Lorenzo Macchiarini 6129400

IMPLEMENTAZIONE DELL'ALGORITMO MIN-CONFLICTS

PRESENTAZIONE DEL PROGETTO

In questo progetto ho implementato l'algoritmo di ricerca locale Min-Conflicts per risolvere un problema di soddisfacimento di vincoli. In particolare l'ho testato su due problemi "giocattolo": le n-regine ed il map coloring. Per generare mappe casuali per l'ultimo problema ho seguito la strategia proposta nell'esercizio 6.10 del libro *Artificial Intelligence a Modern Approach* (S. Russel e P. Norvig, Pearson, 2009). Infine, ho studiato il tempo di esecuzione del programma ed altri suoi aspetti al variare del numero di variabili n.

PRESENTAZIONE DELL'ALGORITMO

L'algoritmo Min-Conflict prende in input un numero massimo di passi dopo il quale fermarsi, le variabili, il loro dominio di appartenenza ed i vincoli del problema. La soluzione del problema si ha quando tutti i vincoli sono rispettati. Per questo ad ogni passo l'algoritmo sceglie randomicamente una variabile in conflitto e cerca di rendere minimi i conflitti con le altre variabili. Tale approccio permette di trovare degli ottimi locali molto velocemente ma non sempre questi sono degli ottimi globali, ovvero soluzioni. Per questo motivo negli algoritmi di ricerca locale si cerca di capire se la soluzione corrente è o in un *minimo locale*, in cui ogni mossa fa peggiorare la soluzione corrente, o in un *plateaux*, uno spazio in cui la soluzione corrente non migliora né peggiora ma rimane sempre costante. In questi casi viene generato nuovamente il problema inizializzato in modo randomico in modo da non ricadere nel solito punto di ottimo locale.

IMPLEMENTAZIONE

Per implementare l'algoritmo ho utilizzato il linguaggio di programmazione Python 3.6 facendo uso delle librerie *math*, *bisect*, *random, csv, timeit*. Sebbene io abbia implementato in modo separato la risoluzione ai due problemi proposti ho reso la base del codice dell'algoritmo abbastanza generale da poter essere usata in entrambi. Ho utilizzato come modello lo pseudocodice dell'algoritmo presentato nel libro *Artificial Intelligence a Modern Approach* al capitolo 6.4.

IMPLEMENTAZIONE PER IL PROBLEMA DELLE n-QUEENS

PROBLEMA

Data una scacchiera n*n contenente n regine voglio trovare una loro configurazione in cui nessuna delle regine attacca una delle altre.

CODICE

Per rappresentare ogni regina ho creato una classe ***Queen*** con attributi ***row*** e ***column***. Per come è definito il problema nessuna delle regine può stare sulla stessa colonna, quindi nel codice ho usato come identificatore per le regine l'attributo ***column***.

Ho inserito l'algoritmo in una classe ***Solver*** che ha come attributi: ***n*** la dimensione del lato della scacchiera; ***variables*** una lista contenente le n regine; ***inConflict*** è una lista contenente le sole regine che sono ancora in conflitto; ***currentConflicts*** è una matrice di dimensione n*n in cui ho rappresentato la scacchiera, in particolare per ogni casella della scacchiera ho associato il numero di regine che riescono a raggiungerla con una mossa. Ciò mi ha permesso di non dover calcolare ad ogni passo il numero di conflitti nelle caselle in cui la regina deve scegliere di andare, perché aggiorno *currentConflicts* ogni volta che una regina si sposta in una casella diversa. Ho scelto di mantenere come variabili anche ***randomRestarts*** che conta quanti random restarts vengono fatti per ogni istanza del problema; ***localMinimaIndex*** e ***plateauxIndex*** degli indici che se superata una certa soglia trovata empiricamente permettono di capire all'algoritmo se si è in un punto di minimo locale o in un plateaux.
Come scritto in precedenza mentre la base dell'algoritmo è generale ogni funzione richiamata è specifica per il problema sottoposto: ***initSolution*** inizializza *currentConflicts* con una matrice di zeri e *variables* con regine su colonne diverse ma con righe scelte randomicamente, quindi chiama ***calculateConflicts*** che calcola quante regine possono raggiungere ogni casella (tramite ***addConflict***) e aggiunge a *inConflict* ogni regina che si trova in conflitto. Ad ogni passo verifico se l'algoritmo ha trovato la soluzione controllando con ***checkSolution*** che non ci siano regine in *inConflict*, nel caso ritorno tutte le regine ed esco dal programma. Se ciò non avviene scelgo casualmente una regina da quelle in conflitto e trovo la casella della colonna a lei associata che le fa fare il minimo numero di conflitti. Se la condizione di restart ***restartCondition*** è verificata allora inizializzo nuovamente il problema in modo randomico con ***randomRestart***. Infine con la funzione ***updateConflicts*** aggiorno i conflitti, aggiungo a *inConflicts* le eventuali regine che risultano avere conflitti dato il nuovo assegnamento e incremento le variabili *plateauxIndex* e *localMinimaIndex* rispettivamente se il nuovo numero di conflitti è lo stesso rispetto al precedente e se il nuovo numero di conflitti è maggiore rispetto al precedente, ovvero se non mi sono avvicinato alla soluzione o se addirittura mi sono allontanato.
Se non è stata trovata la soluzione al problema in ***maxSteps*** passi ritorno *None*.

