Progetto di Quantum Machine Learning

Anno Accademico 2024/2025

Luigi Tarasi Rossella Balice

1 Introduzione

L'obiettivo del nostro progetto è quello di risolvere il problema del graph coloring attraverso tecniche QUBO (Quadratic Unconstrained Binary Optimization) di Quantum Machine Learning. Il protocollo scelto è quello del QAOA (Quantum Alternate Optimization Algorithm). Com'è noto, QUBO è un modello di ottimizzazione generico che può rappresentare moltissimi problemi NP-completi; questi ultimi possono essere tradotti naturalmente in Hamiltoniani diagonali, la cui minimizzazione può avvenire su simulatori quantistici senza troppe difficoltà. Il vantaggio è euristico: non garantito, ma in alcuni casi specifici la risoluzione può essere più efficiente.

2 Fondamenti teorici

2.1 Problema del Graph Coloring

Il Graph Coloring è un problema *NP-hard*; esso consiste nel determinare il *numero cromatico* di un qualsiasi grafo non orientato G = (V,E), ovvero il numero minimo di colori k con cui è possibile colorare tutti i n vertici in modo che nessun arco colleghi due vertici dello stesso colore. L'Hamiltoniano per il suddetto problema può essere costruito nel modo seguente: denotiamo con $x_{v,i}$ la nostra variabile binaria che vale 1 se il vertice v è colorato con il colore i, e 0 altrimenti. La funzione costo sarà dunque:

$$H = A\sum_{v=1}^{n}\left(1-\sum_{i=1}^{k}x_{v,i}\right)^2 + A\sum_{(u,v)\in E}\sum_{i=1}^{k}x_{u,i}x_{v,i} \tag{1}$$

Il primo termine impone il vincolo che ogni vertice abbia esattamente un colore e introduce una penalità energetica ogni volta che questo viene violato. Il secondo termine invece introduce una penalità ogni volta che un arco collega due vertici dello stesso colore. Si noti che la funzione sopra scritta è quadratica in x e tutte le variabili sono binarie (caratteristica di ogni problema QUBO). Per risolvere il problema utilizzando un registro quantistico, sono state effettuate e confrontate diverse scelte di encoding: inizialmente, tramite *one-hot encoding*, il numero di qubits utilizzati è stato $n \cdot k$; si è provato poi ad ottimizzare utilizzando $n \cdot \log_2(k)$ qubits, tramite encoding binario, modificando leggermente l'Hamiltoniano.

2.2 Data encoding su circuiti quantistici

Nei circuiti quantistici variazionali l'encoding dei dati è cruciale. La scelta della codifica determina infatti:

- Il numero di qubits richiesti;
- La dimensione dello spazio di Hilbert;
- La complessità dei vincoli da imporre;

- L'efficienza dell'ottimizzazione.

La codifica deve trasformare variabili discrete (classiche) in stati quantistici di base, mantenendo la semantica del problema.

2.3 One-Hot encoding

Ogni nodo è rappresentato da k qubits, uno per colore. Un nodo ha dunque esattamente uno dei suoi qubits attivi e gli altri spenti. Lo spazio di Hilbert avrà dimensione 2^{nk}.

2.3.1 Esempio con n=3 nodi, k=3 colori

- Nodo 0 $\rightarrow [1, 0, 0]$ (colore 0)
- Nodo 1 $\rightarrow [0, 1, 0]$ (colore 1)
- Nodo 2 $\rightarrow [0, 0, 1]$ (colore 2)

Bitstring totale: 100 010 001 ($n \cdot k = 9$ qubits)
Questa codifica è la più intuitiva e i vincoli da imporre sono semplici (l'Hamiltoniano è quello in eq. 1), ma di contro è la meno efficiente in quanto a numero di qubits, e ci aspettiamo maggiori limitazioni per valori grandi di n, k.

2.4 Binary encoding

L'idea del binary encoding è di utilizzare la rappresentazione binaria come codifica dell'intero colore. Per k colori serviranno $\lceil \log_2 k \rceil$ qubits per nodo. Non c'è bisogno di imporre che ogni nodo debba avere esattamente un colore poiché è intrinseco nell'encoding dei dati, però occorre stare attenti ad evitare assegnazioni non valide, introducendo penalità, come verrà spiegato. Lo spazio di Hilbert avrà dimensione $2^{n\lceil \log_2 k \rceil}$.

2.4.1 Esempio con n=3 nodi, k=3 colori

- Nodo 0 $\rightarrow [0, 0]$ (colore 0)
- Nodo 1 $\rightarrow [0, 1]$ (colore 1)
- Nodo 2 $\rightarrow [1, 0]$ (colore 2)
- Configurazione $[1, 1]$ da ignorare, poiché non rappresenta nessun colore.

Bitstring totale: 00 01 10 ($n \cdot \lceil \log_2 k \rceil = 6$ qubits)
Questa codifica fa utilizzo di molti meno qubits e ci si aspetta che migliori la scalabilità del problema, anche se richiede più attenzione nella costruzione dell'Hamiltoniano.

2.5 Binary encoding nel problema del Graph Coloring

Come preannunciato, vogliamo usare $\lceil log_2 k \rceil$ qubits invece di usarne k per ogni nodo. Ogni colore è rappresentato da una stringa binaria di $m = \lceil log_2 k \rceil$ bit, mentre ogni nodo v è rappresentato da un registro di m qubits (questi codificano il colore assegnato al nodo). La codifica binaria garantisce naturalmente il primo vincolo: ogni nodo ha un solo colore. Rimane quindi da penalizzare tutti gli stati in cui due nodi adiacenti hanno la stessa sequenza binaria, cioè lo stesso colore. Supponendo che u, v siano nodi adiacenti:

- nodo u ha $[u_0, u_1, ..., u_{m-1}]$ qubits
- nodo v ha $[v_0, v_1, ..., v_{m-1}]$ qubits

il termine di penalità sarà il seguente:

$$H_{uv} = \prod_{l=0}^{m-1} \frac{1 + Z_{u_l} Z_{v_l}}{2} \tag{2}$$

esso varrà 1 solo se tutte le coppie di bit sono uguali, altrimenti sarà 0. Adesso bisogna penalizzare tutte le codifiche non valide. Ad esempio nel caso $k = 3$ e $m = 2$ abbiamo $2^m = 4$ combinazioni (00,01,10,11), ma solo 3 sono valide. Perciò penalizziamo lo stato $|11\rangle$ con un termine del tipo:

$$H_{pen} = \sum_{v} \frac{1 - Z_{v_0}}{2} \frac{1 - Z_{v_1}}{2} \tag{3}$$

come si può notare questo termine è minimo (nullo) quando nessun nodo v ha entrambi i qubits a 1. Volendo generalizzare questo ragionamento consideriamo n nodi e k colori generici, usando $m = \lceil log_2 k \rceil$ qubits per ogni nodo. Si può notare che le bitstring "non valide", cioè quelle che non rappresentano nessun colore vi sono solo quando k non è una potenza di 2.

