Alma Mater Studiorum · Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

Problemi di ottimizzazione combinatoria binaria con computer quantistici

Elaborato in:
High Performance Computing

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Presentata da:
Davide Crisante

III Sessione di laurea
Anno Accademico 2020/2021

Dedico questo lavoro a tutti coloro che
mi sono stati vicino durante questo percorso.

Introduzione

La computazione quantistica permette di risolvere problemi sfruttando alcune proprietà della meccanica quantistica. Ciò ha permesso la realizzazione di numerosi algoritmi quantistici capaci di risolvere problemi in maniera più efficiente rispetto alla loro controparte classica. Ad esempio:

- l'algoritmo di fattorizzazione di Shor [18] permette di risolvere il problema della fattorizzazione in numeri primi (problema NP) in tempo polinomiale;
- l'algoritmo di Deutsch-Jozsa [6] permette di determinare se una funzione booleana sia bilanciata o costante in tempo $O(1)$, mentre l'algoritmo classico per risolvere lo stesso problema ha un costo $O(2^n)$.

In questa tesi verranno introdotte le nozione di base della computazione quantistica e, in seguito, saranno analizzati due metodi per la risoluzione di problemi di ottimizzazione combinatoria binaria tramite l'impiego di macchine quantistiche.

Indice

Elenco delle figure

Capitolo 1

Problemi di Ottimizzazione

Per *problemi di ottimizzazione* si intendono i problemi la cui soluzione consiste nel trovare la soluzione "migliore" tra tutte quelle possibili.
Se le variabili di un problema di ottimizzazione sono discrete lo chiameremo **problema di ottimizzazione combinatoria**, in cui la soluzione "migliore" può essere scelta all'interno di un insieme al più numerabile di soluzioni.

L'ottimizzazione combinatoria è uno degli ambiti di ricerca più importanti nel campo dell'ottimizzazione. Infatti, ha molteplici applicazioni pratiche in molti campi ed è un tema su cui la ricerca è molto attiva.

1.1 Formulazione del problema

Data una funzione $f : X \Rightarrow \mathbb{R}$ con X insieme qualsiasi, un problema di ottimizzazione può essere posto sotto forma di un problema di:

- *minimo*: trovare $x_0 \in X \mid f(x_0) \leq f(x)\ \forall x \in X$
- *massimo*: trovare $x_0 \in X \mid f(x_0) \geq f(x)\ \forall x \in X$

Siccome i problemi di ottimizzazione possono avere caratteristiche molto diverse tra di loro, anche la loro formulazione può variare a seconda del problema.

I modelli di ottimizzazione maggiormente usati nell'ambito del Quantum Computing sono:

- Quadratic Unconstrained Binary Optimization (**QUBO**)
- Modello di Ising

1.1.1 Quadratic Unconstrained Binary Optimization

Quadratic Unconstrained Binary Optimization, abbreviato con **QUBO**, è un modello di ottimizzazione combinatoria binaria (in cui le variabili possono assumere solo due valori, in questo caso: 0 e 1).

La formulazione di un problema QUBO è:

$$f(x) = \sum_i Q_{i,i} x_i + \sum_{i<j} Q_{i,j} x_i x_j$$

con $x_i \in \{0, 1\}$ e Q matrice triangolare superiore.

Alcuni problemi esprimibili tramite QUBO sono: apprendimento supervisionato tramite macchine a vettori di supporto(SVM), taglio massimo (Max-Cut), colorazione di grafi [2], partizionamento [12] e molti altri.

1.1.2 Modello di Ising

Il modello di Ising, noto anche con il nome di *modello Lenz-Ising* [9], nasce negli anni '20 del ventesimo secolo per indagare fenomeni ferromagnetici. Originariamente ogni variabile rappresentava il momento magnetico di un atomo ed i coefficienti $J_{i,j}$ descrivevano le interazioni tra gli atomi.
Anch'esso è un modello di ottimizzazione combinatoria binaria le cui variabili possono assumere come valori $+1$ e -1.

La formulazione di un problema di Ising è:

$$f(x) = \sum_i h_i x_i + \sum_i \sum_{j=i+1} J_{i,j} x_i x_j$$

con $x_i \in \{+1, -1\}$, J matrice di coefficienti reali, h vettore di coefficienti reali.

1.1.3 Modelli a confronto

Nonostante i due modelli precedenti siano nati per inseguire obiettivi differenti, essi rappresentano gli stessi problemi.

Notiamo infatti che entrambi hanno dei coefficienti (o *bias*) per ogni variabile e dei coefficienti quadratici per ogni coppia di variabili.
Ciò rende possibile il passaggio delle variabili da un modello all'altro, mantenendo lo stesso risultato.

$$Ising \rightarrow QUBO : x_{qubo_i} = \frac{-x_{ising_i} + 1}{2}$$
$$QUBO \rightarrow Ising : x_{ising_i} = 1 - 2 * x_{qubo_i}$$

I problemi QUBO ed Ising sono NP-Hard, ovvero sono complicati da risolvere almeno quanto i problemi più difficili di NP [14].
I problemi NP sono risolvibili in tempo polinomiale da macchine non deterministiche, mentre le soluzioni sono verificabili in tempo polinomiale da macchine deterministiche.
I problemi NP-Complete sono problemi che possono essere derivati per riduzione, in tempo polinomiale, da problemi NP.
I problemi NP-Hard sono problemi che possono essere derivati per riduzione, in tempo polinomiale, da problemi NP-Complete. Visto che ogni problema NP-Complete può essere trasformato in qualsiasi altro problema NP-Complete, ogni problema NP-Complete è riducibile ad un qualsiasi problema NP-Hard in tempo polinomiale.

1.2 Funzione Hamiltoniana

Una funzione Hamiltoniana è una funzione che associa stati energetici a possibili stati del sistema. Sia problemi Ising che QUBO sono rappresentabili da funzioni Hamiltoniane.
Una funzione Hamiltoniana può essere espressa sotto forma di matrice Hermitiana, ossia una matrice quadrata che coincide con la sua trasposta coniugata. Gli autovalori di una matrice hermitiana sono sempre reali ed ortogonali tra loro, ciò ci tornerà utile nei capitoli successivi.
La relazione tra matrice hamiltoniana H, stati (in forma vettoriale) del sistema ψ e stati energetici E (scalari reali) può essere espressa dall'uguaglianza:

$$H\psi = \psi E$$

Quindi gli stati (o livelli) energetici sono gli autovalori della matrice hamiltoniana. Dato uno stato del sistema associato all'hamiltoniana, possiamo trovare l'energia associata a quello stato come:

$$\psi^T H\psi = E$$

1.3 Esempi problemi di ottimizzazione riconducibili a QUBO e modello di Ising

1.3.1 Max-Cut

Dato un grafo $G = (V, E)$ con V nodi ed E connessioni tra coppie di nodi, il problema Max-Cut consiste nel trovare il modo migliore di separare i vertici connessi tra di loro per massimizzare il numero di collegamenti interrotti dal "taglio".

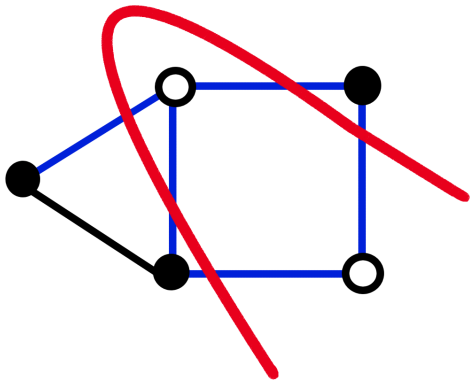

Figura 1.1: Esempio di soluzione ottima di un problema di taglio massimo

Questo problema è NP-Hard e possiamo formularlo tramite il modello di Ising:

$$\min \sum_{(i,j)\in E} x_i x_j = \min \sum_{i=1}^{N} \sum_{j=i+1}^{N} J_{i,j} x_i x_j \quad \text{con } J_{i,j} \in \{0,1\} \quad , x_i \in \{+1,-1\}$$

Dove i valori che $J_{i,j}$ assume sono:

- uguali ad uno se i nodi i e j sono connessi;
- uguali a zero se non lo sono.

Esiste anche una versione più generica del problema Max-Cut che consente di considerare anche coefficienti (o "pesi") associati alle connessioni dei nodi del grafo G:

$$\min \sum_{i=1}^{N} \sum_{j=i+1}^{N} J_{i,j} x_i x_j \quad \text{con } J_{i,j} \in \mathbb{R} \quad , x_i \in \{+1,-1\}$$

Dove i valori che $J_{i,j}$ assume sono:

- uguali a zero se i nodi i e j non sono connessi;
- appartenenti ad $\mathbb{R} - \{0\}$ se i due nodi sono connessi.

1.3.2 Problema del commesso viaggiatore

Il problema del commesso viaggiatore è un problema classico in ambito informatico. La formulazione del problema recita: dato un insieme di città di cui sappiamo la distanza l'una dall'altra, qual è il percorso da fare per attraversare tutte le città una sola volta e tornare nella città di partenza minimizzando la distanza percorsa?
Visto che da questa formulazione sono tralasciati alcuni dettagli, esistono differenti versioni del problema; riportiamo quella di Dantzing-Fulkerson-Johnson [8].
Il problema può essere espresso tramite il modello QUBO:

$$\min_{x=\{x_1,\dots,x_N\}} \sum_{i<j} Q_{i,j} x_i x_j$$

con

$$\sum_{i=1,i\neq j}^{N} x_i x_j = 1 \qquad j = 1,\dots,N;$$

$$\sum_{j=1,i\neq j}^{N} x_i x_j = 1 \qquad i = 1,\dots,N;$$

$$\sum_{i\in S} \sum_{j\neq i, i\in S} x_{i,j} \leq |S| - 1 \qquad \forall S \subsetneq \{1,\dots,N\}, |S| \geq 2$$

1.3.3 Problema del negozio di vernici binario

Questo è una versione semplificata del Paintshop Problem, problema affrontato dall'azienda Volkswagen per rendere più efficienti i propri impianti di verniciazione nel 2020 [19]. Il problema del negozio di vernici binario è formulato come segue: sono date n automobili di modelli diversi disposte su un rullo in ordine casuale, tutte devono essere verniciate seguendo l'ordine in cui si trovano sul rullo. Per ogni modello di automobile ci sono esattamente due esemplari e devono essere colorati di colori diversi. I colori a disposizione sono solo due. Il macchinario che pittura le auto è solo uno e vogliamo

trovare un modo di colorare le macchine che minimizzi i cambi di vernice che il macchinario deve effettuare.
Per risolvere questo problema è stato proposto un approccio [19] che permette di utilizzare $\frac{n}{2}$ variabili. Questo metodo ci indica di che colore pitturare la prima macchina di ogni modello.
Rappresentiamo il problema sotto forma di modello di Ising con $\frac{n}{2}$ variabili e J costruito come segue:
è data una sequenza casuale di auto dove ogni modello è rappresentato da un numero. Il vettore J è inizializzato con tutti gli elementi uguali a zero.
Scorrendo la sequenza di auto da sinistra a destra, leggiamo il valore di due auto (x, y) alla volta. Aggiungiamo a $J_{x,y}$ il valore -1 se non sono state viste in precedenza auto del modello x e y (così prediligiamo il fatto che entrambe siano colorate con lo stesso colore). Altrimenti aggiungiamo il valore $+1$ in modo da sfavorire l'evenienza in cui le due auto siano colorate nello stesso modo.

