Progetto IA

Francesco Vattiato

1000008830

1 Introduzione

Il progetto consiste nella risoluzione dell' Equitable Graph Coloring Problem (EGCP) mediante la Tabu Search. L' EGCP è stata scelta per la sua complessità che ho ritenuto stimolante, mentre la Tabu Search l'ho reputata la scelta migliore per la sua risoluzione.

Abbreviazioni utilizzate

V	Insieme dei nodi
N	Numero totale dei nodi N
K	Insieme dei colori
$\|K\|$	Numero dei colori
$s0$	Soluzione iniziale
C	Insieme dei conflitti

2 Caratteristiche del problema

In questa sezione illustrerò le considerazioni e le scelte fatte per trattare il problema.

2.1 Rappresentazione del grafo

E' stata scelta una matrice di adiacenza triangolare inferiore per rappresentare il grafo.

2.2 Rappresentazione dello stato

La rappresentazione dello stato dei nodi scelta è quella di un array di dimensione N, dove ogni elemento al suo interno indica il colore $k \in K$ assegnato al nodo v $\in$ V del corrispondente indice: ad esempio, l'equitable array $[2, 1, 1, 2, 2]$, con $N = 5$ e $|K| = 2$, abbiamo che ai nodi $1, 2, 3, 4, 5$ vengono assegnati rispettivamente i colori $2, 1, 1, 2, 2$. Una tale rappresentazione dello stato è adatta per un algoritmo di ricerca ed è facile generare una soluzione $s0$ ed i corrispettivi vicini.

2.3 Generazione del vicinato

Dato uno stato s, l'idea iniziale era di generarare i vicini tramite l'aumento e decremento di 1 per ogni elemento di s (senza andare al di sotto di 1). L'evidente vantaggio è la complessità lineare $\mathrm{O}(2\mathrm{N})$ ma gli svantaggi che questo approccio comportano mi hanno indirizzato verso un cambio di idea: infatti, nella generazione del vicinato il vincolo "equitable" non viene rispettato, seppur può essere "fixato" con delle ricerche locali una volta che la ricerca con questo vicinato ha generato una soluzione migliore. Inoltre, la ricerca avanza molto lentamente a causa dei

micro-cambiamenti che porta ad ogni stato ed è molto facile che si blocchi in un minimo locale. E' possibile velocizzare la ricerca con una buona soluzione iniziale (vedi sezione 2.4). L'idea definitiva è stata l'utilizzo dei classici operatore "One Move" e "Swap": nell'implementazione di questo progetto, ad ogni iterazione, viene scelto a caso un nodo e vengono effettuati entrambi gli operatori.

Nel caso "One Move", se il nodo $v \in K_i : |K_i| = \left\lfloor \frac{N}{|K|} \right\rfloor$, vengono inseriti in K_i tutti i possibili nodi (generando un vicino per ogni "Move") $u \in K_j : |K_j| = \left\lceil \frac{N}{|K|} \right\rceil$. Se invece il nodo $v \in K_i : |K_i| = \left\lceil \frac{N}{|K|} \right\rceil$, allora a questo nodo vengono assegnati in tutti i possibili colori $_j : |K_j| = \left\lfloor \frac{N}{|K|} \right\rfloor$ La complessità di questo operatore è $\mathrm{O(N)}$.

Nel caso "Swap", viene effettuato lo scambio di colore tra due nodi u, v con colori $K_i, K_j : i \neq j$. Anche la complessità di questo operatore è $\mathrm{O(N)}$.

2.4 Stato iniziale

Inizialmente, con la generazione dei vicini con incremento e decremento di 1, lo stato $s0$ era dato da una distribuzione casuale di numeri compresi tra 1 ed n, con n parametro scelto: più n è alto, più facilmente la ricerca troverà una soluzione finale. Tuttavia, il modo casuale con cui vengono scelti i nodi non rispetta il vincolo "equitable".

Aspirando di partire da un $s0$ che sia allo stesso "equitable" e più vicino possibile alla soluzione, nel progetto è stata implementata la seguente procedura: scelto un k, se k divide N, ci saranno k insiemi con $\frac{N}{k}$ elementi ognuno, altrimenti ci saranno alcuni insiemi con $\lfloor \frac{N}{k} \rfloor + 1$ elementi ed altri $\lfloor \frac{N}{k} \rfloor$ elementi. In questi insiemi, i primi nodi vengono inseriti estrando casualmente da V dei nodi, inserendoli negli insiemi vuoti. Successivamente, per i restanti nodi, dato v, viene inserito nel primo insieme $i : v \notin C_i$, altrimenti viene inserito in un insieme a caso, rispettando i vincoli di equità tra gli insiemi.

Data la natura sconosciuta del k ottimale, la ricerca partirà da un k relativamente alto. Una volta trovata la soluzione migliore per k, la ricerca ripartirà con $s0$, settata come la soluzione precedentemente trovata, in modo da ricercare una soluzione per $k - 1$. Per far si che la ricerca per k-1, cominci con la soluzione di k, gli elementi che fanno parte del k-esimo gruppo vengono riassegnati negli altri gruppi nel modo descritto nel paragrafo precedente.

