



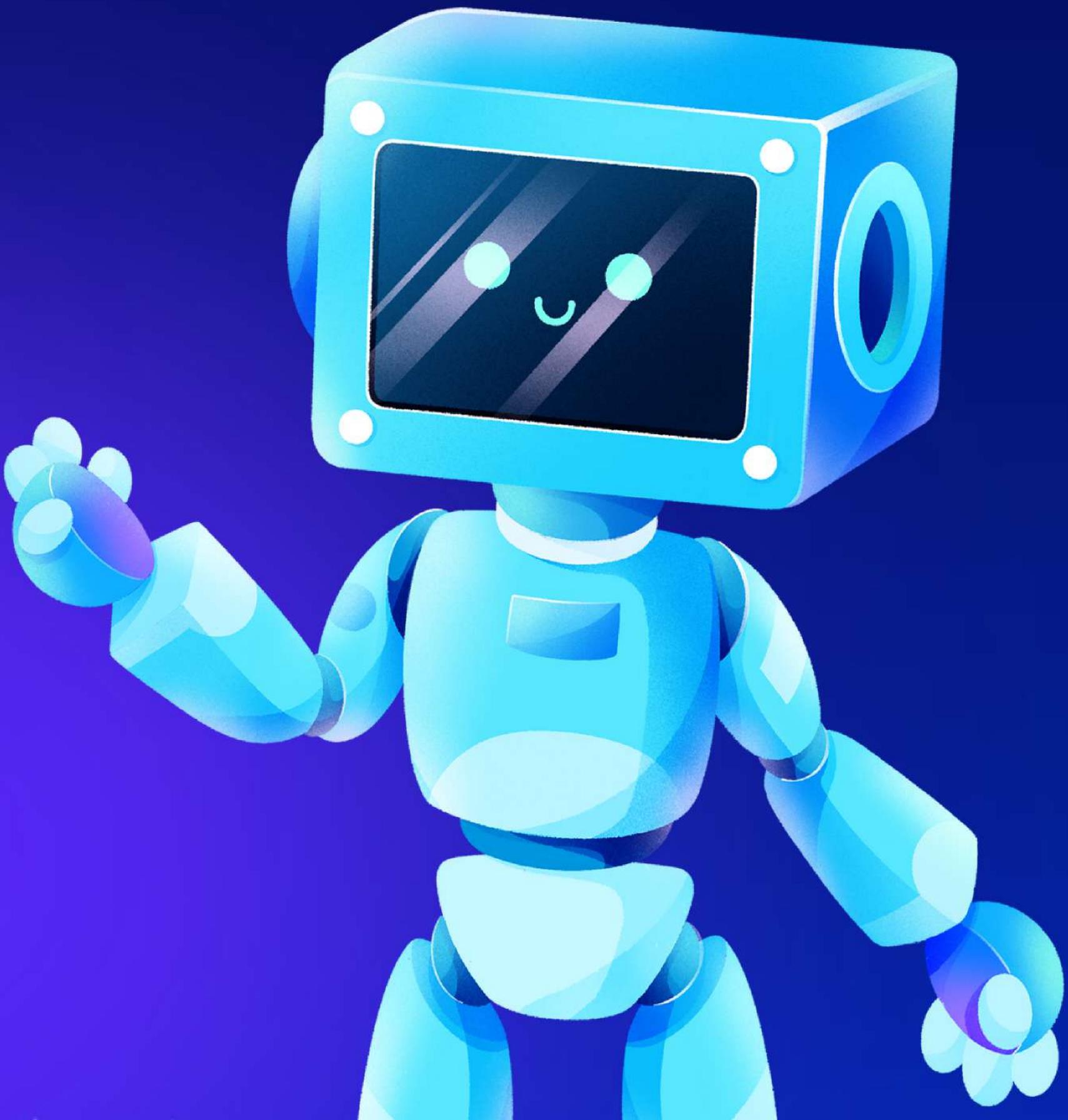
CORSO DI FONDAMENTI DI INTELLIGENZA  
ARTIFICIALE  
A.A. 2023/2024

# SUDOKUWHIZ

Presentazione progetto

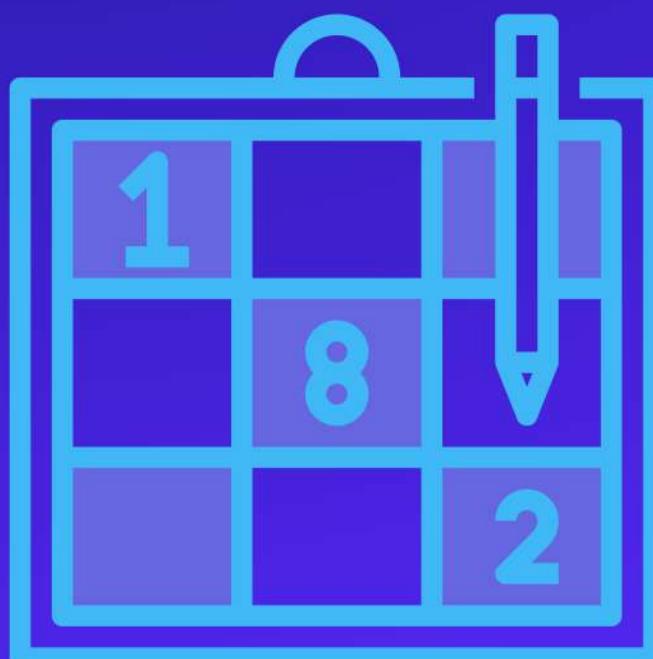
Raffaella Sabatino n. matricola 0512115114