Esempio:
Sequenza:

$$[0, 2, 1, 3, 1, 0, 2, 3]$$

valori di J:

$$J_{0,1} = -1 \qquad J_{0,2} = -1 - 1 = -2$$
$$J_{1,3} = -1 + 1 = 0 \qquad J_{1,2} = -1$$
$$J_{2,3} = -1$$

Formulazione del problema tramite modello di Ising:

$$f(X) = (-1)x_0x_1 + (-2)x_0x_2 + (0)x_1x_3 + (-1)x_1x_2 + (-1)x_2x_3$$

Soluzione:

$$\min f(X) = -5$$

con $X = \{x_0, x_1, x_2, x_3\} = \{-1, -1, -1, -1\}$ oppure $X = \{+1, +1, +1, +1\}$.
In questo tipo di problema le soluzioni ottime sono sempre due, invertendo

i colori infatti il numero di cambi di colore non varia.
Quindi in questo caso coloreremo la prima auto di ogni modello con lo stesso colore. Ciò ci consente di effettuare solo un cambio di colore.

Capitolo 2

Il Calcolo Quantistico

Il calcolo quantistico è un paradigma di calcolo che sfrutta delle proprietà della meccanica quantistica per svolgere computazioni.
Esistono diverse tipologie di computer quantistici, i quali possono essere basati su:

- circuiti quantistici;
- calcolo quantistico adiabatico;
- macchine di Turing quantistiche;
- altro...

Tutti però sono accomunati da tre componenti fondamentali per la computazione: dati, operazioni e risultati.

2.1 Qubit

I **Qubit** sono le unità d'informazione nel calcolo quantistico.
Questi sfruttano alcune proprietà della meccanica quantistica, quali:

- a ogni sistema fisico isolato si associa uno spazio vettoriale di Hilbert (uno spazio vettoriale complesso).

Infatti lo stato di un qubit può essere rappresentato da un vettore unitario complesso;

- l'evoluzione di un sistema quantistico può essere descritta tramite l'uso di un operatore unitario. Questi definiscono le operazioni effettuabili da un computer quantistico sui dati e li chiameremo porte (o gate).
  Il fatto che le porte quantistiche siano operatori unitari implica la possibilità di effettuare computazione reversibile (se non vengono compiute misurazioni dello stato del sistema) [13];

- è possibile misurare lo stato di un sistema quantistico tramite un insieme di operatori M_m che ci permettono di valutare la probabilità che il sistema sia nello stato m;

- lo stato di un sistema quantistico composto da più elementi è dato dal prodotto tensoriale dei vettori degli stati degli elementi.

2.1.1 Notazione Bra-Ket

La notazione Bra-Ket (o notazione di Dirac) viene creata per descrivere uno stato quantistico.
Dato un vettore riga ϕ, lo chiameremo *bra* e lo indicheremo con $\langle\phi|$.
Dato un vettore colonna ψ, lo chiameremo *ket* e lo indicheremo con $|\psi\rangle$.
Con $\langle\phi|\psi\rangle$ indichiamo il prodotto scalare dei due vettori.
Con $|\phi\rangle\langle\psi|$ indichiamo il prodotto vettoriale esterno.

Possiamo descrivere lo stato $|1\rangle$ di un qubit come: $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
e lo stato $|0\rangle$ di un qubit come: $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Dato un qubit, chiamiamo il suo stato $|\psi\rangle$ e lo descriviamo come:

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

con α e β numeri complessi e tali che $\alpha^2 + \beta^2 = 1$.
Lo stato $|\psi\rangle$ può rappresentare una **superposizione** di più stati.
La superposizione è una proprietà della meccanica quantistica per la quale un sistema può trovarsi contemporaneamente in più stati. Nel caso di un singolo qubit: quando verrà misurato il suo stato applicando un operatore adeguato che possiamo chiamare M_1 o M_0, la probabilità di misurare lo stato $|1\rangle$ o $|0\rangle$ seguirà la *legge di Born* [15]:

$$P(M_0 |\psi\rangle = 0) = \alpha^2$$
$$P(M_1 |\psi\rangle = 1) = \beta^2$$

Il processo di misurazione di uno o più qubit elimina l'informazione quantistica presente nel sistema misurato: di un qubit che ha subito una misurazione sappiamo solo l'esito della misurazione.
Ci si riferisce a questa caratteristica come **collasso del vettore di stato** o **collasso**. A causa della natura probabilistica della misurazione di un sistema di qubit, è necessario eseguire lo stesso programma quantistico un numero elevato di volte per ottenere un risultato accurato.

Il *ket* associato ad uno stato è un vettore complesso. La componente complessa del vettore di stato viene chiamata **fase**; questa non altera il risultato atteso da una misurazione, tuttavia influisce sul modo in cui i qubit vengono influenzati dalle porte quantistiche.

Anche quando abbiamo più qubit la notazione *Bra-Ket* ci consente di rappresentare lo stato del sistema. Consideriamo un sistema composto da due

qubit in cui:

$$|00\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \qquad |01\rangle = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$$

$$|10\rangle = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \qquad |11\rangle = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Lo stato $|\psi\rangle$ del sistema è, quindi, esprimibile come:

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle = \begin{bmatrix}\alpha_{00}\\\alpha_{01}\\\alpha_{10}\\\alpha_{11}\end{bmatrix} \quad \text{con} \sum_{x=0}^{1}\sum_{y=0}^{1}|a_{x,y}|^2 = 1$$

Esempio:

Dati due qubit:

$$|\phi\rangle = \alpha_\phi|0\rangle + \beta_\phi|1\rangle$$
$$|\psi\rangle = \alpha_\psi|0\rangle + \beta_\psi|1\rangle$$

Descriviamo il sistema composto dai due qubit come:

$$|\phi\rangle \otimes |\psi\rangle = \begin{bmatrix}\alpha_\phi\\\beta_\phi\end{bmatrix} \otimes \begin{bmatrix}\alpha_\psi\\\beta_\psi\end{bmatrix} = \begin{bmatrix}\alpha_\phi\alpha_\psi\\\alpha_\phi\beta_\psi\\\beta_\phi\alpha_\psi\\\beta_\phi\beta_\psi\end{bmatrix} =$$

$$= (\alpha_\phi\alpha_\psi)|00\rangle + (\alpha_\phi\beta_\psi)|01\rangle + (\beta_\phi\alpha_\psi)|10\rangle + (\beta_\phi\beta_\psi)|11\rangle$$

Lo stato energetico associato ad uno stato $|\psi\rangle$ da una matrice hamiltoniana H è pari a $\langle\psi| H |\psi\rangle$.

2.1.2 Sfera di Bloch

Un modo comune di rappresentare lo stato di un qubit è tramite l'uso della sfera di Bloch. Dato uno stato $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$ possiamo calcolare il valore di γ, φ, θ tali per cui:

$$\alpha = e^{i\gamma} \cos \frac{\theta}{2}$$
$$\beta = e^{i\varphi} e^{i\gamma} \sin \frac{\theta}{2}$$

Possiamo quindi riscrivere $|\psi\rangle$ come:

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

Qubit con diversi valori di γ sono associati allo stesso punto sulla sfera di Bloch in quanto $e^{i\gamma}$ non ha effetti osservabili; questo parametro è noto come *fase globale* e può essere ignorato per quanto riguarda i nostri scopi. Possiamo scrivere lo stato precedente nella forma:

$$|\psi\rangle = \cos \frac{\theta}{2} |0\rangle + e^{i\varphi} \sin \frac{\theta}{2} |1\rangle$$

con $0 \leq \theta \leq \pi$ e $0 \leq \varphi \leq 2\pi$.

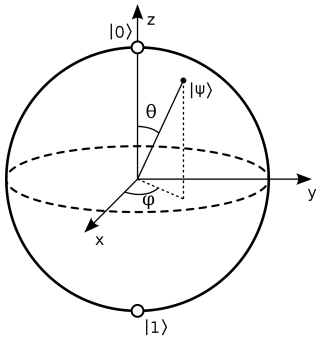

Figura 2.1: Rappresentazione sfera di Bloch
Fonte: en.wikipedia.org/wiki/Bloch_sphere

Ciò ci permette di rappresentare ogni stato di un qubit come un punto su una sfera individuabile dalle coordinate $(\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta) \in \mathbb{R}^3$. Possiamo immaginare in maniera più intuitiva come le varie operazioni modificano lo stato dei qubit associando a ogni operazione una rotazione della sfera di Bloch.

Esempio:
Prendiamo in considerazione la Figura 2.1. Possiamo verificare dove si trovi lo stato $|0\rangle$ sulla sfera di Bloch:

$$|0\rangle = 1\,|0\rangle + 0\,|1\rangle = \cos\frac{0}{2}\,|0\rangle + e^{i\varphi}\sin\frac{0}{2}\,|1\rangle \qquad \forall\varphi \in [0, 2\pi]$$

Quindi lo stato $|0\rangle$ si trova nel punto $(0, 0, 1)$ sulla sfera di Bloch.

2.2 Operazioni sui dati

La manipolazione delle informazioni quantistiche contenute nei qubit può essere svolta in molteplici modi sfruttando diverse tecnologie. Al momento i metodi di manipolazione dei dati su cui la ricerca è più attiva sono:

- **porte logiche quantistiche**;

- **accoppiatori** e **bias**.

2.2.1 Porte logiche quantistiche

Le porte logiche quantistiche sono descrivibili da operatori unitari che agiscono sul vettore di stato del sistema. Dato uno stato $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, possiamo applicargli un qualsiasi operatore unitario $U = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ che ne trasforma lo stato di conseguenza:

$$U|\psi\rangle = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} a\alpha + b\beta \\ c\alpha + d\beta \end{pmatrix}$$

Essendo U unitario, resta sempre vero che: $|a\alpha + b\beta|^2 + |c\alpha + d\beta|^2 = 1$

Operatore identità

L'operatore identità non modifica lo stato del sistema. Esso può essere rappresentato come matrice unitaria:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Porta X di Pauli

La porta X di Pauli comporta una rotazione di 180° rispetto all'asse X della sfera di Bloch.
Essa può essere rappresentata come matrice unitaria:

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

è equivalente alla porta di rotazione:

$$R_X(\theta) = e^{-i\frac{\theta}{2}X} = \cos\frac{\theta}{2}I + i\sin\frac{\theta}{2}X = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

per $\theta = \pi$.

Porta Y di Pauli

La porta Y di Pauli comporta una rotazione di 180° rispetto all'asse Y della sfera di Bloch.
Essa può essere rappresentata come matrice unitaria:

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

è equivalente alla porta di rotazione:

$$R_Y(\theta) = e^{-i\frac{\theta}{2}Y} = \cos\frac{\theta}{2}I + i\sin\frac{\theta}{2}Y = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

per $\theta = \pi$.