$$k < 2^m \Longrightarrow k < 2^{\lceil log_2 k \rceil} \tag{4}$$

il numero di combinazioni non valide è $2^m - k$. Per ognuno di questi stati è possibile costruire un proiettore nel modo seguente:

$$P_{pen}^{(b)} = \prod_{l=0}^{m-1} \left(\frac{1 + (-1)^{b_l} Z_{v_l}}{2} \right) \tag{5}$$

dove $b = (b_0, b_1, ..., b_{m-1})$ è la stringa binaria non ammessa. Infine bisogna sommare su ogni nodo v e su ogni stringa binaria non valida:

$$H_{pen} = \sum_{v} \sum_{non-valid\ b} P_{pen}^{(b)} \tag{6}$$

Dunque l'Hamiltoniano totale di costo sarà la somma dei termini (2) e (6):

$$H_C = \sum_{(u,v) \in E} \prod_{l=0}^{m-1} \left(\frac{1 + Z_{u_l} Z_{v_l}}{2} \right) + \sum_{v} \sum_{non-valid\ b} \prod_{l=0}^{m-1} \left(\frac{1 + (-1)^{b_l} Z_{v_l}}{2} \right) \tag{7}$$

2.6 Estensione ai qudits

Successivamente si è provato ad implementare lo stesso problema simulandolo con qudits al posto dei qubits. In questo caso ogni nodo v del grafo è rappresentato da un qudit di dimensione d = k, dove k è il numero di colori. Lo stato del qudit $|c_v\rangle$ corrisponde dunque al colore assegnato al nodo v, dove $c_v \in \{0, 1, ..., k-1\}$. Anche in questo caso va costruito un hamiltoniano H_C il cui stato fondamentale corrisponda ad una colorazione valida del grafo. Per ogni arco $(u, v) \in E$ vogliamo penalizzare gli stati in cui $c_u = c_v$ per cui definiamo:

$$H_C = \sum_{(u,v) \in E} H_{uv} = \sum_{(u,v) \in E} \sum_{c=0}^{n-1} |c\rangle_u \langle c| \otimes |c\rangle_v \langle c| \tag{8}$$

Come si può notare, a differenza dell'hamiltoniano del caso precedente, l'unica penalità che si è implementata è quella per cui due nodi adiacenti non devono avere lo stesso colore. Il vincolo secondo cui ogni nodo abbia esattamente un colore è implicitamente rispettato dall'encoding dei colori.

2.7 Ottimizzazione del registro quantistico

Per ridurre ulteriormente i tempi di esecuzione e testare l'algoritmo su grafi più complessi, si è pensato di ridurre la degenerazione delle soluzioni imponendo che uno specifico nodo abbia un colore fissato. La scelta del nodo è stata automatizzata per far sì che il nodo scelto fosse quello con il grado minimo, cioè quello che ha il minor numero di connessioni (in questo modo ci sono meno vincoli da imporre sugli altri nodi da aggiungere nell'Hamiltoniano). Così facendo si è lavorato con $n - 1$ nodi, non considerando quello col colore fissato, a costo di aggiungere penalità ai suoi primi vicini. I tempi di convergenza sono notevolmente migliori di quelli ottenuti senza fissare un nodo. Ad

esempio, nel caso del grafo triangolare i tempi di convergenza per il one-hot encording sono di circa 7 minuti che si riducono a circa 8 secondi usando $n-1$ vertici. Di conseguenza tutti i risultati successivi sono stati ottenuti sfruttando questa ottimizzazione.

2.8 QAOA

Il Quantum Approximate Optimization Algorithm (QAOA) è un tecnica generale che può essere usata per trovare soluzioni approssimate a problemi di ottimizzazione combinatoria e si compone dei seguenti passaggi:

1. Definire un Hamiltoniano di costo H_C tale che il suo stato fondamentale codifichi la soluzione del problema di ottimizzazione
2. Definire un Hamiltoniano di mixing H_M che serve ad evitare che il sistema resti bloccato in minimi locali
3. Costruire i circuiti $e^{-i\gamma H_C}$ e $e^{-i\alpha H_M}$, questi sono chiamati rispettivamente layer di costo e layer di mixing
4. Scegliere un parametro n ≥ 1 che rappresenta la profondità del circuito:
$$U(\gamma,\alpha) = e^{-i\alpha_n H_M} e^{-i\gamma_n H_C} ... e^{-i\alpha_1 H_M} e^{-i\gamma_1 H_C} \tag{9}$$
5. Preparare uno stato iniziale, applicare $U(\gamma,\alpha)$ e usare tecniche classiche per ottimizzare i parametri γ_i e α_i.

In sintesi, viene preparato uno stato iniziale e fatto evolvere secondo $U(\gamma,\alpha)$, strutturato a layer (fig. 1).

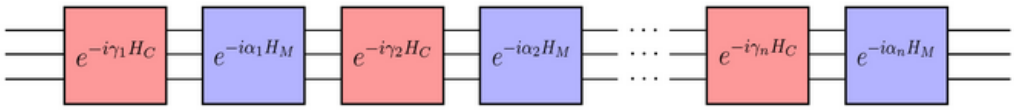

Figura 1

Per quanto riguarda le tecniche di ottimizzazione dei parametri, si è usato il *Gradient Descend* su device "*qulacs.simulator*" e l'ottimizzatore *Adam* su device "*default.qubit*" di PennyLane rispettivamente per l'encoding one-hot e l'encoding binario, mentre con l'utilizzo dei qudits è stato usato un classico ottimizzatore *COBYLA* di *scypy*.

3 Implementazione del codice

3.1 Librerie utilizzate

La versione definitiva dei codici è stata scritta in Python e si è fatto uso di JupyterNoteBook durante la fase di implementazione. Per quanto riguarda i circuiti con qubits, è stata usata la libreria di PennyLane per la simulazione e il QAOA. La costruzione e visualizzazione del grafo (anche di quello "di output", colorato) è avvenuta grazie al pacchetto Networkx e la visualizzazione dell'andamento della loss e gli altri grafici tramite MatPlotLib. Inoltre si è fatto uso della libreria tqdm per la visualizzazione della barra di caricamento durante il training (tool estremamente utile) e si è implementato un early stopping. Altri pacchetti di minore rilevanza sono serviti per l'encoding binario, poiché è servito supporto per la costruzione dei vincoli (sommatorie di produttorie).
Per quanto riguarda l'implementazione con qudits, l'Hamiltoniano è stato costruito nel formalismo bra/ket grazie alla libreria QuTiP (Quantum Tool in Python) e l'ottimizzazione è avvenuta tramite scypy.optimize.

3.2 Struttura generale

I codici fanno uso di diverse funzioni fondamentali, fra queste riportiamo:

- circuit(params): costruisce il circuito QAOA
- qaoaLayer(gamma, alpha): evoluzione di cost + mixer (ripetuta *depth* volte)
- costFunction(params): funzione obiettivo, ovvero il valore dell'Hamiltoniano di costo
- probabilityCircuit(gamma, alpha): ultima run del circuito con i parametri ottimali, assegna le probabilità finali agli stati
- analyzeResults(probs, ...): in diverse versioni chiaramente per ogni encoding, analizza le probabilità massime e verifica la validità delle bitstring (gli stati) a esse associate.

