Applicazione di un algoritmo genetico per la risoluzione del Graph coloring problem

Salvatore Borgesi

Dicembre 2022/Gennaio 2023

1 Introduzione

Il problema da trattare consiste nel colorare i vertici di un grafo considerando un vincolo. Dato un numero di colori, vi è la necessità di colorare i vertici in maniera tale che due nodi adiacenti non abbiano lo stesso colore. Il numero minimo di colori che può essere utilizzato viene definito **chromatic number** $\chi(G)$. La risoluzione a questo problema può essere adottata in diversi contesti applicativi quali:

1. Scheduling.
2. Gioco del Sudoku.
3. Colorazione delle mappe.

La scelta dell'algoritmo genetico per il problema, è stata dettata dal fatto che essa mi consente di migliorare le soluzioni ottenute iterazione per iterazione e portando al successivo step solo le soluzioni che si "avvicinano" di più al raggiungimento dell' ottimo.

2 Rappresentazione della popolazione iniziale

Per l'inizializzazione della popolazione iniziale viene prima di tutto definito un vettore di interi che va da 1 fino al numero massimo di colori che si vuole utilizzare per colorare i vertici del grafo affinché venga rispettato il vincolo di colorazione del problema. Supponiamo dunque, che il numero

di colori sia pari a n = 10. L'array che definisce i colori sarà:[1,2,3...,10]. Data quest'ultima lista, viene successivamente definita una soluzione (rappresentata anche essa da un'array) di lunghezza pari al numero di vertici del grafo. A ciascun cassetto della lista, viene assegnato in maniera casuale uno dei valori numerici definiti nell'arrray precedente. Questa operazione di generazione di un singolo individuo, viene eseguita ripetutamente fino a raggiungere il numero di individui desiderati (rappresentato in un file di configurazione tramite la variabile POPULATION_SIZE).

Algorithm 1 Initialize Population

while $Population \leq POPULATION\_SIZE$ **do**
  $solution \leftarrow []$
  $colors \leftarrow [1...StartColorSize]$
  for `each vertex` **do**
    $color \leftarrow random(colors)$
    $solution[vertex] \leftarrow$ color

3 Funzione di fitness

In un algoritmo genetico, la funzione di fitness consente di comprendere quale tra K individui è il migliore e quindi andrà con una certa probabilità alle iterazioni successive. La fitness definita all'interno della soluzione costruita, esamina ciascun arco del grafo, e per ciascuno viene verificato se due nodi adiacenti hanno lo stesso colore. In caso di esito positivo viene aggiunta una penalità. Successivamente, la somma degli archi che collegano due vertici aventi lo stesso colore, viene moltiplicata per il numero di colori che sta utilizzando la soluzione valutata. Questo prodotto perciò, restituisce zero se tutti i vertici sono colorati in maniera corretta, rispettando i vincoli del problema. Dopo diversi esperimenti ho notato che questa prima "versione" di valutazione ad un determinato punto non riusciva più a migliorare poiché se l'algoritmo trovava due soluzioni con fitness pari a zero, non riusciva a capire quale delle due soluzioni era la migliore. Questo perché, nonostante le soluzioni erano entrambe valide, una di esse potenzialmente poteva avere un valore cromatico più basso. Proprio per tale motivo ho deciso di aggiungere al prodotto citato prima, il numero di colori utilizzato. Facendo un esempio

si supponga di avere due soluzioni corrette nella quale la prima ha 5 colori, invece la seconda ne ha 6. Con la seconda funzione di fitness descritta, nonostante le due soluzioni siano entrambe corrette, la prima sarà migliore in quanto utilizza un minor numero di colori per il grafo.

Algorithm 2 Fitness function

```
count ← 0
colors ← numero di colori usati dalla soluzione
for each edges do
    if colore vertice u == colore vertice v then
        count ← count + 1
    end if
end for
return (colors ∗ count) + colors
```

