

**Studio dell'algoritmo Min-Conflicts con e  
senza *constraint weighting* applicato al  
Map-Coloring Problem**

Pietro Siliani

April 14, 2022

# 1 Introduzione: Testo Elaborato

In questo esercizio si implementano (in un linguaggio di programmazione a scelta) le strategie di ricerca locale min-conflicts e constraint weighting descritte in R&N 2021 6.4. Si comparano quindi le due strategie sul problema della colorazione di mappe, (si possono generare problemi casuali con la strategia suggerita nell'esercizio 6.10 in R&N 2021), studiando empiricamente il tempo di esecuzione in funzione del numero di variabili  $n$  del problema, facendo crescere  $n$  quanto più possibile (nei limiti ragionevoli imposti dall'hardware disponibile).

## 2 Procedimento

**N. B. :** Il linguaggio di programmazione utilizzato per sviluppare ciascun modulo sorgente è stato *Python 3.8*.

### 2.1 Generazione dei Problemi

La procedura seguita per generare automaticamente problemi di Map-Coloring è stata quella descritta nell'esercizio 6.10 in [1]. Un problema di Map-Coloring è identificato attraverso un grafo non orientato  $(V, E)$  tale che:

- Ogni nodo  $v \in V$  rappresenta una variabile, ovvero una regione della mappa.
- Ogni arco  $(u, v) \in E$  indica un confine tra la regione corrispondente ad  $u$  e quella corrispondente a  $v$ , ed un vincolo del tipo  $v.color \neq u.color$ .

I domini delle variabili sono tutti uguali tra loro e corrispondono ad un insieme di colori, ogni colore è identificato da una label numerica che va da 0 a  $k - 1$  dove  $k$  è il numero di colori da utilizzare.

Sono stati generati problemi di diversa dimensione<sup>1</sup> partendo da una dimensione **START\_SIZE** = **20** ed incrementando la dimensione di un passo **SIZE\_INCREMENT** = **20** fino ad un massimo di **MAX\_SIZE** = **600**, per ogni dimensione è stato generato un insieme di 50 problemi, per un totale complessivo di 1500 problemi, per ogni problema è stato fissato  $k = 4$ .

Al fine di implementare la strategia di *constraint weighting* ogni problema presenta un dizionario che mappa ogni vincolo  $(u, v)$  nel corrispondente peso, inizialmente tutti i pesi sono inizializzati ad 1 e sono successivamente aggiornati nel caso in cui si utilizzi *constraint weighting* per la risoluzione dei problemi. Ciascun problema è stato memorizzato sotto forma di un file *.txt*.

### 2.2 Risoluzione dei Problemi e Raccolta Dati

Per risolvere i problemi è stato utilizzato l'algoritmo **MIN-CONFLICTS**, implementato basandosi su quanto riportato nel paragrafo 6.4 di [1] e fissando il valore di **MAX\_STEPS** a 10000. L'implementazione sviluppata utilizza un parametro aggiuntivo: *useWeights*, tale parametro consiste in un booleano che indica se abilitare o disabilitare la strategia di *constraint weighting*.

---

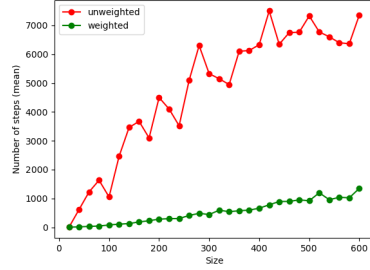
<sup>1</sup>Con dimensione si intende il numero di nodi

Al fine di analizzare le prestazioni di **MIN-CONFLICTS** con e senza *constraint weighting* si è adottata la seguente procedura: inizialmente sono stati caricati i problemi, generati come descritto nel paragrafo precedente, a partire dai file *.txt*, successivamente per ogni problema è stato eseguito l'algoritmo **MIN-CONFLICTS** sia con *constraint weighting* che senza; in entrambi i casi sono stati misurati il tempo di esecuzione (in secondi) ed il numero di passi impiegati dall'algoritmo prima di terminare l'esecuzione. Detto  $S_n$  l'insieme di problemi di dimensione  $n$  è stato poi calcolato, per ogni  $n$ , il tempo di esecuzione medio ed il numero di passi medi dei problemi in  $S_n$ . È stato inoltre registrato il numero complessivo di problemi risolti e non risolti da **MIN-CONFLICTS** con e senza *constraint weighting*. I risultati così ottenuti sono stati infine memorizzati in dei file *.txt*

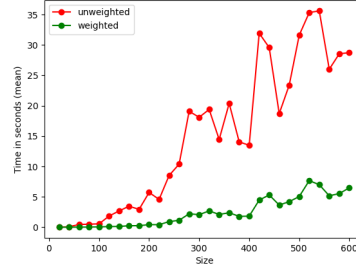
## 2.3 Plotting

I risultati ottenuti come descritto nel paragrafo precedente sono stati impiegati per realizzare dei grafici attraverso l'utilizzo della libreria *matplotlib*. I grafici così ricavati sono riportati e commentati nella sezione Risultati

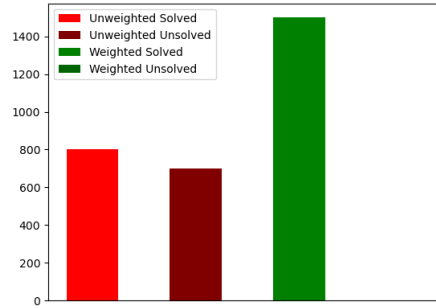
### 3 Risultati



(Figura 1) Numero di passi medio in relazione alla dimensione



(Figura 2) Tempo di esecuzione medio in relazione alla dimensione



(Figura 3) Numero di problemi risolti e non risolti, con e senza *constraint weighting*

Osservando i grafici relativi al numero di passi e al tempo di esecuzione medi in funzione della dimensione dei problemi è evidente il vantaggio derivante dall'utilizzo di *constraint weighting*; senza di esso il massimo del numero medio di passi supera il valore di 7000 passi, ed il massimo del tempo di esecuzione medio è di circa 35 secondi; utilizzando *constraint weighting* si osservano un massimo per il numero medio di passi e per il tempo di esecuzione che si avvicinano rispettivamente a 1500 passi e 7.5 secondi. Globalmente l'impiego di *constraint weighting* ha permesso di ottenere una riduzione percentuale media dell'**84.7%** sui tempi di esecuzione e dell'**86.9%** sul numero di passi rispetto alla versione di standard di **MIN-CONFLICTS**. È possibile infine osservare, in base alla Figura 3, che tutti i problemi avevano soluzione, ciò significa che, per alcuni dei problemi sottoposti all'algoritmo, la versione standard di **MIN-CONFLICTS** avrebbe trovato una soluzione solo dopo un numero di passi superiore a 10000; assumendo quindi di utilizzare un valore di **MAX\_STEPS** tale da permettere a entrambe le versioni dell'algoritmo di trovare una soluzione a tutti i problemi, si sarebbe registrato un aumento del numero di passi e del tempo di esecuzione medi per **MIN-CONFLICTS** senza *constraint weighting*, rendendo

ancora più significativa la riduzione ottenuta tramite l'impiego della suddetta strategia.

## References

- [1] S.J.Russel, P. Norvig, *Artificial Intelligence: A Modern Approach*, Pearson College Div, 4th ed. (2020)
- [2] <https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/>