In particolare, l'intero codice gira sulla variabile k_{colors}, che parte da 2 e si arresta quando viene trovata una soluzione. Ad ogni training completato, viene runnato il probability circuit e ottenute le probabilità, che vengono infine analizzate (non tutte ovviamente, solo quelle sopra una certa threshold) e visualizzate in un istogramma e in un assignment stampato a schermo. Nell'assignment è stata inserita la bitstring, un dizionario nodo-colore, la validità della bitstring (se rispetta i vincoli desiderati o no) e la probabilità associata. Se vi è una soluzione di tipo *True* viene stampata una stringa del tipo "Il numero cromatico del grafo è k", altrimenti "Nessuna colorazione valida trovata con k colori. Provo con $k+1$...".

4 Risultati

Di seguito sono riportati i risultati di diverse risoluzioni del problema per diversi grafi. Si può notare come i tempi di convergenza cambino a seconda dell'encoding dei dati e del sistema considerato (per tempo di convergenza si intende il tempo richiesto per trovare i parametri ottimali, non il tempo totale di run).

GRAFO	NODI	COLORI	SISTEMA	DEPTH	TEMPO
Linea aperta	≤ 6	2	Qubit_1	4	$\leq$ 6m34.1s
			Qubit_2	1	$\leq$ 0.3s
			Qudit	1	$\leq$ 0.1s
Quadrato	4	2	Qubit_1	2	20.8s
			Qubit_2	1	0.3s
			Qudit	1	0.1s
Triangolo	3	3	Qubit_1	1	7.8s
			Qubit_2	1	0.7s
			Qudit	1	0.1s
Pentagono	5	3	Qubit_1	1	24.4s
			Qubit_2	1	1.0s
			Qudit	1	2.7s
Esagono	6	2	Qubit_1	1	23.9s
			Qubit_2	1	0.3s
			Qudit	1	0.1s
Due quadrati	6	2	Qubit_1	1	23.3s
			Qubit_2	1	0.5s
			Qudit	1	0.2s
Tre triangoli	5	3	Qubit_1	4	64m31.2s
			Qubit_2	1	2.7s
			Qudit	1	0.7s
Casa	5	3	Qubit_1	3	30m8s
			Qubit_2	1	2.2s
			Qudit	1	1.9s
Bowtie graph	5	3	Qubit_1	3	59m49.1s
			Qubit_2	1	2.1s
			Qudit	1	2.0s

Tabella 1: Con Qubit_1 ci si riferisce a quelli encodati tramite one-hot; con Qubit_2 ci si riferisce a quelli dell'encoding binario.

GRAFO	NODI	COLORI	SISTEMA	DEPTH	TEMPO
Linea aperta	10	2	Qubit_2	1	0.4s
			Qudit	2	30.6s
Ettagono	7	3	Qubit_2	1	5.3s
			Qudit	1	23.3s
Bull Graph	5	3	Qubit_2	1	1.9s
			Qudit	1	3.2 s
Quattro triangoli adiacenti	6	3	Qubit_2	1	3.2s
			Qudit	1	7.2s
Prisma triangolare	6	3	Qubit_2	1	6.2s
			Qudit	1	6.5s
Reticolo quadrato	9	1	Qubit_2	1	0.6s
			Qudit	1	4.8s
Reticolo triangolare	7	1	Qubit_2	1	8.3s
			Qudit	2	1m1.4s
Pentagono completo	5	5	Qubit_2	1	14.8s
			Qudit	1	18.2s
Aquilone	6	4	Qubit_2	1	2.5s
			Qudit	1	1m57.7s
Random	7	3	Qubit_2	1	8.0s
			Qudit	1	24.7s
Parapluie Graph	7	3	Qubit_2	1	9.4s
			Qudit	1	1m7.0s
Grafene	13	2	Qubit_2	1	1.0s
			Qudit	3	286m24.0s
Petersen Graph	10	3	Qubit_2	1	1m17.2s
			Qudit	1	-

Tabella 2: Con Qubit_2 ci si riferisce a quelli encodati tramite encoding binario. Per quanto riguarda il Petersen Graph, non è stato possibile risolvere il problema simulandolo con Qudits in quanto il carico computazionale era particolarmente elevato.

4.1 Convergenza e istogrammi a confronto per il grafo triangolare

4.1.1 One-hot encoding

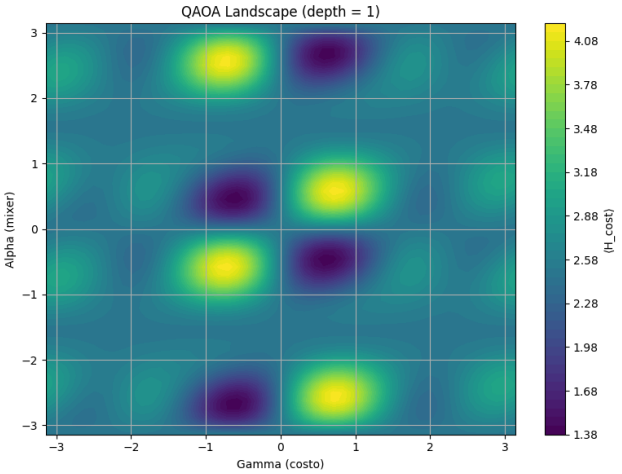

Figura 2: Il grafico rappresenta il panorama della funzione costo (valori per parametri compresi tra $-\pi$ e π)

