Report di progetto - SudokuWhiz

DOROTEA SERRELLI and RAFFAELLA SABATINO

In questo report si descrive come si è pensato di costruire un agente intelligente capace di risolvere il gioco del Sudoku in modo corretto ed efficiente nel tempo, utilizzando i seguenti algoritmi di ricerca: backtracking, ricerca A*, simulated annealing, algoritmi genetici generazionali.

Per poter osservare il comportamento dell'agente nella risoluzione del gioco, si è realizzato un progetto Java che, mediante interfaccia grafica, legge il file .txt contenente la griglia Sudoku inserita dall'utente e fornisce la griglia Sudoku risultante, risolvendola con l'algoritmo di ricerca scelto dall'utente.

Si riporta, dunque, il link al repository Github per la demo: https://github.com/SabatinoRaffaella/SudokuWhiz.

Gli algoritmi di ricerca precedentemente citati sono stati descritti nel loro meccanismo di funzionamento, nel loro adattamento al problema, nella loro complessità, confrontandoli tra loro in termini di prestazioni: i nodi visitati durante la ricerca, i nodi visitati utili per risolvere la griglia Sudoku, il tempo di risoluzione.

Si precisa che l'applicazione Java è stata eseguita su un terminale avente le seguenti caratteristiche:

- Processore: 3.10GHz
- RAM: 16GB

Additional Key Words and Phrases: Sudoku, applicazione Java, Backtracking, Ricerca A*, Simulated annealing, Algoritmi genetici generazionali

ACM Reference Format:

Dorotea Serrelli and Raffaella Sabatino. 2024. Report di progetto - SudokuWhiz. 1, 1 (January 2024), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' address: Dorotea Serrelli; Raffaella Sabatino.

ACM XXXX-XXXX/2024/1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Indice

Comprensione del problema e dell'ambiente

Comprensione del problema e dell'ambiente

1 INTRODUZIONE

1.1 Il gioco Sudoku

Il Sudoku è un gioco di logica nel quale viene proposta una griglia di 9×9 celle, ciascuna delle quali può contenere un numero da 1 a 9, oppure essere vuota; la griglia è suddivisa in 9 righe orizzontali, 9 colonne verticali e in 9 "sottogriglie" di 3×3 celle contigue. Queste sottogriglie sono delimitate da bordi in neretto chiamate *regioni*. Le griglie proposte al giocatore, in generale, hanno da 20 a 35 celle contenenti un numero.

Il gioco fu inventato dal matematico svizzero Eulero da Basilea (1707-1783). La versione moderna del gioco fu pubblicata per la prima volta nel 1979 dall'architetto statunitense Howard Garns all'interno del *Dell Magazines* con il titolo *Number Place*. In seguito fu diffuso in Giappone dalla casa editrice *Nikoli* nel 1984, per poi diventare noto a livello internazionale soltanto a partire dal 2005, quando fu proposto in molti periodici.

Lo scopo del gioco è quello di riempire le caselle bianche con numeri da 1 a 9 in modo tale che in ogni riga, in ogni colonna e in ogni regione siano presenti tutte le cifre da 1 a 9 senza ripetizioni.

1.2 Obiettivi del progetto

In questo progetto si intende costruire un agente intelligente in grado di risolvere il gioco in modo corretto e nel minor tempo possibile.

La soluzione del gioco, ovvero una griglia Sudoku completamente riempita e che rispetta le regole del gioco, verrà determinata applicando i seguenti algoritmi di ricerca: backtracking, ricerca A*, simulated annealing, algoritmi genetici generazionali.

2 DESCRIZIONE DELL'AMBIENTE

2.1 Osservazioni preliminari per la formulazione della specifica PEAS dell'ambiente

Il gioco si svolge in matrici di aspetto 9×9 (le griglie), denotate *matrici Sudoku*, le cui caselle possono contenere un intero da 1 a 9 oppure il valore 0, per denotare la casella bianca o vuota.

Una matrice Sudoku M è suddivisa in 9 blocchi di aspetto 3×3, denotati $B_{\mathrm{h,k}}$ con $h, k = 1, 2, 3$; il blocco $B_{\mathrm{h,k}}$ riguarda, per la matrice M, le righe relative agli indici $3h - 2$, $3h - 1$ e $3h$ e le colonne relative agli indici $3k - 2$, $3k - 1$ e $3k$. In ogni riga, colonna e regione di una matrice Sudoku i valori interi da 1 a 9 non possono essere ripetuti.

La griglia Sudoku proposta o *matrice Sudoku incompleta* è una matrice Sudoku che presenta alcune celle bianche.

Lo scopo del gioco è trasformare la griglia proposta in una matrice completa, cioè in una matrice priva di celle bianche e, quindi, tale che in ogni sua riga, colonna e regione compaiano tutti i numeri dell'insieme $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, ciascuno una sola volta.

Affinché una matrice incompleta sia considerata valida ai fini del gioco, è necessario che la soluzione sia univoca, ovvero non devono sussistere due o più soluzioni differenti. Risulta cruciale, dunque, definire il numero minimo di indizi presenti nella matrice Sudoku, ovvero il numero minimo di valori presenti all'inizio nella griglia Sudoku.

Il numero massimo di indizi di partenza non conta; tuttavia, se ce ne sono troppi il gioco diventa banale e si riduce ad una procedura meccanica di riempimento senza che occorra alcun procedimento logico.

Pertanto, si è deciso di considerare griglie Sudoku che avessero al massimo 30 indizi.

La quantità minima di numeri di partenza è, invece, importante perché, sotto una certa soglia, il rompicapo diventa impossibile da risolvere; infatti, si rischia che la matrice Sudoku ammetta più di una soluzione e ciò la renderebbe non valida (ed esteticamente poco interessante).

Per definire questo valore, ci si basa su uno studio della rivista *Nature* del 2013, condotto da un gruppo di matematici diretto da Gary McGuire dell'Università di Dublino. Tale gruppo di ricerca ha dimostrato che il numero minimo di indizi necessari per risolvere una griglia Sudoku 9×9 è 17; infatti, con meno indizi è impossibile riempire univocamente la griglia del gioco su cui giornalmente milioni di persone si cimentano.

Di seguito si riporta il link al nostro repository di Github per visionare l'abstract Solving the Sudoku Minimum: https://github.com/SabatinoRaffaella/SudokuWhiz/blob/main/Documents/Solving%20the%20Sudoku%20Minimum_abstract.pdf.

Pertanto, per garantire che il Sudoku abbia una sola soluzione, si considereranno griglie Sudoku aventi almeno 17 indizi.

2.2 Specifica PEAS dell'ambiente

Tenendo conto delle osservazioni preliminari fatte nel paragrafo precedente, di seguito si delinea la specifica PEAS dell'ambiente in cui opererà l'agente:

- **Performance**

  La misura delle prestazioni è la capacità dell'agente di risolvere una griglia Sudoku in modo corretto ed efficiente nel tempo, trovando una sola soluzione.

- **Environment**

  L'ambiente è rappresentato dalla griglia Sudoku stessa, che è costituita da una griglia 9×9 parzialmente riempita, divisa in 9 regioni 3×3. L'ambiente include anche le celle vuote — denotate con il valore 0 – da riempire con numeri da 1 a 9, rispettando le regole del gioco. Nel paragrafo successivo si descriveranno le caratteristiche dell'ambiente.

- **Actuators**

  Si intendono gli attuatori dell'agente disponibili per compiere delle azioni, al fine di alterare l'ambiente: inserimento di numeri nelle celle vuote della griglia; cancellazione di un numero inserito dall'agente; spostamento in una nuova cella; aggiornamento della rappresentazione interna della griglia, a seguito di una modifica alla configurazione della griglia.

- **Sensors**

  I sensori dell'agente sono utilizzati per percepire lo stato corrente della griglia Sudoku, ossia per rilevare i numeri già presenti nella griglia e quelli che l'agente può inserire in modo legale nelle celle vuote.

2.2.1 Caratteristiche dell'ambiente. L'agente si interfaccia in un ambiente con le seguenti caratteristiche:

- **completamente osservabile**: l'agente è a conoscenza in ogni istante della configurazione della griglia del Sudoku;
- **singolo agente**: l'unico agente che opera in questo ambiente è quello in oggetto;
- **deterministico**: la configurazione della griglia Sudoku M_j è il risultato solo e soltanto dell'azione dell'agente eseguita sulla configurazione M_i della griglia Sudoku corrente;
- **sequenziale**: la scelta dell'azione dell'agente sulla configurazione della griglia Sudoku corrente dipende dalle azioni fatte nelle configurazioni precedenti (inserimento/cancellazione di un numero, ...);
- **statico**: durante l'esecuzione dell'agente, la griglia Sudoku non muta mentre l'agente pensa alla prossima azione;
- **discreto**: in ogni configurazione della griglia Sudoku c'è un insieme finito di percezioni ed azioni; infatti, la griglia è formata da un numero di celle finito, ciascuna con un numero discreto (da 0 a 9).

