Nonogram

Sabrina Nicosia

8 marzo 2024

Sommario

Il progetto svolto consiste nella creazione di un software, facendo uso del linguaggio Python, che cerca di risolvere il Nonogram, gioco di logica noto anche come "Picross", composto da una griglia, di varie dimensioni, che presenta sequenze di numeri lungo le righe e le colonne, che stanno ad indicare le lunghezze di celle contigue da riempire, con almeno una cella vuota tra ogni gruppo. L'obiettivo è quello di rispettare i vincoli espressi sulle righe e sulle colonne in modo da determinare quali celle devono essere riempite per la costruzione di un'immagine nascosta. Per ottenere ciò, viene implementato un algoritmo metaeuristico, ovvero Tabu Search, che verrà testato su diversi esempi, aventi diverse dimensioni, per valutarne le prestazioni.

1 Struttura del software

Il software sviluppato è stato strutturato in diversi file Python:

- `entities.py`: presenta due classi, ovvero `Nonogram`, che contiene diverse informazioni sulla griglia di gioco, e `Memory`, che implementa una memoria a breve termine;
- `utils.py`: implementa diverse funzioni, tra cui il calcolo della funzione obiettivo, per una determinata soluzione, e la generazione dei *neighbors*; insieme a tutto ciò che serve per la realizzazione del metodo Tabu Search;
- `main.py`: può essere eseguito in due modalità: test e predict; da specificare mediante il parametro `--mode` seguito da una delle due opzioni. La procedura di `predict` prende da terminale il path di un file, in formato .csv, che presenta i vincoli sulle righe e sulle colonne, disposti su due righe. Il flusso principale consiste nella creazione dell'oggetto di classe `Nonogram`, coi rispettivi vincoli richiesti, a cui viene applicato l'algoritmo metaeuristico ed infine restituita la soluzione ottenuta, insieme alla funzione obiettivo associata ad essa. Mentre, per quanto riguarda la modalità di `test`, essa prende da terminale la dimensione della griglia su cui si desidera testare il modello, ed il numero di epoche che si vogliono eseguire per ogni esempio che si andrà a testare; per numero di epoche si intende il numero di volte in cui verrà chiamato l'algoritmo tabu search sul medesimo esempio.

1.1 Classe `Nonogram`

La classe `Nonogram` presenta diversi attributi che contengono alcune informazioni sulla soluzione corrente e sui vincoli richiesti. In particolare, oltre al numero di righe e colonne della griglia e i rispetti vincoli su di esse, vi sono tre matrici: `game_table` che rappresenta la griglia di gioco in cui il valore 1 rappresenta il coloramento della rispettiva cella, mentre, `group_mask_row` e `group_mask_col` sono delle maschere che identificano le sequenze di celle da riempire in ciascuna riga e colonna; ovvero, si avrà il valore -1 se la cella non va riempita, quindi non appartiene a nessun gruppo, altrimenti avrà un valore che rappresenta l'identificativo del gruppo a cui appartiene (la numerazione va in ordine crescente). Inoltre, vengono conservate le lunghezze dei gruppi di celle contigue col valore 1, per ogni riga e colonna, in modo da tener traccia della lunghezza dei gruppi correnti, che possono essere comparati, più facilmente, con i vincoli richiesti. Per facilitare ciò, vengono memorizzati due array; il primo, chiamato `correct_row_groups`, può assumere i valori 0, 1 e 2 in modo da poter distinguere tre casi: le lunghezze dei gruppi non coincidono con quelle espresse nei vincoli; le lunghezze coincidono; le lunghezze dei gruppi coincidono e non si possono effettuare variazioni, ovvero la rispettiva riga non può

essere modificata. Ciò verrà utilizzato dalla procedura che inizializza una soluzione random per far sì che i vincoli sulle righe vengano soddisfatti. Mentre, per quanto riguarda le colonne, viene utilizzato l'attributo `correct_col_groups`, il cui contenuto differisce rispetto al precedente. Ogni elemento è un array di lunghezza pari al numero di gruppi presenti nella rispettiva colonna. Ciascun elemento può assumere il valore 0, il che significa che la lunghezza del relativo gruppo è stata rispettata, altrimenti avrà assegnato il valore 1. Tale vettore viene conservato poiché sarà utile nel calcolo della funzione obiettivo che potrà disporre di un array che mantiene il numero di vincoli violati sulle colonne.