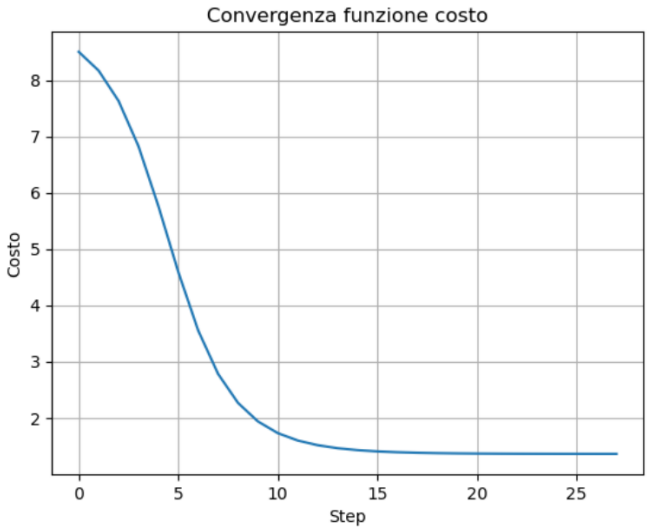

Figura 3

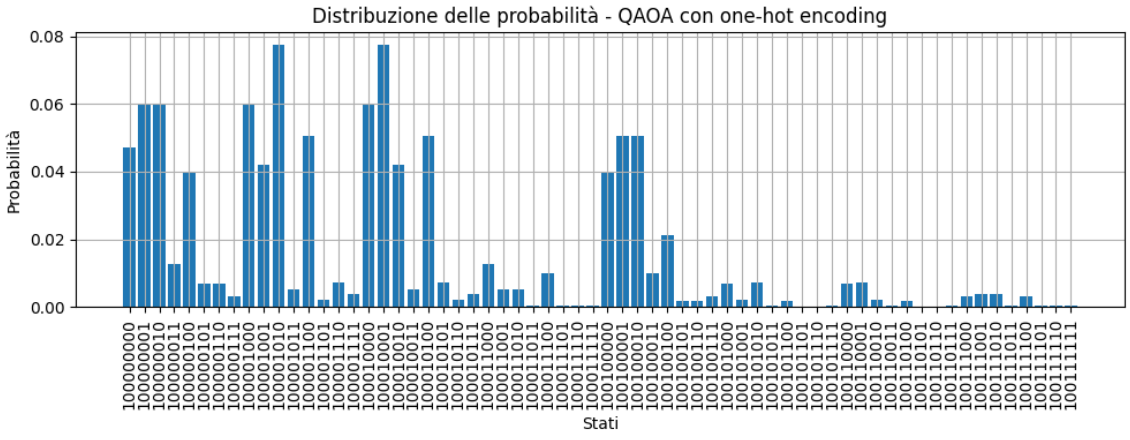

Figura 4

4.1.2 Binary encoding

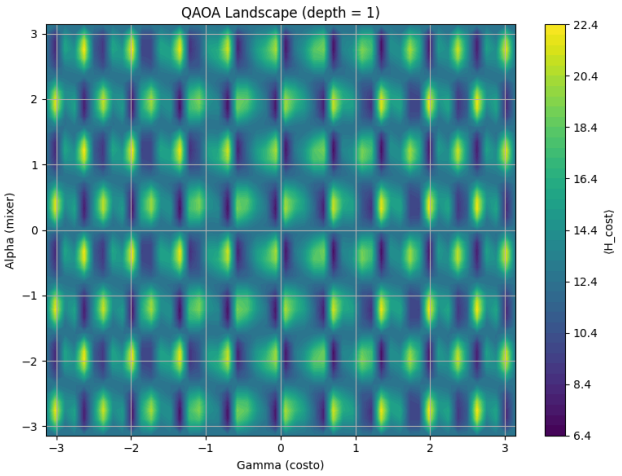

Figura 5: Il grafico rappresenta il panorama della funzione costo (valori per parametri compresi tra $-\pi$ e π)

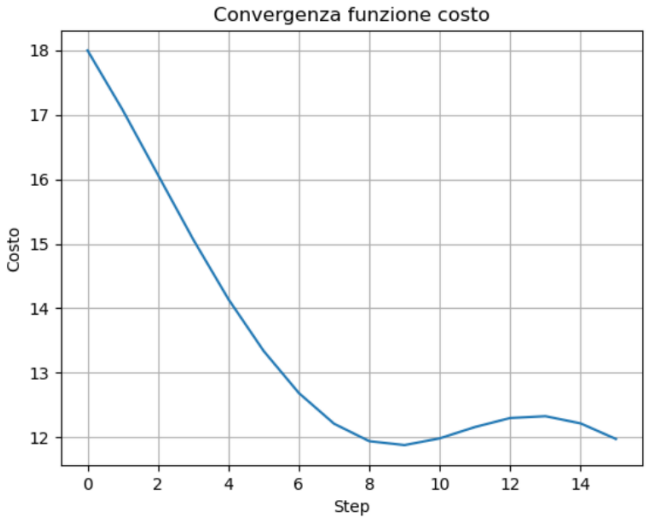

Figura 6

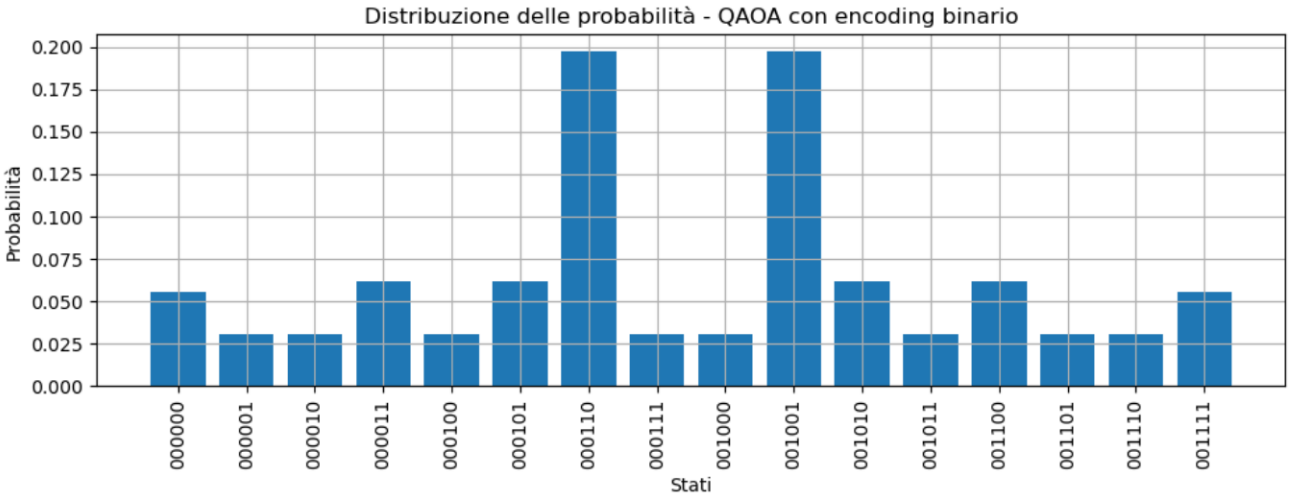

Figura 7

4.1.3 Qudits encoding

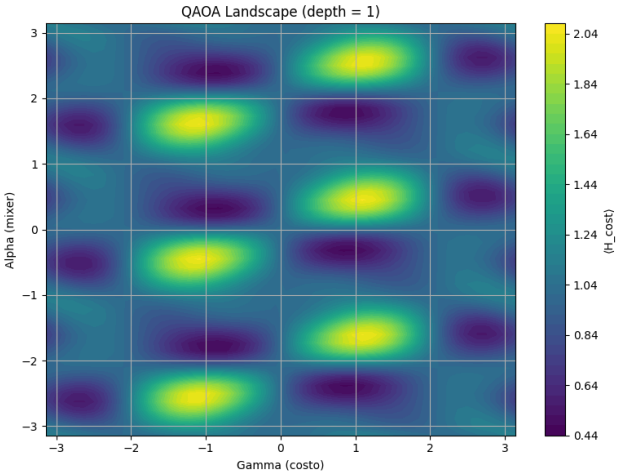

Figura 8: Il grafico rappresenta il panorama della funzione costo (valori per parametri compresi tra $-\pi$ e π)

