601
602
603
604 qc.x(is_treasure_adjacent[0])
605 qc.x(is_shop_adjacent[0])
606 qc.x(phi[0])
607 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_shop_adjacent[0], right_content[0],
608           right_content[1], boss_quality[1]], oracle_qubit)
609 qc.x(phi[0])
610 qc.x(is_shop_adjacent[0])
611 qc.x(is_treasure_adjacent[0])
612
613
614
615 #caso 3) ho adiacenti entrambi
616
617 #se non ho adiacente il tesoro...
618 qc.x(is_treasure_adjacent[0])
619 #Questa volta, per cambiare, metto l'altra riga che flippa il valore del qubit
620 #is_treasure_adjacent[0] dopo aver compiuto TUTTE le operazioni, tanto deve essere
621 #sempre vero che il tesoro non e' adiacente.
622 #...ma ho adiacenti sia il boss che il negozio (is_shop_adjacent = is_boss_adjacent = 1)...
623 #...allora devo capire chi ha piu' importanza tra loro due e la stanza del nemico
624 #Per farlo, devo analizzare il valore massimo che c'e' tra negozio e boss. Ricordiamo che:
625 #1) il negozio puo' avere come valori di qualita' 8, 6, 0 e -1
626 #2) il boss puo' avere come valori di qualita' 7, 3, 1 e 0
627
628 #Quindi: se il negozio vale 8 e ce l'ho a sinistra, al centro o a destra, ci vado.
629 qc.x(left_content[0])
630 qc.x(phi)
631 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_boss_adjacent[0], left_content[0],
632           left_content[1], shop_quality[0], shop_quality[1]], oracle_qubit)
633 qc.x(phi)
634 qc.x(left_content[0])
635
636
637
638 qc.x(center_content[0])
639 qc.x(phi[1])
640 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_boss_adjacent[0], center_content[0],
641           center_content[1], shop_quality[0], shop_quality[1]], oracle_qubit)
642 qc.x(phi[1])
643 qc.x(center_content[0])
644
645
646
647 qc.x(right_content[0])
648 qc.x(phi[0])
649 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_boss_adjacent[0], right_content[0],

```

```

650     right_content[1], shop_quality[0], shop_quality[1]], oracle_qubit)
651 qc.x(phi[0])
652 qc.x(right_content[0])
653
654
655
656 #Mi segno anche che shop_max_quality = 1, se effettivamente il negozio ha la qualita'
657 #piu' alta possibile
658 qc.mcx([shop_quality[0], shop_quality[1]], shop_max_quality[0])
659
660
661
662
663 #Se il boss vale 7 e il negozio NON vale 8 (ovvero vale meno del boss), vado dal boss
664 qc.x(phi)
665 qc.x(shop_max_quality[0])
666 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_shop_adjacent[0], left_content[0],
667         left_content[1], boss_quality[0], boss_quality[1], shop_max_quality[0]],
668         oracle_qubit)
669 qc.x(shop_max_quality[0])
670 qc.x(phi)
671
672
673
674 qc.x(phi[1])
675 qc.x(shop_max_quality[0])
676 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_shop_adjacent[0], center_content[0],
677         center_content[1], boss_quality[0], boss_quality[1], shop_max_quality[0]],
678         oracle_qubit)
679 qc.x(shop_max_quality[0])
680 qc.x(phi[1])
681
682
683
684 qc.x(phi[0])
685 qc.x(shop_max_quality[0])
686 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_shop_adjacent[0], right_content[0],
687         right_content[1], boss_quality[0], boss_quality[1], shop_max_quality[0]],
688         oracle_qubit)
689 qc.x(shop_max_quality[0])
690 qc.x(phi[0])
691
692
693
694 #Anche qui mi segno se boss_max_quality = 1
695 qc.mcx([boss_quality[0], boss_quality[1]], boss_max_quality[0])
696
697
698
699
700
701 #Ora si applica lo stesso discorso per quando lo shop vale 6 e il boss NON vale 7
702 # (ovvero, vale comunque meno del negozio).
703 #In questo caso vado nel negozio
704 qc.x(phi)
705 qc.x(left_content[0])
706 qc.x(shop_quality[0])
707 qc.x(boss_max_quality[0])
708 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_boss_adjacent[0], left_content[0],
709         left_content[1], shop_quality[0], shop_quality[1], boss_max_quality[0]],
710         oracle_qubit)
711 qc.x(boss_max_quality[0])
712 qc.x(shop_quality[0])
713 qc.x(left_content[0])
714 qc.x(phi)
715
716

```

```

717 qc.x(phi[1])
718 qc.x(center_content[0])
719 qc.x(shop_quality[0])
720 qc.x(boss_max_quality[0])
721 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_boss_adjacent[0], center_content[0],
722         center_content[1], shop_quality[0], shop_quality[1], boss_max_quality[0]],
723         oracle_qubit)
724 qc.x(boss_max_quality[0])
725 qc.x(shop_quality[0])
726 qc.x(center_content[0])
727 qc.x(phi[1])
728
729
730
731
732 qc.x(phi[0])
733 qc.x(right_content[0])
734 qc.x(shop_quality[0])
735 qc.x(boss_max_quality[0])
736 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_boss_adjacent[0], right_content[0],
737         right_content[1], shop_quality[0], shop_quality[1], boss_max_quality[0]],
738         oracle_qubit)
739 qc.x(boss_max_quality[0])
740 qc.x(shop_quality[0])
741 qc.x(right_content[0])
742 qc.x(phi[0])
743
744
745
746
747
748 #Infine, se lo shop non vale ne' 8 ne' 6, e il boss non vale 7 ma 3, allora va bene il boss
749 qc.x(phi)
750 qc.x(shop_quality[1])
751 qc.x(boss_quality[0])
752 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_shop_adjacent[0], left_content[0],
753         left_content[1], shop_quality[1], boss_quality[0], boss_quality[1]], oracle_qubit)
754 qc.x(boss_quality[0])
755 qc.x(shop_quality[1])
756 qc.x(phi)
757
758
759
760 qc.x(phi[1])
761 qc.x(shop_quality[1])
762 qc.x(boss_quality[0])
763 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_shop_adjacent[0], center_content[0],
764         center_content[1], shop_quality[1], boss_quality[0], boss_quality[1]], oracle_qubit)
765 qc.x(boss_quality[0])
766 qc.x(shop_quality[1])
767 qc.x(phi[1])
768
769
770
771 qc.x(phi[0])
772 qc.x(shop_quality[1])
773 qc.x(boss_quality[0])
774 qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], is_shop_adjacent[0], right_content[0],
775         right_content[1], shop_quality[1], boss_quality[0], boss_quality[1]], oracle_qubit)
776 qc.x(boss_quality[0])
777 qc.x(shop_quality[1])
778 qc.x(phi[0])
779
780 qc.x(is_treasure_adjacent[0])
781
782 gate = qc.to_gate()
783 gate.name = "Shop vs boss logic"

```

```

784     return gate
785
786
787 def enemies_logic():
788
789     qc = QuantumCircuit(phi, health, attack, treasure_seen, shop_seen, shop_seen_true,
790                          left_content, center_content, right_content, left_seen, center_seen,
791                          right_seen, is_treasure_adjacent, is_shop_adjacent, is_boss_adjacent,
792                          shop_max_quality, boss_max_quality, shop_quality, boss_quality,
793                          oracle_qubit)
794
795     #NEMICI: quando non ci sono alternative migliori
796
797     #Potrebbe accadere che ho adiacenti tre stanze dei nemici non ancora esplorate (per esempio
798     #all'inizio dell'esplorazione). In un caso del genere seleziono un ordine casuale
799     #per valutare le stanze, perche' se altrimenti ogni stanza potesse fornire un kickback
800     #di fase a phi, otterrei come risultato 11. Quindi solo una di queste deve generare il
801     #kickback di fase. Per sapere quindi quando una stanza con nemici e' stata scelta, mettero'
802     #in pratica una procedura un po' strana ma che permette di risparmiare qubit:
803     #per andare in una stanza, anche shop_seen_true[0] deve essere a 0
804     #se ho scelto di andare in una certa stanza, setto shop_seen_true[0] a 1
805     #se quest'ultimo qubit e' a uno, flippo il valore di shop_quality[1], cosi' che
806     #le prossime stanze contenenti nemici non vengano considerate
807     #rimetto a 0 shop_seen_true[0]
808
809     directions = ["left", "center", "right"]
810     random.shuffle(directions)
811     for d in directions:
812         if d == "left":
813             #La spiego per quando devo andare a sinistra, le altre sono analoghe.
814             #Se non ho il tesoro adiacente...
815             qc.x(is_treasure_adjacent[0])
816             #...e il negozio e il boss valgono meno di me (cioe' il negozio vale -1 o 0
817             #e il boss vale 0 o 1)...
818             qc.x(shop_quality[1])
819             qc.x(boss_quality[1])
820             #...e non ho mai esplorato la stanza a sinistra...
821             qc.x(left_seen[0])
822             #...e a sinistra ci sono nemici...
823             qc.x(left_content)
824             #...allora puo' interessarmi andare a sinistra.
825             qc.x(phi)
826             #aggiungo come controllo anche not(shop_seen_true[0]). Potevo aggiungere
827             #un nuovo qubit del tipo enemies_chosen[0], ma non l'ho fatto perche' cosi'
828             #sono in grado di riciclare un vecchio qubit.
829             #Fondamentalmente, a questo punto del circuito, uso shop_seen_true[0] come segue:
830             #se e' a 0, vuol dire che non ho ancora scelto una stanza con nemici dove andare
831             #se e' a 1, vuol dire che invece l'ho scelta
832             #tanto la scelta di entrare in una stanza con nemici verrà considerata solo quando
833             #non e' possibile procedere in una stanza migliore, e mi sono assicurato di
834             #far si' che, a questo punto del circuito, shop_seen_true[0] sia a 1. Questo
835             #ragionamento vale per tutti e tre i casi
836             qc.x(shop_seen_true[0])
837             qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], shop_quality[1], boss_quality[1],
838                     left_seen[0], left_content[0], left_content[1], shop_seen_true[0]],
839                     oracle_qubit)
840             qc.x(shop_seen_true[0])
841             #se effettivamente ho deciso di andare a sinistra, mi segno questo fatto
842             #mettendo a uno il qubit shop_seen_true[0]
843             qc.mcx([is_treasure_adjacent[0], shop_quality[1], boss_quality[1], left_seen[0],
844                     left_content[0], left_content[1]], shop_seen_true[0])
845             #e cambio il valore di shop_quality[1] cosi' che le prossime stanze contenenti
846             #nemici vengano ignorate
847             qc.cx(shop_seen_true[0], shop_quality[1])
848             #ora rimetto a posto shop_seen_true[0]
849             qc.x(shop_quality[1])
850             qc.mcx([is_treasure_adjacent[0], shop_quality[1], boss_quality[1], left_seen[0],

```