4 Algoritmo utilizzato e pseudocodice

In questa sezione della relazione scenderò più nel dettaglio nell'esaminare e descrivere l'algoritmo costruito per la risoluzione del problema.
Come descritto sopra, la prima parte consiste nel tradurre le istanze fornite nel formato *.col* in maniera tale da poter essere rappresentate mediante codice. Per far questo sono state definite due classi: *Graph()* e *Vertex()*. All'interno dell'oggetto grafo saranno presenti la lista di archi e la lista di vertici. Ogni oggetto vertice tiene traccia di chi sono i nodi adiacenti ad esso. Lo pseudocodice, evidenzia come le operazioni più importanti siano quelle di *Selection*, *Crossover*, *Mutation* e di *Replacement* che stanno alla base del funzionamento di un algoritmo genetico. Dopo aver generato la popolazione iniziale, viene determinato mediante il metodo *GetUpperbound()* qual'è il colore con la quale si inizierà a 'dipingere' i nodi del grafo. Questo metodo banalmente, prende come riferimento il numero di archi uscenti per ciascun vertice e ne restituisce il valore più alto . L'algoritmo si conclude quando è stato raggiunto il numero massimo di valutazioni. Qualora il valore della funzione obiettivo è più vantaggioso di un altro (e la soluzione risulta valida) , viene sostituita la AbsoluteBestFitness precedente con quella appena trovata.

Algorithm 3 Genetic algorithm

```
graph ← TranslateDimacsInstance()
Population ← InitializePopulation(graph)
ColoreIniziale ← Getupperbound(graph)
fitnessCount ← 0
while not Stopping criteria do
    nuovaPopolazione ← []
    for i to POPULATION_ SIZE/2 do
        Selection();
        Crossover();
        Mutation();
        Replacement();
        if FitnessAbsoluteBestSolution ≥ ActualBestSolution then
            FitnessAbsoluteBestSolution ← ActualBestSolution
        end if
        if fitnessCount > MAX_ NUMERO_ VALUTAZIONI then
            StoppingCriteria ← True
```

4.1 Selection

Il processo di *Selezione* consiste nel selezionare tra la popolazione generata nella fase di inizializzazione la "*prole*" che sarà portata avanti nelle successive generazioni. In questo progetto sono state implementate tre tipi di operatori di selezione.

4.1.1 Roulette

Tramite questa modalità di selezione, vengono selezionati con probabilità più alta i genitori che hanno una fitness migliore. Per come è stata calcolata la fitness in questo progetto, i genitori che hanno un valore di fitness basso, avranno una maggiore probabilità di essere selezionati.

4.1.2 Random

Questa strategia prevede una selezione casuale dei genitori.

4.1.3 Tournament

Con l'utilizzo di questa strategia vengono estratti K individui dalla popolazione. Di questi K individui, viene selezionato quello che detiene il valore di fitness migliore.

4.1.4 Strategia adottata per la selezione

Mentre i tre metodi sopra citati sono stati tutti implementati nel codice, la decisione finale è stata quella di utilizzare la modalità *Tournament* con un valore di K = 20. L'utilizzo di questo operatore infatti, consente anche a configurazioni non ottimali di fare parte delle future generazioni, dando l'opportunità di spaziare tra differenti soluzioni e non convergere dunque in un ottimo locale. Sono stati effettuati diversi tentativi, ad esempio con con K = 10. Con quest'ultimo infatti, ho notato che anche se l'algoritmo arriva a convergenza più velocemente, rischia dopo diverse iterazioni di rimanere bloccato in un punto di ottimo locale.

5 Crossover

L'operatore genetico di crossover offre l'opportunità di mescolare le soluzioni ottenute mediante l'operatore di selezione ed ottenere perciò dei nuovi figli. Gli operatori di crossover sviluppati durante la realizzazione del prodotto sono il *Single one-point crossover* e il *Two-point crossover*.

5.1 Single one-point crossover

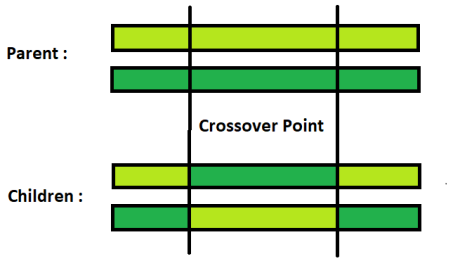
Dai figli generati durante la fase di Selezione, viene selezionato in modo tutto casuale un punto che ci consentirà di tagliare la lista in due parti e di effettuarne uno scambio.

Chromosome1	11011|00100110110
Chromosome2	11011|11000011110
Offspring1	11011|11000011110
Offspring2	11011|00100110110

Single Point Crossover

5.2 Two-point crossover

In questo caso, invece di selezionare un unico punto di taglio, vengono scelti due punti (randomici) delle due liste e vengono successivamente mischiate le due stringhe.