3 FORMULAZIONE DEL PROBLEMA

Di seguito si riporta l'analisi del problema da risolvere:

- **Stato iniziale**

  Lo stato iniziale corrisponde alla configurazione iniziale della griglia 9×9, parzialmente riempita con almeno 17 indizi e massimo 30 indizi, è definita in un file di estensione .txt.
  Ogni riga del file è una riga della griglia 9×9 ed i numeri presenti in una riga sono separati l'uno dall'altro mediante uno spazio. Le caselle vuote sono indicate con il valore 0.

- **Azioni**

  Si elencano di seguito le azioni possibili per l'agente.
  - Inserimento di numeri da 1 a 9 nelle celle vuote, rispettando le regole del gioco.
  - Lettura di un numero in una cella;
  - Cancellazione di un numero inserito in una cella (ad eccezione delle celle definite nel file, contenenti gli indizi);
  - Spostamento verso una nuova cella.

- **Modello di transizione**

Il modello di transizione descrive come lo stato della matrice Sudoku cambia in risposta alle azioni eseguite dall'agente.
Se l'agente inserisce un numero legale x in una cella vuota y (con $1 \leq x \leq 9$), la matrice Sudoku corrente transita ad un nuovo stato, ossia la matrice Sudoku risultante avrà la cella y contenente il valore x. Il valore x scelto non sarà un numero già presente nella riga, nella colonna e nella regione in cui si trova la cella y.
Se l'agente rimuove un numero x che ha precedentemente inserito in una cella y, la matrice Sudoku corrente transita ad un nuovo stato: la matrice Sudoku risultante avrà la cella y priva del valore x, ed è denotata con il valore 0.
Se l'agente decide di leggere il valore di una cella, la configurazione della matrice Sudoku non transita in nuovo stato.
Se l'agente decide di spostarsi in una nuova cella, la configurazione della matrice Sudoku non transita in nuovo stato, bensì cambia la cella che l'agente deve considerare.

- **Test-obiettivo**

  Il test-obiettivo consiste nel verificare se il Sudoku è stato risolto correttamente, ovvero se tutte le regole del gioco sono state rispettate.

- **Costo di cammino**

  Il costo di cammino è definito come il costo dell'esecuzione di una o più azioni necessarie per risolvere il Sudoku. Si suppone che la lettura di un valore della cella e lo spostamento da una cella all'altra hanno un costo pari a 1, mentre le altre operazioni hanno un costo pari a 2.

Algoritmi di ricerca

Algoritmi di ricerca

4 ALGORITMI DI RICERCA ANALIZZATI

Il problema della risoluzione di una griglia Sudoku parzialmente riempita è stato affrontato guardandolo da diverse prospettive.

Innanzitutto, si è guardato al primo possibile approccio per risolvere il gioco, applicato da un giocatore naïve: il backtracking. Successivamente, si è pensato ad una possibile strategia che può utilizzare un giocatore che pratica regolarmente il gioco, però senza aver raggiunto il livello di esperto: la ricerca informata A*.

I due succitati algoritmi di ricerca tradizionale si basano sulla formulazione del problema di tipo incrementale: essi adottano una strategia che mira a riempire, progressivamente, la griglia incompleta 9×9 fornita dall'utente.

Tale strategia prevede di visitare un albero di ricerca, rappresentazione dello Spazio degli stati del problema, al fine di ottenere il cammino che porta verso la soluzione.

D'altra parte, si è pensato di utilizzare per il problema in esame anche algoritmi di ricerca locale, al fine di fornire solamente la configurazione finale della griglia data dall'utente, piuttosto che esplorare in modo sistematico lo spazio di ricerca.

Pertanto, nel documento si sono analizzati per la risoluzione di una griglia Sudoku l'algoritmo Simulated Annealing e gli Algoritmi Genetici Generazionali.

È da precisare che, per motivi di efficienza e di efficacia, si è voluto fornire un'interpretazione del problema a stato completo per quest'ultimi algoritmi: a partire dalla griglia Sudoku 9×9 fornita dall'utente, le celle bianche vengono riempite con valori casuali, presi dall'insieme $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Dinanzi a questa configurazione della griglia, l'agente dovrà adottare una strategia per trasformare la griglia Sudoku casualmente riempita in una soluzione valida per le regole del gioco.

5 BACKTRACKING

5.1 Meccanismo di funzionamento

Il backtracking è un algoritmo utile per risolvere problemi con la ricorsione, costruendo una soluzione incrementalmente.

In generale, il backtracking intende espandere una soluzione parziale $s = (a_1, a_2, a_3, \ldots, a_k)$, dove l'elemento a_i viene scelto da un insieme ordinato S_i di possibili candidati per la posizione i.

L'espansione della soluzione s consiste nel costruire, a partire da s, l'insieme S_{k+1} dei possibili candidati per la posizione $k + 1$ e scegliere un elemento $a_{k+1} \in S_{k+1}$. Fin quando l'estensione genera una soluzione parziale, si continua ad estenderla.

Se l'insieme dei possibili candidati S_{k+1} per la posizione $k + 1$ della soluzione parziale s è vuoto, significa che non c'è la possibilità di estendere la soluzione corrente. Dunque, è necessario ritornare indietro e sostituire l'ultimo elemento a_k nella soluzione s con un altro candidato dell'insieme S_k, con $a_{k'} \neq a_k, a_{k'} \in S_k$.

5.2 Il backtracking in SudokuWhiz

Per il nostro gioco, la soluzione parziale che si intende espandere è la griglia 9×9 del Sudoku, che si suppone riempita fino alla cella posta nella riga i e colonna j ($0 \leq i < 8, 0 \leq j < 8$), denotata con $c_{i,\,j}$.

L'agente si trova nella cella $c_{i,\,j+1}$ e sta valutando l'insieme dei possibili candidati per questa cella, ossia i numeri da 1 a 9.

Si intende, quindi, inserire in modo iterativo i numeri da 1 a 9 uno per uno: se il numero inserito soddisfa tutte e 3 le condizioni, cioè quel particolare numero non è presente nella riga, colonna e regione della cella $c_{i,\,j+1}$, si conferma tale inserimento.

Poi, si passa alla cella vuota successiva e si compie la stessa iterazione dei numeri ma con la matrice Sudoku aggiornata. Man mano che si procede con la nuova cella vuota, si potrebbe raggiungere una cella, denotata con $c_{h,\,k}$ ($i \leq h \leq 8, j + 1 < k \leq 8$) in cui non si può inserire nessun numero dell'insieme $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Quindi, la soluzione costruita fino adesso viene respinta e si torna indietro, provando nuovi valori possibili in quelle celle precedenti. Poiché si assume, per configurazione iniziale della matrice Sudoku, che una matrice Sudoku ha solo una soluzione, si continuerà a respingere una o più soluzioni costruite fino a quando non si trova quella che soddisfa tutte le condizioni per ogni cella inizialmente vuota.

Per questo problema, l'algoritmo di backtracking cercherà di posizionare ogni numero in ogni riga e colonna vuota fino a quando non è risolto.

5.3 Complessità dell'algoritmo backtracking

Nell'algoritmo di backtracking viene costruito un albero delle soluzioni per il problema, dove ciascun nodo interno x è una soluzione parziale e l'arco tra il nodo x e il nodo y nell'albero viene creato se il vertice y è frutto dell'estensione della soluzione parziale x. Le foglie dell'albero sono le soluzioni.

Mediante la verifica del soddisfacimento delle condizioni di unicità per riga, colonna e regione, si effettua una potatura delle soluzioni che non soddisfano le condizioni richieste.

Il backtracking corrisponde ad effettuare una visita in profondità dell'albero delle soluzioni poiché l'intento è quello di trasformare la griglia Sudoku all'inizio incompleta (rappresentata dal nodo radice dell'albero) in una griglia completa (rappresentata da una foglia dell'albero).

L'algoritmo risulta essere completo poiché in modo esaustivo esplora tutte le possibili combinazioni e, per via delle assunzioni fatte sulla configurazione iniziale della matrice Sudoku, troverà una soluzione.

L'algoritmo, inoltre, risulta essere ottimo perché riesce a determinare la soluzione ottima, ma non in termini di efficienza; infatti, la complessità temporale richiesta è esponenziale $O(9^{n \times n})$ poiché, per ogni cella, si hanno a disposizione 9 valori su cui scegliere – anche se si debbano fare delle scelte nel caso peggiore per $81 - 17 = 64$ caselle affinché il Sudoku abbia una sola soluzione – .

La complessità spaziale richiesta è pari a $O(n^2)$ per tenere traccia di una matrice Sudoku di dimensione $n \times n$; infatti, il backtracking mantiene soltanto una singola rappresentazione di uno stato e altera tale rappresentazione anziché crearne di nuove.

Nella seguente tabella si riportano i tentativi fatti per diverse tipologie di Sudoku: nel caso peggiore (17 indizi iniziali), medio (24 indizi iniziali), facile (30 indizi iniziali). Le griglie Sudoku realizzate per fare il test si trovano nel package Griglie_test del progetto Java chiamato SudokuWhiz.