```

851     qc.x(left_content[0], left_content[1]), shop_seen_true[0])
852     qc.x(shop_quality[1])
853     qc.x(phi)
854     qc.x(left_content)
855     qc.x(left_seen[0])
856     qc.x(boss_quality[1])
857     qc.x(shop_quality[1])
858     qc.x(is_treasure_adjacent[0])
859
860
861 if d == "center":
862     #vado al centro
863     qc.x(is_treasure_adjacent[0])
864     qc.x(shop_quality[1])
865     qc.x(boss_quality[1])
866     qc.x(center_seen[0])
867     qc.x(center_content)
868     qc.x(phi[1])
869     qc.x(shop_seen_true[0])
870     qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], shop_quality[1], boss_quality[1],
871             center_seen[0], center_content[0], center_content[1], shop_seen_true[0]],
872             oracle_qubit)
873     qc.x(shop_seen_true[0])
874     qc.mcx([is_treasure_adjacent[0], shop_quality[1], boss_quality[1], center_seen[0],
875             center_content[0], center_content[1], shop_seen_true[0]])
876     qc.cx(shop_seen_true[0], shop_quality[1])
877     qc.x(shop_quality[1])
878     qc.mcx([is_treasure_adjacent[0], shop_quality[1], boss_quality[1], center_seen[0],
879             center_content[0], center_content[1], shop_seen_true[0]])
880     qc.x(shop_quality[1])
881     qc.x(phi[1])
882     qc.x(center_content)
883     qc.x(center_seen[0])
884     qc.x(boss_quality[1])
885     qc.x(shop_quality[1])
886     qc.x(is_treasure_adjacent[0])
887
888
889 if d == "right":
890     #vado a destra
891     qc.x(is_treasure_adjacent[0])
892     qc.x(shop_quality[1])
893     qc.x(boss_quality[1])
894     qc.x(right_seen[0])
895     qc.x(right_content)
896     qc.x(phi[0])
897     qc.x(shop_seen_true[0])
898     qc.mcx([phi[0], phi[1], is_treasure_adjacent[0], shop_quality[1], boss_quality[1],
899             right_seen[0], right_content[0], right_content[1], shop_seen_true[0]],
900             oracle_qubit)
901     qc.x(shop_seen_true[0])
902     qc.mcx([is_treasure_adjacent[0], shop_quality[1], boss_quality[1], right_seen[0],
903             right_content[0], right_content[1], shop_seen_true[0]])
904     qc.cx(shop_seen_true[0], shop_quality[1])
905     qc.x(shop_quality[1])
906     qc.mcx([is_treasure_adjacent[0], shop_quality[1], boss_quality[1], right_seen[0],
907             right_content[0], right_content[1], shop_seen_true[0]])
908     qc.x(shop_quality[1])
909     qc.x(phi[0])
910     qc.x(right_content)
911     qc.x(right_seen[0])
912     qc.x(boss_quality[1])
913     qc.x(shop_quality[1])
914     qc.x(is_treasure_adjacent[0])
915
916
917

```

```

918
919     gate = qc.to_gate()
920     gate.name = "Enemies logic"
921     return gate
922
923
924
925 #definiamo adesso l'oracolo, cioe' la funzione di transizione che deve indicare all'agente
926 #in quale stanza muoversi
927 def oracle_function(qc):
928
929     qc.append(treasure_logic(), [phi[0], phi[1], health[0], health[1], attack[0], attack[1],
930                                 treasure_seen, shop_seen, shop_seen_true, left_content[0], left_content[1],
931                                 center_content[0], center_content[1], right_content[0], right_content[1],
932                                 left_seen, center_seen, right_seen, is_treasure_adjacent, is_shop_adjacent,
933                                 is_boss_adjacent, shop_max_quality, boss_max_quality, shop_quality[0],
934                                 shop_quality[1], boss_quality[0], boss_quality[1], oracle_qubit])
935
936
937
938     qc.append(shop_logic(), [phi[0], phi[1], health[0], health[1], attack[0], attack[1],
939                                 treasure_seen, shop_seen, shop_seen_true, left_content[0], left_content[1],
940                                 center_content[0], center_content[1], right_content[0], right_content[1],
941                                 left_seen, center_seen, right_seen, is_treasure_adjacent, is_shop_adjacent,
942                                 is_boss_adjacent, shop_max_quality, boss_max_quality, shop_quality[0],
943                                 shop_quality[1], boss_quality[0], boss_quality[1], oracle_qubit])
944
945
946
947     qc.append(boss_logic(), [phi[0], phi[1], health[0], health[1], attack[0], attack[1],
948                                 treasure_seen, shop_seen, shop_seen_true, left_content[0], left_content[1],
949                                 center_content[0], center_content[1], right_content[0], right_content[1],
950                                 left_seen, center_seen, right_seen, is_treasure_adjacent, is_shop_adjacent,
951                                 is_boss_adjacent, shop_max_quality, boss_max_quality, shop_quality[0],
952                                 shop_quality[1], boss_quality[0], boss_quality[1], oracle_qubit])
953
954
955
956     qc.append(shop_vs_boss_logic(), [phi[0], phi[1], health[0], health[1], attack[0], attack[1],
957                                 treasure_seen, shop_seen, shop_seen_true, left_content[0], left_content[1],
958                                 center_content[0], center_content[1], right_content[0], right_content[1],
959                                 left_seen, center_seen, right_seen, is_treasure_adjacent, is_shop_adjacent,
960                                 is_boss_adjacent, shop_max_quality, boss_max_quality, shop_quality[0],
961                                 shop_quality[1], boss_quality[0], boss_quality[1], oracle_qubit])
962
963
964
965     qc.append(enemies_logic(), [phi[0], phi[1], health[0], health[1], attack[0], attack[1],
966                                 treasure_seen, shop_seen, shop_seen_true, left_content[0], left_content[1],
967                                 center_content[0], center_content[1], right_content[0], right_content[1],
968                                 left_seen, center_seen, right_seen, is_treasure_adjacent, is_shop_adjacent,
969                                 is_boss_adjacent, shop_max_quality, boss_max_quality, shop_quality[0],
970                                 shop_quality[1], boss_quality[0], boss_quality[1], oracle_qubit])
971
972
973
974
975
976 #e ora definisco la funzione diffuser
977 def diffuser():
978
979     qc = QuantumCircuit(phi, health, attack, treasure_seen, shop_seen, shop_seen_true,
980                         left_content, center_content, right_content, left_seen, center_seen,
981                         right_seen, is_treasure_adjacent, is_shop_adjacent, is_boss_adjacent,
982                         shop_max_quality, boss_max_quality, shop_quality, boss_quality,
983                         oracle_qubit)
984

```

```

985     qc.h(phi)
986     qc.x(phi)
987     qc.mcx(phi, oracle_qubit)
988     qc.x(phi)
989     qc.h(phi)
990
991     gate = qc.to_gate()
992     gate.name = "Diffuser"
993     return gate
994
995
996
997
998
999 #-----VARIABILI PER CONSERVARE L'ESECUZIONE DI UNA ESPLORAZIONE-----
1000 all_explorations_path = []
1001
1002 class Exploration:
1003     def __init__(self):
1004         self.rooms_explored = []
1005         self.health = -1
1006         self.attack = -1
1007         self.outcome = None
1008
1009     def print_exploration(self):
1010         print("In this exploration, the rooms explored were: \n" + str(self.rooms_explored) +
1011             "\n and the stats are: \n " + "health = " + str(self.health) + "\n attack = "
1012             + str(self.attack) + "\n and the outcome was: " + str(self.outcome))
1013
1014 class ExplorationOutcome():
1015     BOSSDEFEATED = "Boss defeated!"
1016     BOSSKILLEDYOU = "The Boss killed you!"
1017     YOUDIED = "You died while exploring the dungeon!"
1018
1019
1020 #-----LOGICA DEL DUNGEON-----
1021 class RoomContent():
1022     ENEMIES = "Enemies"
1023     TREASURE = "Treasure"
1024     SHOP = "Shop"
1025     BOSS = "Boss"
1026
1027 class Room:
1028
1029     #ogni stanza avra' un indice (tre bit rappresentati come stringa) e un contenuto
1030     #(rappresentato anch'esso da una stringa). Altre cose che mi interessa sapere di una stanza
1031     #sono se l'ho gia' visitata e qual e' la sua percentuale qualitativa (quest'ultimo attributo
1032     #sara' l'avventuriero a settarlo)
1033     def __init__(self, index):
1034         self.index = index
1035         self.adjacent_rooms = getAdjacentRooms(self.index)
1036         self.content = None
1037         self.visited = False
1038         self.quality = -10
1039
1040     def room_to_string(self):
1041         print("Stanza = " + self.index + ", content = " + self.content)
1042
1043
1044 class Agent:
1045
1046     DEBUG = False
1047
1048     #definisco il mio agente in termini di:
1049     #la stanza in cui si trova
1050     #se ha visto il tesoro
1051     #se ha visto lo shop

```

