

# Ricerca Locale per CSP utilizzando l'Euristica Min-Conflicts

Francesco Gradi

## 1 Introduzione teorica

Gli algoritmi di ricerca locale possono essere utilizzati per risolvere efficacemente problemi di soddisfacimento di vincoli (CSP), cioè problemi in cui è necessario assegnare dei valori, presi da domini, alle variabili in modo che si abbia una soluzione accettata. Tali algoritmi cercano di avvicinarsi sempre più all'ottimo di un problema, studiando il comportamento di una funzione obiettivo e facendo delle mosse greedy. In questo esperimento cercheremo di risolvere due CSP, in particolare il problema delle **n-Queens** e del **Map Coloring**, utilizzando l'euristica **Min-Conflicts**. l'idea alla base è molto semplice: si tratta di iterare una funzione per  $k$  steps scelti a tavolino e di compiere ad ogni passo una mossa *greedy*.

Alla  $k$  - *esima* iterazione viene scelta in modo casuale una variabile  $X_i$ , e vengono contati il numero di conflitti che ci sarebbero per ogni altro assegnamento (oppure per una parte di assegnamenti possibili). Si sceglie la mossa che provoca **meno conflitti**, in caso di due o più mosse che provocano lo stesso numero di conflitti, viene scelta in modo casuale una di queste. All'inizio di ogni iterazione viene inoltre chiamata una funzione che verifica lo stato delle assegnazioni delle variabili, restituisce la soluzione se tutti i vincoli sono rispettati.

## 2 Implementazione

L'implementazione del problema è stata scritta in *Python 2.7.13*. Il codice dell'algoritmo è generico, può essere utilizzato per risolvere problemi diversi apportando solo piccole modifiche. Le funzioni di supporto interne all'algoritmo devono invece essere reimplementate a seconda del problema; anche la definizione del CSP può essere libera e non strettamente formale, a differenza di un risolutore general purpose. La funzione *isSolution()* restituisce *True* se l'assegnamento corrente è una soluzione, *chooseVar()* invece sceglie casualmente una variabile da assegnare, verranno poi ispezionati il numero di conflitti in *conflicts()* e scelto un nuovo assegnamento greedy. Codice dell'euristica Min-Conflicts:

---

```
def minConflicts(problem, maxSteps):
    current = initialAssignment(problem)

    for k in range(maxSteps):
        if isSolution(current):
            return current
        var = chooseVar(current)
```

```
value = conflicts(current, var)
var = value

return 0
```

---

### 3 n-Queens

Il **problema** delle **regine** consiste nel posizionare in una scacchiera di grandezza  $n \times n$ ,  $n$  regine degli scacchi, in modo che ognuna di esse non possa mangiarne un'altra. Si tratta di un caso che può essere trattato come un CSP, con una matrice quadrata contenente 0 nelle caselle vuote e 1 nelle celle occupate da una regina (variabili); i vincoli sono che non ci possono essere due 1 in ogni riga, in ogni colonna e in ogni diagonale/antidiagonale della matrice. Inoltre la somma di tutte le caselle deve essere  $n$ , dovendoci essere  $n$  queens. Si può risolvere tale problema

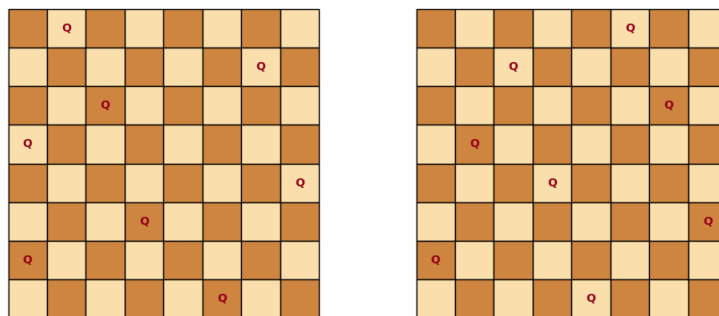


Figure 1: Un assegnamento iniziale e una soluzione con 8 regine prodotta dall'algoritmo e disegnata tramite *drawQueens()*

con una *Backtracking Search*, cioè è possibile eseguire un algoritmo che prova ad assegnare le variabili una dopo l'altra e quando si accorge che è arrivato ad un'inconsistenza fa backtrack, cioè torna indietro nell'albero di ricerca esplorato, e si mette ad eseguire un altro assegnamento. Tale approccio risulta essere però inconcludente quando il problema diventa sufficientemente grande. Trattiamo quindi il CSP come problema di ricerca locale e risolviamolo con l'euristica *Min-Conflicts*.