Livello di difficoltà	MNE	MNS	Tempo medio per trovare una soluzione
Caso peggiore	210.969.538	197	3, 6717
Medio	1.775.908	174	0, 0961
Facile	70.154	125	0, 0201

Tabella 1. Prestazioni del backtraing su diversi livelli di difficoltà del gioco

È da sottolineare che, sottoponendo l'agente alla risoluzione delle diverse griglie Sudoku 9×9, si è notato che anche la diversa configurazione degli indizi nella griglia ha un forte impatto sul tempo di ricerca della soluzione e sul numero di nodi da esplorare.

Come esempio di quanto appena affermato, si riportano i dati della tabella relativi alle griglie Sudoku_casoPeggiore_g e Sudoku_casoPeggiore2_g.

Griglia	MNE	MNS	Tempo medio per trovare una soluzione
Sudoku_casoPeggiore	37.652	103	0, 0226
Sudoku_casoPeggiore2	622.577.597	259	7, 3376941

Tabella 2. Prestazioni del backtraing sulle griglie Sudoku_casoPeggiore e Sudoku_casoPeggiore2

```java
/// ALGORITMO DI BACKTRACKING
/**
 * Risolve la griglia Sudoku passata come parametro utilizzando il Backtracking.
 *
 * @param sudo_m: matrice del sudoku da risolvere.
 * @return sudo_m (soluzione del sudoku) o null nel caso in cui il sudoku non
 *         sia risolvibile.
 *
 */
public int[][] solve_sudoku_backtrack(int sudo_m[][], SolutionStatistics st) {
    if (solveSudoku_basic(sudo_m) == true) {
        st.recordSolutionStatistics(countNodes, exploredNodes.size(), countNodes);
        System.out.println("Numero nodi esplorati: " + countNodes);
        System.out.println("Numero nodi utili per la soluzione: " + exploredNodes.size());
        return sudo_m;
    } else
        return null;
}

private boolean solveSudoku_basic(int sudo_m[][]) {
    int row = 0;
    int col = 0;
    boolean checkBlankSpaces = false;
    /*
     * controllo se il Sudoku è risolto e, in caso negativo,
     * prendo la posizione della prossima cella "vuota"
     */
    for (row = 0; row < sudo_m.length; row++) {
        for (col = 0; col < sudo_m[row].length; col++) {
            if (sudo_m[row][col] == 0) {
                checkBlankSpaces = true;
                break;
            }
        }
        if (checkBlankSpaces == true) {
            break;
        }
    }
    // se non ci sono più celle "vuote" significa che il Sudoku è stato risolto.
    if (checkBlankSpaces == false) {
        return true;
    }
    // cerco di riempire le celle "vuote" con il numero corretto
    for (int num = 1; num <= 9; num++) {
        /*
         * isSafe controlla che num non sia già presente
         * nella riga, colonna, o sotto-griglia 3x3
         * (sotto le funzioni che si occupano di fare questi controlli)
         */
        countNodes++;

        if (isSafe(sudo_m, row, col, num)) {
            sudo_m[row][col] = num;
            String nodeKey = row + "-" + col + "-" + num;
            exploredNodes.add(nodeKey);
            if (solveSudoku_basic(sudo_m)) {
                // table.getModel().setValueAt(num, row, col);
                return true;
            }
            /*
             * se num è stato piazzato in una posizione scorretta,
             * marco nuovamente la cella come "vuota", poi faccio il backtrack su
             * un num differente su cui provare gli altri numeri che non sono stati provati
             */
            sudo_m[row][col] = 0;
        }
    }
    return false;
}
```

Fig. 1. Implemetazione in Java dell'algoritmo backtracking

6 RICERCA A*

6.1 Meccanismo di funzionamento

La ricerca A* è un algoritmo di ricerca informata che sfrutta la conoscenza specifica del dominio del problema per fornire suggerimenti su dove si potrebbe trovare l'obiettivo. I suggerimenti hanno la forma di una funzione euristica, denotata con $h(n)$, intesa come stima del costo del cammino meno costoso dallo stato corrente (associato al nodo n dell'albero di ricerca) ad uno stato obiettivo.

L'algoritmo di ricerca A* procede espandendo il nodo avente il valore minimo della funzione di valutazione $f(n)$:

$f(n)$ = costo stimato del cammino migliore che continua da n fino ad un obiettivo = $g(n) + h(n)$

dove $g(n)$ è il costo del cammino per raggiungere il nodo n a partire dalla radice, e $h(n)$ è il costo stimato dall'algoritmo per raggiungere lo stato obiettivo a partire dal nodo n.

6.2 La ricerca A* in SudokuWhiz

Nella formulazione del nostro problema, lo stato obiettivo è rappresentato dalla matrice completa Sudoku, ovvero la griglia 9×9 avente numero di caselle bianche pari a 0 e che rispetta le regole del gioco.

La funzione $g(n)$, in questo caso, determina il costo di cammino dalla configurazione iniziale della griglia Sudoku alla configurazione corrente M_j (rappresentata dal nodo n) sommando il costo delle operazioni effettuate per raggiungere il nodo n, come riportato nella formulazione del problema.

La funzione euristica $h(n)$, banalmente, è interpretabile come il numero di caselle bianche che devono essere ancora riempite nella griglia corrente M_j; infatti, la soluzione al Sudoku è una griglia 9×9 priva di celle bianche.

A differenza dell'algoritmo di backtracking, si è pensato di rendere più edotto l'agente sulla risoluzione del gioco.

In generale, se in una riga o in una colonna ci sono pochissime celle bianche, vuol dire che l'insieme dei valori assegnabili in una di queste caselle bianche è ristretto rispetto all'insieme $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Nello specifico, se l'agente inserisse correttamente un numero in una casella bianca, diminuirebbe il numero di valori da considerare per le celle bianche rimanenti della stessa riga e della stessa colonna. In questo modo, l'agente sarebbe stimolato a completare la riga/colonna in questione.

Questa intuizione può essere formulata come una funzione euristica $h_2(n)$, ausiliare ad $h(n)$ che valuta quale cella della riga o colonna corrente possiede il minor numero di alternative assegnabili, in modo da favorire l'inserimento di un valore valido. Una volta effettuato l'inserimento, il valore assegnato non verrà considerato nell'insieme delle possibili scelte per le celle che si trovano sulla stessa riga, colonna e regione.

Ricapitolando, ad ogni stadio l'algoritmo A* cerca di prendere la decisione (eseguire il prossimo inserimento di un numero) che minimizza la distanza rimanente dall'obiettivo (numero di celle vuote), garantendo al contempo che le regole del Sudoku siano soddisfatte.

Una coda prioritaria viene utilizzata per gestire i passaggi successivi possibili ed un insieme assicura che uno stato della griglia non venga visitato due volte.

Ogni volta che viene calcolata una decisione su dove effettuare il prossimo inserimento di un valore sulla griglia, viene scelta la cella, appartenente alla riga o alla colonna dell'ultima cella riempita, che ha il minor numero di valori assegnabili, al fine di massimizzare la probabilità che la decisione porti ad un posizionamento corretto.

Le principali strutture dati utilizzate sono:

- Matrice $n \times n$: utilizzata per memorizzare la griglia Sudoku (matrice 9×9).
- Min-coda a priorità per gli stati successori: utilizzata per mantenere gli stati successori ed ordinarli in base al valore della funzione di valutazione $f(n) = g(n)+h(n)+h_2(n)$. La dimensione di questa struttura dati sarà influenzata dal numero massimo di stati successori che possono essere generati in una singola espansione.
- Insieme per gli stati visitati: tiene traccia degli stati visitati durante la ricerca.

6.3 Complessità della ricerca A*

L'algoritmo di ricerca A* risulta essere ottimo in quanto le due funzioni euristiche $h(n)$ e $h_2(n)$ sono entrambe ammissibili e consistenti.

L'ammissibilità deriva dal fatto che le euristiche non sbagliano mai per eccesso la stima del costo per arrivare all'obiettivo e nel numero di valori ammissibili per ogni cella in un dato istante.

La consistenza esiste perché, nel caso di $h(n)$, per ogni nodo n e ogni successore n', il costo stimato per raggiungere l'obiettivo partendo da n non è superiore al costo di passo per arrivare a n' sommato al costo stimato per andare da lì all'obiettivo.

In modo analogo, la funzione euristica $h_2(n)$, la quale valuta il numero di scelte rimanenti per riempire le celle vuote in una data riga o colonna, è anch'essa consistente. Poiché la distanza reale d (costo del cammino) tra due stati successivi nel Sudoku è almeno 1, poiché riempire una cella richiede almeno un passo, la funzione euristica $h_2(n)$ è consistente.

Nel dettaglio, se si considerano due stati A e B, dove B è ottenuto da A riempiendo una cella, $h_2(A)$ rappresenta il numero di opzioni rimanenti nella riga o colonna dell'ultima cella riempita in A, mentre $d(A, B)$ sarà almeno 1, poiché è necessario almeno un passo per raggiungere B da A. Pertanto, $h_2(A) \leq d(A, B)$ come richiesto per la consistenza.

La ricerca A* risulta, poi, essere completa: esiste un numero finito di nodi di costo minore o uguale a C* (il costo di cammino della soluzione ottima).