Al suo interno presenta un metodo che permette di inizializzare una soluzione in modo random, seguendo però un certo criterio. Tale funzione, denominata `initialize_cells_values`, chiama inizialmente il metodo `random_initialization` che, per ogni riga, calcola il numero totale di celle che dovrebbero avere il valore 1 in modo da estrarre in modo random un numero di colonne pari al numero precedentemente calcolato e far sì che ogni riga abbia il numero totale richiesto di celle col valore 1. Dopo di che, aggiorna determinate informazioni, quali matrici delle maschere e i vettori che indicano il soddisfacimento o meno dei vincoli. Tuttavia, tale procedura non è sufficiente per rispettare i vincoli sulle righe, perciò, la funzione più esterna effettua uno swap tra 2 celle estratte randomicamente da una determinata riga fin quando i rispettivi vincoli sono stati soddisfatti, ovvero la lunghezza dei singoli gruppi coincide. Facendo ciò, per arrivare alla soluzione desiderata basterà controllare e soddisfare unicamente i vincoli sulle colonne.

1.2 Classe `Memory`

Questa classe viene utilizzata dall'algoritmo metaeuristico che necessita di una memoria a breve termine che gli permetta di non esplorare soluzioni che ha già visitato di recente, in modo da non entrare in loop, oltre che ridurre il numero di iterazioni da effettuare.

Come attributi presenta un array, che prende il nome di `tabu_list`, la cui dimensione massima viene memorizzata in `max_size`. Quest'ultimo parametro serve a far sì che, se l'array non ha ancora raggiunto la sua dimensione massima il metodo `add` andrà semplicemente ad aggiungere l'elemento passato alla funzione, altrimenti, se la dimensione massima è già stata raggiunta, prima di inserire il nuovo valore viene rimosso l'elemento in cima, ovvero quello che più remoto. Infine, il metodo `clear_memory` si occupa di ripulire la memoria andando a rimuovere un numero di elementi pari alla dimensione corrente della `tabu_list` fratto 2, dal cui risultato si va a prendere l'intero successivo facendo uso di `math.ceil`.

2 Tabu search

Tabu Search è un algoritmo metaeuristico utilizzato per risolvere problemi di ottimizzazione. È particolarmente efficace per i problemi di ottimizzazione combinatoria, in cui l'obiettivo è trovare la migliore disposizione di un insieme finito di elementi. Esso è stato introdotto da Fred Glover alla fine degli anni '80 e da allora viene ampiamente utilizzato in vari campi.

Inizialmente costruisce una prima soluzione per il problema da risolvere, che può essere inizializzata in modo random o seguendo un qualche criterio, come punto di partenza. A partire da tale soluzione fa uso di una procedura di ricerca locale per spostarsi in modo iterativo da una potenziale soluzione ad un'altra, solitamente migliore della precedente, che si trova nelle sue vicinanze, ovvero si ottiene effettuando dei piccoli cambiamenti alla soluzione corrente. Lo spostamento dovrebbe avvenire se la funzione obiettivo definita, che sta ad indicare la bontà di una determinata soluzione, e che quindi dipende prettamente dal problema con cui si sta trattando, migliora. Ciò avviene fino a quando non è stato soddisfatto un criterio di arresto; generalmente, un limite di iterazioni eseguite o una soglia di punteggio ottenuta. Dato che le procedure di ricerca locale spesso si bloccano in aree con punteggi scarsi o in aree in cui i punteggi si stabilizzano, tale metodologia, al fine di evitare queste insidie ed esplorare le regioni dello spazio di ricerca che altrimenti non verrebbero analizzate, permette di spostarsi in una soluzione vicina, seppur incrementa la funzione obiettivo stabilita; oppure, di esplorare soluzioni che si discostano maggiormente da quella corrente, per far sì che non si rimanga bloccati in un minimo locale. Inoltre, per evitare che l'algoritmo entri in loop, quindi oscilli sulle stesse soluzioni in un numero