5.3 Stratgia adottata per il crossover

Inizialmente, nella realizzazione del progetto ho utilizzato il single one-point crossover con una probabilità di 0.9. Ho notato tuttavia, che utilizzare il 2-point crossover anche se con una probabilità più bassa (0.8) riesce ad apportare delle modifiche alle soluzioni generate e per tale motivo è stata mantenuta tale scelta per gli esperimenti.

6 Mutation

L'operazione di mutazione, dato un cromosoma, consente di variarne un 'gene' ottenendo come conseguenza una nuova soluzione. Per questo progetto sono state sviluppate due varianti.

6.1 Random mutation

Rappresenta l'operazione di mutazione più semplice. In breve, viene selezionato un elemento del cromosoma, viene selezionato un colore (in maniera casuale), e quest'ultimo (il colore) viene assegnato all'elemento scelto durante la prima fase.

Algorithm 4 Random mutation

$posizioneDaModificare \leftarrow random(1, numeroVertici)$
$numeroColori \leftarrow numeroColoriPresentiNellaSoluzione$
$soluzione[posizioneDaModificare] \leftarrow random(1, numeroColori)$

Algorithm 5 Vertex mutation

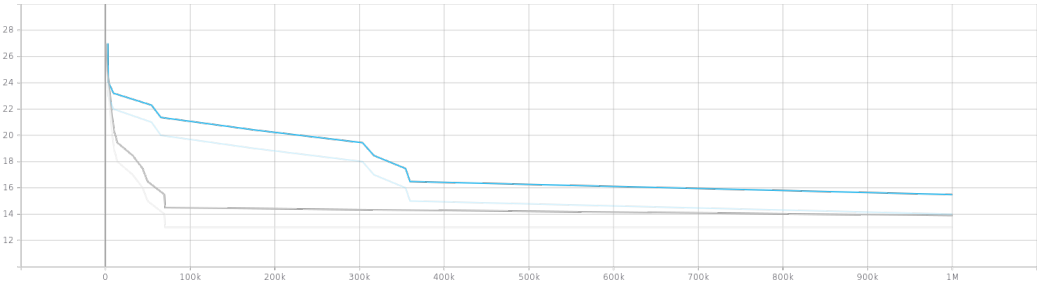
$ColoriModificati \leftarrow 0$
for `vertex in Vertices` **do**
    $neighbors \leftarrow vertex.getNeighbors()$
    for `neighbor in Neighbors` **do**
        if $solution[vertex] == solution[neighbor]$ **then**
            $solution[vertex] \leftarrow random(1, numeroColoriUsatiDallaSoluzione)$
            $coloriModificati \leftarrow coloriModificati + 1$
        end if
    end for
end for
if $ColoriModificati == 0$ **then**
    $posDaModificare \leftarrow random(1, numeroVertici)$
    $posCasuale \leftarrow random(1, numeroVertici)$
    if $colorePosizioneDaModifcare == colorePosizioneCasuale$ **then**
        $colorePosizioneDaModificare \leftarrow random(1, arrayColori)$
    else
        $colorePosizioneDaModificare \leftarrow colorePosizioneCasuale$
    end if

6.2 Vertex mutation

Questo tipo di mutazione è stata costruita per risolvere il problema assegnato. Per ciascun vertice, viene recuperata e ciclata la lista dei suoi nodi adiacenti. Dunque, per ciascun 'vicino' viene effettuato un controllo ovvero se il vertice e i vicini hanno lo stesso colore. In caso di esito positivo viene assegnato randomicamente un colore in un range che va da 1 al numero massimo di colori utilizzati in quell'istante dalla soluzione. Inoltre, viene incrementato un contatore che identifica il numero di colori che sono stati modificati. Dopo aver esaminato ciascun vertice ed il proprio vicinato, se risulta che durante le iterazioni non è stato modificato nessun colore, vuol dire che la soluzione è valida e di conseguenza si sceglierà in maniera casuale una posizione della soluzione alla quale sarà assegnato un colore diverso. Questa seconda parte del codice è stata introdotta poiché senza di essa, dopo un discreto numero di iterazioni, gli individui ottenuti si fossilizzavano in un ottimo locale non andando a decrementare il numero di colori utilizzato.