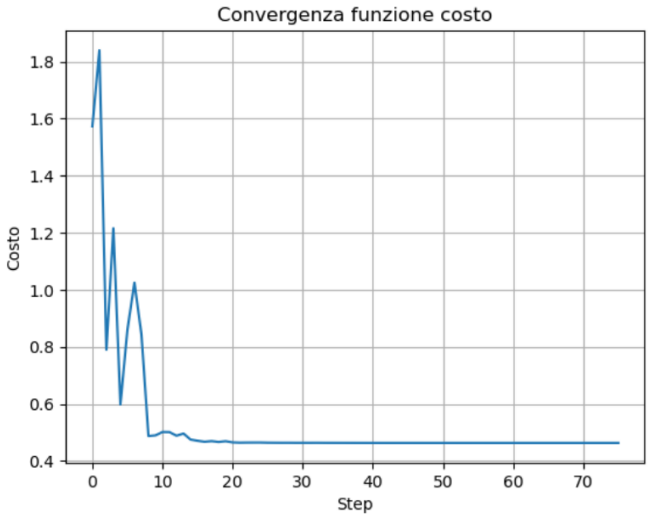

Figura 9

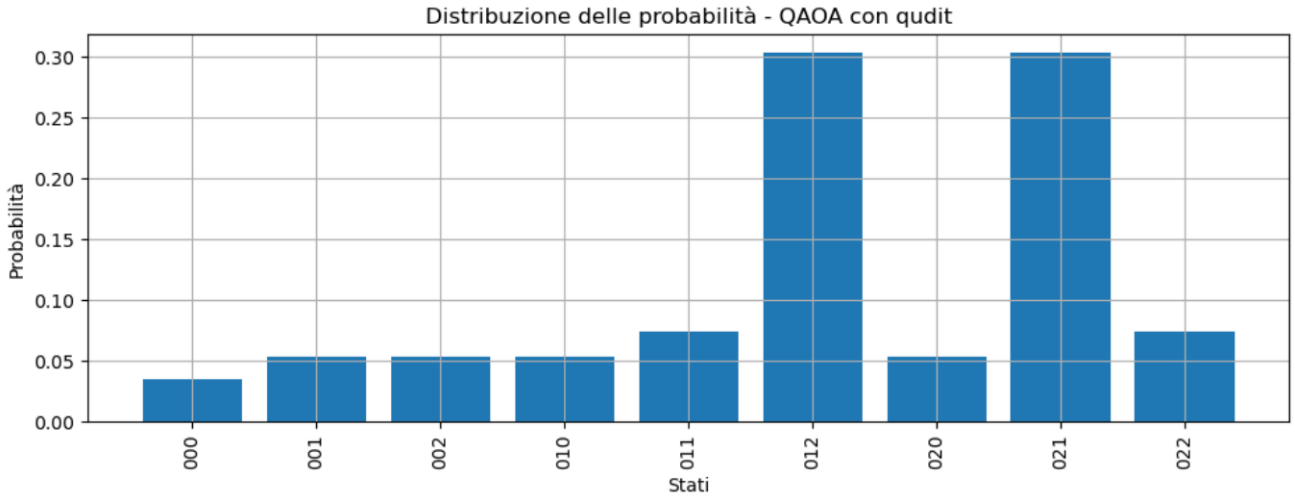

Figura 10

5 Grafi colorati

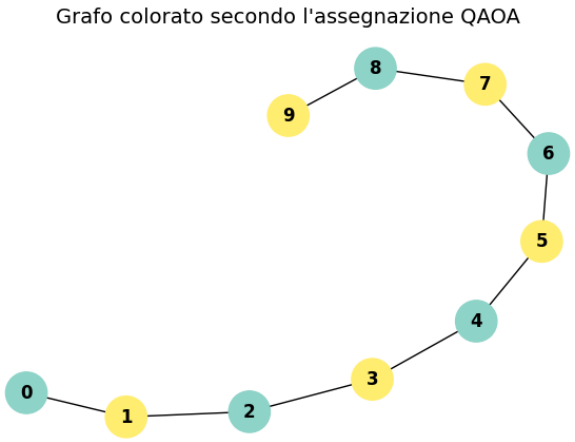

Figura 11: Linea aperta

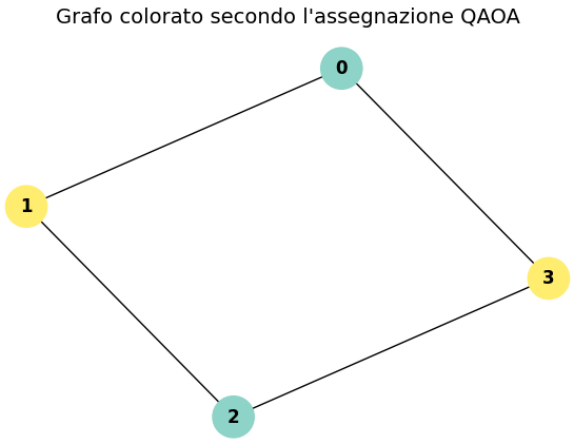

Figura 12: Quadrato

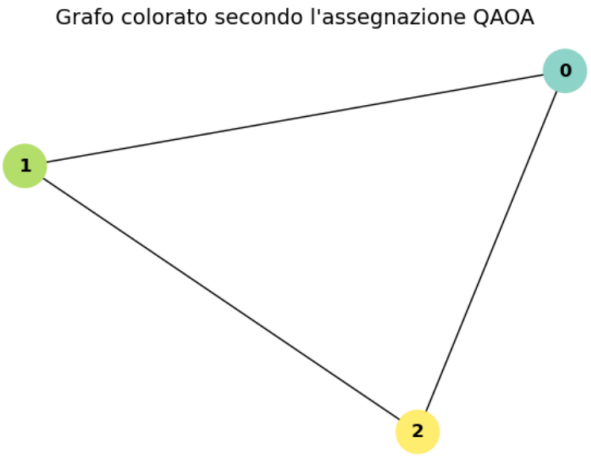

Figura 13: Triangolo

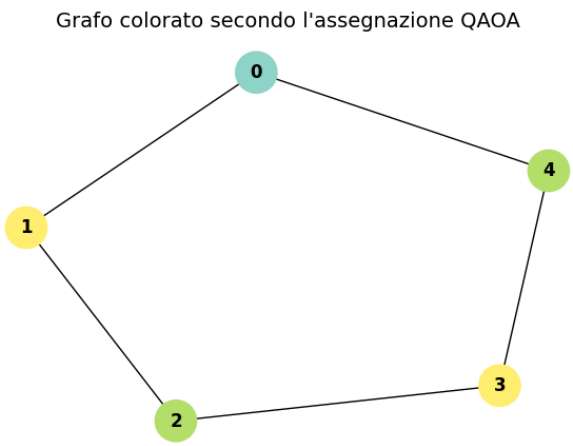

Figura 14: Pentagono

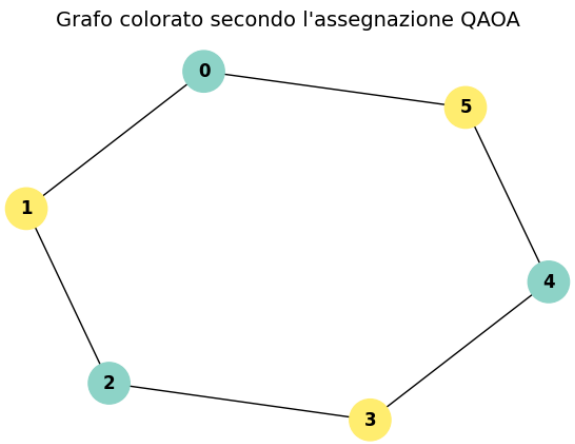

Figura 15: Esagono

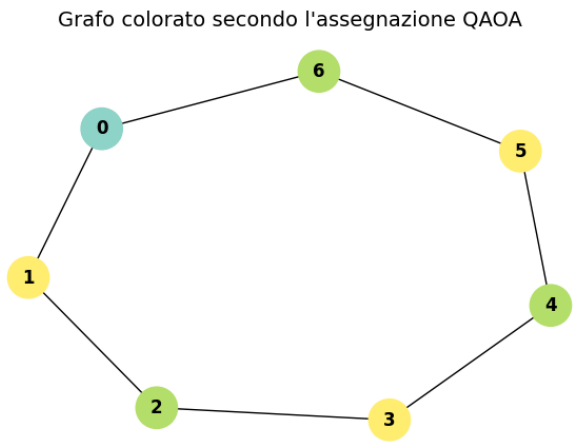

Figura 16: Ettagono

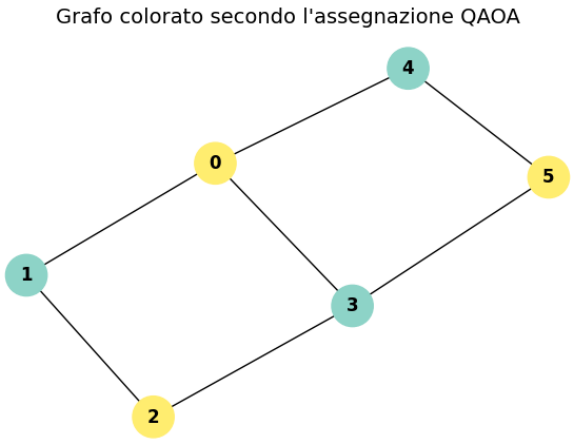

Figura 17: Due quadrati

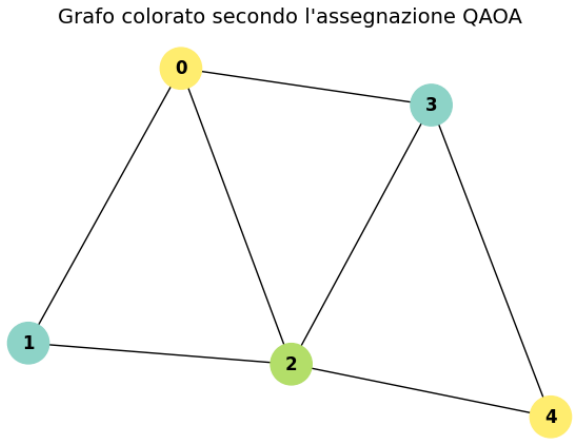