Dorotea Serrelli n. matricola 0512113740





# CONTENUTI



- Il gioco
- Obiettivi del progetto
- Specifiche P.E.A.S.
- Caratteristiche dell'ambiente
- Analisi del problema
- Algoritmi di ricerca
  - Backtracking
  - Ricerca A\*
  - Simulated Annealing
  - Algoritmi genetici
- Conclusioni



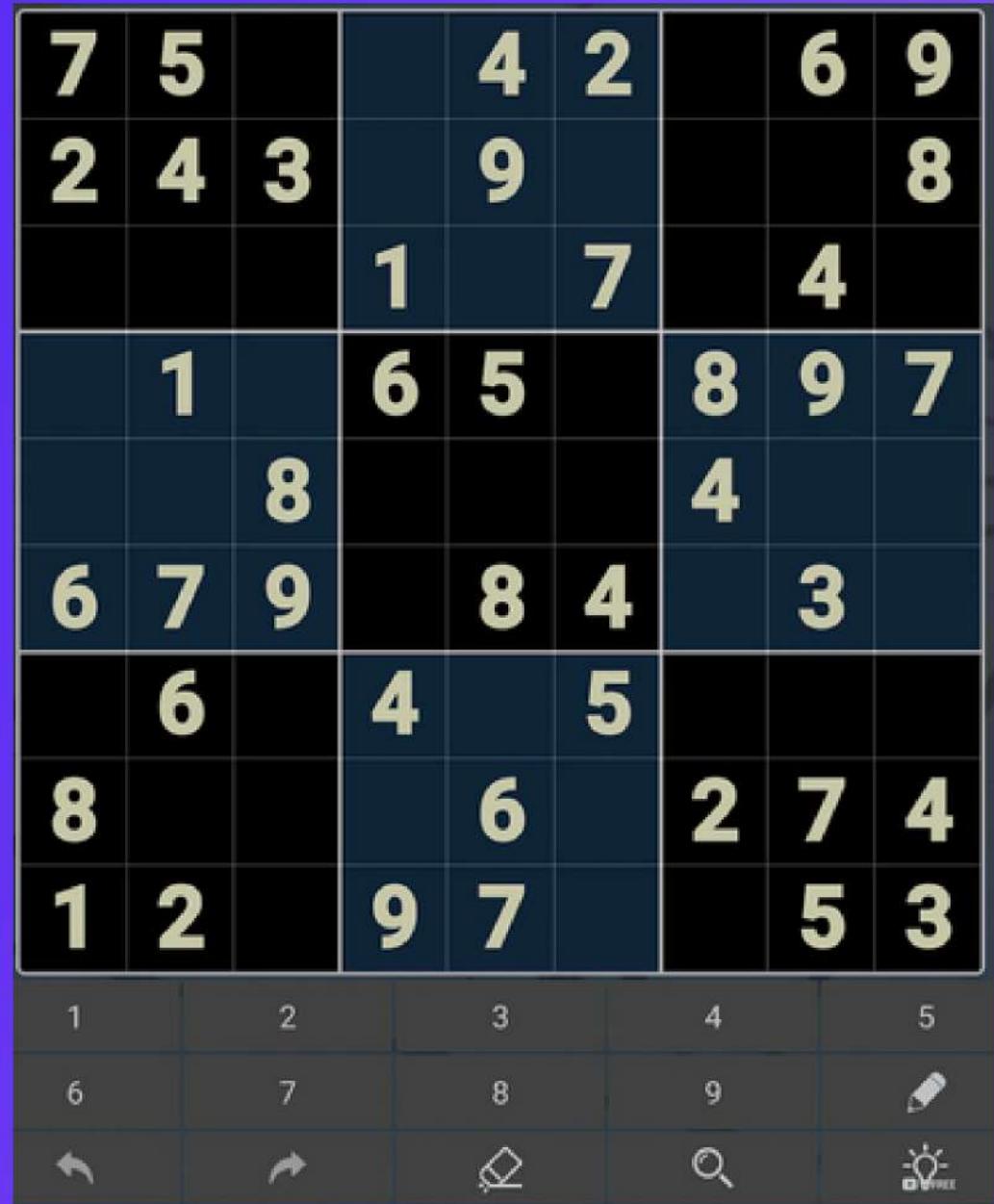
# IL GIOCO

Il Sudoku è un gioco di logica nel quale viene proposta una griglia di  $9 \times 9$  celle, ciascuna delle quali può contenere un numero da 1 a 9, oppure essere vuota.