CONSIDERAZIONI

Per capire quali fossero i parametri in base a cui fare un random restart, ho eseguito l'algoritmo per 40 volte per ogni taglia su problemi con n crescente fino a 200 prendendo però solo le esecuzioni che avessero portato ad una soluzione.
Ho notato che se si arriva ad una soluzione l'algoritmo non assegna mai un valore di minimo conflitto maggiore del precedente, per questo ho imposto che quando *localMinimaIndex* è 1 allora viene eseguito un random restart. Ho notato anche che aumentando la taglia del problema il numero di volte in cui l'algoritmo assegna un valore di minimo conflitto uguale al precedente aumenta fino ad un valore di circa 120 attorno a cui oscilla, come si può vedere in *Figura 1*. Per questo ho assegnato a *plateauxIndex* 120 come valore massimo dopo il quale viene eseguito random restart. Con queste premesse ho voluto tracciare un grafico del numero di random restart e del running time. Ho eseguito l'algoritmo 10 volte per ogni taglia fra 5 e 600 con passo 5.

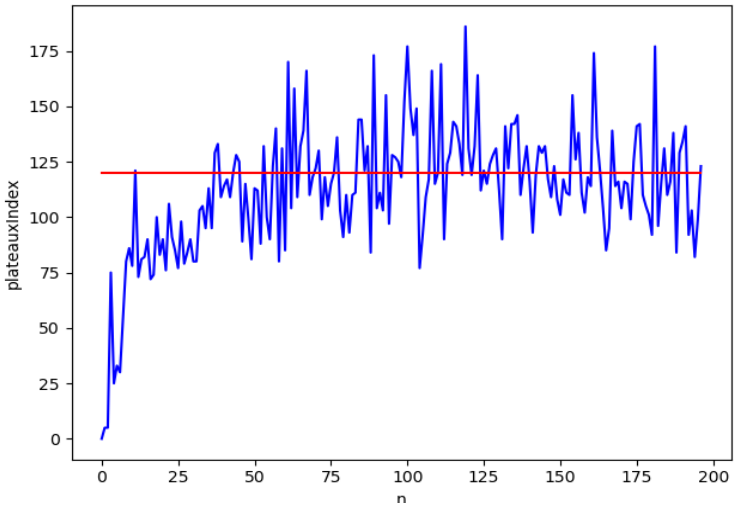

Figura 1: Andamento di plateauxIndex all'aumentare di n

Dai grafici in Figura 2 e Figura 3 è possibile vedere rispettivamente come il numero dei random restarts rimanga molto basso sebbene la taglia del problema aumenti molto e come il running time aumenti linearmente all'aumentare della taglia del problema.

Nell'esecuzione del programma ho dovuto limitare il numero di passi dell'algoritmo a 10000 perché il running time altrimenti sarebbe stato troppo alto ed il mio computer non sarebbe riuscito ad eseguire questi test in tempi ragionevoli. Per questo nei grafici ho voluto rappresentare con i punti rossi le istanze dei problemi per cui l'algoritmo non è stato in grado di trovare la soluzione nel numero di step fissato.