Figura 18: Tre triangoli

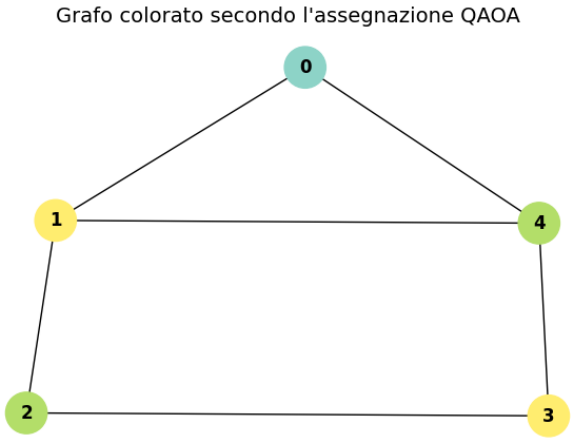

Figura 19: Casa

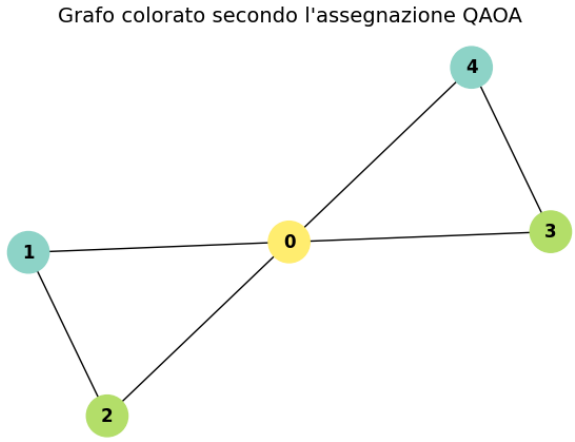

Figura 20: Bowtie graph

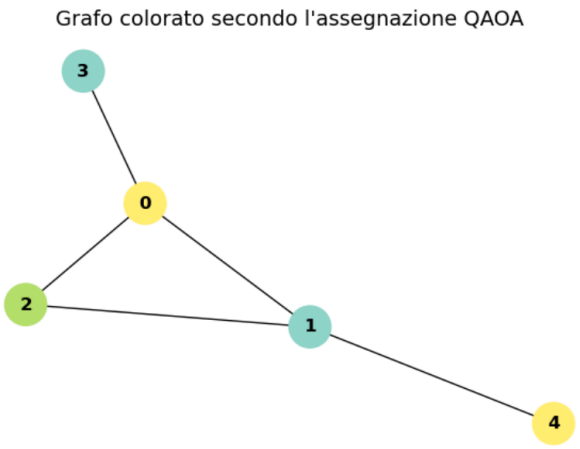

Figura 21: Bull graph

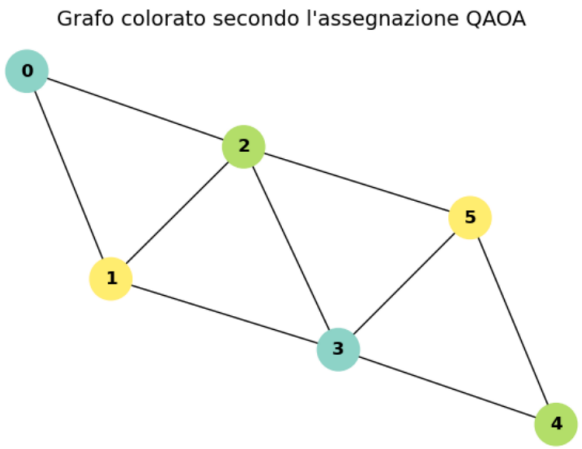

Figura 22: Quattro triangoli adiacenti

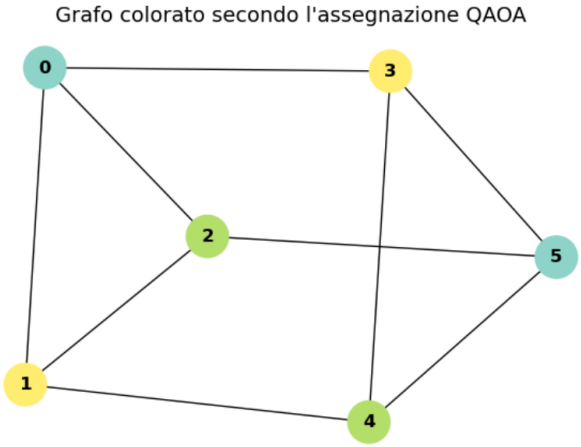

Figura 23: Prisma triangolare

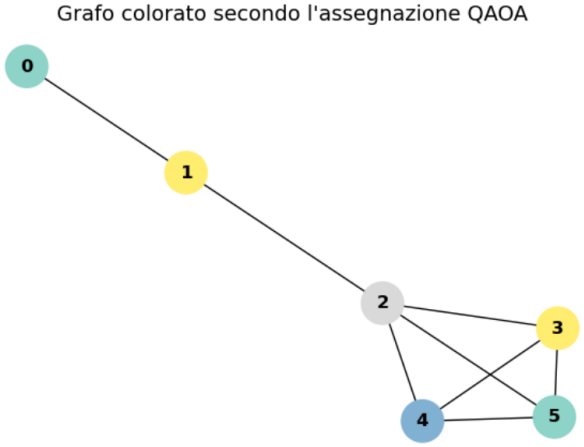

Figura 24: Aquilone

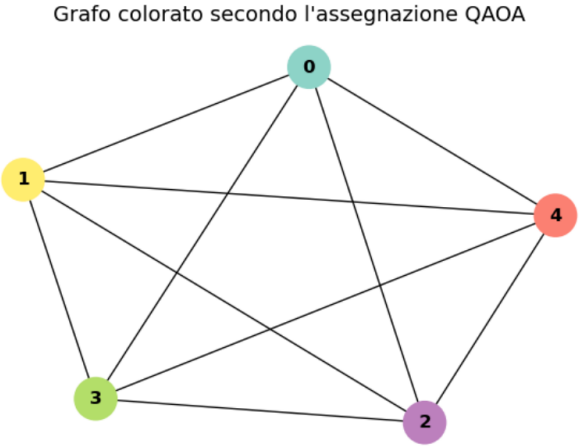

Figura 25: Pentagono completo

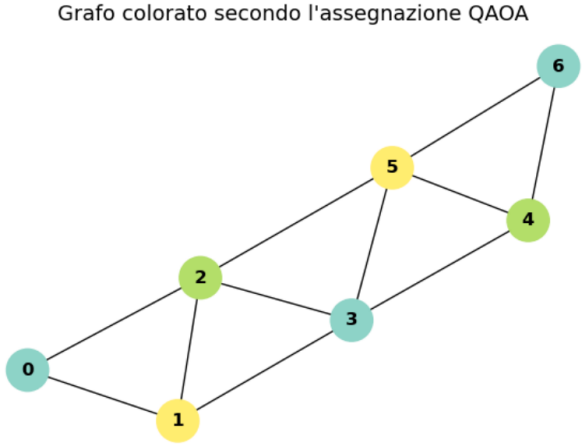

Figura 26: Reticolo triangolare

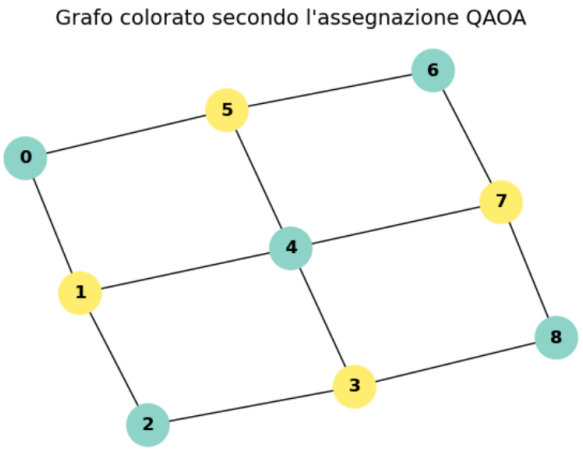

