snAI

SuperNatural AI

Tablut Challenge 23 - 24

Matteo Bostrenghi
Leonardo Gennaioli
Iacopo Sbalchiero
Lorenzo Severini

https://github.com/IacopoSb/snAI

Euristica: Campi

- **state**: lo stato corrente del gioco
- **camps**, **rhombus**, **winPosition**: questi array definiscono posizioni importanti specifiche sulla scacchiera come i campi, un modello romboidale e le posizioni di vittoria rispettivamente
- **NUMWHITE** e **NUMBLACK**: costanti che rappresentano il numero di pedine bianche e nere

```
private final int[][] camps = {
                    {0,3}, {0,4}, {0,5},
                           {1,4},

        {3,0},                                  {3,8},
        {4,0}, {4,1},       {4,4},       {4,7}, {4,8},
        {5,0},                                  {5,8},

                           {7,4},
                    {8,3}, {8,4}, {8,5},
};
```

```
private final int[][] winPosition = {
        {0,1}, {0,2},         {0,6}, {0,7},
{1,0},                                      {1,8},
{2,0},                                      {2,8},

{6,0},                                      {6,8},
{7,0},                                      {7,8},
        {8,1}, {8,2},         {8,6}, {8,7}
};
```

Metodi di Utilità

- **Metodi per individuare posizioni specifiche sulla scacchiera** come la posizione del re, la presenza del re sul trono, il conteggio delle pedine adiacenti, il movimento del re
- **Per controllare la presenza di pedine** o campi vicini a una determinata posizione.
- **Per controllare se una posizione è occupata** e determinare il numero di pedine necessarie per mangiare il re.
- **Per contare il numero di pedine nemiche** in un determinato quadrante, controllare se una pedina nemica può muoversi da un lato specifico
- **Per verificare se una pedina può essere catturata** verticalmente o orizzontalmente.
- **Per calcolare la distanza minima del re da una posizione di vittoria**, contare il numero di pedine bianche che possono essere catturate, calcolare il valore basato sul numero di pedine nere in configurazione romboidale.

White Heuristic - Variabili utilizzate

KING_DISTANCE → distanza del re dai bordi

KING_CAPTURE → numero di pedine nere che circondano il re

WHITE_ALIVE → numero di pedine bianche ancora in vita

BLACK_EATEN → numero di pedine nere catturate

WIN_PATH → numero di vie di vittoria per il re

```
// Weights for the evaluation function
private final int KING_DISTANCE = 6; // Distance of the king from the border
private final int KING_CAPTURE = 80; // Number of black pawns around the king
private final int WHITE_ALIVE = 100;  // Number of white pawns still alive
private final int BLACK_EATEN = 120;  // Number of black pawns already eaten
private final int WIN_PATH = 400;     // Number of winning paths for the king
```

White Heuristic - EvaluateState()

L'euristica calcola un valore totale (stateValue) sommando i contributi ponderati delle variabili appena descritte. Inoltre, ci sono controlli speciali per condizioni di vittoria o sconfitta:

State.Turn.WHITEWIN → il valore dello stato è impostato a Double.POSITIVE_INFINITY per indicare una vittoria sicura.

State.Turn.BLACKWIN → il valore dello stato è impostato a Double.NEGATIVE_INFINITY per indicare una sconfitta.

Percorsi di fuga del re è 2 → il valore dello stato è impostato a Double.POSITIVE_INFINITY, suggerendo una vittoria imminente.

White Heuristic - EvaluateState()

```
double stateValue =
        KING_DISTANCE      * (6 - kingDistanceFromWin()) / 6                                          +
        KING_CAPTURE       * (4 - checkAdjacentPawns(state, kingPosition(state), State.Turn.BLACK.toString()))/4   +
        WHITE_ALIVE        * state.getNumberOf(State.Pawn.WHITE) / NUMWHITE                           +
        BLACK_EATEN        * (NUMBLACK - state.getNumberOf(State.Pawn.BLACK)) / NUMBLACK +
        WIN_PATH           * Arrays.stream(getKingEscapes(state, kingPosition(state))).sum();
```

Casi Speciali:

```
if (state.getTurn().equals(State.Turn.WHITEWIN)) {
    stateValue = Double.POSITIVE_INFINITY;
} else if (state.getTurn().equals(State.Turn.BLACKWIN)) {
    stateValue = Double.NEGATIVE_INFINITY;
}
if (Arrays.stream(getKingEscapes(state, kingPosition(state))).sum() == 2) {
    stateValue = Double.POSITIVE_INFINITY;
}
```

Black Heuristic - Variabili utilizzate

WHITE_EATEN → pedine bianche già catturate.

BLACK_ALIVE → pedine nere ancora vive.

BLACK_SUR_K → pedine nere che circondano il re.