Il problema è definito tramite un vettore di  $n$  elementi ciascuno dei quali rappresenta l'indice della colonna dove si trova la regina, mentre la riga è rappresentata dalla posizione dell'elemento. Scrivendo il CSP in questo modo due regine non possono trovarsi sulla stessa riga per definizione. Il problema può essere inizializzato in due modi diversi: nel primo si assegnano casualmente le variabili, mentre il secondo prevede di assegnare un posto casuale alla prima regina, poi dalla seconda in poi si assegna la variabile cercando di minimizzare i conflitti (una sorta di funzione *conflicts()*); in questo modo l'algoritmo risulta essere *più efficiente* (dai test il risparmio è significativo). Inoltre la funzione che verifica la soluzione restituisce subito *False* se la somma degli indici di colonna è diversa da  $\frac{n(n+1)}{2}$  (formula di Gauss), perchè gli indici di colonna

devono essere tutti diversi e le regine di conseguenza devono occupare necessariamente ogni riga; questo per risparmiare qualche ciclo macchina.

Eseguendo le funzioni in *demo.py* per testare l'algoritmo emerge che nei problemi con  $n$  basso (dai test  $n < 32$ ) l'algoritmo a volte **non riesce a trovare una soluzione** nel limite massimo di steps (10000), il quale è sicuramente più che sufficiente per risolvere ad esempio un problema a 8 queens; questo perché, essendo un algoritmo di ricerca locale, esso si intrappola in minimi locali (e non globali) o in plateau, cioè in situazioni in cui ogni mossa greedy non riesce a convergere alla soluzione effettiva. Si tratta di un problema patologico della local search, che può essere parzialmente mitigato dalla semplice tecnica del *random restart*.

Dai test emerge che il tempo di esecuzione cresce molto al crescere del numero di regine (si impiega in media 10 secondi a risolvere un problema con 128 Queens), ma comunque il problema risulta essere molto più veloce rispetto ai metodi tradizionali per i csp, che esplode dopo le 30 regine. Con l'hardware a disposizione il tempo medio per risolvere il problema delle 256 regine è stato di 151 secondi. Nei grafici sono indicati i tempi di risoluzione, il numero di passi effettuati e il numero di fallimenti dell'algoritmo (cioè quando arriva a 10000 passi e non trova soluzione, si suppone perciò che sia finito in un plateau o in un minimo locale e non globale) fino a 128 queens.

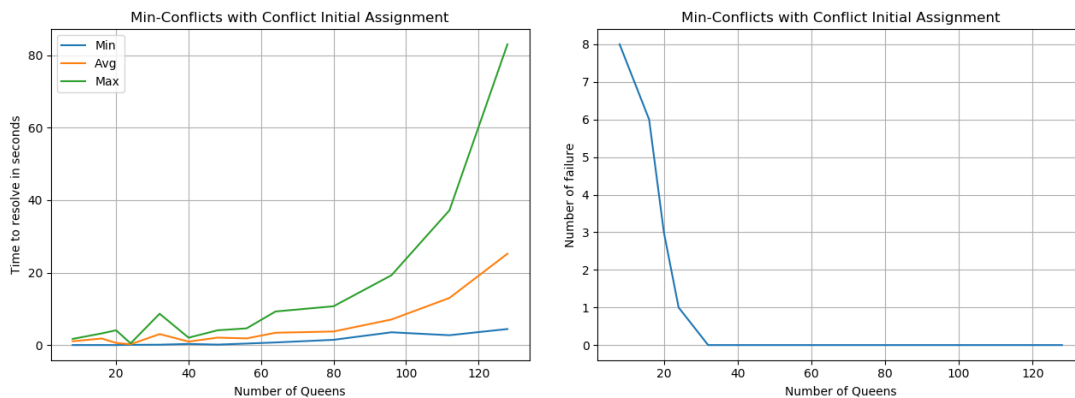


Figure 2: Tempi di esecuzione e numero di fallimenti fino a 128 regine.

Questo algoritmo funziona bene perché le soluzioni sono **densamente distribuite nello spazio** degli stati; in altri termini il problema è **sottovincolato** e questo comporta un buon comportamento nella risoluzione dell'algoritmo di ricerca locale.