La complessità temporale della ricerca A* è $O(b^d)$ dove b è il fattore di ramificazione e d è la profondità della soluzione ottima C*. Nel caso del Sudoku, b dipenderà dal numero di scelte possibili in ogni cella vuota, mentre d sarà la profondità della soluzione ottima. Lo svantaggio di questo algoritmo sta nel fatto che il numero di nodi all'interno dello spazio di ricerca del confine dell'obiettivo cresce esponenzialmente con la lunghezza della soluzione.

Nella seguente tabella si riportano i tentativi fatti per diverse tipologie di Sudoku: nel caso peggiore (17 indizi iniziali), medio (24 indizi iniziali), facile (30 indizi iniziali). Le griglie Sudoku realizzate per fare il test si trovano nel package Griglie_test del progetto Java SudokuWhizWhiz.

Livello di difficoltà	MNG	MNS	Tempo medio per trovare una soluzione
Caso peggiore	385.064	192.532	47, 61
Medio	6.075	3.037	0, 023
Facile	264	132	0, 004

Tabella 3. Prestazioni della ricerca A* su diversi livelli di difficoltà del gioco

Così come è stato notato per l'algoritmo backtracking, si è visto che sottoponendo l'agente alla risoluzione delle diverse griglie Sudoku 9×9, la diversa configurazione degli indizi iniziali nella griglia ha un forte impatto sul tempo di ricerca della soluzione e sul numero di nodi da esplorare.

Come esempio di quanto appena affermato si riportano i dati della tabella relativi sulle griglie Sudoku_casoPeggiore_g e Sudoku_casoPeggiore2_g.

Griglia	MNG	MNS	Tempo medio per trovare una soluzione
Sudoku_casoPeggiore	187	93	0, 0363
Sudoku_casoPeggiore2	421.085	210.535	0, 8613

Tabella 4. Prestazioni della ricerca A* sulle due griglie Sudoku_casoPeggiore e Sudoku_casoPeggiore2 con livello di difficoltà *Caso peggiore*

```java
/// ALGORITMO DI RICERCA A*

public int[][] solveSudoku_AsteriskA(int sudo_m[][], SolutionStatistics st) {
    int exploredNodes = 0; // contatore per tenere traccia dei nodi visitati durante la ricerca
    int totalGeneratedNodes = 0; // numero dei nodi generati
    Set<String> visitedStates = new HashSet<>();

    PriorityQueue<BoardState> queue = new PriorityQueue<>(Comparator.comparingDouble(BoardState::getTotalCost));
    queue.add(new BoardState(sudo_m, costi0, BoardState.heuristic1(sudo_m)));
    visitedStates.add(getGridHash(sudo_m));
    boolean isRoot = true;

    while (!queue.isEmpty()) {
        BoardState currentNode;
        if (isRoot) {
            currentNode = queue.poll();
            isRoot = false;
        } else {
            currentNode = queue.poll();
        }

        if (currentNode.isGoal()) {
            BoardState.copyValues(currentNode.getGrid(), sudo_m);
            st.recordSolutionStatistics(exploredNodes, exploredNodes, totalGeneratedNodes);
            System.out.println("Numero nodi esplorati: " + exploredNodes);
            System.out.println("Numero totale dei nodi generati: " + totalGeneratedNodes);
            return currentNode.getGrid();
        }
        exploredNodes++;
        List<BoardState> successors = currentNode.generateSuccessors();
        totalGeneratedNodes++; // conteggio per ogni stato generato
        totalGeneratedNodes += successors.size();
        for (BoardState successor : successors) {
            String successorHash = getGridHash(successor.getGrid());
            if (!visitedStates.contains(successorHash)) {
                visitedStates.add(successorHash);
                boolean replace = false;
                for (BoardState nodeInQueue : queue) {
                    if (Arrays.deepEquals(successor.getGrid(), nodeInQueue.getGrid()) &&
                            successor.getTotalCost() < nodeInQueue.getTotalCost()) {
                        replace = true;
                        break;
                    }
                }

                if (replace) {
                    queue.remove(successor);
                    queue.add(successor);
                } else {
                    queue.add(successor);
                }
            }
        }
    }
    return sudo_m;
}
```

Fig. 2. Implemetazione in Java dell'algoritmo ricerca A*

7 MIGLIORAMENTO ITERATIVO DELLO STATO INIZIALE: SIMULATED ANNEALING E ALGORITMI GENETICI GENERAZIONALI

Gli algoritmi di ricerca precedentemente illustrati sono progettati per esplorare sistematicamente lo spazio di ricerca. Questa peculiarità è una conseguenza del mantenimento in memoria di uno o più cammini e della memorizzazione delle alternative esplorate in ogni punto del cammino.

Per molti problemi, compreso il gioco del Sudoku, è rilevante la configurazione finale piuttosto che il cammino verso l'obiettivo. Gli algoritmi di ricerca locale, infatti, cercano di migliorare iterativamente un singolo nodo corrente invece di lavorare su cammini multipli e, in generale, si spostano solo nei nodi immediatamente adiacenti.

In questa sezione si tratteranno i seguenti algoritmi: Simulated Annealing e Algoritmi Genetici Generazionali.

7.1 Formulazione del problema a stato completo

Il meccanismo di funzionamento del Simulated Annealing e degli Algoritmi Genetici Generazionali tiene conto della seguente formulazione del problema:

- **Stato iniziale**

La configurazione iniziale della griglia 9×9, parzialmente compilata con almeno 17 indizi e massimo 30, è definita in un file di estensione .txt.
Ogni riga del file è una riga della griglia 9×9 ed i numeri presenti in una riga sono separati l'uno dall'altro mediante uno spazio. Le caselle vuote sono indicate con il valore 0.
Lo stato iniziale della griglia corrisponde alla griglia fornita dall'utente, dove le celle bianche verranno casualmente riempite con valori da 1 a 9, in modo tale che ogni sotto-griglia non possieda dei duplicati.

- **Azioni**

  Le azioni possibili per l'agente sono le seguenti:
  - Inserimento di numeri da 1 a 9 nelle celle vuote, rispettando le regole del gioco.
  - Lettura di un numero in una cella.
  - Spostamento verso una nuova cella.
  - Scambiare di posizione (ossia di riga e colonna) due celle della griglia Sudoku tale che nessuna delle due celle è un indizio.

- **Modello di transizione**

  Il modello di transizione descrive come lo stato della matrice Sudoku cambia in risposta alle azioni dell'agente.
  Se l'agente inserisce un numero x in una cella vuota y (con 1 <= x <= 9), la matrice Sudoku corrente transita ad un nuovo stato, ossia la matrice Sudoku risultante avrà la cella y contenente il valore x. Il valore x scelto non sarà un numero già presente nella riga, nella colonna e nella regione in cui si trova la cella y.
  Se l'agente decide di leggere il valore di una cella, la configurazione della matrice Sudoku non transita in nuovo stato.
  Se l'agente decide di spostarsi in una nuova cella, la configurazione della matrice Sudoku non transita in nuovo stato, bensì cambia la posizione dell'agente.
  Se l'agente decide di scambiare due celle della griglia - non considerate indizi -, la configurazione della matrice Sudoku transita in nuovo stato: date le celle $a_{\mathrm{i,j}} = x$ (riga i, colonna j, valore $x \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$) e la cella $b_{\mathrm{h,k}} = y$ (riga h, colonna k, valore $y \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$), la nuova griglia Sudoku ottenuta dallo scambio delle due celle possiede $a_{\mathrm{i,j}} = y$ e $b_{\mathrm{h,k}} = x$.

- **Test-obiettivo**

  Il test-obiettivo consiste nel verificare se il Sudoku è stato risolto correttamente, ovvero se tutte le regole del gioco sono state rispettate.

- **Costo di cammino**

  Il costo di cammino è definito come il costo dell'esecuzione di una o più azioni necessarie per risolvere il Sudoku. Si suppone che le azioni abbiano lo stesso costo.

8 SIMULATED ANNEALING

8.1 Meccanismo di funzionamento

Simulated Annealing è un algoritmo di ricerca locale che memorizza solo lo stato iniziale, con l'intento di migliorarlo iterativamente, in accordo ad una funzione obiettivo, fino a raggiungere lo stato obiettivo del problema di ottimizzazione.

Il nome dell'algoritmo richiama il processo di annealing fisico; in metallurgia, l'annealing è un processo utilizzato per indurire i metalli o il vetro, riscaldandoli ad altissime temperature e raffreddandoli gradualmente, permettendo così al materiale di cristallizzare in uno stato a bassa energia.

In modo analogo, il simulated annealing, a partire dallo stato iniziale, esplora all'inizio un'ampia regione dello spazio degli stati. Successivamente, viene scelto casualmente uno stato vicino s' dello stato corrente s e, in base alla variazione dei valori della funzione obiettivo $f(s) - f(s')$ - definita *variazione di energia* ΔE - risulta:

- $\Delta E > 0$: il vicino s' verrà sicuramente accettato come mossa migliore di quella corrente;
- $\Delta E \leq 0$: il vicino s' verrà accettato ugualmente come prossima mossa ma con una probabilità $e^{\frac{-\Delta E}{T}}$, inferiore a 1.