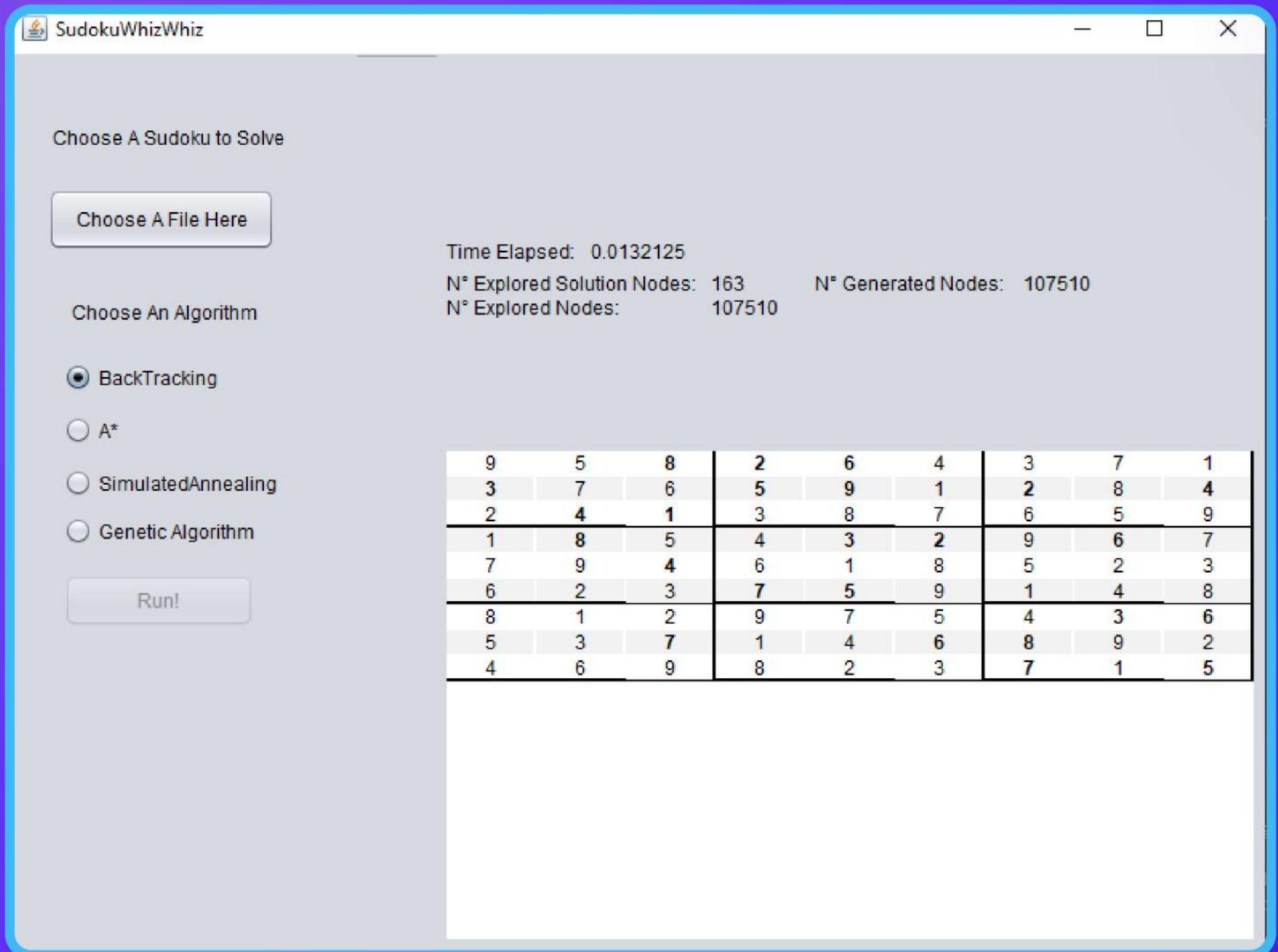
La griglia è suddivisa in 9 righe orizzontali, 9 colonne verticali e in 9 "sottogriglie" (dette regioni) di  $3 \times 3$  celle contigue.

## Obiettivo

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.



# OBIETTIVI DEL PROGETTO



Si intende costruire un agente intelligente in grado di risolvere il gioco in modo corretto e nel minor tempo possibile.

# SPECIFICA P.E.A.S



## Performance

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

## Environment

La griglia Sudoku: una griglia 9X9 parzialmente riempita, suddivisa in 9 sottogrille 3X3 denominate regioni; sono incluse anche le celle vuote -denotate con il valore 0 - da riempire con numeri da 1 a 9.

# SPECIFICA P.E.A.S

Actuators



- Inserimento di numeri nelle celle vuote della griglia;
- Cancellazione di un numero inserito dall'agente;
- Spostamento in una nuova cella;
- Aggiornamento della griglia, a seguito di una modifica alla configurazione della griglia.

Sensors



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.

# CARATTERISTICHE DELL'AMBIENTE



**Singolo Agente:**  
L'unico agente che opera  
è quello in oggetto.



**Completamente  
osservabile:**  
L'agente è a conoscenza  
in ogni istante della  
configurazione della  
griglia Sudoku.



**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 corrente  
della griglia dipende dalle azioni  
fatte precedentemente.

# SPECIFICHE DELL'AMBIENTE

05

## Statico:

La griglia Sudoku non muta mentre l'agente sta scegliendo la prossima azione da compiere.