esiguo di mosse, o che esplori le stesse soluzioni nel breve termine, si fa uso di una memoria che tiene traccia delle più recenti soluzioni precedentemente esplorate in modo da proibire l'analisi di *neighbors* che sono stati già visitati in un determinato numero di mosse precedenti.

2.1 Implementazione

La funzione che implementa l'algoritmo Tabu Search è presente nel file `utils.py` sotto il nome `tabu_search`. In primo luogo, viene chiamato il metodo `initialize_cells_values`, citato già in precedenza, che permette di ottenere la soluzione di partenza, con i vincoli richiesti per ogni riga. Successivamente, viene chiamata su di essa la funzione `objective_function` che permette di ottenere il valore della funzione obiettivo definita, in modo da mantenere il valore della soluzione corrente.

La funzione obiettivo viene calcolata basandosi su diversi criteri che si vogliono attenzionare, e quindi ottimizzare. In particolare, vengono considerati tre valori che vengono ottenuti con le seguenti funzioni:

- `calculate_completeness`: viene calcolata servendosi del vettore `correct_col_groups`. In particolare, viene effettuata la somma dei valori di ogni suo elemento, che è un ulteriore array che presenta il valore 1 se la lunghezza del rispettivo gruppo, relativo ad una colonna, non viene rispettata. Infine viene effettuata un'altra somma dei valori ottenuti sommando ciascuna componente, ottenendo così un valore che rappresenta il numero totale di gruppi che non rispettano la lunghezza richiesta, cioè quante violazioni sulle colonne sono presenti.
- `calculate_exceeding_groups`: estrae un valore positivo, che rappresenta una penalità da assegnare alla soluzione corrente. Per il suo calcolo, viene effettuata una somma del valore assoluto della differenza che vi è tra il numero di gruppi atteso e quello che riporta la soluzione che si sta analizzando; tutto ciò per ogni colonna. In questo modo si ottiene un valore che quantifica, nel complesso, quanti surplus o deficit sono presenti rispetto al numero di gruppi desiderati;
- `calculate_exceeding_len`: analogamente al caso precedente, questa funzione calcola una penalità andando a sommare, per ogni colonna, il valore assoluto della differenza tra il numero totale di celle col valore 1 desiderate rispetto al numero che possiede la soluzione corrente.

La funzione obiettivo finale viene calcolata mediante la funzione `calculate_completeness` che effettua una media pesata dei valori ottenuti con le precedenti tre funzioni, i cui pesi sono stati impostati, in seguito a diverse prove, rispettivamente nel seguente ordine: 0.20, 0.40 e 0.40.

Il flusso del metodo Tabu Search prosegue andando ad identificare le righe i cui gruppi posso subire delle variazioni, tralasciando quelle che non bisogna più modificare. Ciò avviene estraendo gli indici, dall'array `correct_row_groups`, che presentano un valore diverso da 2. Inoltre, vengono inizializzate diverse variabili, tra cui: `count`, che tiene conto del numero di iterazioni eseguite dall'algoritmo, `current_min` che memorizza il valore della funzione obiettivo della soluzione corrente; infine, viene creata un'istanza della classe `Memory` a cui viene impostato, come numero massimo di elementi, un valore pari al numero di righe che debbono essere modificate moltiplicato per 3.

Il fulcro della funzione `tabu_search` consiste in un ciclo *while* che viene eseguito fin tanto la funzione obiettivo della soluzione corrente non raggiunge il valore 0, oppure se è stato raggiunto un numero massimo di iterazioni, impostato a 1000. Al suo interno, viene chiamata la funzione `neighb` che permette di ottenere soluzioni vicine a quella corrente.