Il valore T, detto *temperatura*, influisce sulla probabilità di accettare delle mosse casuali qualitativamente cattive per il valore ΔE: le mosse "cattive" saranno accettate più facilmente all'inizio , in condizioni di T alta, e diventeranno sempre meno probabili a mano a mano che T si abbassa.

Ciò sta ad indicare che il parametro T serve meramente come *parametro di controllo* che, implicitamente, definisce la regione della spazio degli stati esplorati dall'algoritmo in un particolare stadio.

Ad alte temperature, il suddetto algoritmo può attraversare quasi tutto lo spazio degli stati del problema e, dunque, pessime soluzioni vengono facilmente accettate. Successivamente, abbassandosi il valore del parametro di controllo - fenomeno definito come *raffreddamento della temperatura* -, l'algoritmo viene confinato in regioni sempre più ristrette dello spazio degli stati. Nel momento in cui la temperatura scende ad un valore minimo (es. $T = 0$), lo stato corrente verrà restituito come soluzione del problema.

Se il fattore di raffreddamento fa decrescere abbastanza lentamente la temperatura da T a 0, allora, per una proprietà della distribuzione di Boltzmann, tutta la probabilità è concentrata sugli ottimi globali, che l'algoritmo troverà con probabilità tendente a 1.

L'algoritmo simulated annealing può essere visto come un'estensione della tecnica di ottimizzazione locale, così come descritta nell'algoritmo Hill-Climbing: la soluzione iniziale viene ripetutamente migliorata tramite piccole perturbazioni locali fino a quando nessuna di tali perturbazioni migliora la soluzione.

Il simulated annealing randomizza tale procedura in modo da permettere, occasionalmente, dei movimenti in salita, cioè delle perturbazioni che peggiorano la soluzione, cercando di ridurre la probabilità di bloccarsi in una soluzione localmente ottima ma globalmente scadente.

8.2 L'algoritmo Simulated Annealing in SudokuWhiz

In base al meccanismo di funzionamento dell'algoritmo appena descritto, si intende determinare una soluzione ad una griglia Sudoku. Innanzitutto, è necessario definire alcuni parametri:

- **s**tato iniziale

  La griglia Sudoku fornita dall'utente verrà riempita con valori casuali da 1 a 9 in modo tale che ogni sotto-griglia 3×3 sia priva di duplicati.

- **f**unzione obiettivo

  In base alla configurazione della griglia Sudoku, la funzione obiettivo determina il numero totale di duplicati presenti nelle righe e nelle colonne della griglia. Si intende per soluzione globalmente ottima una griglia Sudoku con zero duplicati.

- **t**emperatura (*temperature*)

  In fase di inizializzazione della variabile temperatura, si seleziona una temperatura che consentirà di accettare all'inizio anche mosse non buone qualitativamente rispetto alla mossa corrente. Ciò conferisce all'algoritmo la capacità di esplorare in modo più completo l'intero spazio di ricerca prima di raffreddarsi e stabilizzarsi in una regione più focalizzata. D'altra parte il valore della temperatura non dovrà inficiare il tempo di esecuzione dell'algoritmo o portare a convergere verso un ottimo locale.

- **f**attore di raffreddamento (*cooling factor*)

  Ad ogni iterazione si intende diminuire la temperatura in modo tale che non si è esposti ad una convergenza verso una soluzione sub-ottima né ad uno spreco di risorse di computazione. Dunque, dopo l'iterazione i la temperatura finale T_{i+1} sarà determinata nel modo seguente:

  $$T_{i+1} = \alpha \cdot T_i$$

  dove α è il fattore di raffreddamento, definito come $0 \leq \alpha \leq 1$: per valori di α tendenti a 1 (es. 0.999), la temperatura diminuirà molto lentamente, mentre un valore tipo 0.5 causerà un raffreddamento molto più rapido. Lo schema di raffreddamento appena descritto è detto *geometrico*.

- meccanismo di perturbazione

  Dato uno stato corrente, si determina il suo vicino nel seguente modo:
  (1) casualmente si scelgono due indici i e j, con $1 \leq i, j \leq 9$ in modo tale che la cella $c_{i,\,j}$ non sia un indizio;
  (2) casualmente si scelgono due indici h e k, con $1 \leq h, k \leq 9$ in modo tale che: la cella $c_{h,\,k}$ non sia un indizio,$c_{h,\,k} \neq c_{i,\,j}$ e $c_{h,\,k}$ si trovi nella stessa sotto-griglia di $c_{i,\,j}$;
  (3) scambiare le celle $c_{i,\,j}$ e $c_{h,\,k}$ tra loro.
  Questo meccanismo di perturbazione garantisce che, in ogni iterazione, non ci sono duplicati in ciascuna griglia.

Tenendo conto dei parametri appena definiti, si procederà come segue:

(1) inizializzazione: si riempiono le celle bianche della griglia fornita in maniera casuale, con il vincolo che ogni sotto-griglia non abbia duplicati; si impostano il valore della temperatura e del cooling factor;

(2) generazione dell'individuo: all'interno di un ciclo for, fino a quando la temperatura non scende ad un valore minimo (si è adottato il valore 1), si determina una mossa casuale (secondo lo schema di perturbazione sopra descritto) e si verifica se accettare o meno il vicino generato;

(3) ad ogni mossa generata, si raffredda la temperatura secondo la regola definita nel fattore di raffreddamento.

(4) a temperatura pari a 1 si restituisce lo stato corrente.

8.3 Complessità del Simulated Annealing

La complessità del Simulated Annealing non è facilmente esprimibile in forma di notazione O, come si è visto negli algoritmi di ricerca precedenti e, in generale, negli algoritmi deterministici.

La ragione principale è che il Simulated Annealing è un algoritmo probabilistico che sfrutta il concetto di temperatura per guidare l'esplorazione dello spazio delle soluzioni in modo stocastico.

È possibile, tuttavia, fare delle considerazioni sul tempo di esecuzione, sulla bontà dei risultati ottenuti e sul meccanismo di perturbazione usato per la generazione dei vicini.

I valori dei parametri $temperature$ e $coolingfactor$ sono stati definiti sulla base degli esperimenti condotti sulle griglie presenti nella cartella Griglie_test del progetto in linguaggio Java Sudoku-WhizWhiz. Inizialmente si è sviluppato in Java il suddetto algoritmo seguendo i passaggi definiti nei paragrafo precedenti, ponendo come valori dei parametri di controllo $temperature = 10.000, coolingfactor = 0.99995$. Si è visto che, eseguendo tale implementazione per la griglia sudoku $Sudoku\_facile1\_g$, dopo circa $33ms$ si raggiunge un ottimo locale avente in totale 24 duplicati nelle righe e nelle colonne, partendo da uno stato iniziale avente un valore per la funzione obiettivo pari a 34.

```java
// ALGORITMO DI RICERCA SIMULATED ANNEALING TRADIZIONALE

public int[][] solveSudoku_SimulatedAnnealing_Tradizionale(int sudo_m[][], SolutionStatistics stat) {
    List<Integer> original_entries = m.getOriginalEntriesList(sudo_m);
    BoardRandomizer br = new BoardRandomizer();
    br.generateRandomSudoku(sudo_m);
    m.printMatrix(sudo_m);

    SudokuPuzzle currentSudokuPuzzle = new SudokuPuzzle(sudo_m, original_entries);
    SudokuPuzzle bestSudokuPuzzle = new SudokuPuzzle(sudo_m, original_entries);
    int currentScore = currentSudokuPuzzle.scoreBoard();
    int bestScore = currentScore;
    double temperature = 10000;
    double coolingFactor = 0.99995;
    int count = 0;
    for (double t = temperature; t > 1; t *= coolingFactor) {
        try {

            if (count % 1000 == 0) {
                System.out.printf(format:"Iteration %d,\tT = %.5f,\tbest_score = %d,\tcurrent_score = %d%n",
                        count, t, bestScore, currentScore);
            }
            int[][] candidateData = currentSudokuPuzzle.makeCandidateData(original_entries);
            SudokuPuzzle spCandidate = new SudokuPuzzle(candidateData, original_entries);
            int candidateScore = spCandidate.scoreBoard();

            if (currentSudokuPuzzle.acceptanceProbability(spCandidate, temperature) > Math.random()) {
                currentSudokuPuzzle = spCandidate;
                currentScore = candidateScore;
            }

            if (currentScore < bestScore) {

                bestSudokuPuzzle = new SudokuPuzzle(currentSudokuPuzzle.getData(), original_entries);
                bestScore = bestSudokuPuzzle.scoreBoard();
            }
            if (candidateScore == 0) {
                currentSudokuPuzzle = spCandidate;
                break;
            }
        } catch (Exception e) {
            System.out.println(x:"Hit an inexplicable numerical error. It's a random algorithm-- try again.");
        }

        if (bestScore == 0) {
            System.out.println(x:"\nSOLVED THE PUZZLE.");

        }
        count++;
    }
    stat.recordSolutionStatistics(count, count, count);
    return bestSudokuPuzzle.getData();
}
```