06

## 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 avente un numero discreto (da 0 a 9).

# ANALISI DEL PROBLEMA

## STATO INIZIALE

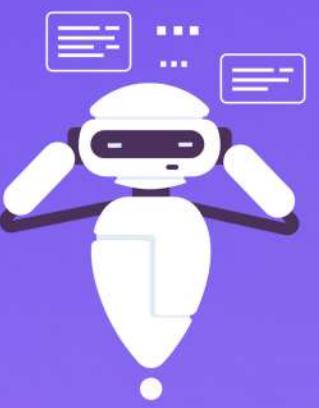
Sudoku_casoPeggiore2_g								
File	Modifica	Formato	...	...	...	...	...	...
0 0 0 0 0 0 0 0 0								
0 0 0 0 0 3 0 8 5								
0 0 1 0 2 0 0 0 0								
0 0 0 5 0 7 0 0 0								
0 0 4 0 0 0 1 0 0								
0 9 0 0 0 0 0 0 0								
5 0 0 0 0 0 0 7 3								
0 0 2 0 1 0 0 0 0								
0 0 0 0 4 0 0 0 9								

La griglia 9×9, parzialmente riempita con almeno 17 indizi e massimo 30 indizi, è definita in un file di estensione .txt.



## AZIONI

- 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 - eccetto un indizio - in una cella;
- Spostamento verso una nuova cella.



## MODELLO DI TRANSIZIONE

Il modello di transizione descrive come cambia la configurazione della griglia Sudoku in risposta alle azioni eseguite dall'agente, verificando, in ogni istante, se vengono rispettate le regole del gioco

# ANALISI DEL PROBLEMA



## TEST OBIETTIVO

Si verifica se la griglia Sudoku è stata completamente riempita, in accordo con le regole del gioco.



## COSTO DI CAMMINO

Il costo dell'esecuzione di una o più azioni necessarie per risolvere il Sudoku.

Si definiscono i seguenti costi:

- Lettura di una cella: 1;
- Spostamento da una cella all'altra: 1;
- Inserimento/Cancellazione: 2

# ALGORITMI DI RICERCA



```

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;
    }
    for (int num = 1; num <= 9; num++) {
        /*
         * isSafe controlla che num non sia già presente
         * nella riga, colonna, o sotto-griglia 3x3
         */
        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)) {
                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;
}

```

# BACKTRACKING



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  $s = (a_1, a_2, a_3, \dots, a_k)$ , dove l'elemento  $a_i$  viene scelto da un insieme ordinato  $S_i$  di possibili candidati per la posizione  $i$ .



## MECCANISMO DI FUNZIONAMENTO

Ad ogni passo si tenta di inserire, in modo iterativo, un numero da 1 a 9, verificando che rispetti tre condizioni: quel numero non è presente nella riga, colonna e regione della cella in cui viene inserito.

In caso affermativo se ne conferma tale inserimento.

Successivamente si passa alla prossima cella vuota ma con la griglia Sudoku aggiornata con il valore precedentemente inserito, fino a quando non si raggiunge una cella, 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.



## COMPLESSITÀ

- **Completo:** in modo esaustivo esplora tutte le possibili combinazioni, trovando una soluzione.
- **Ottimo:** riesce a determinare la soluzione ottima.
- **complessità temporale:** è esponenziale  $O(9^{n \times n})$  poiché, per ogni cella, si hanno a disposizione 9 valori tra cui scegliere.
- **complessità spaziale:** è pari a  $O(n^2)$ .



## RICERCA A\*

```
// 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, cost:0, 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;  
}
```

L'algoritmo procede espandendo il nodo avente il valore minimo della funzione di valutazione  $f(n) = g(n) + h(n)$ .

- $g(n)$  = costo del cammino per raggiungere il nodo  $n$  a partire dalla radice;
- $h(n)$  = costo stimato dall'algoritmo per raggiungere lo stato obiettivo a partire dal nodo  $n$ .

## MECCANISMO DI FUNZIONAMENTO



- In ogni stadio, l'algoritmo A\* cerca di effettuare un inserimento nella 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.
- $h(n)$  = numero di celle vuote (per  $h(n) = 0$  si ha uno stato obiettivo).
- $h_2(n)$  = numero di valori nel range  $1 \leq x \leq 9$  assegnabili in una cella.
- $g(n)$  = costo totale delle azioni effettuate per raggiungere il nodo  $n$
- Strutture dati: min-coda a priorità, insieme degli stati visitati, matrice

## COMPLESSITÀ



- **completo**: esiste un numero finito di nodi di costo  $c \leq C^*$
- **ottimo** per ammissibilità e consistenza di  $h(n)$  e  $h_2(n)$
- **complessità temporale**:  $O(b^r)$  con  $\text{costo(azione)} > r > 0$
- **complessità spaziale**:  $O(b^m)$



# MIGLIORAMENTO ITERATIVO: SIMULATED ANNEALING E GGA

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.



# FORMULAZIONE DEL PROBLEMA A STATO COMPLETO



+-----+	+-----+	+-----+
1 7 9   7 5 4   1 5 8		
3 5 6   9 6 1   4 3 2		
2 8 4   3 8 2   6 7 9		
+-----+	+-----+	+-----+
4 6 5   7 3 9   8 9 1		
9 1 3   2 1 8   5 7 6		
2 8 7   6 4 5   2 4 3		
+-----+	+-----+	+-----+
5 8 7   1 2 6   9 4 7		
1 9 6   8 4 5   5 3 6		
4 3 2   3 9 7   1 2 8		
+-----+	+-----+	+-----+

- **Stato iniziale:** la griglia fornita dall'utente, dove le celle bianche vengono casualmente riempite con valori da 1 a 9, in modo tale che ogni regione non possieda duplicati
- **Azioni:** inserimento di un numero, lettura di un numero, spostamento verso una nuova cella, scambio di posizione di due celle non indizi;
- **Modello di transizione**
- **Test obiettivo**
- **Costo di cammino:** le azioni hanno lo stesso costo.