In particolare, la funzione `neighb`, itera per ogni riga, che può essere modificata, e va ad estrarre gli identificatori univoci dei gruppi presenti in essa. Iterando su quest'ultimi, estrae per ognuno di essi gli indici di colonna in cui sono posizionati e chiama la funzione `get_admissible_range` per ottenere un range che identifica di quanto può essere spostato il rispettivo gruppo, sia in avanti, che indietro.

La funzione `get_admissible_range` estrae diversi valori necessari all'esecuzione dell'analisi che deve effettuate, tra cui: indice della prima e dell'ultima cella relativa al gruppo che si intende spostare; se esiste un gruppo precedente, indice dell'ultima cella relativo ad esso, mentre, se esiste un gruppo successivo, indice della prima cella. Successivamente, la procedura può eseguire il calcolo del range ammissibile in due modalità, che vengono gestite attraverso la scelta casuale di un numero compreso tra 0 e l'intero successivo della divisione tra il numero di colonne per due; se il numero estratto è pari a 0, verranno analizzate tutte le posizioni in cui il gruppo può essere spostato, altrimenti, si potranno effettuare solo dei passi di una posizione, in avanti o indietro. Quindi, nel primo caso, verrà analizzato il range di movimento che il gruppo di celle può effettuare, andando a controllare l'esistenza di un gruppo precedente, per far sì che, nel caso in cui fosse presente, venga lasciata una cella di divisione; lo stesso procedimento viene eseguito per verificare se l'esistenza di un gruppo successivo. Nel caso in cui non vi fosse un gruppo, precedente o successivo, viene controllato che il gruppo non superi la lunghezza della griglia. In tal modo viene estratto il valore massimo di spostamento sia a destra che a sinistra, i cui valori sono espressi come spostamenti positivi nel primo caso, e negativi nell'altro. In entrambe le modalità, prima di restituire il range di spostamenti possibili, viene verificato che ogni spostamento non sia già stato analizzato nel breve termine, andando a chiamare la funzione `check_history`. Tale funzione controlla se la configurazione di una determinata riga, ottenuta effettuando un certo spostamento del gruppo in questione, è presente nell'istanza della classe `Memory`, chiamata cache. In caso positivo, lo spostamento non viene restituito tra quelli ammissibili, altrimenti verrà incluso.

A questo punto, la funzione `neigb` possiede i valori di offset che rappresentano di quanto il relativo gruppo può essere shiftato. Se non vi sono valori ammissibili si passa al successivo gruppo della riga, se presente, oppure alla prossima riga da analizzare. Altrimenti, per ogni valore nel range ammissibile, viene costruita una soluzione andando ad effettuare il rispettivo spostamento del gruppo in questione, ed aggiunta la configurazione ottenuta alla lista dei neighbors da restituire. In quest'ultimo caso, verrà aggiunta la riga modificata nella cache, in modo da non ripetere la stessa configurazione nel breve termine. Una volta analizzate tutte le righe, e i rispettivi gruppi presenti, viene restituita la lista di neighbors ottenuta.

All'intero del `while`, se la lista restituita da `neighb` non presenta alcun elemento, viene chiamato il metodo `clear_memory` che rimuove alcuni elementi dalla memoria, in modo che alla prossima iterazione non si rischia di non ottenere soluzioni, dopo di che si passa all'iterazione successiva col comando `continue`. In caso contrario, vengono calcolate le funzioni obiettivo delle soluzioni vicine a quella corrente, in modo da estrarne il valore minimo e passare alla soluzione corrispondente, anche se quest'ultima non presenta un valore inferiore rispetto al valore della soluzione precedente. Infine la procedura restituisce il valore della funzione obiettivo dell'ultima soluzione ottenuta.