Figura 27: Reticolo quadrato

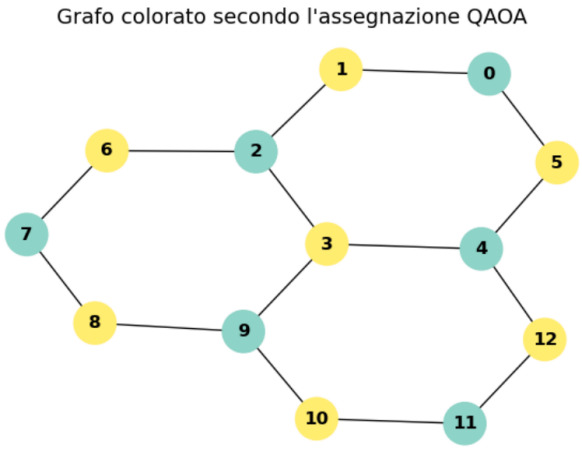

Figura 28: Grafene

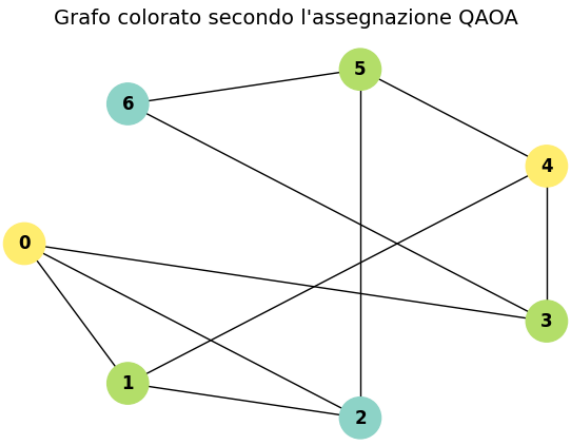

Figura 29: Random graph

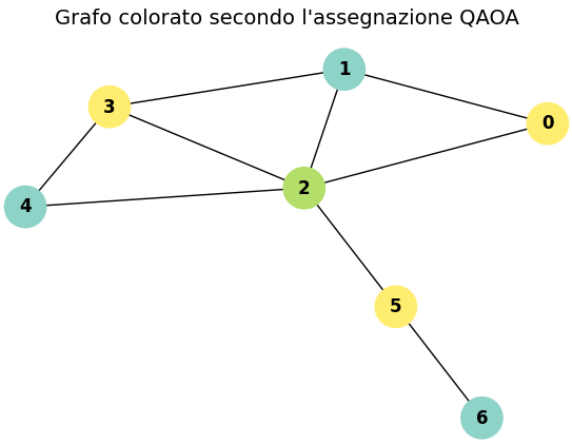

Figura 30: Parapluie graph

6 Esempio di output

Come precedentemente accennato, l'algoritmo è stato implementato in modo che alla fine esso fornisca il numero cromatico del grafo preso in input; cioè il numero minimo di colori che servono per colorare i vari nodi per far sì che due nodi collegati non abbiano lo stesso colore. È riportato un esempio del grafo di Petersen (numero cromatico 3, ricavato tramite binary encoding).