# 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.

## MECCANISMO DI FUNZIONAMENTO

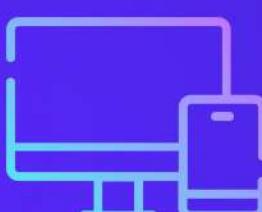


- **Funzione obiettivo:** numero di duplicati totali nelle righe e nelle colonne.
- A partire dallo stato corrente  $s$ , si genera casualmente uno stato vicino  $s'$  mediante lo scambio di posizione di due celle di una sottogriglia.
- Se  $f(s') < f(s)$  allora lo stato vicino  $s'$  verrà accettato sicuramente come prossima mossa.
- Se  $f(s') \geq f(s)$  lo stato vicino  $s'$  verrà accettato, ma con una probabilità inferiore a 1

$$e^{\frac{-\Delta E}{T}},$$

- **decremento della temperatura:**  $T' = \text{temp} * \text{cooling\_factor}$

## COMPLESSITÀ



- Essendo un algoritmo probabilistico, la complessità del Simulated Annealing non è facilmente esprimibile in forma di notazione O.
- È possibile, tuttavia, fare delle considerazioni sul tempo di esecuzione, sulla bontà dei risultati ottenuti e sul meccanismo di perturbazione.

```
// 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("Hit an inexplicable numerical error. It's a random algorithm-- try again.");
        }

        if (bestScore == 0) {
            System.out.println("\nSOLVED THE PUZZLE.");
        }
        count++;
    }
    stat.recordSolutionStatistics(count, count, count);
    return bestSudokuPuzzle.getData();
}
```

# COMPLESSITÀ ALGORITMO S.A.

```
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
```

Esecuzione dell'algoritmo S.A. per risolvere  
una griglia Sudoku di difficoltà facile  
(Sudoku\_facile\_1\_g)

**Tempo impiegato: circa 7 minuti**  
**Best score: 24!**

Aumento temperatura e cooling factor?  
**Tempo impiegato: circa 15 minuti!!!**

```
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             if (count % 1000 == 0) {
228                 System.out.printf(format:"Iteration %d,\tT = %.5f,\tbest_score = %d,\tcurrent_score = %d%n",
229                                 count, t, bestScore, currentScore);
230             }
231             int[][] candidateData = currentSudokuPuzzle.makeCandidateData(original_entries);
232             SudokuPuzzle snCandidate = new SudokuPuzzle(candidateData, original_entries);
233         } catch (Exception e) {
234             e.printStackTrace();
235         }
236         count++;
237     }
238
239     return bestSudokuPuzzle.getSolution();
240 }
```

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
```

# MIGLIORAMENTI PER S.A.

01

- 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.

02

- Il meccanismo di perturbazione viene modificato: dato uno stato corrente, uno stato vicino viene generato scambiando di posizione, in una sottogriglia, una cella - non indizio - con una cella che è un duplicato nella riga e nella colonna a cui appartiene.

# ALGORITMO S.A. CON MIGLIORAMENTI

```
// 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();
}
```

PROBLEMS	OUTPUT	DEBUG CONSOLE	TERMINAL	PORTS
9				
	Iteration 225000, T = 1,00366,	best_score = 7, current_score = 19		
	Iteration 226000, T = 1,00366,	best_score = 7, current_score = 20		
	Iteration 227000, T = 1,00366,	best_score = 7, current_score = 21		
	Iteration 228000, T = 1,00366,	best_score = 7, current_score = 19		
	Iteration 229000, T = 1,00366,	best_score = 7, current_score = 17		
	Iteration 230000, T = 1,00366,	best_score = 7, current_score = 18		
	Iteration 231000, T = 1,00366,	best_score = 7, current_score = 21		
	Iteration 232000, T = 1,00366,	best_score = 7, current_score = 19		
	Iteration 233000, T = 1,00366,	best_score = 7, current_score = 16		
	Iteration 234000, T = 1,00366,	best_score = 7, current_score = 16		
	Iteration 235000, T = 1,00366,	best_score = 7, current_score = 24		
	Iteration 236000, T = 1,00366,	best_score = 7, current_score = 19		
	Iteration 237000, T = 1,00366,	best_score = 7, current_score = 24		
	Iteration 238000, T = 1,00366,	best_score = 7, current_score = 17		
	Iteration 239000, T = 1,00366,	best_score = 7, current_score = 21		
	Iteration 240000, T = 1,00366,	best_score = 7, current_score = 19		
	Iteration 241000, T = 1,00366,	best_score = 7, current_score = 21		
	Iteration 242000, T = 1,00366,	best_score = 7, current_score = 20		
	Iteration 243000, T = 1,00366,	best_score = 7, current_score = 21		
	Iteration 244000, T = 1,00366,	best_score = 7, current_score = 19		
	Iteration 245000, T = 1,00366,	best_score = 7, current_score = 20		
	Iteration 246000, T = 1,00366,	best_score = 7, current_score = 20		
	Iteration 247000, T = 1,00366,	best_score = 7, current_score = 23		
	Iteration 248000, T = 1,00366,	best_score = 7, current_score = 15		
	Iteration 249000, T = 1,00366,	best_score = 7, current_score = 15		
	□			

Esecuzione dell'algoritmo S.A. per risolvere  
una griglia Sudoku di difficoltà facile  
(Sudoku\_facile\_1\_g)

**Initial score : 43**  
**best score: 7**  
**tempo impiegato: circa 7 minuti**

# ALGORITMI GENETICI GENERAZIONALI



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.

## MECCANISMO DI FUNZIONAMENTO



Ispirandosi alla teoria dell'evoluzione di Charles Darwin, in particolare ai concetti di selezione naturale, adattamento e sopravvivenza di un individuo nell'ambiente, 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, nel nostro caso matrici 9x9 con le celle bianche riempite con valori casuali in modo tale che le sottogriglie 3x3 siano prive di duplicati.

### (2) Valutazione

La funzione di fitness adottata per valutare quanto bene un individuo della popolazione si adatta al problema è il calcolo del numero totale di duplicati nelle righe e nelle colonne della griglia Sudoku.

### (3) Selezione

Visto che si tiene conto del tempo di esecuzione e si intende tenere traccia degli individui più promettenti per il valore di fitness, si adotta l'algoritmo di selezione RouletteWheel e si promuove l'elitismo. Dunque, prima dell'esecuzione del Roulette Wheel, si effettuerà direttamente una copia dei primi *ELITISM\_COUNTER* individui della generazione corrente più promettenti per il valore di fitness.