Porta Z di Pauli

La porta Z di Pauli comporta una rotazione di 180° rispetto all'asse Z della sfera di Bloch.
Essa può essere rappresentata come matrice unitaria:

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

è equivalente alla porta di rotazione:

$$R_Z(\theta) = e^{-i\frac{\theta}{2}Z} = \cos\frac{\theta}{2}I + i\sin\frac{\theta}{2}Z = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

per $\theta = \pi$.

Porta S

La porta S comporta una rotazione di 90° rispetto all'asse Z della sfera di Bloch.

Essa può essere rappresentata come matrice unitaria:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

è equivalente alla porta di rotazione: $R_Z(\frac{\pi}{2})$.

Porta T

La porta T comporta una rotazione di 90° rispetto all'asse Z della sfera di Bloch.
Essa può essere rappresentata come matrice unitaria:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{bmatrix}$$

è equivalente alla porta di rotazione: $R_Z(\frac{\pi}{4})$.

Porta X Controllata

Le porte controllate sono porte che agiscono su due o più qubit allo stesso tempo.
La porta X controllata, chiamata CNOT o CX, agisce su due qubit. In presenza di stati "classici" (i qubit sono nello stato *0* o *1*, senza superposizioni), la porta CNOT mappa gli stati $|q_0, q_1\rangle \rightarrow |q_0, q_0 \oplus q_1\rangle$ come se fosse una porta XOR classica che legge q_0 (qubit di controllo) e scrive il risultato dello XOR con q_1 nel secondo qubit (qubit obiettivo).
In presenza di stati non classici i qubit si influenzano a vicenda a causa delle fasi. Questo fenomeno è noto come **phase-kickback** ed è sfruttato da un grande numero di algoritmi quantistici.
La matrice della porta CNOT, come tutte le porte che operano su più qubit, è data dal prodotto tensoriale delle porte applicate a ognuno dei qubit su cui essa opera. Nel caso della porta CNOT il qubit di controllo non viene modificato (se non a causa della fase), quindi a esso viene applicata la matrice identità I; mentre sul qubit obiettivo viene applicata la porta X. Quindi:

$$CNOT = I \otimes X = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In Figura 2.2 possiamo vedere una rappresentazione grafica della porta CNOT.

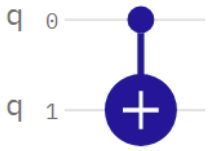

Figura 2.2: Porta CNOT con q_0 qubit di controllo e q_1 qubit obiettivo

Esempio:
CNOT con stati classici (senza superposizione):

$$|0,0\rangle \rightarrow |0,0\rangle$$
$$|0,1\rangle \rightarrow |0,1\rangle$$
$$|1,0\rangle \rightarrow |1,1\rangle$$
$$|1,1\rangle \rightarrow |1,0\rangle$$

Esempio:
CNOT con stati non classici:

$$\frac{1}{2}\begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \rightarrow \frac{1}{2}\begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$

Notiamo che in questo caso le probabilità di collasso dei qubit non variano, ma c'è un'inversione di fase nel qubit di controllo.

Porta di Hadamard

Comporta una rotazione di 180° rispetto all'asse $\frac{\hat{x}+\hat{z}}{\sqrt{2}}$ della sfera di Bloch. Rappresentazione come matrice unitaria:

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

è equivalente a:

$$R_Y(\pi/2)Z = XR_Y(\pi/2)$$

Porte di Clifford

Sono elementi di un insieme di operatori chiamato gruppo di Clifford. Tutti gli operatori appartenenti al gruppo di Clifford sono generati dalle porte: Hadamard, CNOT ed S.
Notiamo che tutte le porte Pauli sono generabili dalla porta S e dalla porta di Hadamard, quindi queste porte rientrano nel gruppo di Clifford.
Per il teorema di Gottesman-Knill [10] i circuiti quantistici che usano queste porte possono essere efficientemente simulati da un computer classico.
Questo insieme di operatori non è tuttavia universale in quanto:

- non tutti gli operatori sono compresi nel gruppo di Clifford;
- alcuni operatori non generabili dal gruppo di Clifford non possono essere approssimati con una sequenza finita di operatori di Clifford.

Aggiungendo la porta T all'insieme di Clifford otteniamo un insieme di porte quantistiche **universali** (noto come "*Extended Clifford*" o "*Clifford+T*").
Con "porte quantistiche universali" intendiamo un insieme di porte che ci offrono la possibilità di ottenere ogni operatore che vogliamo e di applicarlo su un numero arbitrario di qubit. Ciò ci permette di riprodurre tutto quello che potrebbe fare un computer classico, oltre a tutto quello che solo un computer quantistico può fare.

Circuiti quantistici

Un metodo diffuso per rappresentare programmi quantistici per macchine basate su porte quantistiche sono i circuiti quantistici. Questi forniscono una rappresentazione grafica del programma che la macchina eseguirà e permettono di rendere più semplice la comprensione del programma.
Un circuito quantistico:

- va letto da sinistra a destra, poichè questo è l'ordine in cui i comandi saranno eseguiti;
- ogni riga indica un qubit;
- permette di indicare l'uso degli operatori presentati precedentemente tramite varie icone.

Esempio:
La computazione $HTSZYX\,|0\rangle$ è rappresentabile sotto forma di circuito come in Figura 2.3:

Figura 2.3: Esempio circuito con misurazione finale.

A seguito della misurazione, il risultato viene scritto su un registro classico, indicato in Figura 2.3 dalla riga "c1".

Creazione di un matrice hamiltoniana

Possiamo costruire una matrice hamiltoniana trasformando ogni variabile di un problema QUBO in:

$$x_i \to \frac{I - Z_i}{2}$$

ed ogni variabile di un problema di Ising in:

$$x_i \rightarrow Z_i$$

Esempio:
Dato un problema di Ising :

$$x_0x_1 + x_1x_2 + x_0x_3 + x_2x_3$$

possiamo esprimerlo sotto forma della matrice hamiltoniana:

$$\begin{aligned} H_P = &\frac{1}{2}\big(Z_0 \otimes Z_1 \otimes I_2 \otimes I_3\big) + \frac{1}{2}\big(I_0 \otimes Z_1 \otimes Z_2 \otimes I_3\big) + \\ &\frac{1}{2}\big(Z_0 \otimes I_1 \otimes I_2 \otimes Z_3\big) + \frac{1}{2}\big(I_0 \otimes I_1 \otimes Z_2 \otimes Z_3\big) \end{aligned}$$

2.2.2 Accoppiatori e Bias

I dati vengono manipolati tramite due componenti:

- accoppiatori (*couplers*);
- bias.

I **bias** agiscono sui singoli qubit e permettono di influenzare la probabilità che un certo qubit collassi in uno stato alto o basso.
Gli **accoppiatori** agiscono su coppie di qubit ed assegnano a ciascuna coppia una "forza di accoppiamento" che indica la correlazione tra i due qubit.
Esempi di accoppiamento:

- forza di accoppiamento = 0: qubit completamente indipendenti;
- forza di accoppiamento = 1: qubit che tendono a collassare in stati opposti;
- forza di accoppiamento =-1: qubit che tendono a collassare in stati uguali;

Bias e accoppiatori permettono di configurare un sistema in modo che rappresenti problemi di Ising o QUBO attraverso un processo chiamato *"minor embedding"*. Questo si occupa di allocare le risorse della macchina scegliendo quali qubit manipolare e in quale modo, a seconda della topologia del processore. Questo compito è spesso demandato a moduli che allocano automaticamente le risorse della macchina e può anche essere reso completamente trasparente allo sviluppatore tramite un middleware apposito (come Dwave *Virtual Full-Yield Chip - VFYC*).
Accoppiatori e bias sono usati da macchine chiamate "*quantum annealers*".

2.2.3 Avanzamento tecnologico

Attualmente la computazione quantistica si trova in una fase nota come **NISQ** (*Noisy Intermediate-Scale Quantum computers*). I computer quantistici infatti, per motivi tecnologici, sono molto propensi a errori, sfruttano un numero ridotto di qubit (50-100), i quali non sono tutti connessi tra di loro (gli annealer usano molti qubit, ma soffrono comunque degli altri problemi). Le connessioni tra qubit (ad esempio tramite CNOT) e tutte le componenti hardware, che agiscono sui qubit, introducono del rumore sui dati su cui agiscono. Ciò ha un impatto significativo sui risultati.
IBM mette a disposizione una descrizione accurata dello stato delle proprie macchine quantistiche. Possiamo consultare il "*noise model*" di una macchina per capire quanto questa sia propensa a commettere errori e quali qubit siano direttamente connessi tra di loro. In Figura 2.4 vediamo una rappresentazione grafica del noise model della macchina *ibmq_ guadalupe*, da questa notiamo che, ad esempio, il qubit "3" è meno preciso degli altri nell' applicare la porta X e che il CNOT tra i qubit 2 e 3 è quello più rumoroso.

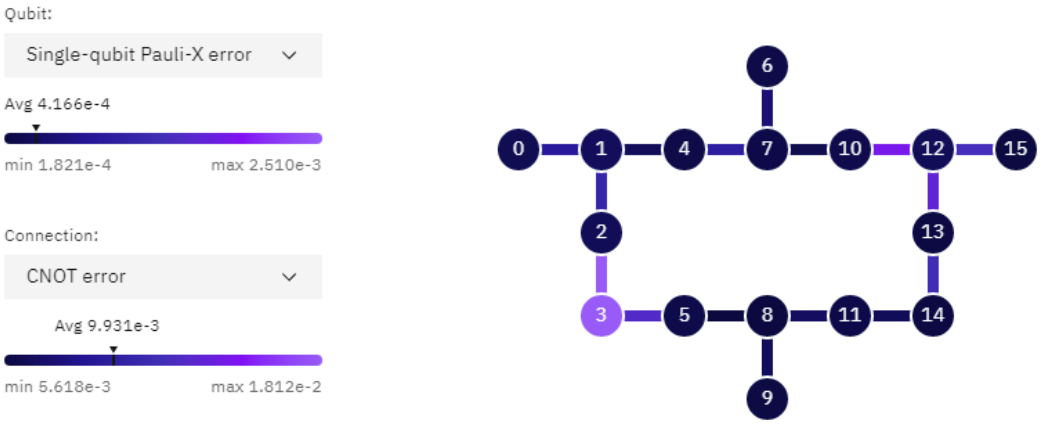

Figura 2.4: Errori riguardanti porta X su singolo qubit e CNOT tra qubit connessi della macchina *ibmq_ guadalupe*, macchina con 16 qubit

La gestione e la correzione degli errori quantistici sono argomenti molto importanti per il calcolo quantistico in questa era tecnologica. Vi sono già svariati tentativi di risoluzione del problema [7][3], tuttavia la ricerca è ancora molto attiva in questo ambito.
Ad oggi le aziende più influenti nel settore del calcolo quantistico (**IBM**, **Honeywell**, **Google**, **IonQ**) stimano di riuscire ad implementare computer quantistici *fault-tollerant* indicativamente entro il 2030, grazie all'impiego di *Surface-Codes* [7]. Questi necessitano di un numero elevato di *qubit fisici* per modellare un *qubit logico* più resistente alle perturbazioni.