Fig. 3. Implemetazione in Java dell'algoritmo Simulated Annealing tradizionale

```
Iteration 0,     T = 10000,00000,        best_score = 34,        current_score = 34
Iteration 1000,  T = 9512,28235, best_score = 31,        current_score = 43
Iteration 2000,  T = 9048,35156, best_score = 31,        current_score = 42
Iteration 3000,  T = 8607,04749, best_score = 31,        current_score = 42
Iteration 4000,  T = 8187,26659, best_score = 27,        current_score = 35
Iteration 5000,  T = 7787,95915, best_score = 27,        current_score = 41
Iteration 6000,  T = 7408,12664, best_score = 27,        current_score = 45
Iteration 7000,  T = 7046,81924, best_score = 27,        current_score = 42
Iteration 8000,  T = 6703,13343, best_score = 27,        current_score = 37
Iteration 9000,  T = 6376,20978, best_score = 27,        current_score = 44
Iteration 10000,         T = 6065,23078, best_score = 27,        current_score = 39
Iteration 11000,         T = 5769,41877, best_score = 27,        current_score = 43
Iteration 12000,         T = 5488,03404, best_score = 27,        current_score = 50
Iteration 13000,         T = 5220,37293, best_score = 27,        current_score = 46
Iteration 14000,         T = 4965,76613, best_score = 27,        current_score = 38
Iteration 15000,         T = 4723,57696, best_score = 27,        current_score = 40
Iteration 16000,         T = 4493,19977, best_score = 27,        current_score = 42
Iteration 17000,         T = 4274,05849, best_score = 27,        current_score = 45
Iteration 18000,         T = 4065,60512, best_score = 27,        current_score = 44
Iteration 19000,         T = 3867,31838, best_score = 27,        current_score = 47
Iteration 20000,         T = 3678,70244, best_score = 27,        current_score = 41
Iteration 21000,         T = 3499,28563, best_score = 27,        current_score = 44
```

Fig. 4. Risultati dell'algoritmo Simulated Annealing tradizionale per la griglia facile1 - esecuzione

```
210     // ALGORITMO DI RICERCA SIMULATED ANNEALING
211     public int[][] solveSudoku_SimulatedAnnealing(int sudo_m[][]) {
212         List<Integer> original_entries = m.getOriginalEntriesList(sudo_m);
213         BoardRandomizer br = new BoardRandomizer();
214         br.generateRandomSudoku(sudo_m);
215         m.printMatrix(sudo_m);
216
217         SudokuPuzzle currentSudokuPuzzle = new SudokuPuzzle(sudo_m, original_entries);
218         SudokuPuzzle bestSudokuPuzzle = new SudokuPuzzle(sudo_m, original_entries);
219         int currentScore = currentSudokuPuzzle.scoreBoard();
220         int bestScore = currentScore;
221         double temperature = 10000;
222         double coolingFactor = 0.99995;
223
224         int count = 0;
225         for (double t = temperature; t > 1; t *= coolingFactor) {
226             try {
227
228                 if (count % 1000 == 0) {
229                     System.out.printf(format:"Iteration %d,\tT = %.5f,\tbest_score = %d,\tcurrent_score = %d%n",
230                             count, t, bestScore, currentScore);
231                 }
232                 int[][] candidateData = currentSudokuPuzzle.makeCandidateData(original_entries);
233                 SudokuPuzzle neCandidate = new SudokuPuzzle(candidateData, original_entries);
```

PROBLEMS 9 OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Iteration 151000,       T = 5,26011,    best_score = 25,        current_score = 42
Iteration 152000,       T = 5,00356,    best_score = 25,        current_score = 42
Iteration 153000,       T = 4,75953,    best_score = 25,        current_score = 42
Iteration 154000,       T = 4,52740,    best_score = 25,        current_score = 44
Iteration 155000,       T = 4,30659,    best_score = 25,        current_score = 42
Iteration 156000,       T = 4,09655,    best_score = 25,        current_score = 48
Iteration 157000,       T = 3,89675,    best_score = 25,        current_score = 39
Iteration 158000,       T = 3,70670,    best_score = 25,        current_score = 44
Iteration 159000,       T = 3,52592,    best_score = 25,        current_score = 44
Iteration 160000,       T = 3,35396,    best_score = 25,        current_score = 47
Iteration 161000,       T = 3,19038,    best_score = 25,        current_score = 36
Iteration 162000,       T = 3,03478,    best_score = 25,        current_score = 41
Iteration 163000,       T = 2,88677,    best_score = 25,        current_score = 41
Iteration 164000,       T = 2,74597,    best_score = 25,        current_score = 46
Iteration 165000,       T = 2,61205,    best_score = 25,        current_score = 39
Iteration 166000,       T = 2,48465,    best_score = 25,        current_score = 45
Iteration 167000,       T = 2,36347,    best_score = 25,        current_score = 42
Iteration 168000,       T = 2,24820,    best_score = 25,        current_score = 36
Iteration 169000,       T = 2,13855,    best_score = 25,        current_score = 43
Iteration 170000,       T = 2,03425,    best_score = 25,        current_score = 43
Iteration 171000,       T = 1,93504,    best_score = 25,        current_score = 39
Iteration 172000,       T = 1,84066,    best_score = 25,        current_score = 37
Iteration 173000,       T = 1,75089,    best_score = 25,        current_score = 44
Iteration 174000,       T = 1,66550,    best_score = 25,        current_score = 45
Iteration 175000,       T = 1,58427,    best_score = 25,        current_score = 43
Iteration 176000,       T = 1,50700,    best_score = 25,        current_score = 40
Iteration 177000,       T = 1,43350,    best_score = 25,        current_score = 45
Iteration 178000,       T = 1,36359,    best_score = 24,        current_score = 39
Iteration 179000,       T = 1,29708,    best_score = 24,        current_score = 40
Iteration 180000,       T = 1,23382,    best_score = 24,        current_score = 36
Iteration 181000,       T = 1,17364,    best_score = 24,        current_score = 37
Iteration 182000,       T = 1,11640,    best_score = 24,        current_score = 44
Iteration 183000,       T = 1,06196,    best_score = 24,        current_score = 37
Iteration 184000,       T = 1,01016,    best_score = 24,        current_score = 42
```

Fig. 5. Risultati dell'algoritmo Simulated Annealing tradizionale per la griglia facile1 - terminazione

Si è provato a manipolare i parametri di controllo in modo da agevolare la ricerca di una soluzione che si avvicinasse il più possibile alla soluzione ottima globalmente, ovvero ad una griglia Sudoku avente 0 duplicati, ottenendo, però, come conseguenza un rallentamento del tempo di esplorazione

nello spazio di ricerca ed un leggero miglioramento nell'ottimo locale trovato (il valore della funzione obiettivo osservato in media cadeva nel range $17 \leq x \leq 24$).

In particolare, per la griglia *Sudoku _facile1 _g*, avente livello di difficoltà facile , si è visto che, ponendo dei valori elevati per i parametri di controllo, ad esempio $temperature = 1000000000000f$ e $coolingfactor = 0.999999999995$, dopo $15minuti$ circa viene restituita come soluzione un ottimo locale avente 7 duplicati totali nelle righe e nelle colonne.

Visto che siamo interessati al tempo di esecuzione, si è provato a considerare altre strategie utili al raggiungimento di una griglia Sudoku completa che si avvicini il più possibile ad avere un numero di duplicati pari a 0. Approfondendo la tematica, si è arrivati ad implementare il seguente algoritmo

```
// ALGORITMO DI RICERCA SIMULATED ANNEALING - VARIANTE
public int[][] solveSudoku_SimulatedAnnealing(int sudo_m[][]) {
    List<Integer> original_entries = m.getOriginalEntriesList(sudo_m);
    BoardRandomizer br = new BoardRandomizer();
    br.generateRandomSudoku(sudo_m);
    m.printMatrix(sudo_m);
    SudokuPuzzle sudokuPuzzle = new SudokuPuzzle(sudo_m);
    // sudokuPuzzle.randomizeOnZeroes();
    SudokuPuzzle bestSudokuPuzzle = new SudokuPuzzle(sudo_m, original_entries);
    int currentScore = sudokuPuzzle.scoreBoard();
    int bestScore = currentScore;
    double temperature = 1000;
    double coolingFactor = 0.995;
    int max_iterations = 250000;

    while (temperature > 1) {
        for (int count = 0; count < max_iterations; count++) {
            try {
                if (count % 1000 == 0) {
                    System.out.printf(format:"Iteration %d,\tT = %.5f,\tbest_score = %d,\tcurrent_score = %d%n",
                            count, temperature, bestScore, currentScore);
                }
                int[][] candidateData = sudokuPuzzle.makeCandidateDataDup(original_entries);
                SudokuPuzzle spCandidate = new SudokuPuzzle(candidateData, original_entries);
                int candidateScore = spCandidate.scoreBoard();
                double deltaS = currentScore - candidateScore;
                if (candidateScore < currentScore || Math.exp(deltaS / temperature) - Math.random() > 0) {
                    sudokuPuzzle = spCandidate;
                    currentScore = candidateScore;
                    if (currentScore < bestScore) {
                        bestSudokuPuzzle = new SudokuPuzzle(sudokuPuzzle.getData(), original_entries);
                        bestScore = bestSudokuPuzzle.scoreBoard();
                    }
                }
                if (candidateScore == 0) {
                    sudokuPuzzle = spCandidate;
                    break;
                }
            } catch (Exception e) {
                System.out.println(x:"Hit an inexplicable numerical error. It's a random algorithm-- try again.");
            }

            if (bestScore == 0) {
                System.out.println(x:"\nSOLVED THE PUZZLE.");