3 Tabu Search e dettagli implementativi

In questa sezione descriverò l'implementazione della Tabu Search focalizzandomi sulle singole parti.

3.1 Funzione obiettivo e conflitti

La funzione obiettivo da minimizzare tiene conto dei conflitti tra i nodi adiacenti a seguito di una errata assegnazione dei colori. Sono state definite due funzioni da minimizzare: la baseline-evaluate e l'evaluate. La funzione baseline, data la matrice di adiacenza e lo stato s, ritorna un array *evaluation* di dimensione N, dove ogni entry indica se un nodo è in conflitto e, nel caso in cui lo fosse, con quanti nodi, quindi se $evaluation_v = 0$, nessun conflitto sul nodo v, altrimenti

sarà > 0.
Il conflitto per un nodo v viene calcolato come:

$$C_{v,u} = \begin{cases} 1, & \text{Se } adj_{v,u} \times s_u = K_v \\ 0, & \text{Altrimenti} \end{cases} \tag{1}$$

$$C_v = \sum_{u \in V, c \neq v} C_{v,u} \tag{2}$$

dove con C_v si intendono conflitti totali di $v, C_{v,u}$ conflitti di v con u, $adj_{v,u}$ l'entry(v,u) della matrice di adiacenza, s_u l' u-esimo elemento dello stato corrente, K_v il colore di v. In altre parole, dato un nodo v, si moltiplicano la sua la riga e la sua colonna della matrice di adiacenza con lo stato corrente: se da queste due moltiplicazioni ci sono uno o più valori uguale al colore di v, si ottengono i nodi con cui va in conflitto.
Questo calcolo, nella funzione baseline, è ripetuto per tutti i nodi. Tuttavia, essendo un calcolo oneroso per N molto grandi, questa funzione viene chiamata solo una volta per iterazione affinché possa verificare i conflitti generali.

Per valutare i vicini di uno stato si chiama la funzione fitness che, dati la matrice, lo stato vicino e due vertici se il vicino è stato generato da una "Swap" operation o un vertice se è stato generato da una "Move" operation: tutto quello che serve per valutare i vicini di uno stato è ottenibile calcolando i conflitti su uno/due nodi, infatti calcolare i conflitti sui nodi non variati rappresenterebbe uno spreco di tempo che si manifesterebbe in modo più grave con N elevati.

3.2 Memorie

La tabu search dispone di 3 tipologie di memorie: la short-term (tabu-list), la mid-term(intensification) e la long-term(diversification). La dimensione della tabu-list è stata impostata in modo statico a 10 nella quale verranno conservati gli stati (migliore,nuovo-migliore).
Non sono state implementate le altre due memorie in quanto non ho considerato la mid-term adatta a questo problema, mentre per la long-term ho provato diverse strategie (e.g perturbazione della soluzione dopo tot iterazioni non miglioranti, penalizzazioni su nodi visitati troppe volte) ma nessuna di queste sembra aver fatto la differenza.

3.3 Overview della ricerca

La ricerca parte da una soluzione iniziale s0 generata con la funzione "`initial_solution`" nel caso la ricerca stia partendo dal k dato, altrimenti la soluzione best trovata nella ricerca k+1 viene ridotta in k colori come descritto nella sezione 2.

Al momento di generare i vicini, piuttosto che generare tutti i possibili vicini, si considera casualmente un nodo candidato conflittuale, dato il vettore base ritornato dalla funzione "`baseline_evaluate`": in questo modo, la generazione del vicinato si limiterà alle sole possibili operazioni "One Move" e "Swap" che coinvolgono il nodo scelto.

Ottenuti i vicini, si sceglie la soluzione che non sia presente nella tabu-list e con valore dalla funzione "evaluate" minore e di conseguenza si aggiornerà la soluzione best attuale e la tabu-list anche se il vicino scelto non diminuisce i conflitti. Nel caso in cui uno stato vicino sia presente nella lista, l'algoritmo non calcolerà la funzione "evaluate" e passerà al vicino successivo, in modo tale da evitare calcoli non indispensabili alla ricerca.

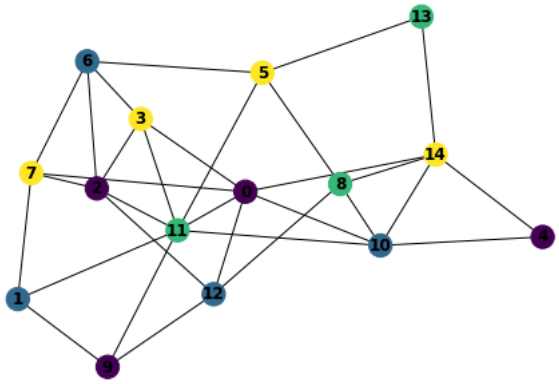

Figure 1: Grafo random di 20 nodi partizionato in 4 colori.