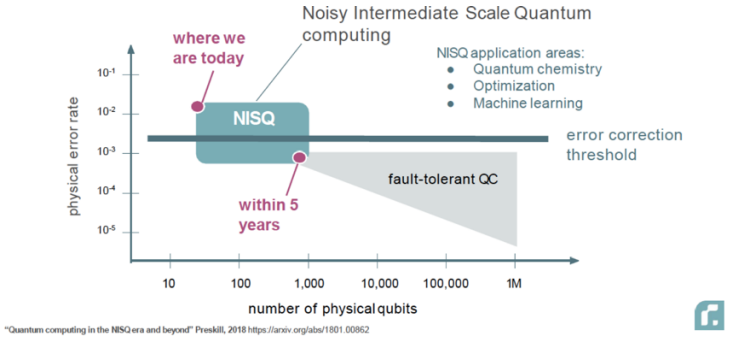

Figura 2.5: Ere del calcolo quantistico
Fonte: Rigetti Computing Inc.

In Figura 2.5 vediamo il rapporto tra frequenza di errori e numero di qubit fisici. Questa evidenzia la necessità di avere un numero elevato di qubit fisici per poter ottenere risultati sempre più precisi.

Capitolo 3

Risoluzione di problemi di ottimizzazione tramite computazione quantistica

Negli anni sono stati proposti svariati metodi per risolvere problemi di ottimizzazione sfruttando macchine quantistiche. Di seguito sono descritti due dei più rilevanti.

3.1 Quantum Annealing

Il Quantum Annealing (**QA**) è un tecnica per trovare minimi globali in una data funzione. L'idea alla base del QA nasce e si sviluppa negli anni '70 ed '80 in cui vennero descritte diverse strategie per simulare questo metodo su macchine classiche [16][1]. Le macchine quantistiche che sfruttano il QA sono chiamate *annealer*.
Il quantum annealing sfrutta il teorema adiabatico [11], il quale afferma che se un sistema si trova in uno stato energetico basso e poi muta con la giusta velocità, il sistema si adatterà alle variazioni per restare nello stato energetico basso del sistema finale.

Il tempo con cui il sistema si deve evolvere dallo stato iniziale a quello finale è direttamente proporzionale a $\frac{1}{\Delta^2_{min}}$ con Δ^2_{min} distanza minima tra stato energetico basso del sistema e il primo stato eccitato. Per problemi il cui stato basso arriva a essere molto vicino a uno stato eccitato, i tempi di anneal possono quindi diventare lunghi.

L'evoluzione del sistema può essere descritta da una hamiltoniana dipendente dal tempo:

$$H(t) = (1 - \frac{t}{T})H_i + \frac{t}{T}H_f$$

con:

- H_i hamiltoniana di *base*, di solito rappresenta tutti i qubit in superposizione ($H^{\otimes n} |\psi\rangle$);
- H_f hamiltoniana del *problema*, che descrive il problema/funzione di cui vogliamo trovare punti di minimo globale;
- T tempo totale di anneal.

I computer che sfruttano questa tecnologia sono limitati a risolvere una piccola classe di problemi rispetto alle macchine basate su porte logiche quantistiche. Tuttavia questa limitatezza consente loro di utilizzare un numero molto maggiore di qubit (migliaia).
Il quantum annealing sfrutta l'evoluzione naturale dei sistemi quantistici per trovare i punti di minimo di un problema dato. Una volta impostato il sistema non possiamo fare altro che aspettare la fine del processo di anneal. Ciò lo rende un metodo molto più semplice di quelli usati dalle macchine basate su porte quantistiche che invece consentono di controllare e manipolare l'evoluzione del sistema nel tempo.

3.2 Quantum Approximate Optimization Algorithm

Quantum Approximate Optimization Algorithm (**QAOA**) è un algoritmo quantistico che permette di trovare soluzioni (non sempre ottime) a problemi di ottimizzazione.

3.2.1 Caratteristiche

QAOA è un algoritmo creato per computer basati su porte logiche quantistiche e fa parte della famiglia degli algoritmi quantistici variazionali. Questi sono caratterizzati dall'uso ibrido di macchine quantistiche e macchine ed ottimizzatori classici. Ciò porta molti benefici nell'era NISQ: consente infatti di eseguire circuiti molto più brevi, ragion per cui gli effetti degli errori quantistici sui risultati restano limitati. Inoltre possiamo utilizzare una grande varietà di ottimizzatori ben noti e già utilizzati in altri ambiti (ad esempio ottimizzatori basati sulla discesa del gradiente). Gli ottimizzatori hanno il compito di leggere lo stato finale del circuito quantistico $\left|\psi(\vec{\beta}, \vec{\gamma})\right\rangle$ e modificare di conseguenza i parametri $(\vec{\beta}, \vec{\gamma})$ al fine di minimizzare $\left\langle\psi(\vec{\beta}, \vec{\gamma})\right| H_P \left|\psi(\vec{\beta}, \vec{\gamma})\right\rangle$, con H_P matrice hamiltoniana associata al problema di cui vogliamo trovare i punti di minimo globale. In altri termini l'algoritmo cerca l'autovalore minimo della matrice hamiltoniana del problema modificando i vettori di stato. QAOA utilizza un numero di qubit pari a quello delle variabili del problema, per questo motivo attualmente non è molto usato per risolvere problemi reali. Nasce come un adattamento del modello adiabatico per i computer basati su porte quantistiche ed è strutturato in cinque fasi:

- inizializzazione dati della macchina quantistica;
- evoluzione del sistema quantistico in base al valore dei parametri;
- misurazione dei qubit;
- ottimizzazione dei parametri tramite l'ottimizzatore classico;

- aggiornamento dei parametri.

Fasi che vengono ripetute fin quando non si soddisfa un criterio di arresto dell'ottimizzatore. In Figura 3.1 vediamo una rappresentazione grafica dell'algoritmo QAOA.

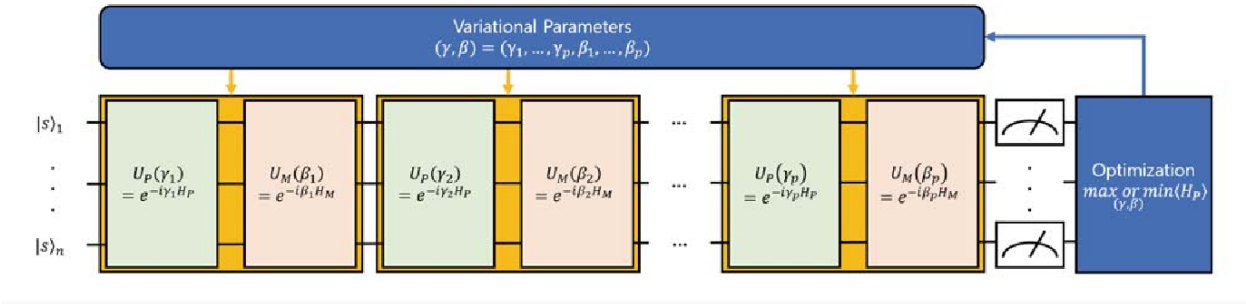

Figura 3.1: Struttura algoritmo QAOA [4]

3.2.2 Struttura dell' algoritmo

Inizializzazione macchina quantistica

Lo stato iniziale è dato da $H^{\otimes n} |0^{\otimes n}\rangle$ con tutti i qubit in superposizione.

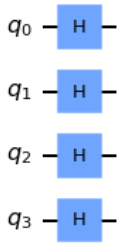

Figura 3.2: Inizializzazione QAOA con $n = 4$ qubit

Evoluzione del sistema

Per far evolvere il sistema viene utilizzato un insieme di operatori che simulano l'evoluzione a partire dallo stato iniziale; la velocità e la modalità con cui il sistema evolve è data dal numero di passi che QAOA può fare (p) e dal valore dei parametri che vengono modificati dall'ottimizzatore classico. Il numero di passi di QAOA può essere scelto arbitrariamente, tuttavia va tenuto in considerazione che un circuito più lungo porta più rumore nei dati. Per questa ragione, attualmente, il numero p di passi da preferirsi è molto piccolo (2-5). Vengono utilizzati $2p$ parametri tali per cui lo stato misurato al termine del circuito quantistico è:

$$\left|\psi(\vec{\beta},\vec{\gamma})\right\rangle = U(\beta_1,\gamma_1)U(\beta_2,\gamma_2)\dots U(\beta_p,\gamma_p)$$

con:

$$U(\beta_i,\gamma_i) = U_M(\beta_i)U_P(\gamma_i)$$

e con:

$$U_M(\beta) = e^{-i\beta H_B}$$
$$U_P(\gamma) = e^{-i\gamma H_P}$$

operatori controllati dai parametri.
L'operatore U_M applica l'hamiltoniana H_B in base al valore del parametro β. Questo operatore è detto *mixer* e H_B è una hamiltoniana che applica ad ogni variabile la porta X di Pauli.

Esempio:
Per un sistema composto da quattro qubit:

$$H_B = \big(X_0\otimes I_1\otimes I_2\otimes I_3\big) + \big(I_0\otimes X_1\otimes I_2\otimes I_3\big) + \big(I_0\otimes I_1\otimes X_2\otimes I_3\big) + \big(I_0\otimes I_1\otimes I_2\otimes X_3\big)$$

L'operatore U_P applica l'hamiltoniana H_P in base al valore del parametro γ. H_P è l' hamiltoniana associata al problema.

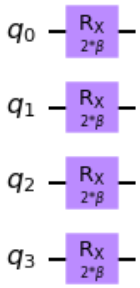

Figura 3.3: Operatore *mixer* $U_B(\beta)$

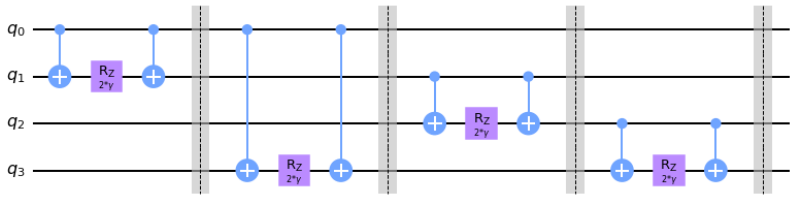

Figura 3.4: Circuito operatore associato ad un problema con $U_P(\gamma) = e^{-i\gamma H_P} = e^{-i\gamma Z_0 Z_1} e^{-i\gamma Z_1 Z_2} e^{-i\gamma Z_2 Z_3} e^{-i\gamma Z_0 Z_3}$

Come vediamo in Figura 3.3 possiamo rappresentare l'operatore mixer come delle porte di rotazione attorno all'asse X, in questo modo possiamo applicare H_B nella misura indicata dal parametro β. Lo stesso vale per le porte di rotazione attorno all'asse Z in Figura 3.4. In questo caso sfruttiamo anche il phase-kickback tramite porta CNOT per apportare modifiche alla fase di tutti i qubit che devono subirle in base a quanto specificato dall'hamiltoniana H_P.
In un una computazione adiabatica i parametri $(\vec{\beta}, \vec{\gamma})$ iniziali hanno: valori di β alti e valori bassi di γ, mentre quelli finali hanno: valori di β bassi e

valori alti di γ.
In Figura 3.5 vediamo i parametri ottimali per risolvere delle istanze di Max-Cut e notiamo che sfruttano la computazione adiabatica.