RHOMBUS_POS → posizione nella formazione "rombo" per bloccare le fughe del re.

```
private final int WHITE_EATEN = 0; // White pawns already eaten
private final int BLACK_ALIVE = 1; // Black pawns still alive
private final int BLACK_SUR_K = 2; // Black pawns surrounding the king
private final int RHOMBUS_POS = 3; // Formation to prevent the king from escaping
private final int BLOCKED_ESC = 3; // Pawns that block king escapes
```

Black Heuristic - Fasi di gioco

EARLY GAME → quando il numero di pedine bianche è maggiore di 6

MID GAME → quando il numero di pedine bianche è maggiore di 4 e minore o uguale a 6

LATE GAME → quando il numero di pedine bianche è minore o uguale a 4

```
double stateValue = 0.0;
int gamePhase = 0;

int numbOfWhite = state.getNumberOf(State.Pawn.WHITE);
if (numbOfWhite > 4 && numbOfWhite <= 6)
    gamePhase = 1;
else if (numbOfWhite <= 4)
    gamePhase = 2;
```

Black Heuristic - Valore iniziale

PAWNS_AGGRESSION_WEIGHT → importanza dell'aggressività nel catturare le pedine bianche

```
private final double PAWNS_AGGRESSION_WEIGHT = 2.0;
```

possibleCatches → numero di pedine bianche che possono essere catturate

numbOfWhite → numero di pedine nere che impediscono al re di fuggire

```
double possibleCatches = getPossibleCatches();
if (getPossibleCatches() > 0)
    stateValue += (possibleCatches / numbOfWhite) * PAWNS_AGGRESSION_WEIGHT;
```

Black Heuristic - Early Game

Valutazione in base a:

- percentuale di pedine bianche mangiate
- percentuale di pedine nere vive
- percentuale di pedine nere che circondano il re
- percentuale di pedine nere nel rombo

```
earlyGameWeights = new Double[4];
earlyGameWeights[WHITE_EATEN] = 45.0;
earlyGameWeights[BLACK_ALIVE] = 35.0;
earlyGameWeights[BLACK_SUR_K] = 15.0;
earlyGameWeights[RHOMBUS_POS] = 5.0;
```

```
if (gamePhase == 0) { // Early Game

    stateValue += percWhiteEaten * earlyGameWeights[WHITE_EATEN];
    stateValue += percBlackAlive * earlyGameWeights[BLACK_ALIVE];
    stateValue += percSurroundKing * earlyGameWeights[BLACK_SUR_K];
    stateValue += pawnsInRhombus * earlyGameWeights[RHOMBUS_POS];
```

Black Heuristic - Mid Game

Valutazione in base a:

- percentuale di pedine bianche mangiate
- percentuale di pedine nere vive
- percentuale di pedine nere che circondano il re
- percentuale di pedine nere nel rombo
- numero di pedine nere che impediscono al re di fuggire

```
midGameWeights = new Double[4];
midGameWeights[WHITE_EATEN] = 42.5;
midGameWeights[BLACK_ALIVE] = 32.5;
midGameWeights[BLACK_SUR_K] = 20.0;
midGameWeights[RHOMBUS_POS] = 2.5;
midGameWeights[BLOCKED_ESC] = 2.5;
```

```
else if (gamePhase == 1) { // Mid Game

    stateValue += percWhiteEaten * midGameWeights[WHITE_EATEN];
    stateValue += percBlackAlive * midGameWeights[BLACK_ALIVE];
    stateValue += percSurroundKing * midGameWeights[BLACK_SUR_K];
    stateValue += pawnsInRhombus * midGameWeights[RHOMBUS_POS];
    stateValue += blockingPawns * midGameWeights[BLOCKED_ESC];
```

Black Heuristic - Late Game

Valutazione in base a:

- percentuale di pedine bianche mangiate
- percentuale di pedine nere vive
- percentuale di pedine nere che circondano il re
- numero di pedine nere che impediscono al re di fuggire

```
lateGameWeights = new Double[4];
lateGameWeights[WHITE_EATEN] = 40.0;
lateGameWeights[BLACK_ALIVE] = 30.0;
lateGameWeights[BLACK_SUR_K] = 25.0;
lateGameWeights[BLOCKED_ESC] = 5.0;
```

```
else { // Late Game

    stateValue += percWhiteEaten * lateGameWeights[WHITE_EATEN];
    stateValue += percBlackAlive * lateGameWeights[BLACK_ALIVE];
    stateValue += percSurroundKing * lateGameWeights[BLACK_SUR_K];
    stateValue += blockingPawns * lateGameWeights[BLOCKED_ESC];
```

Black Heuristic - Valore finale

Un’aggiunta al valore finale dello stato viene dato dal fatto se il re può essere catturato alla prossima mossa.

Aggiunge il valore 10 a stateValue → alla mossa viene attribuito un valore alto per poter superare in valore tutte le altre mosse non vincenti calcolate

```
stateValue += canCaptureKing();

return stateValue;
```