Equitable Coloring Graph: Risoluzione con algoritmo genetico

DANILO SPINELLI
1000008680

1.0 Il problema

Dato un grafo non orientato G = (V, E) e un intero positivo k, la k-colorazione equa di G è una partizione dell'insieme dei vertici V in k insiemi disgiunti, tali che le cardinalità di due insiemi indipendenti qualsiasi, differiscano di uno al massimo. Il problema della colorazione equa è trovare il k più piccolo per il quale esista una colorazione del grafo equa. Questo problema è noto essere NP-hard e quindi computazionalmente impegnativo. In questa documentazione, presento la mia soluzione per risolvere il problema attraverso un algoritmo genetico.

2.0 Il Progetto

Il progetto è stato implementato completamente in linguaggio Python ed è composto da due script:

(1) main.py

(2) equitable_graph_colorizer.py

2.1 Main

Nel Primo script è presente il main del progetto:

```
if __name__ == '__main__':
    path = 'testGraph/GEOM30b.col'
    params = {
        'iteration_number': 5,
        'K': 10,
        'population_size': 1000,
        'mutation_probability': 0.8,
        'crossover_probability': 0.8,
        'max_improvements': 50,
    }

    egc = EquitableGraphColorizer(path, params)

    risultato = egc.findKMin()
```

Qui vengono inizializzate le variabili fondamentali per l'esecuzione dell'algoritmo genetico:

- K: numero di colori da utilizzare per colorare equamente il grafo.
- iteration_number: quante volte decrementare K di 1 dopo aver trovato una soluzione ammissibile.
- population_size: dimensione della popolazione.
- mutation_probability: probabilità che un individuo della popolazione muti.
- crossover_probability: probabilità che avvenga il crossover fra due individui.
- max_improvements: numero di cicli massimi per migliorare la popolazione affinchè si trovi una soluzione ammissibile.

Oltre a queste variabili viene specificato il path del file contenente il grafo. Successivamente viene creato un oggetto di classe *EquitableGraphColorizer* e viene richiamato il metodo *findKMin()* del secondo script.

Nella seconda parte dello script troviamo il seguente codice:

```
mioGrafo = egc.loadGraph('testGraph/GEOM30b.col')
    G = nx.Graph()
    for x in mioGrafo.vertices:
        G.add_node((x))
    for i in mioGrafo.edges:
        G.add_edge(i[0], i[1])

    nx.draw(G, with_labels=True, node_color=colors, label=all_color_number)
    print("K: ",len(all_color))
    color_patch = mpatches.Patch(label=all_color_number)
    plt.legend(handles=[color_patch])
    plt.show()
```

Attraverso la libreria networkX disegno il grafo con la soluzione che l'algoritmo genetico ha trovato (un esempio di output è mostrato in figura 1).

2.2 Equitable Graph Colorizer

Lo script *equitable_graph_colorizer.py* è composto da 2 classi:

1. Graph
2. EquitableGraphColorizer

La prima classe è la più semplice rappresentazione di un grafo che si possa avere, cioè una lista di vertici e di archi.