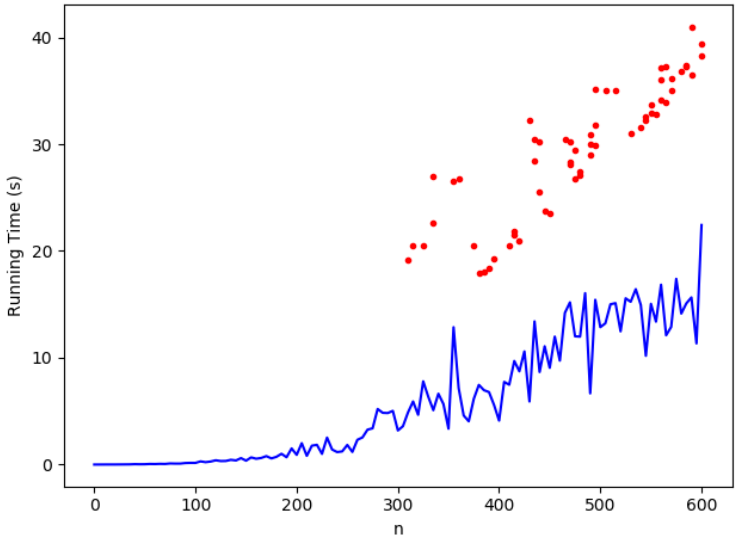

Figura2: Andamento del numero di Random Restarts all'aumentare di n

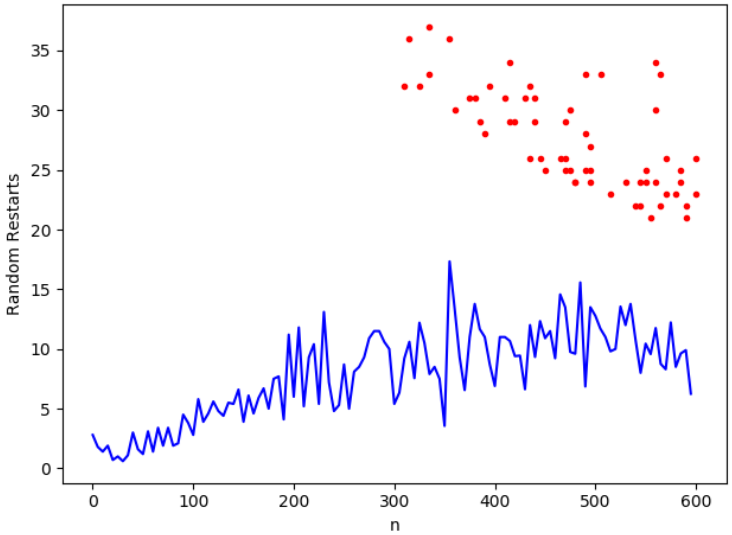

Figura 3: Andamento del tempo di esecuzione all'aumentare di n

IMPLEMENTAZIONE PER IL PROBLEMA DEL MAP COLORING

PROBLEMA

Data una mappa voglio sapere se ogni regione può essere colorata con uno del k colori in modo che le regioni a lei adiacenti non abbiano il suo stesso colore.

CODICE

Il codice che ho implementato per questo esercizio si divide in due parti: la generazione della mappa e l'applicazione dell'algoritmo Min-Conflict per trovare una soluzione al problema.
Per la generazione casuale delle mappe ho seguito l'idea presentata nell'esercizio 6.10 del libro *Artificial Intelligence a Modern Approach*. Da un quadrato di lato n ho preso casualmente n punti e ad ogni passo ne ho collegato uno con il punto più vicino fra quelli disponibili. Ho ripetuto questo metodo fino a quando non era possibile creare altri collegamenti (archi) senza che ci fossero intersezioni con quelli già creati. Per controllare se due archi si intersecano ho utilizzato l'algoritmo presentato al capitolo 33.1 nel libro *Introduction to Algorithms* (T. Cormen, C. Leiserson, R. Rievest, C. Stein, The MIT Press, 2014).
Ho creato la classe ***Region*** per rappresentare una regione che ha come attributi: ***name*** l'identificatore, ***x*** e ***y*** le coordinate in cui viene creata, ***color*** il colore che le è associato e ***arcs*** una lista in cui ho inserito tutte le regioni a cui è collegata da un arco.
Ho creato la classe ***MapGenerator*** che ha come attributi: ***regions*** l'insieme delle regioni create, ***arcs*** l'insieme degli archi che connettono regioni ed ***n*** il numero di regioni da creare. Le funzioni presenti sono: ***generateMap*** che crea le n regioni, assegna loro una posizione ammissibile e chiama la funzione *regionConnection*; ***regionConnection*** che fino a quando ci sono regioni disponibili a creare un arco ne prende una in modo randomico e prova a creare archi che da questa raggiungono le altre regioni. Se l'arco trovato non è già stato creato o non interseca altri archi allora viene inserito nella lista ***momentaryArcs***. Una volta che sono stati controllati tutti gli archi possibili si sceglie come arco da creare quello che collega la regione alla regione a distanza minore, infine aggiunge l'informazione dell'arco nelle liste *arcs* delle due regioni.