3 Test del modello

Per la fase di test del modello, è necessario eseguire in modalità di `test` il file `main.py`. Ciò permette di testare le griglie presenti all'interno della cartella `Examples`, che contiene due sottocartelle rispettivamente per gli esempi con dimensione 10x10 e 15x15, che verranno selezionate sulla base del parametro passato dopo il comando `--dim`, ovvero 10 oppure 15. Una volta estratti i file di una determinata dimensione, per ciascuno di essi viene chiamata la procedura tabu search un numero di volte pari al parametro relativo alle epoche da eseguire, in modo da ottenere, per ogni epoca, il valore della funzione obiettivo della soluzione restituita. I valori ottenuto sulle varie epoche verranno disposti in ordine crescente ed aggiunti ad un array che mantiene i valori per tutti gli esempi da analizzare.

Per ogni dimensione della griglia sono presenti 10 esempi di cui la metà hanno una difficoltà semplice, mentre i restanti sono più complessi. Per stimare matematicamente la difficoltà di ogni esempio, viene utilizzato il metodo `get_difficult` presente nella classe `Nonogram`. Esso quantifica la difficoltà di un determinata griglia di esempio andando a calcolare, per ogni riga, il numero di celle da riempire ed il rispettivo numero di gruppi presenti; ciò viene calcolato anche per ogni colonna. Come valore finale, effettua una somma pesata tra il numero totale di celle da riempire, relativo alle righe, fratto il numero di gruppi totali ottenuti e lo stesso rapporto tra i valori ottenuti per le colonne. In questo modo, si è

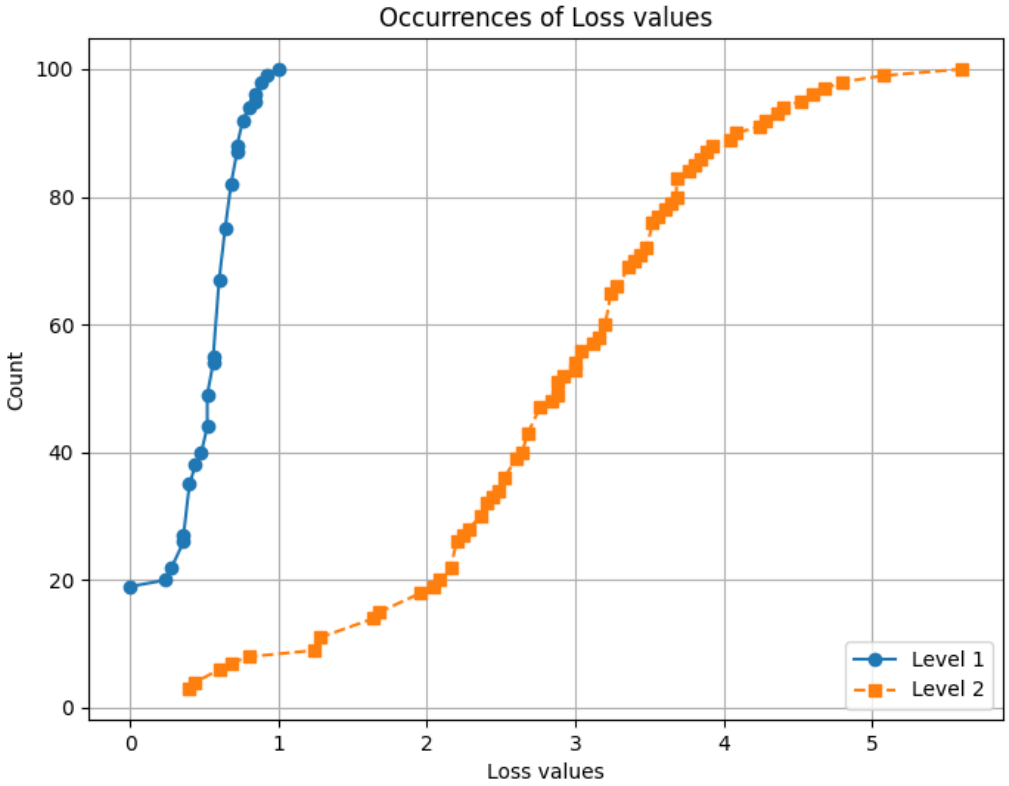

Figura 1: Grafico relativo all'esecuzione di 100 epoche degli esempi in esame con dimensione pari a 10x10, i quali sono stati suddivisi in due categorie.