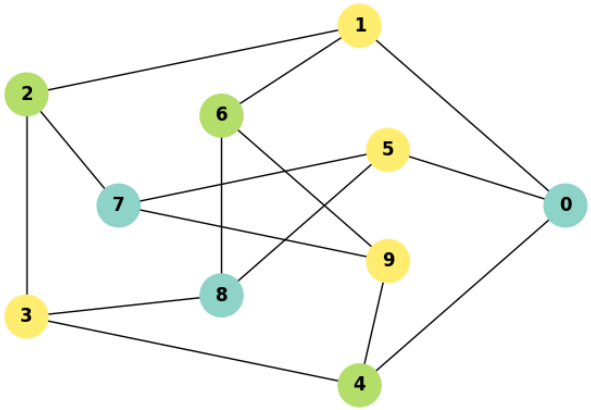

Figura 31

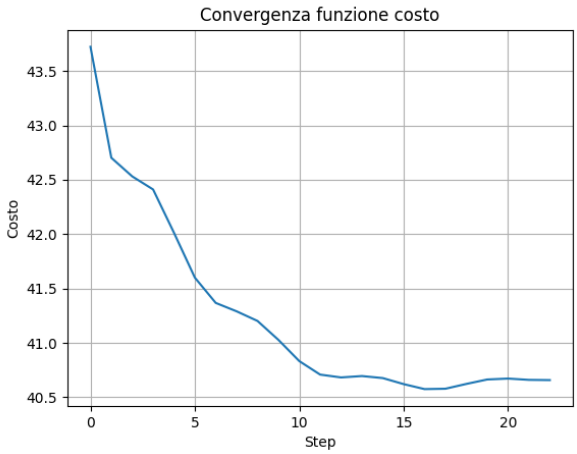

Figura 32

```
Training Progress k=2...:  14%|                                                  | 14/100 [00:00<00:04, 21.03it/s]
Early stopping at step 14
Training Progress k=2...:  14%|                                                  | 14/100 [00:00<00:04, 20.15it/s]
Bitstring | Assegnamento | Valido | Probabilità
--------------------------------------------------
0101111100 | {0: 0, 1: 1, 2: 0, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 0, 9: 0} | False | 0.05263688890088796
0110110011 | {0: 0, 1: 1, 2: 1, 3: 0, 4: 1, 5: 1, 6: 0, 7: 0, 8: 1, 9: 1} | False | 0.05263688890088796

Nessuna colorazione valida trovata con 2 colori. Provo con 3...
Training Progress k=3...:  22%|                                                  | 22/100 [01:06<04:15,  3.28s/it]
Early stopping at step 22
Training Progress k=3...:  22%|                                                  | 22/100 [01:07<03:57,  3.05s/it]
Bitstring | Assegnamento | Valido | Probabilità
--------------------------------------------------
0001000l100100101001 | {0: 0, 1: 1, 2: 0, 3: 1, 4: 2, 5: 1, 6: 0, 7: 2, 8: 2, 9: 1} | True | 0.0005185015008750409
0001001001l000010110 | {0: 0, 1: 1, 2: 0, 3: 2, 4: 1, 5: 2, 6: 0, 7: 1, 8: 1, 9: 2} | True | 0.0005185015008750406
0001100001l010010100 | {0: 0, 1: 1, 2: 2, 3: 0, 4: 1, 5: 2, 6: 2, 7: 1, 8: 1, 9: 0} | True | 0.0005185015008750406
0001100010l010010100 | {0: 0, 1: 1, 2: 2, 3: 0, 4: 2, 5: 2, 6: 2, 7: 1, 8: 1, 9: 0} | True | 0.0005185015008750406
0001100l100110000001 | {0: 0, 1: 1, 2: 2, 3: 1, 4: 2, 5: 1, 6: 2, 7: 0, 8: 0, 9: 1} | True | 0.0005185015008750409
0001100l101010000001 | {0: 0, 1: 1, 2: 2, 3: 1, 4: 2, 5: 2, 6: 2, 7: 0, 8: 0, 9: 1} | True | 0.0005185015008750406
0010000l100100101001 | {0: 0, 1: 2, 2: 0, 3: 1, 4: 2, 5: 1, 6: 0, 7: 2, 8: 2, 9: 1} | True | 0.0005185015008750409
0010001001l000010110 | {0: 0, 1: 2, 2: 0, 3: 2, 4: 1, 5: 2, 6: 0, 7: 1, 8: 1, 9: 2} | True | 0.0005185015008750406
0010010001010l101000 | {0: 0, 1: 2, 2: 1, 3: 0, 4: 1, 5: 1, 6: 1, 7: 2, 8: 2, 9: 0} | True | 0.0005185015008750409
0010010010010l101000 | {0: 0, 1: 2, 2: 1, 3: 0, 4: 2, 5: 1, 6: 1, 7: 2, 8: 2, 9: 0} | True | 0.0005185015008750406
0010011001010l000010 | {0: 0, 1: 2, 2: 1, 3: 2, 4: 1, 5: 1, 6: 1, 7: 0, 8: 0, 9: 2} | True | 0.0005185015008750409
0010011001l001000010 | {0: 0, 1: 2, 2: 1, 3: 2, 4: 1, 5: 2, 6: 1, 7: 0, 8: 0, 9: 2} | True | 0.0005185015008750405
Il numero cromatico del grafo è 3.
Miglior costo trovato: 40.57554452048411
Parametri corrispondenti: [[0.48450866]
 [0.35917952]]
```

Figura 33

7 Conclusioni

In questo progetto abbiamo affrontato il problema della colorazione dei grafi utilizzando l'algoritmo QAOA, sperimentando diverse codifiche (one-hot, binaria e qudit) per rappresentare i colori dei nodi. Abbiamo analizzato i vantaggi e le limitazioni di ciascun approccio in termini di efficienza computazionale e risorse richieste, valutando

anche le prestazioni su grafi con diverse complessità. Particolare attenzione è stata dedicata alla costruzione esplicita dell'Hamiltoniano e all'implementazione di strategie di ottimizzazione per ridurre il numero di variabili, come la fissazione del colore di un nodo a grado minimo. I risultati ottenuti dimostrano che tali ottimizzazioni possono ridurre sensibilmente i tempi di simulazione e migliorare la scalabilità dell'algoritmo. Questo lavoro rappresenta un primo passo verso l'implementazione efficiente di algoritmi di ottimizzazione combinatoria su dispositivi quantistici e offre la possibilità di risolvere problemi attuali quali:

- **colorazione di mappe**: assumendo che i nodi siano le regioni e che gli archi che li collegano rappresentino i confini comuni, si riuscirebbe a colorare le varie regioni in modo tale che quelle confinanti abbiano colori diversi;
- **pianificazione di orari universitari**: associando i corsi ai nodi e collegando con gli archi solo i nodi che hanno studenti in comune, si potrebbero assegnare fasce orarie (rappresentate dai colori) ai corsi in modo tale che gli studenti che devono seguirli non abbiamo sovrapposizioni;
- **assegnazione di frequenze radio**: assegnando ad ogni nodo un dispositivo e collegando tra di loro i dispositivi fisicamente vicini, si riuscirebbe ad evitare interferenze assegnando frequenze radio diverse (rappresentate dai colori) a nodi adiacenti;
- **pianificazione di tornei**: nei campionati ogni squadra può rappresentare un nodo, e gli archi rappresentano le partite che vengono affrontate; i colori dunque saranno rappresentativi dei turni di gioco o le giornate di disputa;
- **transpiler su circuiti quantistici**: mappatura qubit logico - qubit fisico, ogni nodo è un qubit logico mentre ogni arco collega qubits che devono interagire. Colorare il grafo significa assegnare qubit fisici (colori) evitando conflitti; qubit che devono interagire devono essere su qubit fisici connessi.