```

public class Algorithm {
    /* GA parametri */
    private static final int GRID_NO = 9; // numero di griglie nel Sudoku
    private static final double MUTATION_RATE = 0.9;
    private static final double CROSSOVER_RATE = 0.5;
    private static final int POPULATION_SIZE = 200;
    private static final int ELITISM_POOL = POPULATION_SIZE / 2;
    private static final int MATING_POOL = POPULATION_SIZE - ELITISM_POOL;

    // Evoluzione della popolazione

    public static Population evolvePopulation(Population pop, int[][] board, List<Integer> hints) {
        Population newPopulation = new Population(pop.size());
        List<Integer> rhints = hints;

        FitnessCalc fc = new FitnessCalc();
        // Elitismo
        Individual elites[] = fc.getElites(pop);
        pop.removeElites(elites);

        for (int i = 0; i < ELITISM_POOL; i++) {
            newPopulation.saveIndividual(i, elites[i]);
        }

        // Si calcolano le probabilità che hanno gli individui di essere scelti.
        List<Individual> parents = selectParents(pop);
        // Dal mating pool generato si effettua il crossover
        for (int i = 0; i < parents.size() - 1; i++) {
            int fatherIndex = new Random().nextInt(parents.size());
            int motherIndex = new Random().nextInt(parents.size());

            Individual father = parents.get(fatherIndex);
            Individual mother = parents.get(motherIndex);
            Individual[] children = crossover(father, mother);
            newPopulation.saveIndividual(newPopulation, i + ELITISM_POOL, children[0]);
            newPopulation.saveIndividual(newPopulation, i + 1 + ELITISM_POOL, children[1]);
        }

        // Mutazione
        for (int i = ELITISM_POOL; i < newPopulation.size(); i++) {
            if (newPopulation.getIdByPhisicIndex(i) != null) {
                mutate(newPopulation.getIdByPhisicIndex(i), rhints);
            }
        }
    }

    return newPopulation;
}

```

#### (4) Crossover

Al fine di creare nuovi individui ritenuti soluzioni ammissibili al problema, gli individui selezionati vengono combinati tra loro mediante l'algoritmo di crossover Uniform: si prendono casualmente dal padre alcune regioni 3x3 e dalla madre le rimanenti per il primo figlio; viceversa per il secondo figlio.

#### (5) Mutazione

Alcuni individui della nuova popolazione possono subire delle mutazioni casuali tramite scramble mutation di m celle della sottogriglia n che non siano indizi. Nel caso in cui il numero di celle m da permutare è troppo grande rispetto al numero di celle in n non indizi, si sceglie un valore casuale tra 2 e il numero (9 - numero indizi in quella sottogriglia).

#### (6) Sostituzione della popolazione

La nuova popolazione, derivata dall'applicazione sequenziale degli operatori genetici, sostituisce completamente la popolazione della generazione precedente, includendo l'elitismo.

#### (7) Terminazione

Il processo di evoluzione della popolazione si ripete attraverso un numero prefissato di generazioni o fino a che non si ottiene un individuo che abbia in totale 0 duplicati nelle righe e nelle colonne.



## COMPLESSITÀ

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 sui meccanismi di selezione-crossover e mutazione usati per la generazione della nuova generazione.



## ... COMPLESSITÀ