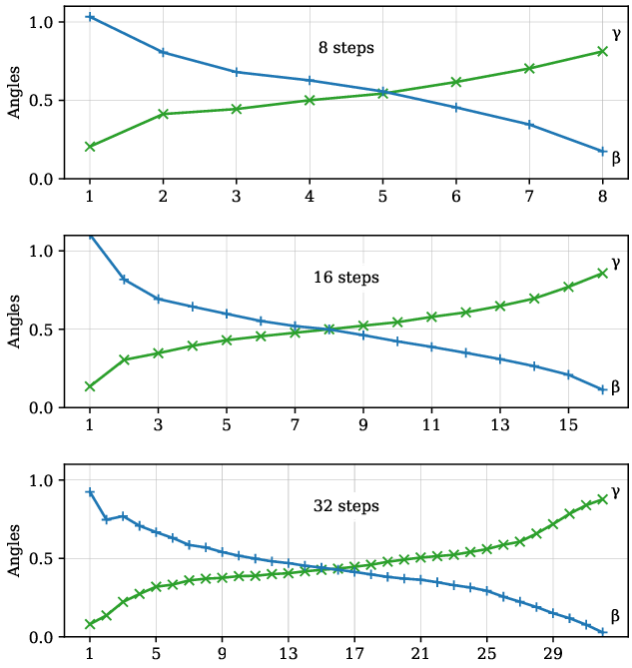

Figura 3.5: Valori ottimali per parametri $(\vec{\beta}, \vec{\gamma})$ in un problema di taglio massimo [5]

QAOA può anche operare sfruttando computazioni diabatiche [20]; ciò in alcuni casi ci permette di trovare soluzioni ottime a problemi con gap energetici molto piccoli in tempi minori rispetto a quelli che sarebbero richiesti nel quantum annealing [20][17].

Misurazione dei qubit

Viene misurato lo stato di tutti i qubit, il risultato della misurazione viene mandato all'ottimizzatore. Il circuito quantistico di QAOA con $p = 1$ e $U_P(\gamma) = e^{-i\gamma H_P} = e^{-i\gamma Z_0 Z_1} e^{-i\gamma Z_1 Z_2} e^{-i\gamma Z_2 Z_3} e^{-i\gamma Z_0 Z_3}$ è quindi:

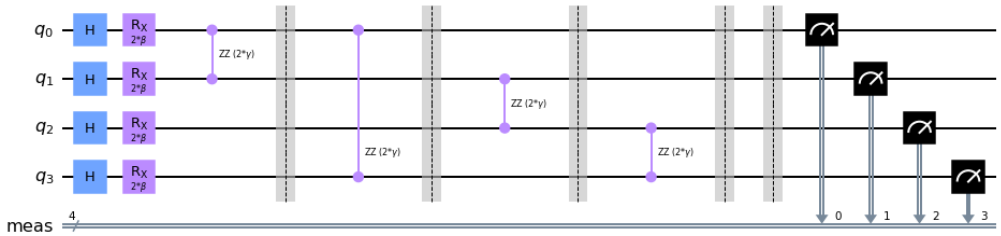

Figura 3.6: Esempio circuito QAOA

Nota: le porte ZZ in figura 3.3 sono equivalenti alle porte in figura 3.6, cambia solo la notazione.

Capitolo 4

Implementazione

Per implementare gli algoritmi precedentemente discussi sono stati usati:

- **Qiskit**, SDK Python sviluppato e mantenuto da IBM per eseguire programmi quantistici su: macchine basate su porte quantistiche reali e simulatori;
- **dwave**, SDK Python sviluppato e mantenuto da DWave per eseguire programmi quantistici su: quantum annealer reali e simulatori;
- **NumPy**, libreria Python per il calcolo scientifico.

I primi due mettono a disposizione macchine quantistiche reali in maniera gratuita, abbiamo però delle limitazioni:

- in Qiskit possiamo utilizzare solo macchine quantistiche aventi un massimo di cinque qubit;
- in dwave non possiamo usare la tecnologia VFYC e possiamo sfruttare le macchine quantistiche per un totale di un minuto al mese.

Analizzeremo le performance dei due metodi nella risoluzione del "**Binary Paintshop Problem**" accostandoli ad algoritmi classici per la risoluzione dello stesso problema.

4.1 Impostazione del problema

Il problema del negozio di vernici binario, descritto in 1.3.3, necessita dei seguenti dati iniziali:

- numero di automobili presenti sul rullo;
- disposizione dei diversi modelli di automobile.

Generiamo in maniera casuale questi dati:

```
CAR_PAIR_COUNT = 5
CAR_SEQUENCE = np.random.permutation([x for x in range(
    CAR_PAIR_COUNT)] * 2)
```

Il risultato è una lista di 10 automobili identificate dallo stesso numero intero se appartenenti allo stesso modello.

Esempio:

$$[3, 1, 2, 0, 2, 3, 1, 0, 4, 4]$$

Fatto ciò creiamo i vettori h e J richiesti dal modello di Ising tramite la funzione *init_coeff*:

```
def init_coeff():
    same = -1
    different = 1
    appeared_already = set()
    for car0, car1 in zip(CAR_SEQUENCE[:-1], CAR_SEQUENCE
    [1:]):
        if car0 == car1:
            continue
        if car0 in appeared_already:
            appeared_already.add(car0)
            if car1 in appeared_already:
                yield [(car0, car1), same]
            else:
                yield [(car0, car1), different]
```

```
14          else:
15              appeared_already.add(car0)
16              if car1 in appeared_already:
17                  yield [(car0, car1), different]
18              else:
19                  yield [(car0, car1), same]
```

Dopo alcune elaborazioni di questi dati abbiamo h e J sotto forma di dizionari. h è un dizionario vuoto e J è un dizionario che ha:

- per chiave una tupla che indica la coppia di variabili a cui è associato il valore;
- per valore un numero intero.

4.2 QAOA

Qiskit mette a disposizione una classe apposita per implementare l'algoritmo QAOA. Questa è figlia della classe VQE (*Variational Quantum Eigensolver*), da cui eredita svariati aspetti, tra cui, l'implementazione della classe *MinimumEigensolver*. Quest'ultima ci consente di sfruttare tutti gli ottimizzatori messi a disposizione dalla libreria NumPy.
La classe QAOA in fase di inizializzazione richiede che il problema da risolvere venga espresso sotto forma di funzione QUBO in un oggetto di tipo *QuadraticProgram*.
Costruiamo l'oggetto *qubo* di tipo *QuadraticProblem*:

```
1 qubo_temp, _ = ising_to_qubo(h, J)
2
3 linear_qubo = {(i1):qubo_temp[i1, i2]  for i1, i2 in
     qubo_temp.keys() if i1 == i2}
4 quadratic_qubo = {(i1, i2):qubo_temp[i1, i2]  for i1, i2 in
     qubo_temp.keys() if i1 != i2}
5
6 qubo = QuadraticProgram()
7 for i in range(CAR_PAIR_COUNT):
```

```
    qubo.binary_var(f"car_{str(i)}")

qubo.minimize(linear=linear_qubo, quadratic=quadratic_qubo)
```

Successivamente possiamo passare *qubo* alla classe QAOA, ma prima dobbiamo decidere come impostare tutti gli altri parametri di QAOA.
Dobbiamo infatti scegliere:

- *backend* - su quale macchina o simulatore far eseguire il nostro programma;
- *reps* - nella letteratura inerente QAOA corrisponde al parametro p;
- *optimizer* - quale ottimizzatore classico utilizzare (se omesso userà l'ottimizzatore di default di VQE);
- *initial_point* - valori iniziali per i vettori $\vec{\beta}$ e $\vec{\gamma}$, inizializzare opportunamente QAOA può portare a risultati qualitativamente migliori [20][17] (se omessò userà valori iniziali di default).

Creiamo un istanza di QAOA con:

- *backend* = *ibmq_santiago*, macchina quantistica messa a disposizione da IBM;
- *reps* = 3.

```
provider = IBMQ.load_account()
backend = provider.backend.ibmq_santiago
quantum_instance = QuantumInstance(backend)
qaoa_mes = QAOA(quantum_instance=quantum_instance, reps=3)

qaoa = MinimumEigenOptimizer(qaoa_mes)
```

Non ci resta che eseguire il programma ed ottenere i risultati:

```
qaoa_results = qaoa.solve(qubo)
```

4.3 Quantum Annealing

La libreria *dwave* mette a disposizione tre diversi mezzi per eseguire programmi quantistici:

- *ExactSolver* - per fare dei semplici test sulla propria CPU locale;
- *DWaveSampler* - per eseguire programmi su macchine quantistiche reali;
- *LeapHybridSampler* - il programma viene svolto da una macchina ibrida. Permette di effettuare *minor embedding* in maniera completamente automatica per problemi che usano fino a sessantaquattro qubit.

DWaveSampler e *LeapHybridSampler* possono sfruttare dei processori quantistici aventi fino a 2000 qubit.
Dopo aver scelto un sampler possiamo mandare il programma in esecuzione fornendo i dizionari h e J.
A seconda del sampler possono essere impostati diversi parametri.

ExactSolver:

```
sampler = ExactSolver()
sample = sampler.sample_ising(h, J)
```

DWaveSampler:

```
sampler = EmbeddingComposite(DWaveSampler(solver={'
    topology__type': 'chimera'}))
sample = sampler.sample_ising(h, J, num_reads=1024)
```

Il minor embedding può essere fatto in maniera automatica o manuale, in questo caso per sfruttare il minor embedding automatico dobbiamo usare la funzione *EmbeddingComposite*. Il parametro "*topology__type*" serve per indicare la topologia del processore quantistico che si vuole utilizzare ed il parametro "num_reads" serve per indicare quante volte dev essere eseguito il programma.

LeapHybridSampler:

```
sampler = LeapHybridSampler(solver={'category': 'hybrid'})
sample = sampler.sample_ising(h, J)
```

Il minor embedding viene svolto in maniera automatica per problemi che usano fino a sessantaquattro qubit.