## 4 Map Coloring

Il **problema** del **Map Coloring** consiste nel colorare le regioni di una mappa geografica in modo che *stati vicini non abbiano lo stesso colore*. In termini di CSP, le regioni sono nodi di un grafo, le variabili del problema, a cui sono assegnati i colori (domini del problema); i vincoli sono che ciascun nodo vicino (collegato quindi con un arco) deve avere un colore diverso dal nodo stesso preso in considerazione. L'esempio base (presente sul RN) è la carta geografica dell'Australia, ogni regione deve essere colorata diversamente; la mappa dell'Australia è costruita tramite un'apposita funzione e viene visualizzata a schermo tramite la libreria *networkx* combi-

nata con *matplotlib*. Ogni nodo dovrà anche disporre di coordinate  $x$  e  $y$  per essere visualizzati. Si nota che per trovare una soluzione è necessario disporre di **almeno 3 colori**, i tempi di esecuzione sono intorno ai 0.0003 secondi (output del problema in Figure3). Il prossimo passo

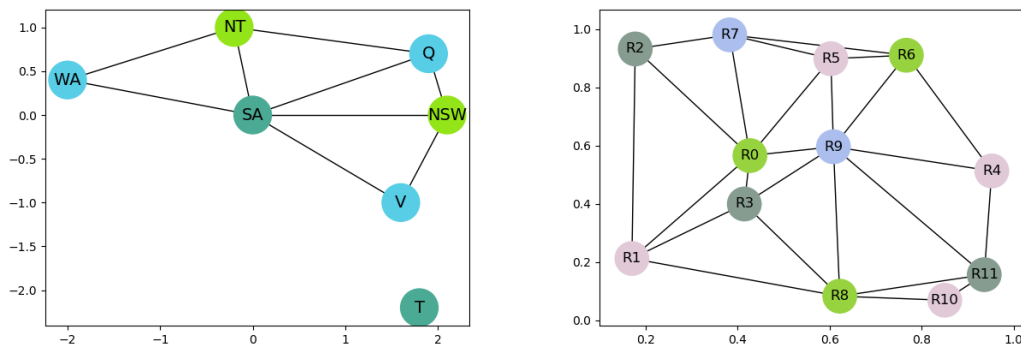


Figure 3: Mappa dell’Australia con colori assegnati e mappa casuale con 12 nodi e 4 colori.

è riuscire a generare **mappe casuali** con  $n$  regioni: si dispongono sul piano  $n$  punti a caso, successivamente si generano archi tra nodi vicini con il vincolo che gli archi non possono interscarsi a vicenda. Per fare ciò abbiamo fatto ricorso all’*Algoritmo di Delaunay*, che crea una triangolazione dei punti che rispetta il vincolo sopra citato. A questo punto è possibile applicare l’euristica Min Conflicts.

Per la mappa da 12 nodi in Figure3 si sono resi necessari 4 colori; in generale (come si vede anche dai test successivi) non è necessario un gran numero di colori, perché al crescere dei nodi, il numero di archi non aumenta tantissimo. Questo fa anche sì che l’algoritmo abbia pochi vicini da esplorare e rende rapida l’esecuzione. Ad esempio per risolvere una mappa di 1024 nodi casuali sono sufficienti (solitamente) 5 colori, mentre il tempo di esecuzione medio si aggira sui 3 secondi. A differenza di n-Queens, i controlli da fare nelle funzioni di supporto dell’euristica sono più semplici, questo impatta sulla complessità dell’algoritmo; inoltre non si registrano casi di minimi locali o plateau (a differenza delle regine).

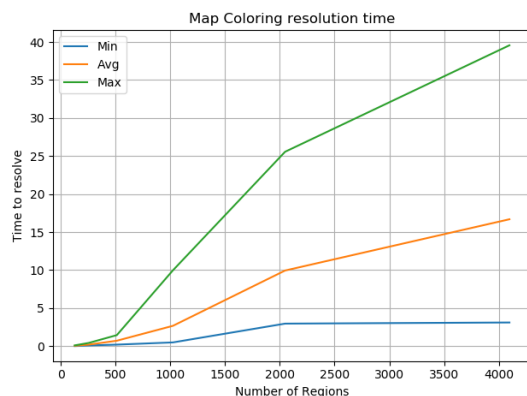


Figure 4: Tempi di esecuzione dell’algoritmo con mappe casuali fino a 4096 nodi.