```

1052 #-i suoi punti salute
1053 #-i suoi punti attacco
1054
1055 #la qualita' della stanza puo' essere intesa sia come la probabilita' di non prendere danno
1056 #quando si parla di stanze coi nemici, sia come la probabilita' di vincere contro il boss
1057 #se si tratta della stanza del boss, e sara' sempre 100% quando si tratta del tesoro e puo'
1058 #variare per il negozio. Sia il negozio che il boss usano la stessa formula per indicare
1059 #la loro qualita', che ha come parametri i punti vita e i punti attacco
1060 def __init__(self, current_room, dungeon):
1061     self.current_room = current_room
1062     self.dungeon = dungeon
1063     self.treasure_seen = False
1064     self.shop_seen = False
1065     self.health_points = 4
1066     self.attack_points = 1
1067
1068
1069 #definisco una funzione che iteri nel dizionario e mi restituisca la chiave di valore
1070 #piu' alto (serve per capire dove dovrei andare secondo l'algoritmo di Grover).
1071 #Questa funzione serve anche ad aggiornare il valore shop_seen
1072 def choice_quantum_movement(self, counts, dungeon, current_room_index):
1073     choice = ""
1074     max = -1
1075     for k in counts.keys():
1076         if counts[k] > max:
1077             max = counts[k]
1078             choice = k
1079
1080     shop_seen_string = choice[0]
1081     if shop_seen_string == "1":
1082         if Agent.DEBUG:
1083             print("From the measure, I can tell i saw the shop!")
1084         self.shop_seen = True
1085
1086     next_room_string = choice[2] + choice[3]
1087
1088     #next_room_string mi dice semplicemente in che stanza ho scelto di andare:
1089     #00 = sinistra (devo cambiare il primo bit indice)
1090     #01 = centro (devo cambiare il secondo bit indice)
1091     #10 = destra (devo cambiare il terzo bit indice)
1092
1093     next_room_index = ""
1094     if next_room_string == "00":
1095         opposite_bit = opposite(current_room_index[0])
1096         next_room_index = opposite_bit + current_room_index[1] + current_room_index[2]
1097     elif next_room_string == "01":
1098         opposite_bit = opposite(current_room_index[1])
1099         next_room_index = current_room_index[0] + opposite_bit + current_room_index[2]
1100     elif next_room_string == "10":
1101         opposite_bit = opposite(current_room_index[2])
1102         next_room_index = current_room_index[0] + current_room_index[1] + opposite_bit
1103     else:
1104         if Agent.DEBUG:
1105             print("TOO MUCH NOISE IN SELECTING NEXT ROOM! Retry this room")
1106         next_room_index = current_room_index
1107
1108     next_room = dungeon[next_room_index]
1109
1110     return next_room
1111
1112
1113
1114
1115 #funzione che implementa l'intera esplorazione del mio agente
1116 def explore(self):
1117
1118     #eo sara' l'oggetto che maniene il riepilogo di questa esplorazione

```

```

1119     eo = Exploration()
1120
1121     #finche' non ho trovato il boss
1122     while not self.current_room.content == RoomContent.BOSS:
1123
1124         eo.rooms_explored.append(self.current_room.index)
1125
1126         if Agent.DEBUG:
1127             print("+++++++\n")
1128             print("Stanza = " + str(self.current_room.index))
1129             print("Contenuto = " + str(self.current_room.content))
1130             print("Vista = " + str(self.current_room.visited))
1131
1132         #-----PARTE IN CUI APPLICO GLI EFFETTI DELLA STANZA IN CUI SONO APPENA ENTRATO-----
1133
1134         #se sono entrato in una stanza con nemici, lancio una monetina e forse prendo danno
1135         if (self.current_room.content == RoomContent.ENEMIES and not
1136             (self.current_room.index == "000" and self.current_room.visited == False)):
1137             #switch del danno e poi scegli
1138             l = []
1139             r = 1
1140             if self.attack_points == 1:
1141                 for i in range(25):
1142                     l.append(1)
1143                     for i in range(75):
1144                         l.append(0)
1145                         r = random.choice(l)
1146             if self.attack_points == 2:
1147                 for i in range(50):
1148                     l.append(1)
1149                     for i in range(50):
1150                         l.append(0)
1151                         r = random.choice(l)
1152             if self.attack_points == 3:
1153                 for i in range(75):
1154                     l.append(1)
1155                     for i in range(25):
1156                         l.append(0)
1157                         r = random.choice(l)
1158             if self.attack_points == 4:
1159                 r = 1
1160
1161             if r == 0:
1162                 self.health_points -= 1
1163                 if Agent.DEBUG:
1164                     print("Damage taken! Argh!")
1165                 elif Agent.DEBUG:
1166                     print("No damage taken! Phew!")
1167
1168         #se sono entrato nella stanza del tesoro o nel negozio, me lo segno, e aggiorno
1169         #il contenuto di queste stanze con dei nemici
1170         if self.current_room.content == RoomContent.TREASURE:
1171             self.treasure_seen = True
1172
1173         #ottengo il potenziamento casuale
1174         values = [1, 2]
1175         power_up_types = ["health", "attack"]
1176         v = random.choice(values)
1177         p = random.choice(power_up_types)
1178         if Agent.DEBUG:
1179             print("Power up found = " + str(v) + " " + p)
1180         if v == 1 and p == "health" and self.health_points <= 3:
1181             self.health_points += 1
1182             if Agent.DEBUG:
1183                 print("+1 h")
1184         if v == 2 and p == "health" and self.health_points <= 3:
1185             if self.health_points <=2:

```

```

1186         if Agent.DEBUG:
1187             print("+2 h")
1188             self.health_points += 2
1189         else:
1190             if Agent.DEBUG:
1191                 print("+2 h, but really +1 h")
1192                 self.health_points += 1
1193         if v == 1 and p == "attack":
1194             if Agent.DEBUG:
1195                 print("+1 a")
1196                 self.attack_points += 1
1197         if v == 2 and p == "attack":
1198             if Agent.DEBUG:
1199                 print("+2 a")
1200                 self.attack_points += 2
1201
1202         self.current_room.content = RoomContent.ENEMIES
1203
1204     if self.current_room.content == RoomContent.SHOP:
1205         self.shop_seen = True
1206
1207         #tolgo un punto vita e ne aggiungo uno di danno
1208         self.health_points -= 1
1209         self.attack_points += 1
1210
1211         self.current_room.content = RoomContent.ENEMIES
1212
1213     if Agent.DEBUG:
1214         print("Health = " + str(self.health_points))
1215         print("Attack = " + str(self.attack_points))
1216
1217 #-----STANZA ESPLORATA, ORA DECIDO SE SONO MORTO O MENO. SE NON
1218 #SONO MORTO, MI SEGNO CHE HO ESPLORATO QUESTA STANZA-----
1219
1220     #se un nemico mi ha ucciso, esco dal ciclo
1221     if self.health_points == 0:
1222         break
1223
1224     #segno che mi sono visto la stanza corrente
1225     self.current_room.visited = True
1226
1227 #-----ADESSO C'e' LA FUNZIONE DI TRANSIZIONE, OVVERO LA LOGICA
1228 #CHE PORTA A SCEGLIERE LA PROSSIMA STANZA-----
1229
1230     #considero le stanze adiacenti
1231     adj_r_index = self.current_room.adjacent_rooms
1232     adjacent_rooms = []
1233     for i in adj_r_index:
1234         adjacent_rooms.append(dungeon[i])
1235
1236
1237
1238
1239     #A causa del rumore il personaggio potrebbe non riuscire a prendere una decisione
1240     #circa la prossima stanza da esplorare (ovvero la misura sarebbe 11, che non
1241     #corrisponde a nessuna stanza adiacente). Finche' questo e' vero, ri-eseguiamo
1242     #il circuito
1243
1244     new_room_chosen = False
1245     while not new_room_chosen:
1246         #Parte la parte (gioco di parole) quantistica.
1247         #costruisco il circuito che mi serve per fare i calcoli
1248         qc_curr = QuantumCircuit(phi, health, attack, treasure_seen, shop_seen,
1249             shop_seen_true, left_content, center_content, right_content, left_seen,
1250             center_seen, right_seen, is_treasure_adjacent, is_shop_adjacent,
1251             is_boss_adjacent, shop_max_quality, boss_max_quality, shop_quality,
1252             boss_quality, oracle_qubit, measure_bits, shop_bit)

```