```
public class Algorithm {  
    /* GA parametri */  
    private static final int GRID_NO = 9; // numero di griglie nel Sudoku  
    private static final double MUTATION_RATE = 0.9;  
    private static final double CROSSOVER_RATE = 0.5;  
    private static final int POPULATION_SIZE = 200;  
    private static final int ELITISM_POOL = POPULATION_SIZE / 2;  
    private static final int MATING_POOL = POPULATION_SIZE - ELITISM_POOL;
```

Altri parametri:

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

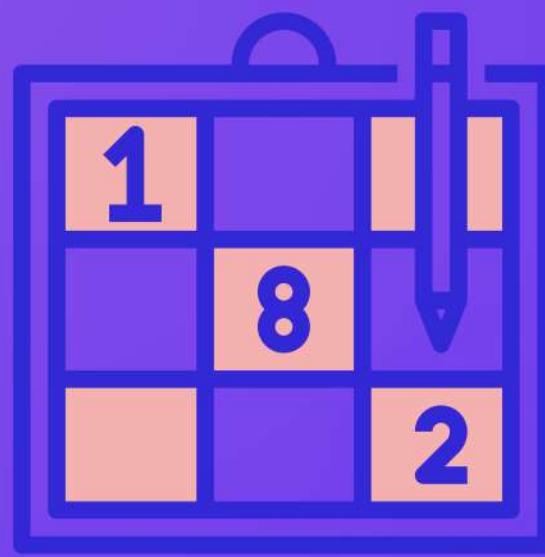
Si è osservato che l'esecuzione dell'algoritmo per tutte le tipologie di difficoltà delle griglie Sudoku termina per aver superato il numero massimo di generazioni (tempo di esecuzione: tra 4s e 5s).

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 <= score <= 61
- **medio:** 56 <= score <= 62
- **caso peggiore:** 57 <= score <= 71

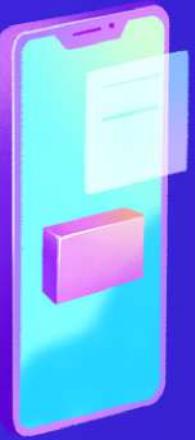
# CONCLUSIONI

- 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.
- 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.



GRAZIE PER  
L'ATTENZIONE!





# RIFERIMENTI BIBLIOGRAFICI

- 1.“There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration” by Gary McGuire, Bastian Tugemann and Gilles Civario (2013) pp. 5--7
- 2.“Artificial Intelligence: A Modern Approach - Volume 1 Fourth Edition” by Stuart Russell and Peter Norvig (2021), Pearson
- 3.“Sudoku”: <https://it.wikipedia.org/wiki/Sudoku>
- 4.“Simulated Annealing for beginners” by Lee Jacobson (2013):  
<https://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6>
- 5.“Creating a Genetic Algorithm for beginners” by Lee Jacobson (2012):  
<https://www.theprojectspot.com/tutorial-post/creating-a-genetic-algorithm-for-beginners/3>

# RIFERIMENTI BIBLIOGRAFICI



6. "Applying a genetic algorithm to the traveling salesman problem" by Lee Jacobson (2012):  
[https://www.theprojectspot.com/tutorial\\_post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5](https://www.theprojectspot.com/tutorial_post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5)
7. "Algoritmi Genetici con Esempio" by Marco Cianetti (2019) :  
<https://marcocianetti.com/articoli/algoritmi-genetici-con-esempio>

# SLIDES DI BACKUP: RISULTATI BACKTRACKING

(01)

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

(02)

Griglia	MNE	MNS	Tempo medio per trovare una soluzione
Sudoku_casoPeggior	37.652	103	0,0226
Sudoku_casoPeggior2	622.577.597	259	7,3376941

Tabella 2. Prestazioni del backtraing sulle griglie Sudoku\_casoPeggior e Sudoku\_casoPeggior2

# SLIDES DI BACKUP: RISULTATI A\*

(01)

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

(02)

Griglia	MNG	MNS	Tempo medio per trovare una soluzione
Sudoku_casoPeggior	187	93	0,0363
Sudoku_casoPeggior2	421.085	210.535	0,8613

Tabella 4. Prestazioni della ricerca A\* sulle due griglie Sudoku\_casoPeggior e Sudoku\_casoPeggior2 con livello di difficoltà *Caso peggiore*

# SLIDES DI BACKUP: COMPLESSITÀ G.A.

Esecuzione dell'algoritmo G.A. per risolvere  
una griglia Sudoku di difficoltà facile  
(Sudoku\_facile\_3\_g)

**Tempo impiegato: circa 4s**

**Best score: 51!**

Aumento taglia popolazione, probabilità di  
crossover e di mutazione?

Leggero miglioramento nel valore di fitness  
risultante

```
336     //ALGORITMO DI RICERCA GENETICO
337     public int[][] solveSudoku_GeneticAlgorithm(int sudo_m[][]){
338         List<Integer> hints = m.getOriginalEntriesList(sudo_m);
339         /*GA parametri*/
340         int POPULATION_SIZE = 200;
341         Population startingPopulation;
342         int MAX_GENERATIONS = 5000;
343         startingPopulation = new Population(POPULATION_SIZE, sudo_m);