4.4 NumPyMinimumEigensolver

Usando lo stesso *QuadraticProgram* di QAOA è possibile usare dei metodi della libreria NumPy per affrontare questa tipologia di problemi.

```
np_mes = NumPyMinimumEigensolver()
np_min = MinimumEigenOptimizer(np_mes)
np_result = np_min.solve(qubo)
```

4.5 Algoritmo greedy

L'algoritmo greedy proposto per risolvere i problemi di binary paintshop inizia a scorrere la lista assegnando alla prima auto un colore. Continua a scorrere la lista senza cambiare colore fin quando non è necessario (fin quando non incontra un auto di un modello già incontrato). In quel caso cambia colore e continua con il nuovo colore fin quando può. Continua in questo modo fino a quando non raggiunge la fine della lista.

```
def color_changes_greedy(paint_bitstring, echo=False):
    paint_zero= set()
    paint_one = set()
    status = 0
    counter = 0
    path = []
    result = np.zeros(len(paint_bitstring) // 2)
    for i in range(len(paint_bitstring)):
        path.append(status)
```

```
10          if (paint_bitstring[i] not in paint_zero) and (
    paint_bitstring[i] not in paint_one):
11              result[paint_bitstring[i]] = status
12
13          if status == 0 and (paint_bitstring[i] not in
    paint_zero):
14              paint_zero.add(paint_bitstring[i])
15          elif status == 0 and (paint_bitstring[i] in
    paint_zero):
16              counter += 1
17              status = 1
18              paint_one.add(paint_bitstring[i])
19
20          elif status == 1 and (paint_bitstring[i] in paint_one
    ):
21              counter += 1
22              status = 0
23              paint_zero.add(paint_bitstring[i])
24
25          elif status == 1 and (paint_bitstring[i] not in
    paint_one):
26              paint_one.add(paint_bitstring[i])
27          if echo:
28              print(i,status, CAR_SEQUENCE[i])
29
30      return list(map(int, result)), counter
```

4.6 Algoritmo Casuale

L'algoritmo casuale scorre la lista assegnando in maniera casuale quale colore assegnare ad una auto scegliendo tra i colori disponibili per quell' auto.

```
1 def color_changes_random(paint_bitstring, echo=False):
2     paint_zero= set()
3     paint_one = set()
4     status = np.random.randint(0,2)
5     counter = 0
```

```
    path = []
    result = np.zeros(len(paint_bitstring) // 2)
    for i in range(len(paint_bitstring)):
        path.append(status)
        if (paint_bitstring[i] not in paint_zero) and (paint_bitstring[i] not in paint_one):
            result[paint_bitstring[i]] = np.random.randint(0,2)

        if status == 0 and (paint_bitstring[i] not in paint_zero):
            paint_zero.add(paint_bitstring[i])
        elif status == 0 and (paint_bitstring[i] in paint_zero):
            counter += 1
            status = 1
            paint_one.add(paint_bitstring[i])

        elif status == 1 and (paint_bitstring[i] in paint_one):
            counter += 1
            status = 0
            paint_zero.add(paint_bitstring[i])

        elif status == 1 and (paint_bitstring[i] not in paint_one):
            paint_one.add(paint_bitstring[i])
        if echo:
            print(i,status, CAR_SEQUENCE[i])

    return list(map(int, result)), counter
```

Capitolo 5

Analisi

Prima di eseguire i programmi è necessario decidere su quali dispositivi questi verranno eseguiti.
Gli approcci greedy, casuale e tramite *NumPyMinimumEigensolver* verranno eseguiti sulla CPU locale dato che con un numero di variabili ridotto (5-30) anche un computer classico non particolarmente potente riesce a portare a termine la computazione nel giro di pochi minuti.
Possiamo eseguire programmi su dei veri quantum annealer; useremo il LeapHybridSampler che permetterà di sottomettere in maniera semplice problemi che necessitano di un massimo di 64 qubit.
Tramite Qiskit possiamo sottomettere problemi a macchine quantistiche reali con massimo cinque qubit; quindi possiamo utilizzare un simulatore che possa sfruttare il modello di rumore di una macchina reale.

5.1 Configurazione del simulatore per QAOA

Qiskit mette a disposizione svariati simulatori più o meno efficienti e precisi. Il simulatore che imiterebbe in maniera più fedele il comportamento di una macchina reale è "*qasm_simulator*"; questo simulatore ha un suo modello di rumore ma può anche sfruttare modelli di macchine quantistiche reali.

In alternativa possiamo utilizzare un simulatore che non introduca rumore e che lavori in maniera esatta con il vettore di stato. Questo ha prestazioni migliori rispetto ai simulatori precedentemente descritti ma simula un sistema quantistico ideale, privo di rumore.
Visto che l'algoritmo QAOA permette di usare circuiti brevi, gli effetti del rumore su questi ultimi sono da analizzare per poter scegliere un simulatore che:

- dia risultati coerenti con quelli che restituirebbero macchine quantistiche reali;
- porti a termine la computazione in tempi ragionevoli.

Creazione dei simulatori

qasm_simulator con modello di rumore della macchina *imbq_montreal*:

```
noise_model = NoiseModel.from_backend(FakeMontreal())
backend = Aer.get_backend('qasm_simulator')
quantum_instance_montreal = QuantumInstance(backend,
    noise_model=noise_model)
```

qasm_simulator con modello di rumore di default:

```
backend = Aer.get_backend('qasm_simulator')
quantum_instance_qasm = QuantumInstance(backend)
```

qasm_simulator senza rumore:

```
backend = QasmSimulator(method='statevector')
quantum_instance_statevector = QuantumInstance(backend)
```

Comparazione dei risultati

Per testare l'accuratezza dei modelli sono state eseguite 20 istanze diverse del *binary paintshop problem* con 25 variabili ed un numero alto di passi p. Il numero di step p è grande perchè più il circuito è lungo, più sono evidenti gli effetti del rumore sui dati.

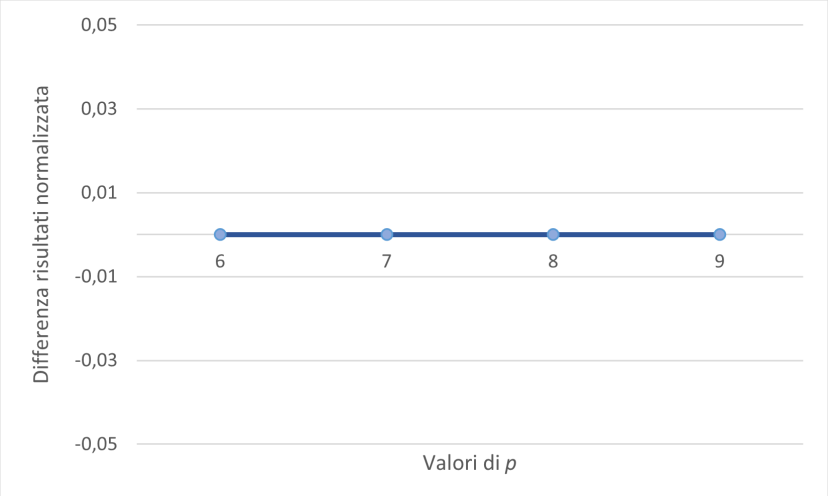

Figura 5.1: Differenza normalizzata dei risultati di qasm con noisemodel *ibmq_montreal* e noisemodel di default

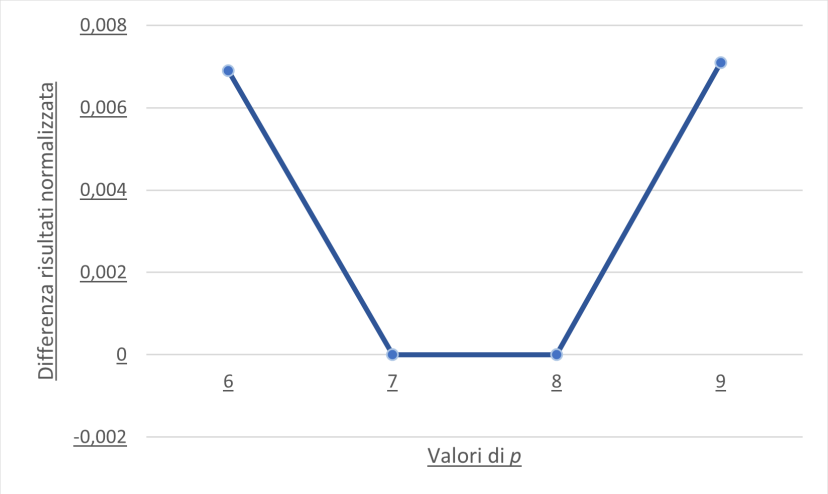

Figura 5.2: Differenza normalizzata dei risultati di qasm con noisemodel *ibmq_montreal* e assenza di rumore

Dalla Figura 5.1 si nota che non c'è una differenza evidente tra i risultati dei simulatori con noise model di default di qasm e quelli con noise model di *ibmq_montreal*. Il simulatore che non introduce rumore, invece, ha restituito

dati lievemente diversi da quelli che immettono del rumore. Tuttavia la differenza è molto piccola ($\sim 0.7\%$) nonostante siano circuiti relativamente lunghi (esposti a molto rumore). Visto che useremo circuiti più brevi, su cui l'effetto del rumore sarà ancora minore, decidiamo di usare il simulatore che non immette rumore per ridurre i tempi di calcolo per le simulazioni.

5.2 Struttura del programma

Il programma è stato implementato in un Jupyter Notebook strutturato in questo modo:

1. dichiarazione di funzioni necessarie a risolvere istanze di un problema di binary paintshop;
2. creazione dati iniziali;
3. risoluzione del problema tramite l'impiego delle funzioni e dei metodi precedentemente descritti;
4. salvataggio dei risultati in memoria secondaria.

Sono state eseguite istanze di QAOA con valori di $p \in [1, 4]$ in quanto è stato provato che, per alcuni problemi, aumentare il valore del parametro p permette di ottenere risultati migliori [20].
Ogni esecuzione del programma quindi genera i dati iniziali del problema per poi risolverlo con i metodi descritti in precedenza.

5.3 Risultati

I valori in Figura 5.3 sono dati dalla media aritmetica dei risultati ottenuti da dieci esecuzioni del programma per ogni numero di variabili (corrispondente al numero di modelli di automobile) del problema.
Alcuni metodi sono stati testati per problemi grandi fino a 64 variabili. Ciò è stato possibile in quanto:

- gli algoritmi "casuale" e "greedy" hanno un costo computazionale molto basso;

- il metodo "QA" sfrutta il *LeapHybridSampler* (discusso in 4.3).

Invece i metodi NumPy e QAOA sono stati testati fino ad un massimo di 25 variabili. Dopo questa soglia i programmi non sono eseguibili dalle macchine a mia disposizione (memoria richiesta maggiore di 16 GiB).