in grado di distinguere i picross più facili da quelli più complessi, facendo sì che si possano ottenere statistiche differenti per ciascuna categoria. In particolare, viene effettuata tale distinzione in modo da effettuare una media dei valori della funzione obiettivo, per ogni epoca eseguita, per gli esempi appartenenti alla stessa categoria e vedere in media come si comporta l'algoritmo.

3.1 Risultati ottenuti

Viene estratto un grafico che presenta nell'asse delle ascisse i valori della funzione obiettivo, mentre nell'asse delle ordinate il conteggio dei valori inferiori al valore della funzione obiettivo. In questo modo si può visualizzare, in media, quante volte l'algoritmo ha restituito un valore inferiore ad una certa soglia. Come citato in precedenza, i picross sono stati suddivisi in due categorie, in base alla difficoltà riscontrata, che riportano l'etichetta *Label 1* per gli esempi pià semplici, mentre *Label 2* per quelli più complessi. Inoltre, sono state provate due dimensioni differenti delle griglia di gioco, ovvero 10x10 e 15x15.

Il grafico ottenuto per gli esempi aventi una dimensione pari a 10x10 è mostrato in figura 1. Si può notare una marcata differenza tra gli esempi di *Level 1* e quelli di *Level 2*; infatti, i primi, in media, hanno ottenuto un valore di loss che oscilla tra 0 e 1, mentre, i valori degli esempi più complessi rientrano in un range che possiede un valore massimo parecchio superiore. Inoltre, emerge che non tutti gli esempi più complessi hanno ottenuto almeno un valore pari a 0, segno del fatto che non è stata riscontrata la soluzione esatta al problema. Al contrario, gli esempi più semplici hanno ottenuto la soluzione richiesta in, almeno, 20 epoche eseguite.