```

1253
1254     #lo inizializzo
1255     initialize_dungeon_circuit(qc_curr, self, adjacent_rooms)
1256
1257     #applico l'oracolo di grover
1258     oracle_function(qc_curr)
1259
1260     #applico il diffuser
1261     qc_curr.append(diffuser(), [phi[0], phi[1], health[0], health[1], attack[0],
1262                                 attack[1], treasure_seen, shop_seen, shop_seen_true, left_content[0],
1263                                 left_content[1], center_content[0], center_content[1], right_content[0],
1264                                 right_content[1], left_seen, center_seen, right_seen, is_treasure_adjacent,
1265                                 is_shop_adjacent, is_boss_adjacent, shop_max_quality, boss_max_quality,
1266                                 shop_quality[0], shop_quality[1], boss_quality[0], boss_quality[1],
1267                                 oracle_qubit])
1268
1269     #metto le misure
1270     qc_curr.measure(phi, measure_bits)
1271     qc_curr.measure(shop_seen, shop_bit)
1272
1273     #aggiungo il circuito all'array di circuiti
1274     qc_list.append(qc_curr)
1275
1276     #eseguo il circuito
1277     simulator = Aer.get_backend('qasm_simulator')
1278     result = execute(qc_curr, backend = simulator).result()
1279     counts = result.get_counts(qc_curr)
1280
1281     #aggiungo i counts di questa esecuzione all'array di counts
1282     counts_list.append(counts)
1283
1284     #scelgo dove andare
1285     next_room = self.choice_quantum_movement(counts, self.dungeon,
1286                                              self.current_room.index)
1287
1288     if not next_room.index == self.current_room.index:
1289         new_room_chosen = True
1290         if Agent.DEBUG:
1291             print("I chose a new room! I'm in room " + str(self.current_room.index) +
1292                  ", but I'm going to room " + str(next_room.index))
1293     else:
1294         if Agent.DEBUG:
1295             print("I didn't choose a new room! I'm gonna retry...")
1296
1297     self.current_room = next_room
1298
1299
1300
1301
1302     outcome = None
1303     if self.health_points == 0:
1304         outcome = ExplorationOutcome.YOUDIED
1305         if Agent.DEBUG:
1306             print("Sei morto!")
1307     else:
1308         probability_of_victory = 5 * (1 + self.health_points) * self.attack_points
1309         if Agent.DEBUG:
1310             print("La tua probabilita' di vittoria, con " + str(self.health_points) +
1311                  " punti vita e " + str(self.attack_points) + " punti attacco, e' del " +
1312                  str(probability_of_victory) + "%")
1313     l = []
1314     for i in range(probability_of_victory):
1315         l.append(1)
1316     for i in range(100 - probability_of_victory):
1317         l.append(0)
1318     r = random.choice(l)
1319     if r == 1:

```



```

1387
1388 list_bits = []
1389
1390 for i in reversed(range(number_of_bits)):
1391     current_pow_of_two = pow(2, i)
1392     if current_pow_of_two <= num:
1393         list_bits.append(1)
1394         num -= current_pow_of_two
1395     else:
1396         list_bits.append(0)
1397
1398 res = ""
1399 for i in list_bits:
1400     res += str(i)
1401
1402 return res
1403
1404
1405
1406
1407 dungeon = {}
1408 def full_exploration():
1409     rooms = []
1410     for i in range(8):
1411         rooms.append(Room(intToBinary(i, 3)))
1412
1413 #la prima stanza contiene sempre nemici
1414 rooms[0].content = RoomContent.ENEMIES
1415 rooms[0].visited = False
1416
1417 #questo e' il dungeon di prova usato nella Tesi
1418 rooms[1].content = RoomContent.ENEMIES
1419 rooms[1].visited = False
1420 rooms[2].content = RoomContent.ENEMIES
1421 rooms[2].visited = False
1422 rooms[3].content = RoomContent.BOSS
1423 rooms[3].visited = False
1424 rooms[4].content = RoomContent.ENEMIES
1425 rooms[4].visited = False
1426 rooms[5].content = RoomContent.SHOP
1427 rooms[5].visited = False
1428 rooms[6].content = RoomContent.TREASURE
1429 rooms[6].visited = False
1430 rooms[7].content = RoomContent.ENEMIES
1431 rooms[7].visited = False
1432
1433
1434 if Agent.DEBUG:
1435     for i in rooms:
1436         print("contenuto della stanza " + str(i.index) + " = " + str(i.content) +
1437             ", le sue stanze adiacenti sono " + str(i.adjacent_rooms))
1438
1439 #metto adesso le stanze in un dizionario cosi' che per il mio agente sia piu' semplice
1440 #vederne il contenuto
1441
1442 for i in rooms:
1443     dungeon[i.index] = i
1444
1445 if Agent.DEBUG:
1446     for i in rooms:
1447         i.room_to_string()
1448
1449
1450 a = Agent(dungeon["000"], dungeon)
1451 a.explore()
1452
1453

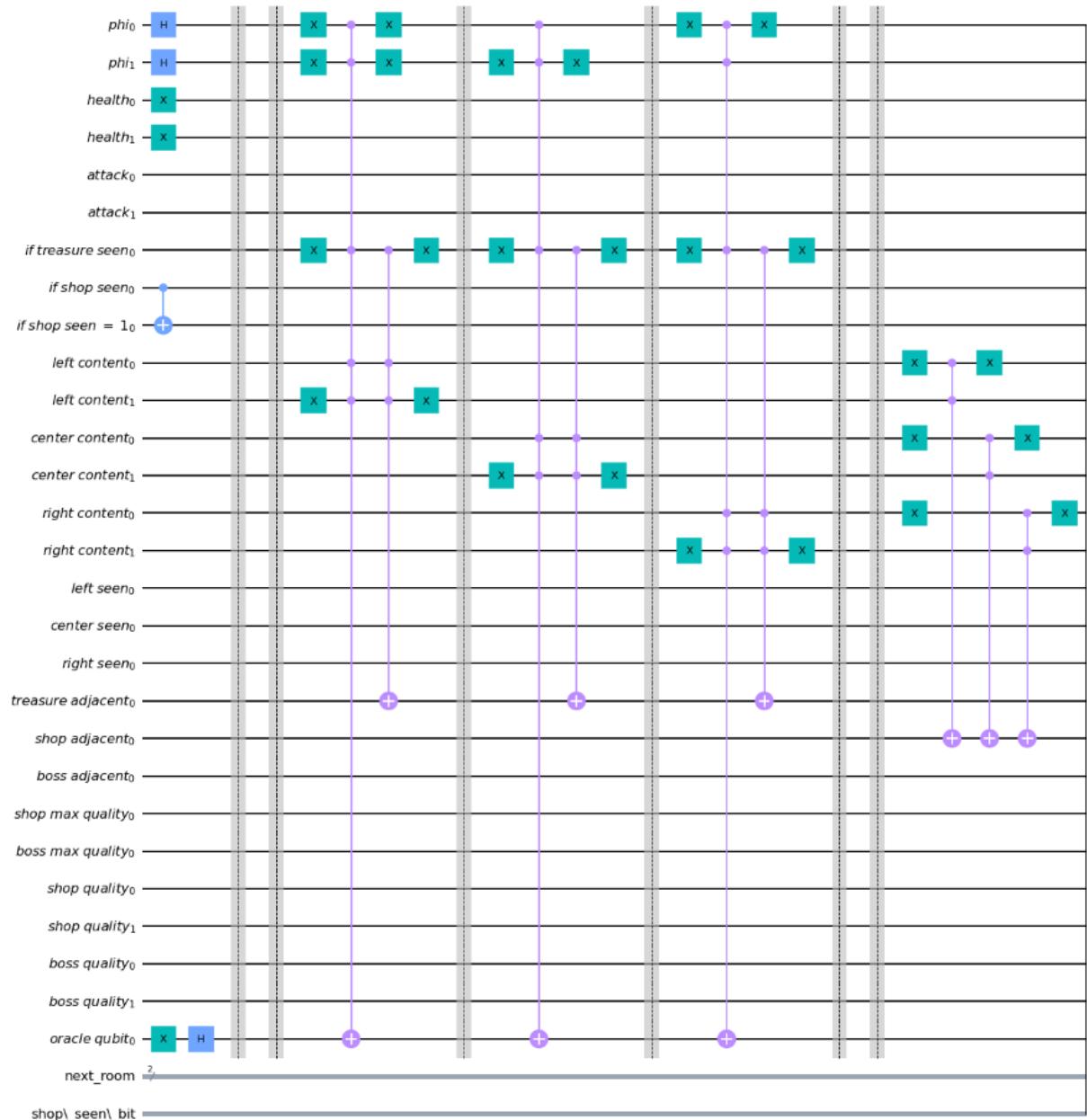
```

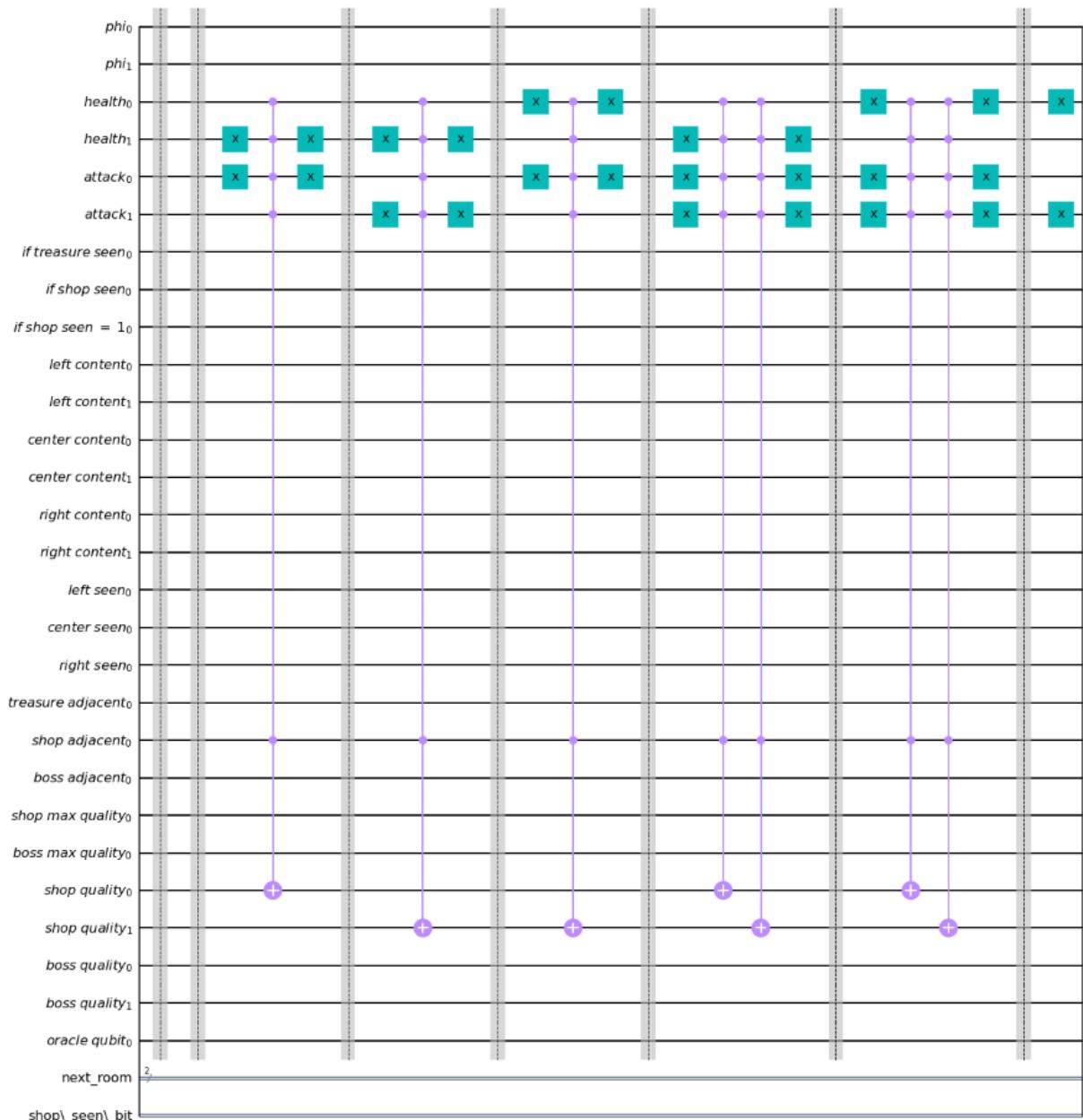
```