Nella classe ***Solver***, come per le n Queens, ho implementato l'algoritmo Min Conflict. Gli attributi della classe sono: ***variables*** la lista delle regioni; ***inConflict*** la lista delle regioni che sono in conflitto con almeno una delle regioni adiacenti; ***n*** il numero delle regioni; ***k*** il numero di colori ammessi per colorare la mappa; ***plateauxIndex, localMinimaIndex*** e ***randomRestarts*** con scopi analoghi a quelli delle stesse 3 variabili del problema delle n Queens. L'algoritmo inizializza il problema associando colori randomici ad ogni regione. Chiamando ***updateValue*** controlla se la regione è in conflitto con altre e nel caso la aggiunge a *inConflict*. Il controllo della soluzione e la scelta della nuova variabile da controllare sono analoghe a quelle viste nel problema delle n Queens. Nella funzione in cui scelgo il colore che genera il minor numero di conflitti ***getMinConflicting*** ho voluto calcolare anche il numero di conflitti iniziale per capire se il valore trovato dalla funzione migliorasse la soluzione o meno. In quest'ultimo caso incremento *localMinimaIndex* se peggiora e *plateauxIndex* se non migliora. Se non è stata trovata la soluzione in ***maxSteps*** passi esco dal programma ritornando *None*.

CONSIDERAZIONI

A differenza di quanto ho registrato per il problema delle n Queens, in questo caso la buona riuscita nella ricerca della soluzione e il tempo di esecuzione dipendono fortemente dalla taglia del problema e, in particolare, ho notato che dipende da come viene generata la mappa. Il numero di archi e quindi di vincoli presenti in ogni problema cresce linearmente all'aumentare di n come si può vedere in *Figura 4*, ciò rende molto difficoltosa la ricerca di una soluzione. Infatti pur variando il numero di colori disponibili fra 3 e 5 l'algoritmo non trova la soluzione a problemi con dimensione maggiore di 20. Nella *Figura 5* i puntini rappresentano i problemi risolti mentre i punti più grossi i problemi che dopo *maxSteps* passi non hanno trovato la soluzione; il colore blu indica che sono stati usati 3 colori, il verde indica che ne sono stati usati 4 e il giallo 5.

Si vede che da un certo punto in poi, a mio parere basso, l'algoritmo non riesce a trovare più soluzioni ai problemi proposti.

In questo algoritmo ho posto come limite massimo a *localMinimaIndex* e *plateauxIndex* il valore 1 perché sperimentalmente ho visto che se l'algoritmo riesce a trovare una soluzione non passa da assegnamenti che peggiorano o lasciano invariato il numero di conflitti.

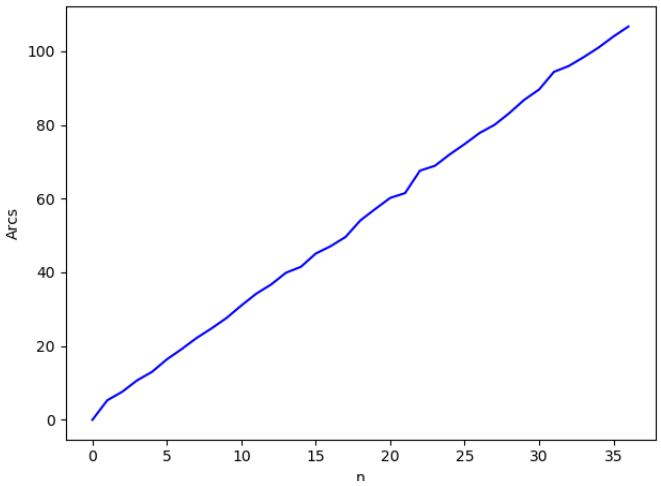

Figura 4: Numero di archi al crescere di n

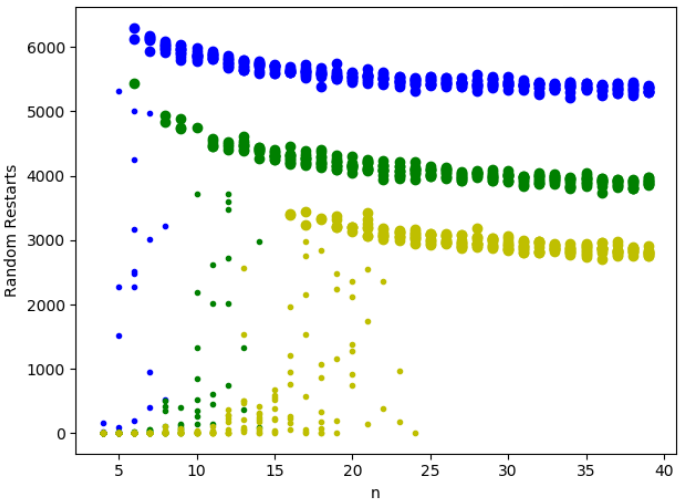

Figura 5: Mostra il numero di Random Restarts al variare della dimensione del problema e del numero di colori usati per la risoluzione

SPECIFICHE DEL CALCOLATORE USATO

Il calcolatore usato per tali test è un ASUS N550LF; CPU: Intel Core i7-4500U 2-Core 2.4GHz; Sistema Operativo: Windows 10 Home; Memoria Secondaria: Samsung SSD 860 EVO 500 Gb; RAM: 8gb