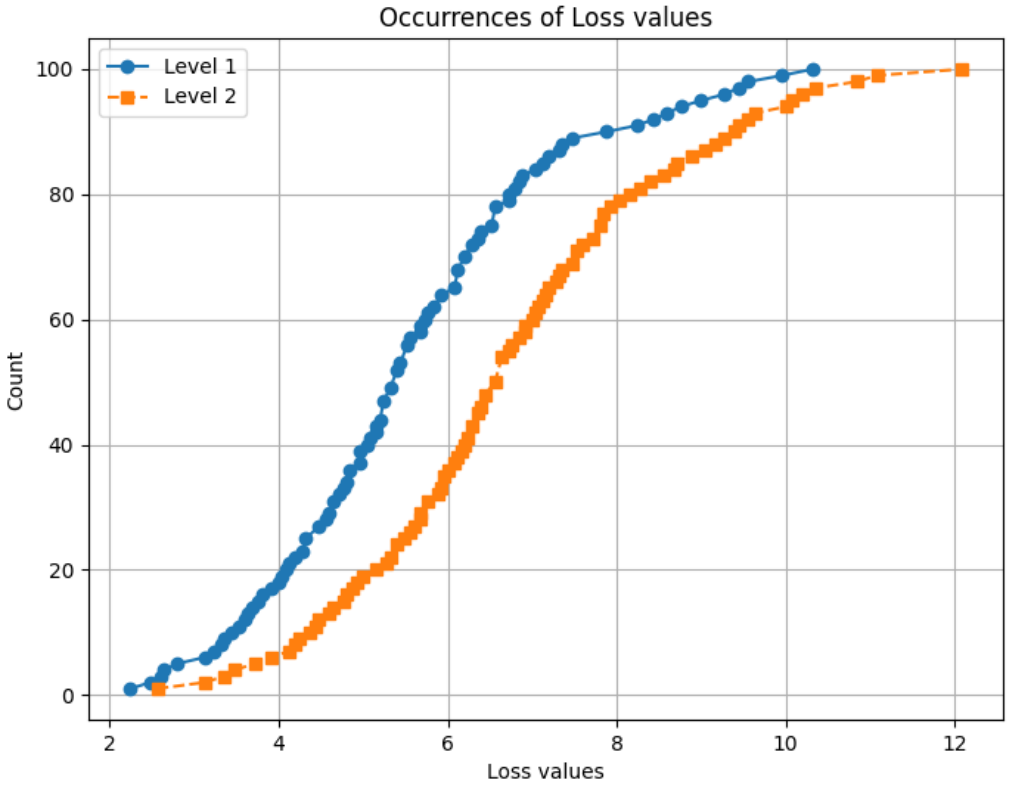

Figura 2: Grafico relativo all'esecuzione di 100 epoche degli esempi in esame con dimensione pari a 15x15, i quali sono stati suddivisi in due categorie.

Per quando riguarda, invece, gli esempi con dimensione 15x15, il grafico ottenuto è mostrato in figura 2. Sebbene le due curve siano più vicini e seguendo un andamento piuttosto simile, è comunque presente una certa differenza che sta ad indicare che nei picross più semplici la funzione obiettivo assume un valore inferiore, seppur di poco, in media. Rispetto al grafico precedente, si può notare come sia aumentato sia il valore minimo della funzione obiettivo, che il valore massimo, che tra esempi di livello 2 è duplicato, mentre, tra esempi di livello 1 è passato ad un valore 10 volte superiore. Ciò mostra come, seppur gli esempi più semplici siano ancora distinguibili in questo caso, rispetto a ciò che viene rappresentato in figura 2 la differenza in termini di prestazioni dell'algoritmo cresce enormemente, segno del fatto che l'aumentare della griglia è un fattore che incide maggiormente sulla difficoltà dell'algoritmo a convergere, e quindi sulla probabilità di ottenere la soluzione desiderata.

4 Conclusione

Nel seguente lavoro è stato sviluppato un software in grado di risolvere, attraverso un metodo metaeuristico, un gioco di logica, noto come *Picross*. A tal fine, sono state definite diverse strutture e classi necessarie per rappresentare la griglia di gioco e i parametri ad essa correlati. Fatto ciò, è stato implementato l'algoritmo scelto per la risoluzione del problema in questione, ovvero Tabu Search, il quale ha richiesto la definizione di alcune funzioni, come, ad esempio, la generazione di soluzioni vicine, il cui criterio di creazione è stato definito basandosi sul funzionamento del gioco in esame e i vincoli richiesti. Allo stesso modo, è stato stabilito un criterio di arresto ed un meccanismo che permettesse all'algoritmo di non entrare in loop.

Una volta ultimato il processo di creazione dell'algoritmo proposto, si è passati alla fase di test del modello, in modo tale da valutarne le prestazioni, sia basandosi sulla dimensione della griglia, che della difficoltà dell'esempio in questione. Dai grafici mostrati è emerso che, per esempi aventi una dimensione della griglia pari a 10x10 la funzione obiettivo ottiene valori piuttosto bassi, che raggiungono anche lo 0, per quanto riguarda i picross più semplici, segno del fatto che è più probabile ottenere la soluzione desiderata. Al contrario, gli esempi più complessi, nella maggior parte dei casi ottengono un valore superiore a 2, e solo nel 10% dei casi un valore che si avvicina allo 0. Aumentando la griglia di gioco ad una dimensione pari a 15x15, i valori della funzione obiettivo sono incrementati notevolmente. Ciò dimostra come la dimensione della griglia aumenti la difficoltà del problema in esame ed abbassi di conseguenza la probabilità di convergenza dell'algoritmo implementato.