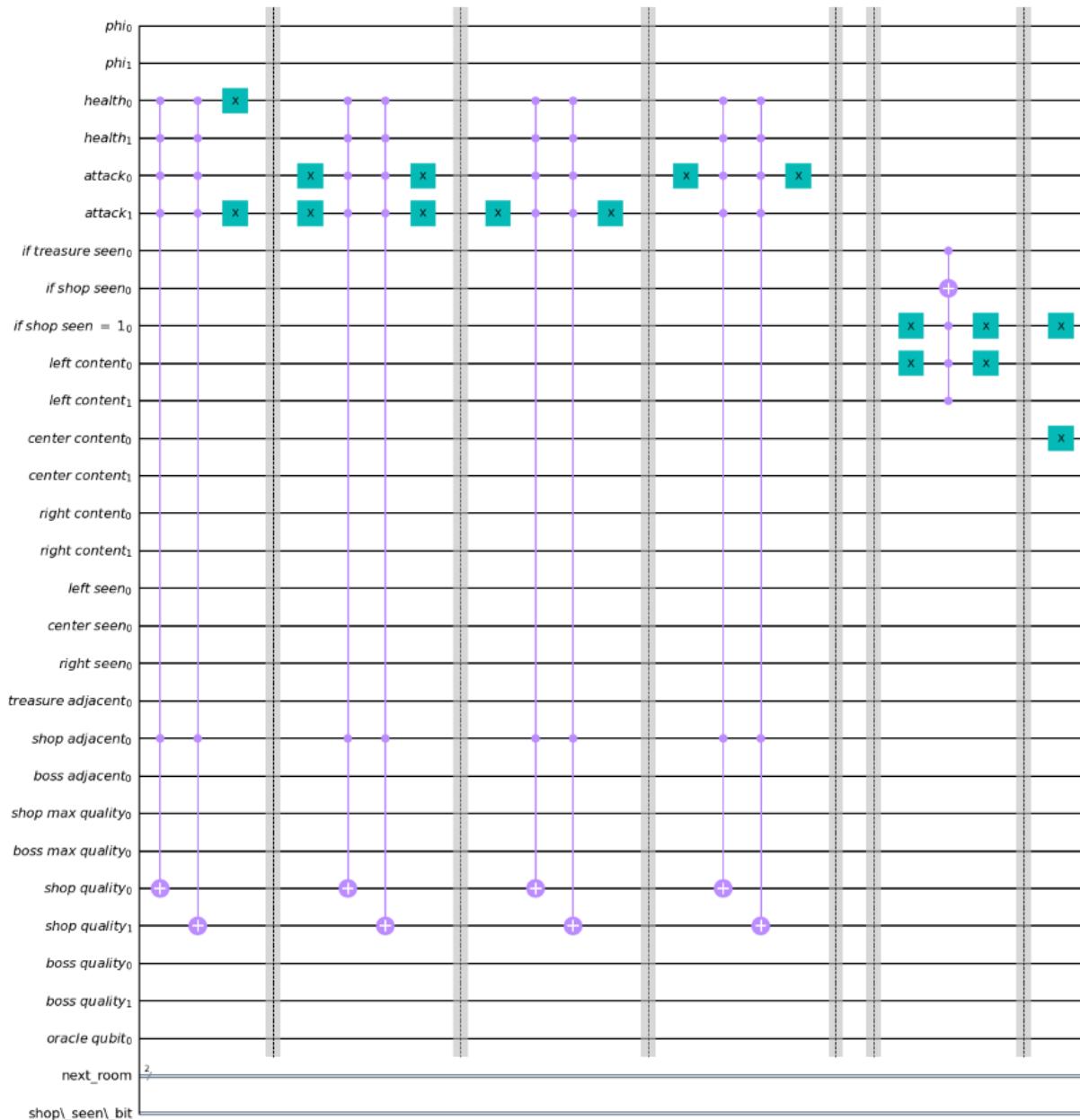
1454
1455 #per la tesi ho fatto anche qui 100 esplorazioni, ma ho dovuto lasciare il computer
1456 #acceso tutta la notte. Ora lo metto a uno cosi' che se uno volesse copia-incollare
1457 #questo codice per eseguirlo non dovrebbe aspettare 10 ore prima che termini, ma
1458 #qualche minuto.
1459 for k in range(1):
1460     full_exploration()
1461
1462
1463
1464 #giusto per vedere qualcosa
1465 qc_list[0].draw('mpl')
1466 plot_histogram(counts_list[0])
1467
1468
1469
1470 #settando a true questa variabile e' possibile visualizzare il contenuto di una delle liste
1471 #di percorsi con cui abbiamo lavorato finora.
1472 visualize_a_list = True
1473 if visualize_a_list:
1474     #e' possibile modificare la lista da assegnare a f_list per vedere
1475     #un diverso tipo di output.
1476     #Le liste possibili includono:
1477     #all_explorations_path
1478     #explorations_path_boss_reached
1479     #explorations_path_high_stats
1480     #explorations_path_shortest
1481
1482     f_list = all_explorations_path
1483     f = open("dungeon_summary.txt", "w")
1484     for i in range(len(f_list)):
1485         if i < 10:
1486             f.write("00")
1487         elif i>=10 and i<100:
1488             f.write("0")
1489         f.write(str(i) + "    exploration = " + str(f_list[i].rooms_explored))
1490
1491         #assumo che un agente non possa esplorare piu' di 20 stanze
1492         blank_spaces = 20 - len(f_list[i].rooms_explored)
1493         for j in range(blank_spaces):
1494             f.write("        ")
1495
1496         f.write(", h = " + str(f_list[i].health) + ", a = " + str(f_list[i].attack) +
1497             ", outcome = " + str(f_list[i].outcome) + "\n")
1498     f.close()
1499
1500
1501     f = open("dungeon_summary.txt", "r")
1502     print(f.read())