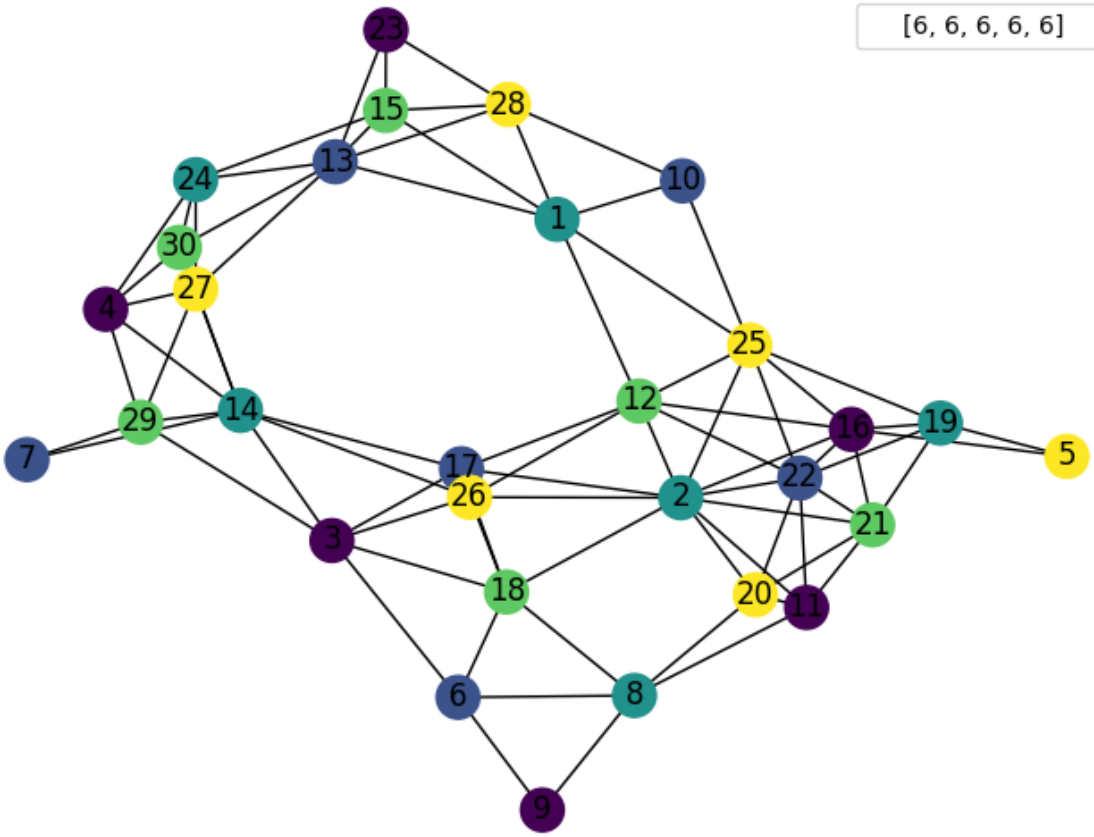

Figure 1: Esempio di output. I numeri in alto a sinistra rappresentano le cardinalità delle classi di colore utilizzate.

```
class Graph:
    def __init__(self):
        self.vertices = []
        self.edges = []

    def __str__(self):
        return 'Nodes: {}, Edges: {}.'.format(len(self.vertices), len(self.edges))
```

La seconda classe invece è composta da 14 metodi che costituiscono il vero e proprio algoritmo genetico.
I metodi implementati sono i seguenti:

1. loadGraph()
2. findKMin()
3. applyGeneticAlgorithm()
4. initializePopulation()
5. calculateFitnessFunction()

6. getColor()
7. selectEvenPopulation()
8. applyMutation()
9. selectAmmissibleSolution()
10. calculateCardinality()
11. selectColorByCardinality()
12. stopConditionReached()
13. createColorClasses()
14. applyCrossover()

Vediamoli tutti per comprendere il funzionamento dell'algoritmo genetico.

1. loadGraph()

Metodo utilizzato per leggere i file di tipo *.col* ed impostare il grafo per eseguire la colorazione.

2. findKMin()

Richiamato direttamente dal main. Qui viene invocato il metodo *applyGeneticAlgorithm()* che non appena trova una soluzione ammissibile, viene inserita nella lista result e viene decrementato K. Se non è stata trovata una soluzione viene restituito l'ultimo elemento conservato in result.

```
    def findKMin(self):
        results = []
        for n in range(self.iteration_number):
            for i in range(self.K):
                self.colors.append(i)
            ris = self.applyGeneticAlgorithm()
            if ris:
                results.append(ris)
                print("Solution found with K = ", self.K)
            else:
                break
            self.K -= 1
            self.colors = []
        return results
```

2. applyGeneticAlgorithm()

Questo metodo raccoglie tutti i passi dell'algoritmo genetico:
1) Inizializzazione della popolazione
2) Selezione degli individui migliori
3) Operazione di crossover
4) Operazione di mutazione

```
    def applyGeneticAlgorithm(self):
    print("Try to found solution with K =", self.K)
    solutions = []
    population = self.initializePopulation()
    t = 0
    program_starts = time.time()
    while not self.stopConditionReached(population, t, solutions, program_starts):
        print("CYCLE: ", t, " IN: ", self.max_improvements)
        population = self.selectEvenPopulation(population)
        population = self.applyCrossover(population)
        population = self.applyMutation(population)
        t += 1
    return solutions
```