            }
        }
        temperature *= coolingFactor;
    }
    return bestSudokuPuzzle.getData();
}
```

Fig. 6. Implementazione in Java della variante dell'algoritmo Simulated Annealing

di ricerca che adotta lo stesso meccanismo di ricerca adottato dal Simulated Annealing, ma in modo più articolato:

- ad ogni abbassamento della temperatura si generano più di un vicino casuale del nodo corrente - nell'implementazione abbiamo considerato al più 250.000 stati vicini da generare per i motivi che verranno delineati - ;
- il meccanismo di perturbazione per la generazione dei vicini viene modificato, per rendere la generazione degli stati vicini un po' meno casuale; dato uno stato corrente, si determina il suo vicino nel seguente modo:

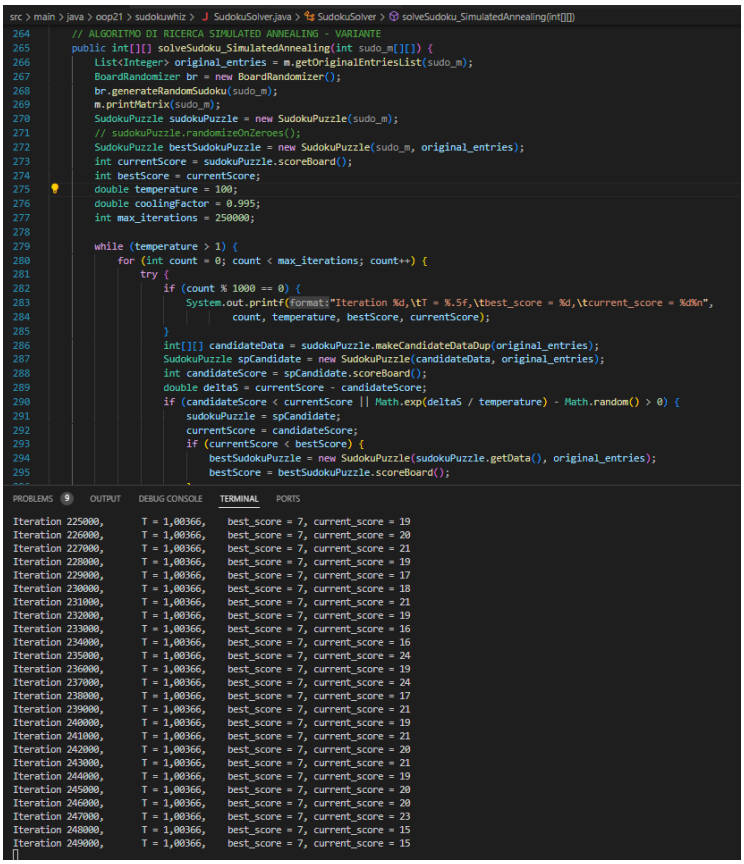

Fig. 7. Risultati della variante dell'algoritmo Simulated Annealing per la griglia facile1

(1) casualmente si scelgono due indici i e j, con $1 \leq i, j \leq 9$ in modo tale che la cella $c_{i,\,j}$ non sia un indizio;

(2) casualmente si scelgono due indici h e k, con $1 \leq h, k \leq 9$ in modo tale che: la cella $c_{h,\,k}$ non sia un indizio,$c_{h,\,k} \neq c_{i,\,j}$, $c_{h,\,k}$ si trovi nella stessa sotto-griglia di $c_{i,\,j}$ e il valore presente in $c_{h,\,k}$ sia un duplicato nella riga h oppure un duplicato nella colonna k;

(3) scambiare le celle $c_{i,\,j}$ e $c_{h,\,k}$ tra loro.

Questo meccanismo di perturbazione garantisce che, in ogni iterazione, non ci sono duplicati in ciascuna griglia e di ridurre la possibilità di aumentare il numero di duplicati generati.

I risultati ottenuti dall'esecuzione di questo algoritmo sono promettenti; infatti, testando questa strategia di ricerca sulla griglia Sudoku *Sudoku _facile1 _g*, in meno di 7 minuti è stata restituita come griglia risultante una soluzione localmente ottima avente numero totale di duplicati pari a 7, partendo da una griglia inizialmente riempita, in modo casuale, con 43 duplicati totali.

Visto il tempo di esecuzione riscontrato per una griglia di difficoltà facile, i valori dei parametri di controllo non sono stati ulteriormente aumentati rispetto a quelli delineati nell'implementazione.

9 ALGORITMI GENETICI GENERAZIONALI

9.1 Meccanismo di funzionamento

Gli algoritmi genetici rappresentano una procedura ad alto livello (meta-euristica), ispirati alla genetica, per definire un algoritmo di ricerca al fine di risolvere problemi di ottimizzazione computazionalmente difficili.

Il meccanismo di funzionamento degli algoritmi genetici si ispira alla teoria dell'evoluzione di Charles Darwin, in particolare ai concetti di selezione naturale, adattamento e sopravvivenza di un individuo nell'ambiente.

In modo analogo al processo di selezione naturale, negli algoritmi genetici si intende selezionare, a partire da una popolazione iniziale di individui, le soluzioni migliori (gli individui che si adattano meglio nell'ambiente) rispetto ad un obiettivo, definito da una funzione di fitness. Successivamente, tali soluzioni vengono ricombinate fra loro (riproduzione degli individui), dando vita a nuove soluzioni candidate che ereditano le caratteristiche dei genitori e ne sviluppano delle proprie. In questo modo, il processo evolutivo porterà la popolazione verso un punto di ottimo o verso una condizione di arresto, restituendo in tal caso una soluzione sub-ottima.

Gli algoritmi genetici generazionali strutturano il processo evolutivo di una popolazione nel seguente modo:

(1) Inizializzazione

In questa fase si genera una popolazione iniziale di individui, spesso scelti in modo casuale o mediante specifiche strategie di campionamento. Ogni individuo rappresenta una possibile soluzione al problema in esame e ha un set di caratteristiche (geni) che influiscono sulla sua performance.

(2) Valutazione

Ogni individuo nella popolazione viene valutato l'abilità di un individuo di competere con gli altri e di sopravvivere nell'ambiente circostante. Questa valutazione è basata su una funzione di fitness che misura quanto bene un individuo si adatta al problema. Essa assegna un punteggio ad ogni individuo e la probabilità che un individuo venga selezionato per la riproduzione è basata su questo punteggio.

(3) Selezione

Gli individui vengono selezionati per la riproduzione in base al loro valore di fitness. Gli individui più adatti hanno una maggiore probabilità di essere selezionati per tramandare i propri geni alla generazione successiva[1].

(4) Crossover

Gli individui selezionati vengono combinati attraverso operazioni di crossover per generare una nuova popolazione. Questa fase simula la ricombinazione genetica tra genitori.

(5) Mutazione

Alcuni individui della nuova popolazione possono subire delle mutazioni casuali: alcuni geni, scelti casualmente dal materiale genetico dell'individuo, vengono modificati. Ciò introduce variazioni genetiche occasionali per esplorare nuove regioni dello spazio delle soluzioni.

(6) Sostituzione della popolazione

La nuova popolazione, derivata dall'applicazione sequenziale degli operatori genetici - selezione, crossover e mutazione - sostituisce completamente la popolazione della generazione precedente. Questo costituisce una nuova "generazione".

(7) Terminazione

Il processo di selezione, crossover, mutazione e sostituzione della popolazione si ripete attraverso un numero prefissato di generazioni o fino a che non viene soddisfatto un criterio di terminazione (ad esempio, raggiungimento di una soluzione accettabile, assenza di miglioramenti, tempo di esecuzione).

9.2 Gli algoritmi genetici generazionali in SudokuWhiz

Tenendo conto del processo evolutivo adottato dagli algoritmi genetici generazionali descritto nel paragrafo precedente, si delinea l'adattamento di questo meccanismo nella risoluzione del gioco del Sudoku:

(1) Inizializzazione

In questa fase si genera una popolazione iniziale di individui: ogni individuo è una griglia Sudoku $9x9$ completa, dove ogni cella bianca è riempita casualmente con un numero da 1 a 9, in modo tale che ogni regione $3x3$ non contenga alcun duplicato.
In questo modo, ci si focalizza sul far evolvere la generazione corrente in modo che abbia un numero totale di duplicati per le righe e le colonne inferiore alla generazione precedente.

(2) Valutazione

Ogni individuo nella popolazione viene valutato per il numero totale di duplicati presenti nelle rispettive righe e colonne.

(3) Selezione

Visto che si tiene conto del tempo di esecuzione richiesto dagli algoritmi genetici per risolvere una griglia Sudoku e che si vuole tenere traccia degli individui più promettenti per il valore di fitness, si intende adottare l'algoritmo di selezione RouletteWheel e promuovere l'elitismo.
Dunque, prima dell'esecuzione del Roulette Wheel, si effettuerà direttamente una copia dei primi k individui della generazione corrente più promettenti per il valore di fitness. Il valore k, denotato *ELITISM _COUNTER*, verrà stimato sperimentalmente, durante l'esecuzione degli algoritmi genetici.

(4) Crossover

Al fine di creare nuovi individui ritenuti soluzioni ammissibili al problema - possiedono 9 sotto-griglie prive di duplicati -, si è pensato di adottare l'algoritmo di crossover Uniform: dal mating pool ottenuto dall'applicazione dell'operatore di selezione, si prelevano due individui - denotati *genitori* - per generare esattamente due nuovi individui - detti *figli* - .
Un figlio è una griglia $9x9$ dove l'i-esima sottogriglia $3x3$ ($1 \leq i \leq 9$) è scelta, in modo casuale, tra le i-esime sottogriglie dei due genitori. L'algoritmo di crossover, inoltre, viene realizzato in modo che ogni figlio ha almeno una sottogriglia di entrambi i genitori, al fine di evitare di avere delle copie identiche dei genitori.
La probabilità con cui viene selezionata una sottogriglia $3x3$ per il nuovo individuo, detta *CROSSOVER _RATE* verrà stabilita in fase sperimentale.

[1]Ne "L'origine della specie", Darwin afferma che una generazione di individui che sopravvive tende a produrre più individui, ai quali trasmette caratteristiche ereditabili che consentiranno alla prole di sopravvivere di più e di "portare avanti la specie".

(5) Mutazione

Al fine di applicare delle perturbazioni piccole ma significative e di avere degli individui ottenuti dalla mutazione che rispettino la codifica definita nel punto (1) del suddetto elenco, l'algoritmo di mutazione adottato è Scramble.
In base ad un parametro detto *MUTATION _RATE*, definito in fase di sperimentazione, verrà scelta casualmente la sottogriglia $3x3$ alla quale effettuare una permutazione di k delle sue celle. Il numero delle celle da permutare k viene scelto casualmente tra un numero compresa tra 2 (il semplice swap di posizione delle due celle) e il numero di celle della sottogriglia che non sono indizi della griglia Sudoku fornita in input.

(6) Sostituzione della popolazione

La nuova popolazione, derivata dall'applicazione sequenziale degli operatori genetici - selezione, crossover e mutazione - sostituisce completamente la popolazione della generazione precedente.

(7) Terminazione

Il processo di selezione, crossover, mutazione e sostituzione della popolazione si ripete attraverso un numero prefissato di generazioni o fino a che non si raggiunge una soluzione che ha, in assoluto, il miglior valore per la funzione di fitness: una griglia Sudoku avente in totale 0 duplicati nelle righe e nelle colonne.

9.3 Complessità degli Algoritmi Genetici generazionali

La complessità degli Algoritmi genetici Generazionali, analogamente al Simulated Annealing, non è facilmente esprimibile in forma di notazione O poiché, tramite l'applicazione degli operatori genetici, include nella strategia di ricerca una componente di casualità.

È possibile, tuttavia, fare delle considerazioni sul tempo di esecuzione, sulla bontà dei risultati ottenuti e sulla convergenza e/o stagnazione dell'algoritmo in una regione dello spazio di ricerca.