6.3 Strategie adottate per la Mutation

Per alcune istanze , sono stati condotti degli esperimenti sia applicando la random mutation che la "Vertex mutation" sviluppata ad Hoc con alcuni accorgimenti legati alla rappresentazione del problema.



Il grafico in figura mostra un esempio di due iterazioni dove linea azzurra rappresenta l'andamento del numero di colori in funzione delle valutazioni effettuate utilizzando la random mutation mentre, la linea grigia, mostra l'andamento per la vertex mutation. Si può facilmente notare che per quest'ultimo, il numero di colori converge più velocemente e inoltre raggiunge un valore più basso rispetto al primo caso. Ulteriori aspetti e considerazioni saranno considerate nella parte legata ai risultati ottenuti.

7 Replacement

Questa fase consente di sostituire la popolazione. Fondamentalmente sono state utilizzate due tipi di replacement. La prima è la $\mu + \lambda$ *replacement*. Questa soluzione mette insieme la popolazione attuale con quella appena generata. La nuova popolazione dunque sarà formata dagli individui che presentano il valore di fitness più basso. Questo replacement avviene con una certa probabilità (impostata dal file di configurazione a 0.2). Altrimenti la popolazione appena costruita rimpiazzerà totalmente la popolazione precedente .

8 Struttura del progetto

Il progetto è stato realizzato utilizzando Python. La struttura data è la seguente:

- *instances*: contiene tutte le istanze dimacs da utilizzare
- *logs* : contiene i grafici sia con le prove effettuate con la vertex mutation che con la random mutation.
- *src*: codice sorgente.
- *results*: risultati di ciascuna iterazione. In particolare per ciascuna istanza è stato generato un file contenente diverse informazioni quali Best color, Mean color , Std , Tempo di esecuzione.
- *result vertex mutation*: Risultati con l'utilizzo della vertex mutation.

Ulteriori informazioni riguardanti l'installazione e l'esecuzione del codice sono disponibili alla pagina github .

9 Risultati ottenuti e conclusioni

Sono stati realizzati due esperimenti ove la modifica più sostanziale è stato l'utilizzo nel primo caso della mutazione random, mentre nel secondo caso è stata utilizzata la Vertex mutation. In tabella vengono riportati alcuni dei valori del primo esperimento:

Nome istanza	Miglior colore	Media	Std	Tempo di esecuzione
queen5_5.col	5	6.8	0.74	28m
queen6_6.col	8	9.1	0.7	41m
queen7_7.col	10	10	0	53m
queen8_8.col	14	14.2	0.39	1h and 30m
queen8_12.col	18	19.3	0.89	3h and 12m
queen9_9.col	14	15.1	0.53	2h and 14m
DSJC125_1.col	12	12.4	0.6	347m
DSJC125_5.col	27	28.5	0.89	2h and 1m
DSJC125_9.col	52	52.9	1.13	4h and 16m
le450_15b.col	40	42.4	2.09	4h and 40m
le450_15c.col	57	60.4	2.28	8h and 54m
le450_15d.col	56	59	2.36	8h and 48m

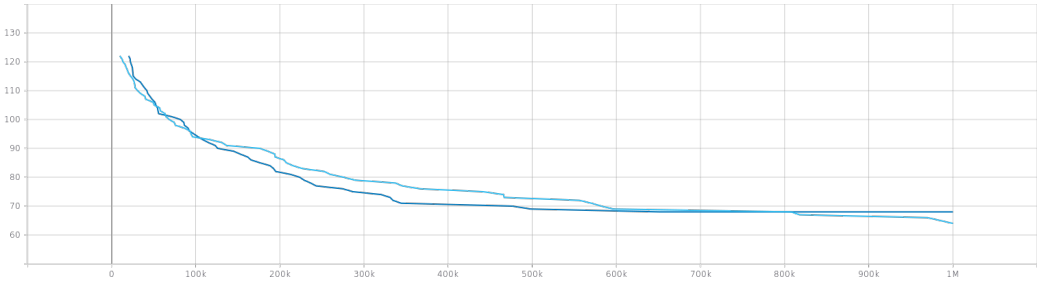
Table 1: Risultati ottenuti con la random mutation