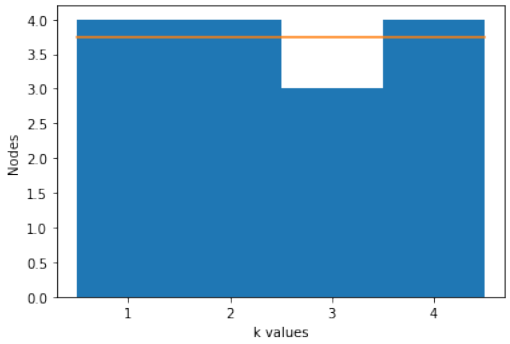

Figure 2: Distribuzione dei nodi per colore.In arancione il valore medio.

L'ultima cosa che farà la ricerca sarà quella di calcolare la "`baseline_evaluate`" in modo tale da valutare in modo globale la nuova soluzione: se ogni nodo non si troverà in una situazione di conflitto, la ricerca si può concludere, altrimenti il vettore ritornato dalla funzione servirà per l'iterazione successiva.
Tutto questo verrà ripetuto fino a quando i criteri scelti non saranno soddisfatti.

4 Fase sperimentale

Tutti gli esperimenti sono stati svolti con i seguenti criteri di terminazione: 30 minuti di tempo per trovare la soluzione per un dato k e numero totale di conflitti uguale a 0. Una volta che verrà soddisfatto di uno di questi criteri, l'esecuzione sul k corrente si concluderà. Inoltre, se la conclusione è data dal raggiungimento dei 30 minuti da parte della ricerca, non si proseguirà nella ricerca della soluzione con $k - 1$.
Come valore iniziale di k, è scelto la metà di N in modo da non eseguire soluzioni per k superflui (nel caso in cui k $\approx$ N) o essere troppo vicini ad un k infattibile (nel caso in cui k sia troppo basso). Alla fine della ricerca è possibile visualizzare il risultato attraverso un grafo (fig. 1) ed un istogramma (fig. 2).
Ogni prova è stata settata con un seed diverso.

Le prime istanze su cui la mia implementazione della Tabu Search è stata provata, sono

grafi generati in maniera random con numero di nodi e archi variegato: con questi grafi, è stato possibile comprendere meglio il problema e capire come migliorare (tutto descritto nelle sezioni precedenti). Consolidato l'algoritmo, sono state testate le istanze huck e tutte le myciel, ottenendo il k minimo registrato in letteratura.
Per quanto riguarda le istanze fortnitemi, non sono riuscito a leggere i file nel modo corretto, quindi ho reperito le istanze dal seguente sito Graph Coloring Instances. Elenco di seguito i risultati ottenuti sulle istanze fornite:

Istanza	Best	Media	Dev. Std
`DSJC125.5`	31	31.2	0.4
`DSJC250.5`	67	67.4	0.5
`queen6_6`	7	7	0
`queen8_12`	15	15.2	0.4

L'implementazione sembra trovare delle buone soluzioni per le istanze queen; sembrerebbe invece bloccarsi in dei minimi locali nelle istanze `DSJC125.5` ed `DSJC250.5` dove la ricerca non riesce a proseguire in profondità. Quest'ultimi due grafi risultano essere più complessi, non per il numero di nodi che li compongo, ma principalmente per l'elevato numero di archi per nodo: ad una maggiore connettività del grafo, il "landscape" in cui la ricerca procede per individuare il minimo diventa più complesso rispetto ad un grafo dove sono presenti meno archi, ed è per questo che la mia implementazione della Tabu Search non riesce a proseguire.
Tuttavia, è possibile uscire da questi minimi locali con un'adeguata long-term memory in grado di attivarsi nel momento in cui ci si trova in uno di questi minimi: per questo, come già anticipato nella sezione 3, ho provato l'inserimento di long-term memory ma, purtroppo, nessuno dei tentativi è andato a buon fine. Un'altra possibilità potrebbe essere quella di assegnare più tempo massimo di esecuzione per ogni k.

5 Conclusioni

La Tabu Search e la mia implementazione di questa si è dimostrata una buona scelta per la risoluzione dell' EGCP. I risultati ottenuti dimostrano che la logica dietro la mia implementazione è valida e sicuramente migliorabile se il tempo a disposizione fosse stato maggiore: infatti, come già discusso nella sezione sperimentale, la ricerca si blocca su dei minimi locali e ciò si può evitare con le giuste soluzioni, come ad esempio una valida long-term memory. In ogni caso, la versione della Tabu-Search presentata in questo progetto renderebbe di più se inserita all'interno di un altro algoritmo, come accade per esempio nel caso del GRASP, perché le ricerche locali da sole, come lo è d'altronde anche la Tabu-Search, si dimostrano spesso limitate nell'esplorazione del landscape di un problema. Ad ogni modo, lo sviluppo di questo progetto è servito a maturare in me la consapevolezza di quanto siano difficili e in funzione di questo, imporante risolvedere nel modo più veloce ed accurato possibile i problemi NP: trattare, non solo in teoria ma anche in pratica, problemi di tale complessità aiuta a capire quanto siano importanti i problemi di classe NP nel mondo dell'informatica e la necessità di sviluppare un pensiero critico quando ci si trova davanti ad un tale situazione.