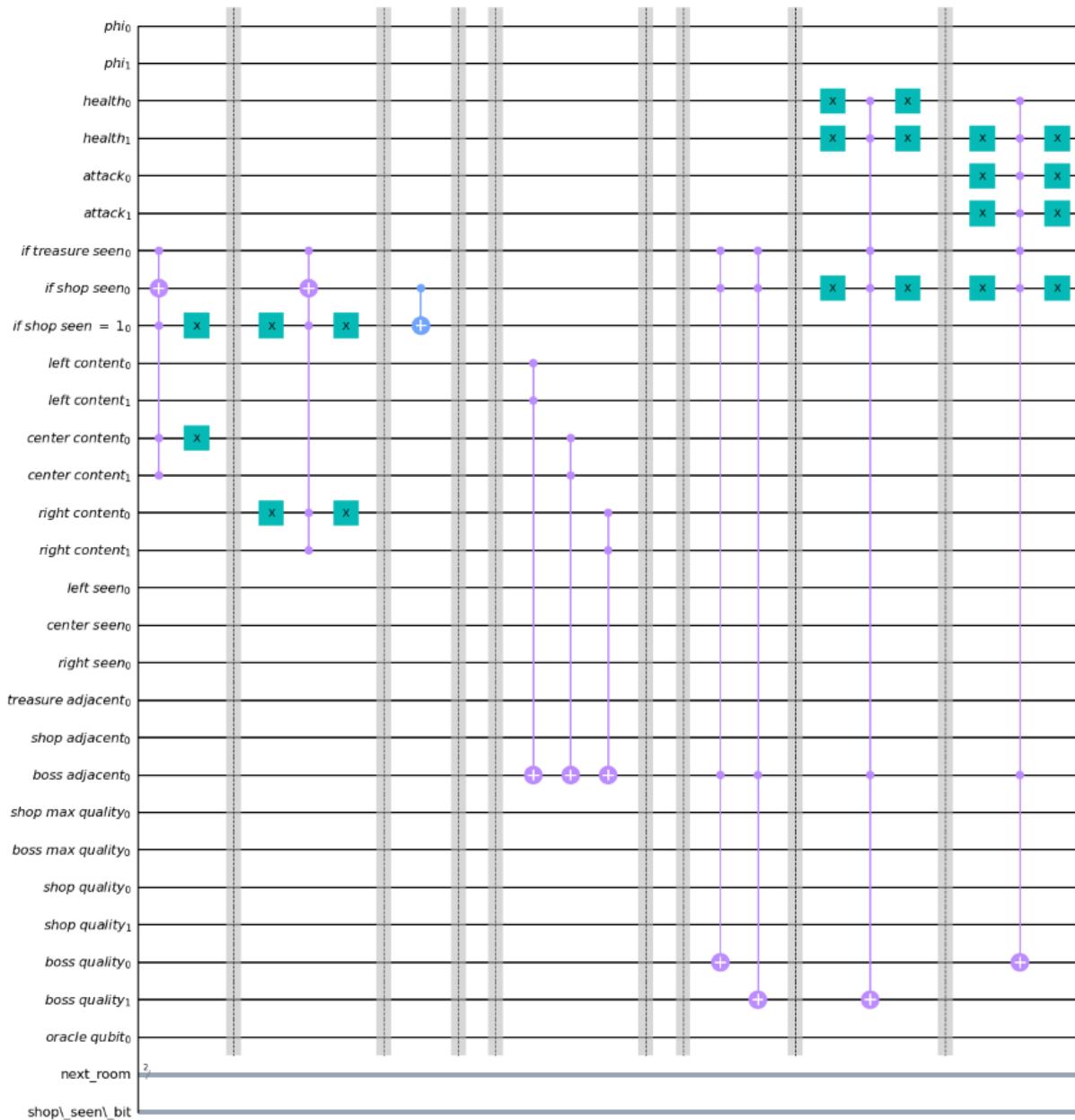
```

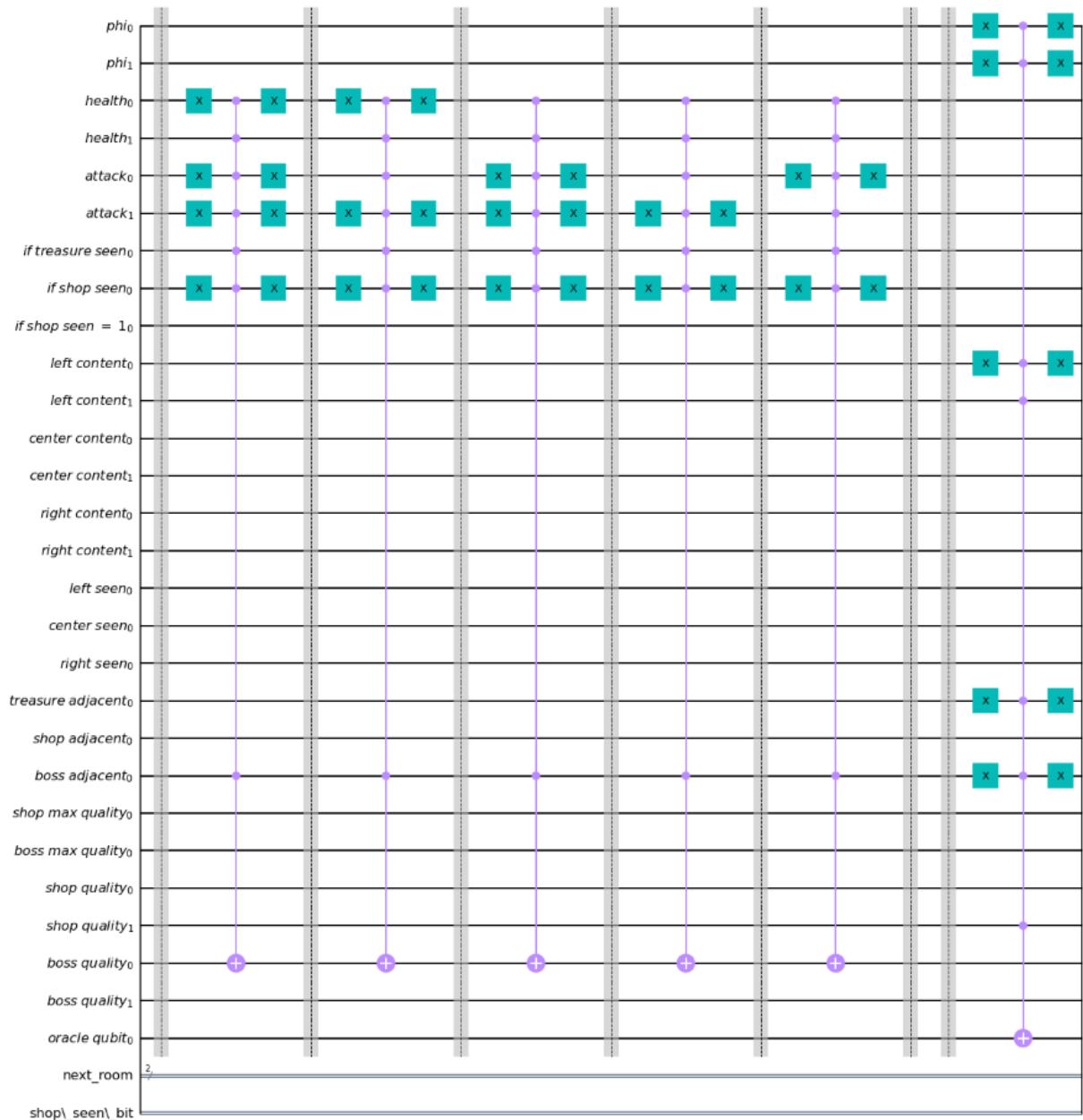
Di seguito il circuito completo generato dal codice dell'appendice C, ovvero quello relativo all'esplo-
razione mediante Grover del labirinto a otto stanze. In particolare il comando che lo stampa è quello alla
riga 1465.

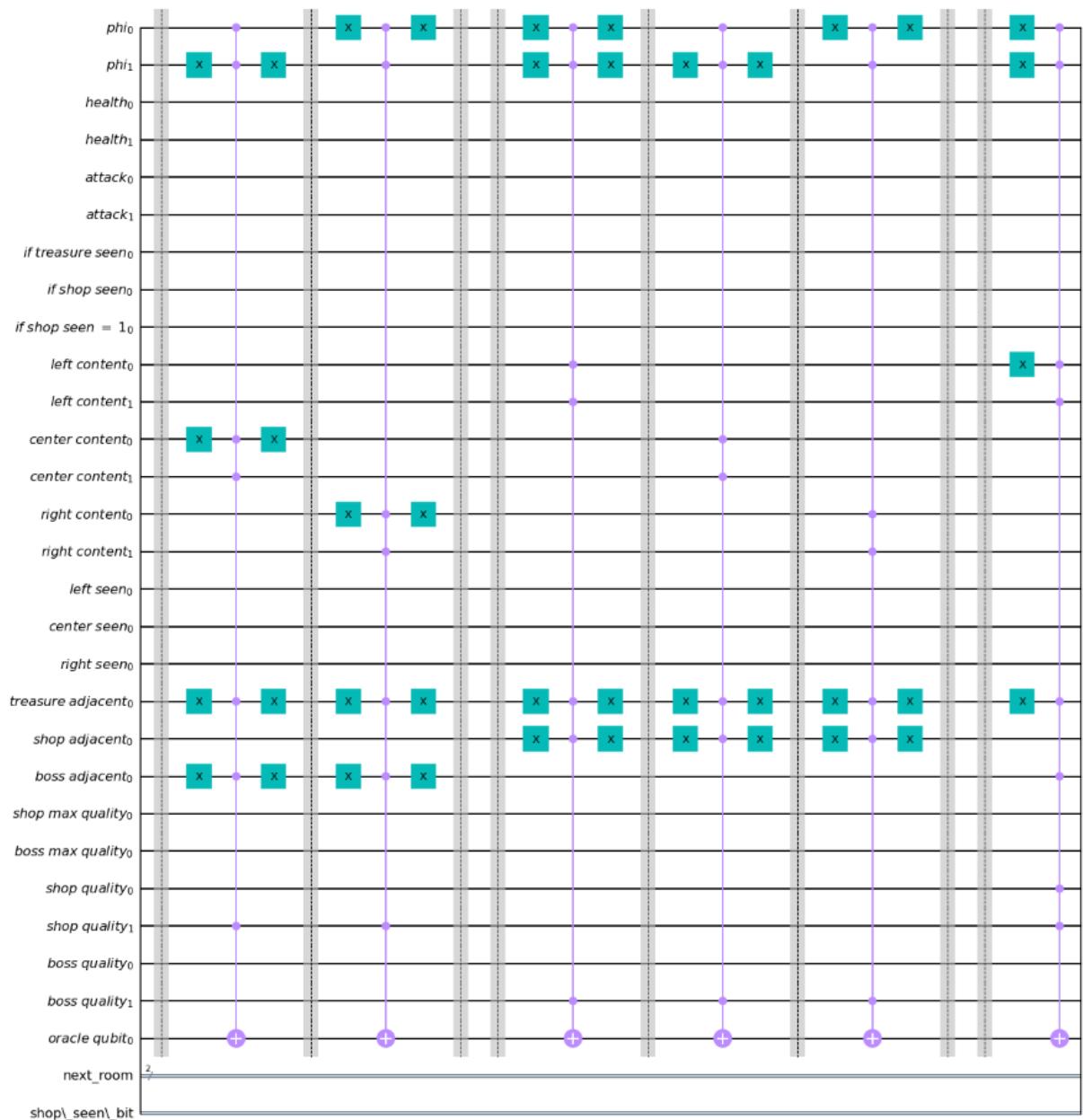


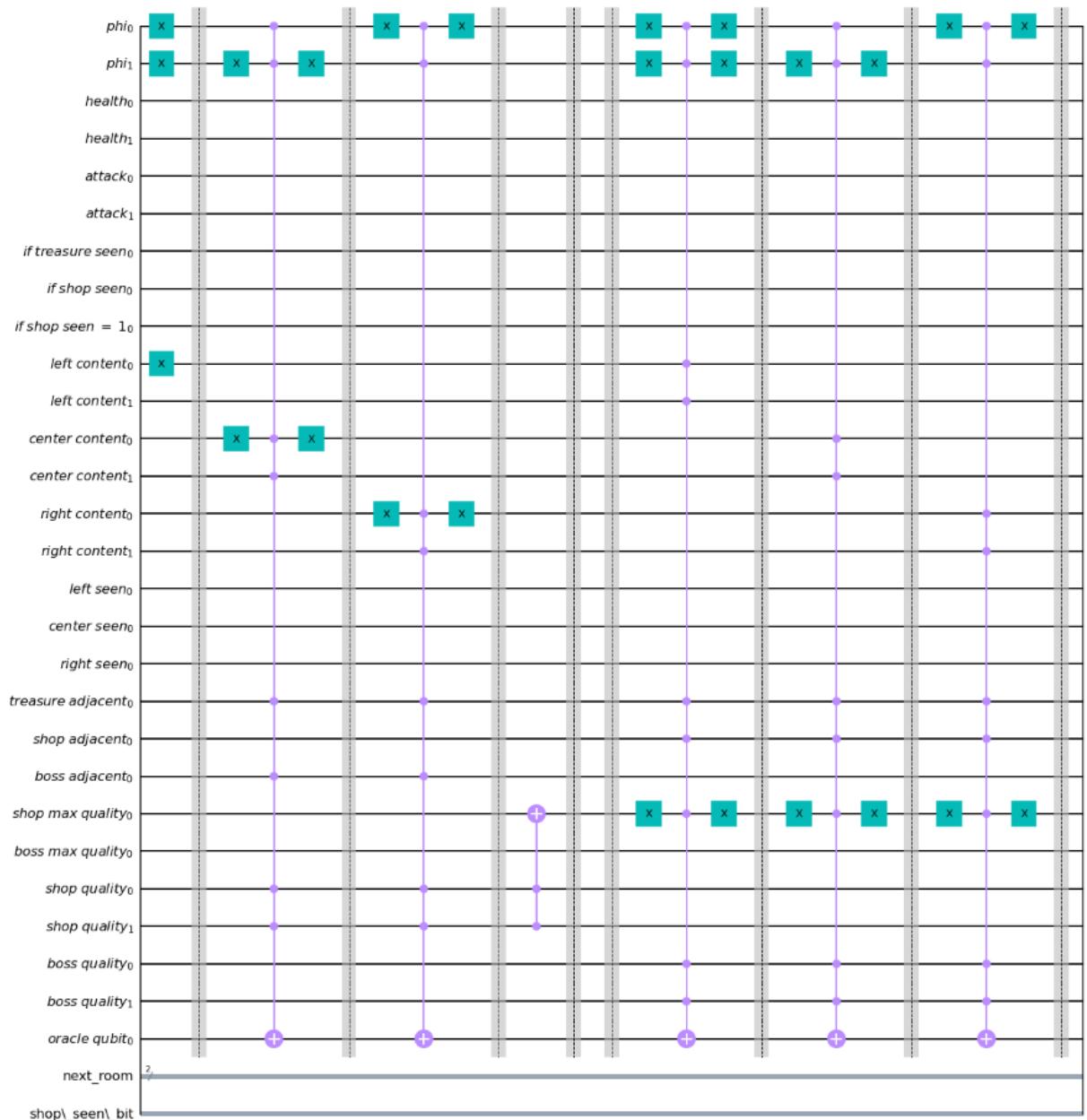


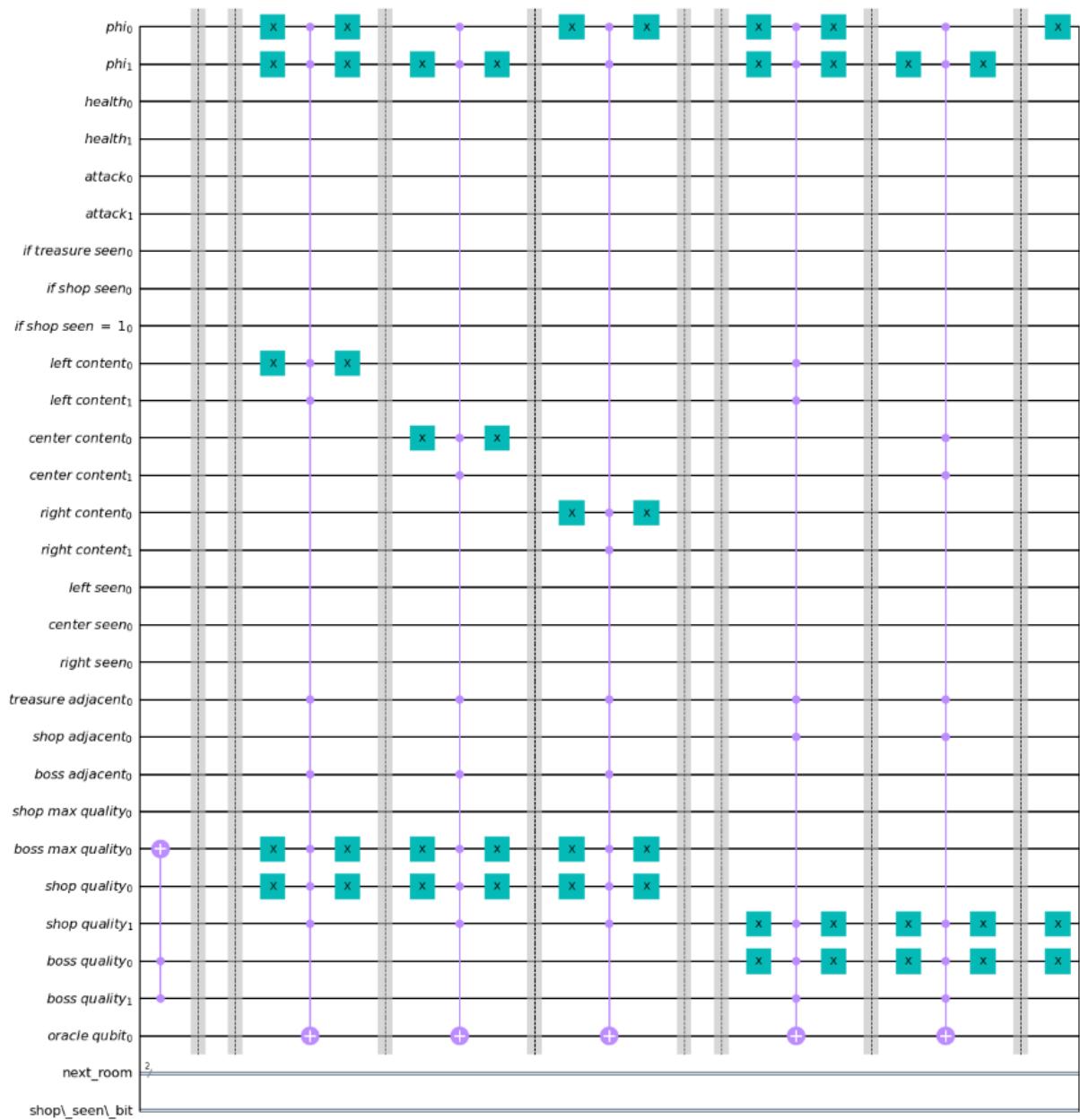


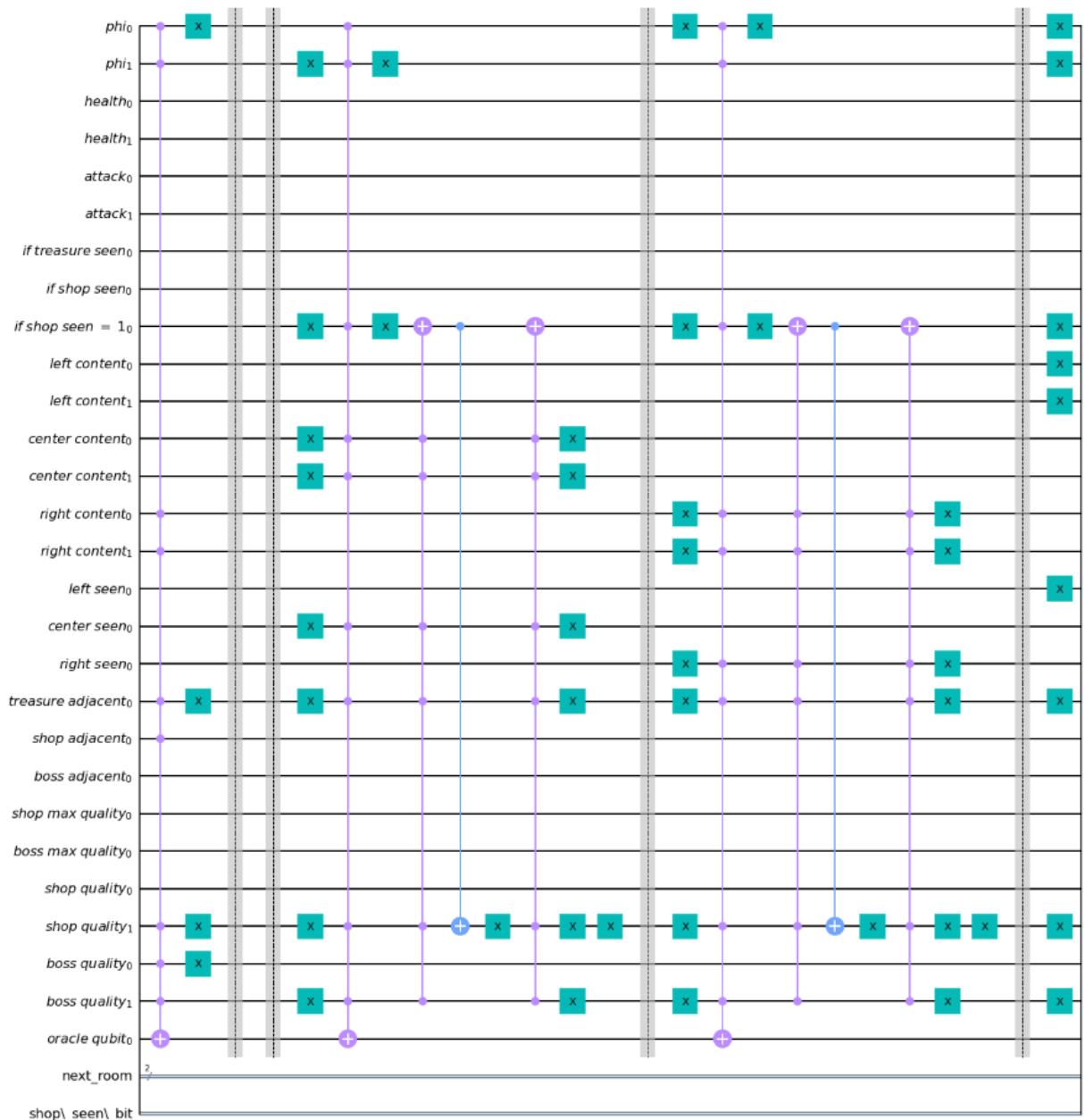


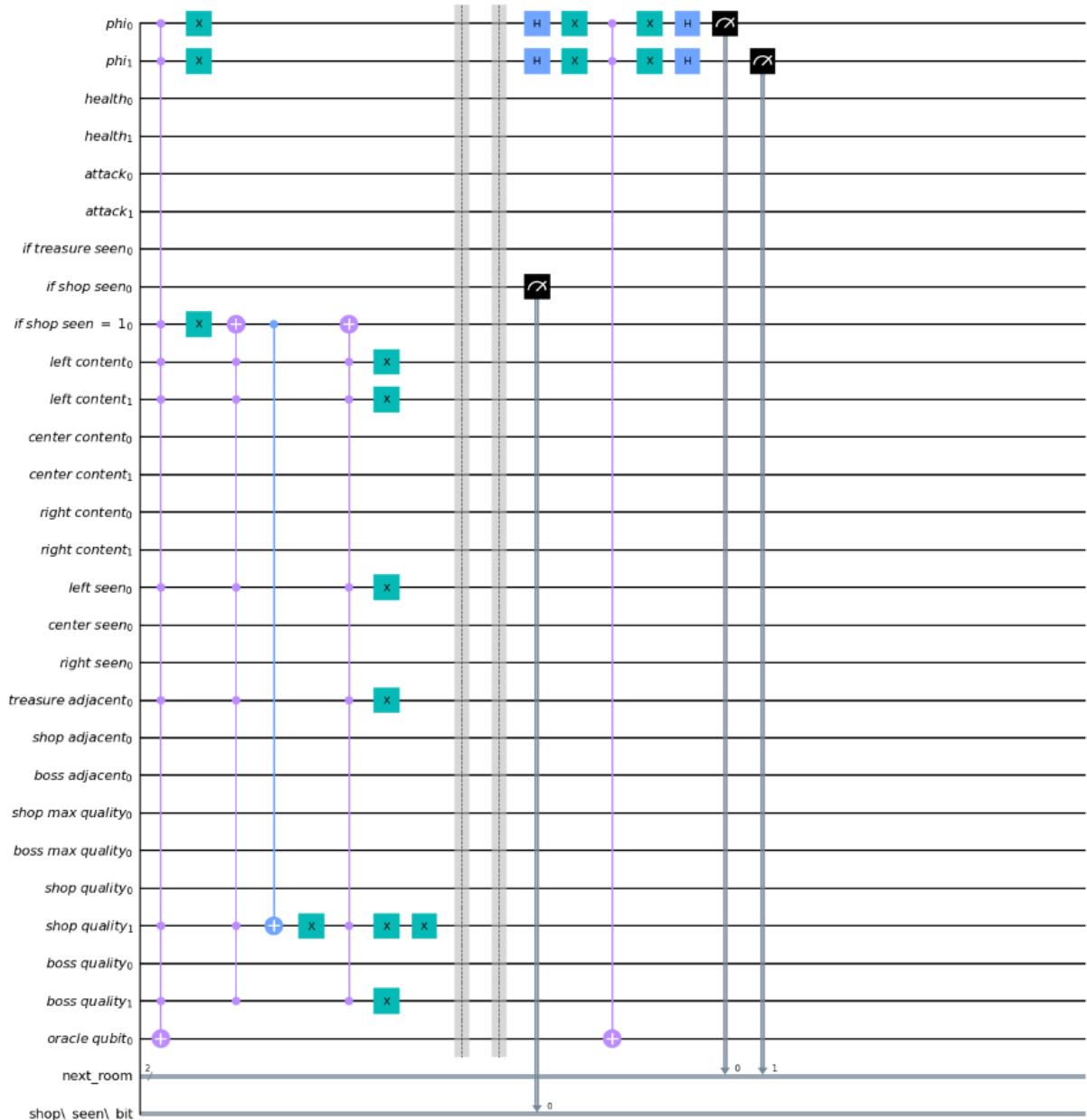












Bibliografia

- [1] Mobius digital. <https://www.mobiusdigitalgames.com/>.
- [2] Arya K. Babbush R. et al. Arute, F. Quantum supremacy using a programmable superconducting processor. <https://www.nature.com/articles/s41586-019-1666-5>.
- [3] Anna Bernasconi, Paolo Ferragina and Fabrizio Luccio. *Elementi di Crittografia*. Pisa University Press, 2015.
- [4] IQIM Caltech. Stephen hawking faces paul rudd in epic chess match (feat. keanu reeves). https://www.youtube.com/watch?v=Hi0BzqV_b44.
- [5] Chris Cantwell. Quantum Chess. <https://quantumchess.net>.
- [6] J. Clement. Topic: Video game industry. <https://www.statista.com/topics/868/video-games/>.
- [7] Intel. Reinventing data processing with quantum computing. <https://www.intel.com/content/www/us/en/research/quantum-computing.html>.
- [8] Jack Krupansky. Quantum advantage now: Generation of true random numbers. <https://jackkrupansky.medium.com/quantum-advantage-now-generation-of-true-random-numbers-237d89f8a7f2>.
- [9] Luciano Lenzini. Oltre i bit: la logica illogica dei quantum computer. https://www.italian.tech/blog/quantum/2021/06/04/news/cosa_sono_i_quantum_computer_e_perche_ci_cambieranno_la_vita-304214663/.