I risultati della tabella ci danno alcune informazioni di cui vale la pena confrontare con la tabella 2 che rappresenta invece gli esperimenti (di tutte le istanze assegnate) effettuati con la vertex mutation:
Confrontando le due tabelle si può notare come nelle istanze più piccole il miglior valore cromatico dei grafi è uguale, tuttavia si osservi che non in tutte le iterazioni (runs) viene ottenuto sempre lo stesso valore. Questo è deducibile dai valori della media. Nelle istanze di medie dimensioni (queen8_8.col e 8_12.col) nella tabella 1 si nota come il numero di colori utilizzati per colorare il grafo è più basso (così come la media). Osservando infine le istanze più grandi, nonostante il numero di colori utilizzato per ciascuna valutazione diminuisce in modo più deciso utilizzando la vertex mutation, i risultati ottenuti mutando in modo casuale il colore dei vertici sono migliori.

Nome istanza	Miglior colore	Media	Std	Tempo di esecuzione
queen5_5.col	5	5	0	24m
queen6_6.col	8	8	0	37m
queen7_7.col	10	10.1	0.3	53m
queen8_8.col	12	12.8	0.4	1h and 19m
queen8_12.col	17	18.2	0.6	2h and 18m
queen9_9.col	15	15.8	0.4	1h and 48m
DSJC125_1.col	11	12.3	0.64	50m
DSJC125_5.col	27	28.9	0.83	3h and 7m
DSJC125_9.col	52	53.3	0.78	4h and 50m
le450_15b.col	47	50.2	1.53	7h and 33m
le450_15c.col	61	64.3	1.9	18h and 37m
le450_15d.col	63	65.5	2.06	1day and 10h

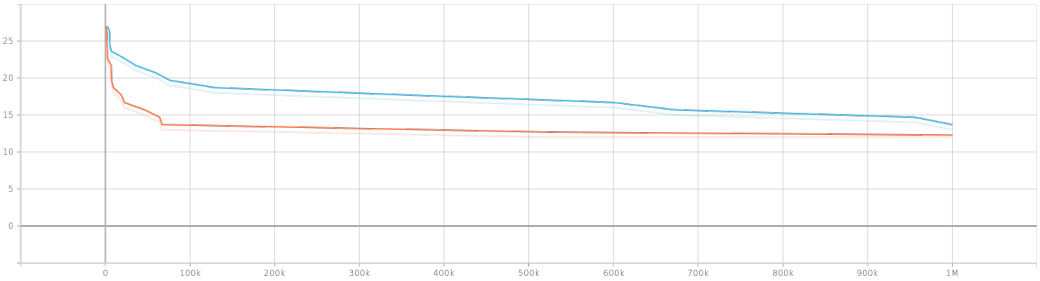
Table 2: Risultati ottenuti con la vertex mutation

Figure 1: linea azzurra: Random mutation - linea blu : Vertex mutation



La figura sovrastante mostra come nel caso del grafo *le450_15d.col* la vertex mutation converge più velocemente, tuttavia alla fine delle valutazioni essa utilizzerà un numero maggiore di colori rispetto alla random mutation.

Figure 2: linea azzurra: Vertex mutation - linea arancione : Random mutation



Da questo grafico (che rappresenta il comportamento per l'istanza queen8_8.col) può essere dimostrato come nel caso di istanze medie la "vertex mutation" non solo converge prima ma (generalmente), riesce a colorare il grafo con un numero più basso di colori rispetto alla random mutation. Come descritto nella documentazione github, del progetto, è possibile vedere i grafici di ciascuna istanza (sia per la random che per la vertex mutation), lanciando il comando *tensorboard –logdir logs*

10 Conclusioni

Dai vari esperimenti e dall' approfondimento di questa famiglia di algoritmi si può notare come risultano fondamentali due caratteristiche:

- Il tempo di esecuzione
- Il risultato ottenuto

Inoltre è essenziale conoscere ed approfondire le peculiarità del problema che si sta provando ad affrontare. Con tale conoscenza infatti, è possibile costruire degli operatori ad hoc che hanno il doppio compito di trovare la soluzione migliore e di non bloccarsi in un ottimo locale. Un implementazione futura, data anche la natura del problema potrebb essere quella di adottare un approccio ibrido utilizzando una local search. Il tempo di esecuzione inoltre, ha inciso in maniera non indifferente, dunque sarebbe opportuno ottimizzare il codice creato o utilizzare un linguaggio più performante come c++.