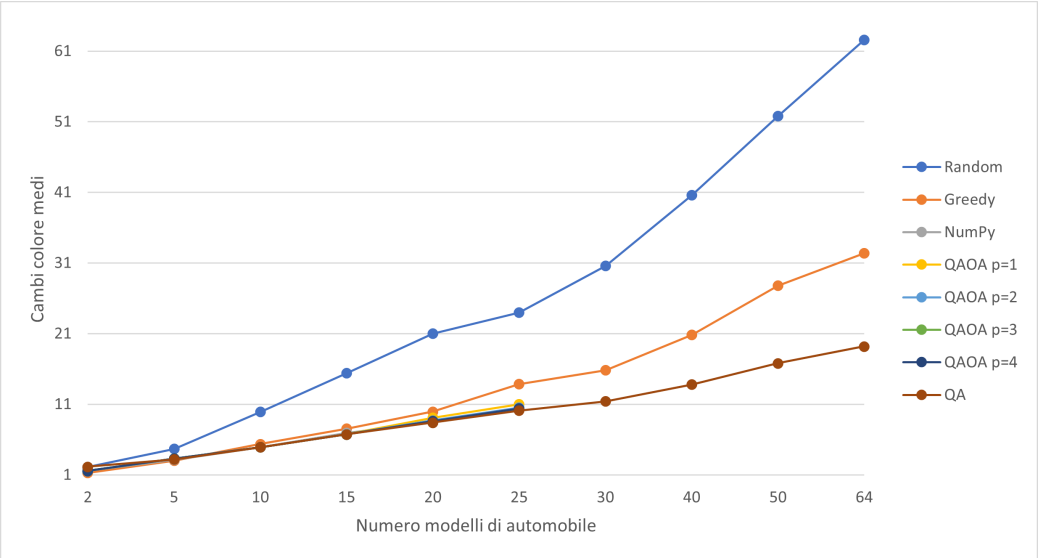

Figura 5.3: Media risultati per diversi numeri di variabili

In Figura 5.3 notiamo che gli algoritmi quantistici ed il metodo che sfrutta solo la libreria NumPy danno risultati simili per problemi che usano fino a 25 variabili. Per problemi più grandi notiamo che il risultato fornito da QA non aumenta in maniera repentina. Ci aspetteremmo di ottenere risultati simili utilizzando QAOA con un valore adeguato di p visto che QAOA, come QA, può sfruttare la computazione adiabatica [11].

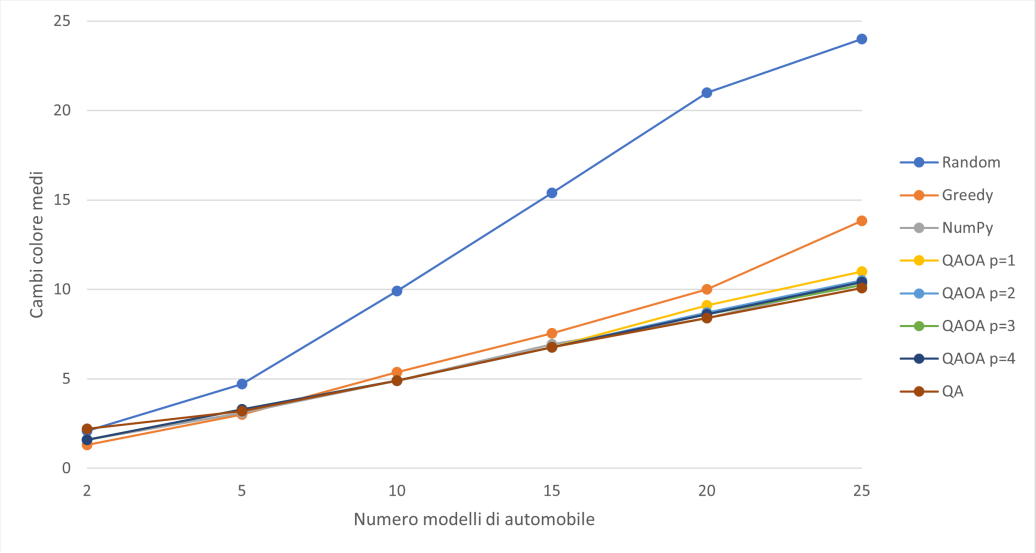

Figura 5.4: Media risultati per numero di variabili fino a 25

In Figura 5.4 notiamo che per un numero ridotto di variabili (fino a ~ 15), tutti i metodi (tranne l'algoritmo casuale) forniscono soluzioni qualitativamente simili. Da 20 variabili in poi notiamo che il risultato dell'algoritmo greedy inizia ad allontanarsi sempre più dai risultati degli algoritmi quantistici presi in considerazione. Il metodo che sfrutta solo la libreria NumPy è perfettamente in linea con gli algoritmi quantistici analizzati.

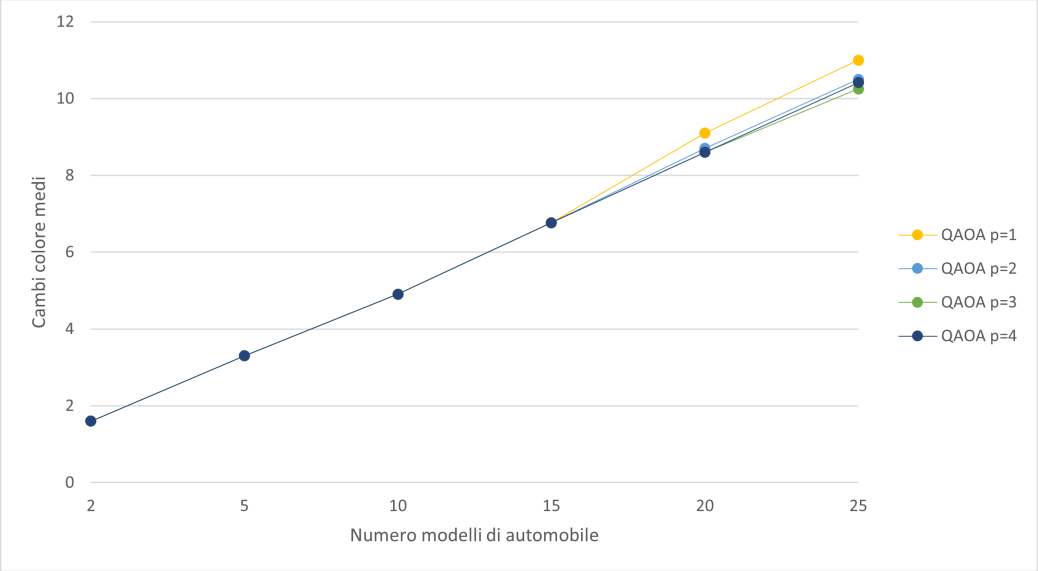

Figura 5.5: Media risultati QAOA per numero di variabili fino a 25

In Figura 5.5 sono riportati solo i risultati di QAOA; da questi notiamo che per diversi numeri di passi p si ottengono risultati diversi. Infatti è chiaro che, per problemi composti da 20 e 25 variabili, QAOA con $p = 1$ restituisce risultati peggiori rispetto ad istanze di QAOA con valori di p maggiori (che danno risultati simili tra loro).
Se fosse stato possibile affrontare problemi con un numero di variabili maggiore avremmo potuto osservare delle differenze più evidenti al variare del parametro p di QAOA.

Capitolo 6

Conclusioni

Dai dati analizzati nei capitoli precedenti si evince che, sebbene non privi di errori, i metodi per risolvere problemi di ottimizzazione combinatoria binaria che sfruttano macchine quantistiche possono restituire soluzioni qualitativamente comparabili a quelle ottenibili tramite algoritmi classici.
Sia il quantum annealing che QAOA consentono di risolvere in maniera efficiente problemi di ottimizzazione su macchine quantistiche attuali. Questo non è scontato, basti pensare che molti algoritmi quantistici sono stati teorizzati decenni prima della creazione delle prime macchine quantistiche e che tutt'ora esistono algoritmi quantistici che necessitano di un numero di qubit talmente elevato da essere inutilizzabili con la tecnologia attualmente a nostra disposizione.

Appendice A

Programma in formato Jupyter Notebook

Nelle pagine seguenti è riportato il codice Python utilizzato per ottenere i risultati mostrati nel grafico in Figura 5.3.
Il programma è stato implementato in un Jupyter Notebook in quanto la separazione in celle e la possibilità d'inserire annotazioni in formato Markdown rendono il programma più leggibile.
Ogni esecuzione del programma genera e risolve un'istanza del problema del negozio di vernici binario.

1 Dichiarazione di funzioni necessarie a risolvere istanze di un problema di binary paintshop

[ ]:
```
import numpy as np
from itertools import product
import copy
import matplotlib.pyplot as plt
from scipy import optimize

from qiskit.algorithms import QAOA, NumPyMinimumEigensolver
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import Aer, execute
from qiskit.utils import algorithm_globals, QuantumInstance
from qiskit.circuit import Parameter
from qiskit_optimization.algorithms import MinimumEigenOptimizer,␣
↪RecursiveMinimumEigenOptimizer
from qiskit_optimization import QuadraticProgram
from qiskit import IBMQ
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer import QasmSimulator
from qiskit.test.mock import FakeMontreal

from dimod.utilities import ising_to_qubo
from dimod.reference.samplers import ExactSolver
from dwave.system import DWaveSampler, EmbeddingComposite
```

[ ]:
```
def color_changes(paint_bitstring):
    color_sequence = []
    painted_once = set()
    for car in CAR_SEQUENCE:
        if car in painted_once:
            color_sequence.append(not paint_bitstring[car])
        else:
            color_sequence.append(paint_bitstring[car])
            painted_once.add(car)
    paint_change_counter = 0
    for color0, color1 in zip(color_sequence[:-1], color_sequence[1:]):
        if color0 != color1:
            paint_change_counter += 1
    return paint_change_counter
```

[ ]:
```
def init_coeff():
    same = -1
    different = 1
    appeared_already = set()
    for car0, car1 in zip(CAR_SEQUENCE[:-1], CAR_SEQUENCE[1:]):
```

```
        if car0 == car1:
            continue
        if car0 in appeared_already:
            appeared_already.add(car0)
            if car1 in appeared_already:
                yield [(car0, car1), same]
            else:
                yield [(car0, car1), different]
        else:
            appeared_already.add(car0)
            if car1 in appeared_already:
                yield [(car0, car1), different]
            else:
                yield [(car0, car1), same]
```

[ ]:
```
def coeff_to_dict(lista):
    quad = dict()
    for item in lista:
        if (item[0][0], item[0][1]) in quad.keys():
            quad[item[0][0], item[0][1]] += item[1]
        elif (item[0][1], item[0][0]) in quad:
            quad[item[0][1], item[0][0]] += item[1]
        else:
            quad[item[0][0], item[0][1]] = item[1]
    return quad
```

[ ]:
```
def color_changes_from_initial(initial, echo=False):
    counter = 0
    dynamic_sequence = initial
    dynamic_sequence

    for i in range(CAR_PAIR_COUNT*2 - 1):
        if (dynamic_sequence[CAR_SEQUENCE[i]] != dynamic_sequence[CAR_SEQUENCE[i␣
↪+ 1]])\
            or\
            (CAR_SEQUENCE[i] == CAR_SEQUENCE[i + 1]):
            counter += 1
        if echo:
            print(i,dynamic_sequence[CAR_SEQUENCE[i]], CAR_SEQUENCE[i])
        dynamic_sequence[CAR_SEQUENCE[i]] = ␣
↪int((dynamic_sequence[CAR_SEQUENCE[i]]) ^ True)
        #print(CAR_SEQUENCE[i], CAR_SEQUENCE[i+1], dynamic_sequence, counter)
    return counter
```

QAOA:

[ ]:
```
def exec_qaoa(p, noisy=False):
    noise_model = NoiseModel.from_backend(FakeMontreal())
#     algorithm_globals.random_seed = 42
    algorithm_globals.massive=True

    if noisy:
        noise_model = NoiseModel.from_backend(FakeMontreal())

        backend = Aer.get_backend('qasm_simulator')
#         quantum_instance = QuantumInstance(backend,
#                                            seed_simulator=algorithm_globals.
 ↪random_seed,
#                                            seed_transpiler=algorithm_globals.
 ↪random_seed,
#                                            noise_model=noise_model)
        quantum_instance = QuantumInstance(backend,
                                           seed_simulator=algorithm_globals.
 ↪random_seed,
                                           seed_transpiler=algorithm_globals.
 ↪random_seed)

        qaoa_mes = QAOA(quantum_instance=quantum_instance, reps=p,␣
 ↪include_custom = True)
        qaoa_noisy = MinimumEigenOptimizer(qaoa_mes)
        return qaoa_noisy.solve(qubo)
    else:
        backend = QasmSimulator(method='statevector')
        quantum_instance = QuantumInstance(backend,
                                           seed_simulator=algorithm_globals.
 ↪random_seed,
                                           seed_transpiler=algorithm_globals.
 ↪random_seed)
        qaoa_mes = QAOA(quantum_instance=quantum_instance, reps=p,␣
 ↪include_custom = True)
        qaoa_noiseless = MinimumEigenOptimizer(qaoa_mes)    # using QAOA
        return qaoa_noiseless.solve(qubo)
```

Greedy:

[ ]:
```
def color_changes_greedy(paint_bitstring, echo=False):
    paint_zero= set()
    paint_one = set()
    status = 0
    counter = 0
    path = []
    result = np.zeros(len(paint_bitstring) // 2)
    for i in range(len(paint_bitstring)):
```

```
        path.append(status)
        if (paint_bitstring[i] not in paint_zero) and (paint_bitstring[i] not in␣
↪paint_one):
            result[paint_bitstring[i]] = status

        if status == 0 and (paint_bitstring[i] not in paint_zero):
            paint_zero.add(paint_bitstring[i])
        elif status == 0 and (paint_bitstring[i] in paint_zero):
            counter += 1
            status = 1
            paint_one.add(paint_bitstring[i])

        elif status == 1 and (paint_bitstring[i] in paint_one):
            counter += 1
            status = 0
            paint_zero.add(paint_bitstring[i])

        elif status == 1 and (paint_bitstring[i] not in paint_one):
            paint_one.add(paint_bitstring[i])
        if echo:
            print(i,status, CAR_SEQUENCE[i])

    return list(map(int, result)), counter
```

NumPy:

[ ]:
```
def exec_np():
    exact_mes = NumPyMinimumEigensolver()
    exact = MinimumEigenOptimizer(exact_mes)  # using the exact classical numpy␣
↪minimum eigen solver
    return exact.solve(qubo)
```

Casuale:

[ ]:
```
def color_changes_random(paint_bitstring, echo=False):
    paint_zero= set()
    paint_one = set()
    status = np.random.randint(0,2)
    counter = 0
    path = []
    result = np.zeros(len(paint_bitstring) // 2)
    for i in range(len(paint_bitstring)):
        path.append(status)
        if (paint_bitstring[i] not in paint_zero) and (paint_bitstring[i] not in␣
↪paint_one):
            result[paint_bitstring[i]] = np.random.randint(0,2)
```

```
        if status == 0 and (paint_bitstring[i] not in paint_zero):
            paint_zero.add(paint_bitstring[i])
        elif status == 0 and (paint_bitstring[i] in paint_zero):
            counter += 1
            status = 1
            paint_one.add(paint_bitstring[i])

        elif status == 1 and (paint_bitstring[i] in paint_one):
            counter += 1
            status = 0
            paint_zero.add(paint_bitstring[i])

        elif status == 1 and (paint_bitstring[i] not in paint_one):
            paint_one.add(paint_bitstring[i])
        if echo:
            print(i,status, CAR_SEQUENCE[i])

    return list(map(int, result)), counter
```

2 Creazione dati iniziali

[ ]:
```
CAR_PAIR_COUNT = 5
CAR_SEQUENCE = np.random.permutation([x for x in range(CAR_PAIR_COUNT)] * 2)

CAR_SEQUENCE
```

[ ]:
```
linear = [0]*CAR_PAIR_COUNT

quadratic = coeff_to_dict(list(init_coeff()))

J = quadratic
h = {}
qubo_temp, _ = ising_to_qubo(h, J)
```

[ ]:
```
linear_qubo = {(i1):qubo_temp[i1, i2]  for i1, i2 in qubo_temp.keys() if i1 ==␣
 ↪i2}
quadratic_qubo = {(i1, i2):qubo_temp[i1, i2]  for i1, i2 in qubo_temp.keys() if␣
 ↪i1 != i2}
```

Creazione oggetto *QuadraticProgram*:

[ ]:
```
qubo = QuadraticProgram()
for i in range(CAR_PAIR_COUNT):
    qubo.binary_var(f"car_{str(i)}")

qubo.minimize(linear=linear_qubo, quadratic=quadratic_qubo)
```

3 Risoluzione del problema

3.1 QAOA:

[ ]:
```
max_p = 4

best_result_qaoa_noisy = np.zeros((max_p+1, CAR_PAIR_COUNT), dtype=int)
best_result_qaoa_noiseless = np.zeros((max_p+1, CAR_PAIR_COUNT), dtype=int)

for p in range(1, max_p+1):
    print(f"Evaluating P={p} noisy")
    best_result_qaoa_noisy[p] = list(map(int, exec_qaoa(p, True).x))
    print(f"Evaluating P={p} noise-less")
    best_result_qaoa_noiseless[p] = list(map(int, exec_qaoa(p, False).x))
```

3.2 NumPy:

[ ]:
```
best_result_np = list(map(int, exec_np().x))
```

3.3 QA:

[ ]:
```
qa_sim = True             #use a simulator if True
if qa_sim:
    sampler = ExactSolver()
    sample = sampler.sample_ising(h, J)
else:
    from dwave.system import LeapHybridSampler
    sampler_auto = LeapHybridSampler(solver={'category': 'hybrid'})
    sample = sampler_auto.sample_ising(h, J)

best_result_qa = list(map(lambda x : int(x>0), list(sample.first.sample.
 ↪values())))

# sample.to_pandas_dataframe().sort_values("energy")
```

3.4 Greedy:

[ ]:
```
result_greedy, CC_greedy = color_changes_greedy(CAR_SEQUENCE, echo=False)
```

4 Salvataggio dei risultati in memoria secondaria:

[ ]:
```
random_r = color_changes_random(CAR_SEQUENCE)
CC_random = color_changes_from_initial(random_r[0])
```

```
reference_res = color_changes(CAR_SEQUENCE)
random_res = CC_random
greedy_res = color_changes_from_initial(result_greedy.copy())

qa_res = color_changes_from_initial(best_result_qa.copy())
np_res = color_changes_from_initial(best_result_np.copy())
```

[ ]:
```
print(f"Number of car pairs:\t{CAR_PAIR_COUNT}")
print(f"Car sequence:\t\t{CAR_SEQUENCE}\n")

print(f"Random Solution:\t\t{random_r[0]}\t Color changes:{random_res}")
print(f"Greedy Solution:\t\t{result_greedy}\t Color changes:{greedy_res}")

for p in range(1, max_p + 1):
    print(f"QAOA Noisy \tp={p} Solution:\t{list(best_result_qaoa_noisy[p])}"\
        f"\t Color changes:{color_changes_from_initial(best_result_qaoa_noisy[p].
 ↪copy())}")
    print(f"QAOA Noise-Less p={p} Solution:
 ↪\t{list(best_result_qaoa_noiseless[p])}"\
        f"\t Color changes:
 ↪{color_changes_from_initial(best_result_qaoa_noiseless[p].copy())}")
print(f"QA   Solution:  \t\t{best_result_qa}\t Color changes:{qa_res}")
print(f"NumPy   Solution:  \t\t{best_result_np}\t Color changes:{np_res}")
```

[ ]:
```
from pandas import DataFrame
to_print = DataFrame({"Sequence": [str(CAR_SEQUENCE)], "Random":random_res,\
                      "Greedy": greedy_res, "NumPy": np_res})
for p in range(1, max_p + 1):
    to_print[f"QAOA_p{p}_noisy"] =␣
 ↪color_changes_from_initial(best_result_qaoa_noisy[p].copy())
    to_print[f"QAOA_p{p}_noiseless"] =␣
 ↪color_changes_from_initial(best_result_qaoa_noiseless[p].copy())
to_print["QA"] = qa_res
to_print
```

[ ]:
```
filename = f'values_{CAR_PAIR_COUNT}.csv'
import os

if os.path.exists(filename):
    header = False
else:
    header = True

with open(filename, 'a') as f:
    to_print.to_csv(filename, header=header, mode='a', index=False)
```

Bibliografia

[1] B. Apolloni, C. Carvalho, and D. de Falco. Quantum stochastic optimization. *Stochastic Processes and their Applications*, 33(2):233–244, 1989.

[2] R. L. Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, 37(2):194–197, 1941.

[3] A.R. Calderbank, E.M. Rains, P.M. Shor, and N.J.A. Sloane. Quantum error correction via codes over gf(4). *IEEE Transactions on Information Theory*, 44(4):1369–1387, 1998.

[4] Jaeho Choi and Joongheon Kim. A tutorial on quantum approximate optimization algorithm (qaoa): Fundamentals and applications. *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 138–142, 2019.

[5] Gavin E. Crooks. Performance of the quantum approximate optimization algorithm on the maximum cut problem. *arXiv: Quantum Physics*, 2018.

[6] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, 1992.

[7] Austin G. Fowler, Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3), Sep 2012.

[8] S. Johnson G. Dantzig, R. Fulkerson. Solution of a large-scale traveling-salesman problem, Novembre 1954.

[9] Giovanni Gallavotti. Statistical mechanics. *Springer, Berlin, Heidelberg*, 1999.

[10] Daniel Gottesman. The Heisenberg Representation of Quantum Computers. *arXiv e-prints*, pages quant–ph/9807006, July 1998.

[11] Tosio Kato. On the adiabatic theorem of quantum mechanics. *Journal of the Physical Society of Japan*, 5(6):435–439, 1950.

[12] Richard E. Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2):181–203, 1998.

[13] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961.

[14] Jan van Leeuwen. Algorithms and complexity. *Elsevier*, 1998.

[15] Michael A Nielsen and Chuang Isaac L. Quantum computation and quantum information. *Cambridge Univ. Press*, 2007.

[16] Martin Pincus. Letter to the editor—a monte carlo method for the approximate solution of certain types of constrained optimization problems. operations research 18(6):1225-1228. 1970.

[17] Stefan H. Sack and Maksym Serbyn. Quantum annealing initialization of the quantum approximate optimization algorithm. *Quantum*, 5:491, July 2021.

[18] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring, 1994.

[19] Michael Streif, Sheir Yarkoni, Andrea Skolik, Florian Neukart, and Martin Leib. Beating classical heuristics for the binary paint shop problem with the quantum approximate optimization algorithm. *Physical Review A*, 104(1), Jul 2021.

[20] Leo Zhou, Sheng-Tao Wang, Soonwon Choi, Hannes Pichler, and Mikhail D. Lukin. Quantum approximate optimization algorithm: Performance, mechanism, and implementation on near-term devices. *Physical Review X*, 10(2), Jun 2020.