- [10] Andy Matuschak and Michael Nielsen. Quantum computing for the very curious. <https://quantum.country/qcvc>.
- [11] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2010.
- [12] Peter Norvig, Stuart Russell and Francesco Amigoni. *Intelligenza artificiale, un approccio moderno*. Pearson Italia, 2010.
- [13] Michaël Rollin. Quantum PokéMon Fight. <https://fullstackquantumcomputation.tech/blog/post-quantum-pokemon-fight/#chap0>.
- [14] serebii.net. Critical hits. <https://www.serebii.net/games/criticalhits.shtml>.
- [15] TensorFlow. Introduction to Quantum Chess (Quantum Summer Symposium 2020). <https://www.youtube.com/watch?v=ec-Mb80JuRg>.
- [16] Wikipedia. Legge di Moore — Wikipedia, the free encyclopedia. https://it.wikipedia.org/wiki/Legge_di_Moore.
- [17] Wikipedia. IBM Q System One — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/IBM_Q_System_One, 2021.

- [18] Wikipedia. Minecraft — Wikipedia, the free encyclopedia. <https://it.wikipedia.org/wiki/Minecraft>, 2021.
- [19] Wikipedia. No Man's Sky — Wikipedia, the free encyclopedia. https://it.wikipedia.org/wiki/No_Man's_Sky, 2021.
- [20] Wikipedia. Pokémon — Wikipedia, the free encyclopedia. <https://it.wikipedia.org/wiki/Pok%C3%A9mon>, 2021.
- [21] Wikipedia. Quantum supremacy — Wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Quantum%20supremacy>, 2021.
- [22] Wikipedia. Roguelike — Wikipedia, the free encyclopedia. <https://it.wikipedia.org/wiki/Roguelike>, 2021.