PROBLEMS 19 OUTPUT DEBUG CONSOLE TERMINAL PORTS

Generation: 4976 Fittest: 53
Generation: 4977 Fittest: 53
Generation: 4978 Fittest: 54
Generation: 4979 Fittest: 53
Generation: 4980 Fittest: 57
Generation: 4981 Fittest: 54
Generation: 4982 Fittest: 51
Generation: 4983 Fittest: 53
Generation: 4984 Fittest: 51
Generation: 4985 Fittest: 51
Generation: 4986 Fittest: 53
Generation: 4987 Fittest: 53
Generation: 4988 Fittest: 54
Generation: 4989 Fittest: 51
Generation: 4990 Fittest: 52
Generation: 4991 Fittest: 52
Generation: 4992 Fittest: 54
Generation: 4993 Fittest: 51
Generation: 4994 Fittest: 54
Generation: 4995 Fittest: 54
Generation: 4996 Fittest: 56
Generation: 4997 Fittest: 56
Generation: 4998 Fittest: 53
Generation: 4999 Fittest: 53
Generation: 5000 Fittest: 51
Solution found!
Generation: 5000
Genes:
ga.Individual@1c34a505
Verifico il trasferimento alla matrice degli input dei dati
5 2 3 5 8 3 8 3 2
9 8 1 7 6 9 1 4 9
4 7 6 1 4 2 5 7 6
4 9 6 5 2 4 5 3 4
2 1 8 8 3 6 2 1 8
7 5 3 7 1 9 9 6 7
9 6 2 8 5 6 7 4 3
7 8 3 7 1 3 2 8 9
4 5 1 2 9 4 6 1 5
```

# SLIDES DI BACKUP: OPERATORI G.A.

SELEZIONE:

```
// Roulette Wheel selection
private static Individual rouletteWheelSelection(Population population) {
    FitnessCalc c = new FitnessCalc();
    int totalFitness = c.calculateTotalFitness(population);
    int randomValue = new Random().nextInt(totalFitness);
    int cumulativeFitness = 0;
    for (Individual individual : population.getIndividuals()) {
        cumulativeFitness += individual.getFitness();
        if (cumulativeFitness >= randomValue) {
            return individual;
        }
    }
    return population.getIndividual(population.size() - 1);
}

// Creazione del mating pool mediante Roulette Wheel selection
private static List<Individual> selectParents(Population population) {
    List<Individual> parents = new ArrayList<>();
    for (int i = 0; i < MATING_POOL; i++) {
        Individual selected = rouletteWheelSelection(population);
        parents.add(selected);
    }
    return parents;
}
```

# SLIDES DI BACKUP: OPERATORI G.A.

## CROSSOVER:

```
// Metodo che effettua il crossover
private static Individual[] crossover(Individual indiv1, Individual indiv2) {

    int[][] fatherGrid = indiv1.getSudo_m();
    int[][] motherGrid = indiv2.getSudo_m();

    Individual child1 = new Individual(fatherGrid);
    Individual child2 = new Individual(motherGrid);

    // Si prende un numero casuale di celle dal padre
    Set<Integer> fatherSubmatrices = new HashSet<>();
    for (int i = 0; i < GRID_NO; i++) {
        if (Math.random() <= CROSSOVER_RATE) {
            fatherSubmatrices.add(i);
        }
    }

    // Generazione del primo figlio
    for (int i = 0; i < GRID_NO; i++) {
        int[][] source = fatherSubmatrices.contains(i) ? motherGrid : fatherGrid;
        Individual.copySubgrid(i, source, child1.getSudo_m());
    }

    // Generazione del secondo figlio
    for (int i = 0; i < GRID_NO; i++) {
        int[][] source = fatherSubmatrices.contains(i) ? fatherGrid : motherGrid;
        Individual.copySubgrid(i, source, child2.getSudo_m());
    }

    return new Individual[] { child1, child2 };
}
```

## MUTAZIONE:

```
public static void mutate(Individual indiv, List<Integer> hints) {
    Random rand = new Random();
    if (Math.random() <= MUTATION_RATE) { // probabilità che indiv subisca una mutazione
        ManageMatrix m = new ManageMatrix();
        int x = rand.nextInt(bound:9);
        int y = rand.nextInt(bound:9);
        int start[] = m.identify_Subgrid_startRowCol(x, y);
        int start_row = start[0];
        int start_col = start[1];
        int[] temp = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

        int indiv_sudo_m[][] = indiv.getSudo_m();
        int counter = 0;
        // In un array temporaneo vengono copiati gli elementi della sottogriglia scelta
        for (int i = start_row; i < start_row + 3; i++) {
            for (int j = start_col; j < start_col + 3; j++) {
                if (hints.contains(i * 9 + j)) {
                    temp[counter] = 0;
                } else {
                    temp[counter] = indiv_sudo_m[i][j];
                }
                counter++;
            }
        }
        // Si effettua la permutazione delle posizioni delle celle
        int random_swaps = rand.nextInt(bound:9);
        counter = 0;
        int first_cell;
        int second_cell;
        while (random_swaps > 0) {
            for (int i = 0; i < 9; i++) {
                do {
                    first_cell = rand.nextInt(bound:9);
                    second_cell = rand.nextInt(bound:9);
                } while (temp[first_cell] == 0 || temp[second_cell] == 0);
                int temp_cell = temp[first_cell];
                temp[first_cell] = temp[second_cell];
                temp[second_cell] = temp_cell;
                random_swaps--;
            }
        }
        for (int i = start_row; i < start_row + 3; i++) {
            for (int j = start_col; j < start_col + 3; j++) {
                if (hints.contains(i * 9 + j))
                    ;
                else {
                    if (temp[counter] != 0) {
                        indiv_sudo_m[i][j] = temp[counter];
                    }
                }
                counter++;
            }
        }
        indiv.setSudo_m(indiv_sudo_m);
    }
}
```