```
    //ALGORITMO DI RICERCA GENETICO
public int[][] solveSudoku_GeneticAlgorithm(int sudo_m[][]) {
    List<Integer> hints = m.getOriginalEntriesList(sudo_m);
    /*GA parametri*/
    int POPULATION_SIZE = 200;
    Population startingPopulation;
    int MAX_GENERATIONS = 5000;
    startingPopulation = new Population(POPULATION_SIZE,sudo_m);
    int generationCount = 0;
    FitnessCalc fit = new FitnessCalc();
        while (generationCount<MAX_GENERATIONS && startingPopulation.getFittest().getFitness() < fit.calculateTotalFitness(startingPopulation)) {
            generationCount++;
            System.out.println("Generation: " + generationCount + " Fittest: " + startingPopulation.getFittest().getFitness());
            startingPopulation = Algorithm.evolvePopulation(startingPopulation,sudo_m,hints);
            //ManageMatrix m= new ManageMatrix();
            //m.printMatrix(startingPopulation.getFittest().getSudo_m());
        }
        System.out.println(x:"Solution found!");
        System.out.println("Generation: " + generationCount);
        System.out.println(x:"Genes:");
        System.out.println(startingPopulation.getFittest());
        ManageMatrix m = new ManageMatrix();
        m.printMatrix(startingPopulation.getFittest().getSudo_m());
        return startingPopulation.getFittest().getSudo_m();
}
```

Fig. 8. Implementazione in Java di un algoritmo genetico

I valori dei parametri di controllo, utili per l'evoluzione della popolazione iniziale, sono impostati come:

- numero di individui per ogni generazione: 200;
- numero massimo di generazioni: 5000;
- numero di elementi promettenti da preservare dal processo di selezione: 100;

- dimensione del mating pool (insieme di individui scelti in fase di selezione): 100 (POPULATION _SIZE - ELITISM _POOL);
- probabilità di mutazione: 0.4;
- probabilità di crossover: 0.5.

A partire da questi valori iniziali per i parametri di controllo, si sono analizzati i risultati ottenuti eseguendo il programma Java per le griglie Sudoku presenti in Griglie _test.

Si è osservato che l'esecuzione dell'algoritmo per tutte le tipologie di difficoltà delle griglie Sudoku termina per aver superato il numero massimo di generazioni (tempo di esecuzione: tra 4s e 5s). In particolare, si riporta per ogni categoria di difficoltà della griglia Sudoku il range di valori di fitness degli individui forniti come soluzioni ottime localmente, ma non globalmente:

- facile: $55 \leq score \leq 61$;
- medio: $56 \leq score \leq 62$;
- caso peggiore: $57 \leq score \leq 71$;

Visto che per le categorie delle griglie Sudoku si è raggiunta una soluzione localmente ottima che risiede in un intervallo di punteggio di fitness tra 55 e 70, si è pensato di modificare i parametri di controllo - aumento o diminuzione - per vedere eventuali miglioramenti.

Nonostante si siano provate diverse combinazioni di valori dei parametri di controllo, gli effetti di tale modifiche si sono concretizzati in lievi abbassamenti, in termini di poche unità, dei valori di fitness degli individui restituiti come soluzioni, a discapito del tempo di esecuzione.

10 CONCLUSIONI

Il presente progetto è stato svolto tenendo conto delle conoscenze acquisite durante il corso di Fondamenti di Intelligenza Artificiale. In particolare, il gruppo di lavoro ha ritenuto interessante applicare gli algoritmi di ricerca per risolvere il gioco del Sudoku.

Le motivazioni di tale scelta risiedono nella diffusione e nella popolarità del gioco del Sudoku; la versione moderna del Sudoku è diventata nota a livello internazionale soltanto a partire dal 2005, benché appassioni numerosi giocatori sin dalla seconda metà degli anni ’80 del secolo scorso.

Nel progetto presentato si è inteso costruire un agente intelligente in grado di in grado di risolvere il gioco in modo corretto e nel minor tempo possibile.

Le impostazioni generali degli algoritmi, le informazioni ed i suggerimenti del docente, le conoscenze acquisite durante il corso di Fondamenti di Intelligenza Artificiale hanno consentito al gruppo di individuare diverse strategie per risolvere il Sudoku, delle quali viene data ampia descrizione nel progetto.

Il risultato ottenuto è stato raggiunto attraverso una serie di sperimentazioni. L’insieme delle soluzioni raggiunte, tutte ugualmente significative ai fini di una completa ed ampia trattazione dell’argomento affrontato, ci ha concretamente evidenziato quale debba essere il nostro approccio rispetto alla risoluzione di un problema.

Partendo da un’approfondita analisi della descrizione dell’ambiente, della formulazione del problema, si è giunti, attraverso l’implementazione degli algoritmi di ricerca, ad un ventaglio di soluzioni possibili per, poi, scegliere quella che risulta essere completa, ottima, efficiente nella complessità temporale e spaziale.

Concludiamo esprimendo un positivo apprezzamento sui risultati raggiunti nel progetto e, più in generale, sull’esperienza maturata grazie alle conoscenze acquisite durante il corso di Fondamenti di Intelligenza Artificiale.

GLOSSARIO

MNE acronimo di numero Medio dei Nodi Esplorati dall'algoritmo di ricerca durante la sua esecuzione . 9

MNG acronimo di numero Medio dei Nodi Generati dall'algoritmo di ricerca A* durante la sua esecuzione . 13

MNS acronimo di numero Medio dei Nodi Soluzione esplorati dall'algoritmo di ricerca durante la sua esecuzione per determinare la soluzione. 9, 13

PEAS acronimo di Performance Environment Actuators Sensors; descrive l'ambiente operativo - la misura di prestazione, l'ambiente esterno, gli attuatori e i sensori dell'agente - . 4

Spazio degli stati si intende lo stato iniziale, l'insieme di azioni possibili e il modello di transizione definiti nella formulazione di un problema da risolvere. 7