- [23] Wikipedia. Storia dei videogiochi — Wikipedia, the free encyclopedia. https://it.wikipedia.org/wiki/Storia_dei_videogiochi, 2021.
- [24] Wikipedia. The Binding of Isaac: Rebirth — Wikipedia, the free encyclopedia. https://it.wikipedia.org/wiki/The_Binding_of_Isaac:_Rebirth, 2021.
- [25] James Wootton. Creating infinite worlds with quantum computing. <https://medium.com/qiskit/creating-infinite-worlds-with-quantum-computing-5e998e6d21c2>.
- [26] James Wootton. Gamers, physicists come together at fifth annual quantum jam in helsinki. <https://www.ibm.com/blogs/research/2019/03/quantum-ferris-wheel/>.
- [27] James Wootton. Getting to know your quantum processor. <https://medium.com/qiskit/getting-to-know-your-quantum-processor-ea418867615f>.
- [28] James Wootton. The history of games for quantum computers. <https://decodoku.medium.com/the-history-of-games-for-quantum-computers-a1de98859b5a>.
- [29] James Wootton. Hunt the quantupus. <https://decodoku.itch.io/hunt-the-quantupus>.
- [30] James Wootton. Introducing the world's first game for a quantum computer. <https://decodoku.medium.com/introducing-the-worlds-first-game-for-a-quantum-computer-50640e3c22e4>.
- [31] James Wootton. Quantum battleships: The first multiplayer game for a quantum computer. <https://decodoku.medium.com/quantum-battleships-the-first-multiplayer-game-for-a-quantum-computer-e4d600ccb3f3>.
- [32] James Wootton. Using a simple puzzle game to benchmark quantum computers. <https://decodoku.medium.com/understanding-quantum-computers-through-a-simple-puzzle-game-a290dde89fb2>.
- [33] Noson S. Yanofsky and Mirco A. Mannucci. *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008.