UNISACheckers

MODULO FIA

Alexandru M. Burlacu; Giovanni Sorrentino

Fondamenti di Intelligenza Artificiale 2021/2022

SOMMARIO

Introduzione

Il progetto prevede un sito che permetta all'utente di effettuare partite di Dama contro un'intelligenza artificiale.

L'utente sarà in grado di selezionare una difficoltà (Facile – Intermedio – Difficile) prima dell'inizio di una partita e, in base a tale scelta, l'agente agirà di conseguenza.

Abbiamo deciso di optare per un sito del genere in quanto analizzando le altre piattaforme le abbiamo trovate poco intuitive e distanti dall'utente che vi si approccia.

Descrizione dell'agente

OBBIETTIVI

L'idea proposta è quella di utilizzare un algoritmo di ricerca di tipo Minimax poiché la dama è un gioco a somma zero e quindi si presta bene ad algoritmi di questa tipologia.

L'agente dovrà essere in grado di offrire una diversa prestazione in base alla difficoltà scelta dall'utente e avere un tempo di risposta inferiore ad un secondo.

SPECIFICHE PEAS

PEAS	
Performance	La performance dell'agente è proporzionale alla bontà delle mosse scelte dallo stesso in funzione della posizione corrente della damiera e della difficoltà selezionata dall'utente.
Environment	L'ambiente in cui lavora l'agente è rappresentato dall'insieme di tutte le possibili configurazioni che la damiera può assumere nel corso della partita. • **Dinamico**: la damiera cambia configurazione ad ogni turno. • **Completamente osservabile**: l'agente ha accesso allo stato completo dell'ambiente in ogni momento. • **Deterministico**: le configurazioni della damiera sono determinate esclusivamente dalla configurazione corrente e dalle mosse eseguite • **Sequenziale**: l'agente calcola la migliore mossa in base alla totalità delle mosse scelte in precedenza dall'utente • **Discreto**: c'è un numero finito di mosse possibili e un numero finito di configurazioni della damiera. • **Multi-agente**: trattandosi di un problema di teoria dei giochi, l'agente interagisce con l'utente, che è considerabile come un secondo agente.
Actuators	Gli attuatori dell'agente consistono nella migliore mossa da attuare data una configurazione della damiera e il colore che ha il tratto.
Sensors	I sensori consistono in una matrice di caratteri che rappresenta la damiera

ANALISI DEL SISTEMA

Il gioco della dama consiste in una sfida tra due giocatori, ognuno dei quali utilizza un lato della damiera. Ogni lato è dotato di 12 pedine in grado di muoversi esclusivamente in diagonale in avanti di una casa e catturare una pedina avversaria "scavalcandola". Condizione necessaria e sufficiente affinché questo possa avvenire, è che la casa al di là della pedina avversaria (casa di destinazione della prima), sia vuota.

Quando una pedina raggiunge la traversa più lontana dal proprio lato, essa diventa una dama, e acquisisce la possibilità di muoversi anche all'indietro (esclusivamente in diagonale).

Esistono numerose versioni del gioco della dama giocate in tutto il mondo. Le seguenti sono le regole che il team ha ritenuto essere più comuni:

- Il gioco si svolge su una scacchiera 8x8, esclusivamente sulle case scure.
- Quando un giocatore si trova di fronte alla possibilità di catturare un pezzo, questa mossa non è obbligatoria.
- Le dame possono essere catturate sia da altre dame che da semplici pedine.
- Tutti i pezzi sono considerati "volanti", ovvero, nel caso della possibilità di una cattura multipla, essi sono in grado di fermarsi tra una cattura e l'altra e passare il turno.

Un giocatore vince nel caso in cui cattura tutti i pezzi avversari, oppure l'avversario non ha più mosse valide disponibili.

L'obiettivo dell'agente è quello di trovare la mossa migliore in una determinata configurazione della damiera.

Un primo possibile approccio a cui il team ha pensato consiste nell'utilizzare una rete neurale allenata attraverso apprendimento non supervisionato grazie ai GA. Il team ha preso spunto da AlphaZero. Esso è un motore scacchistico sviluppato dall'azienda DeepMind, il quale è stato in grado di sconfiggere StockFish (il più forte motore fino ad allora) allenandosi soltanto per quattro ore[1]. Questa opzione, tuttavia, è stata scartata per motivi di budget.

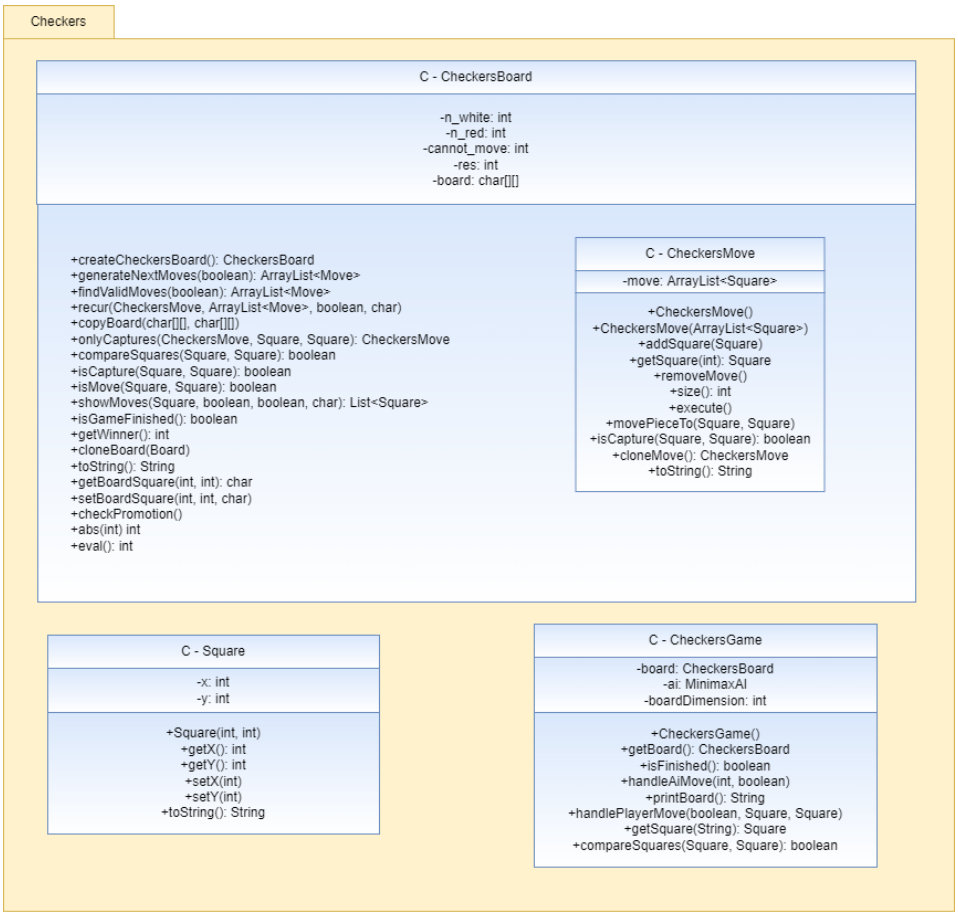
In seguito, si è giunti alla conclusione che il miglior approccio a questo problema fosse quello di implementare un algoritmo minimax.

Classi utilizzate nel progetto

Il progetto è suddiviso nei seguenti package

La classe MinimaxAI implementa l'algoritmo Minimax.

Board e Move sono interfacce utilizzate dalla Classe MinimaxAI per comunicare con le classe Board e Move.



La classe CheckersBoard gestisce la damiera rappresentandola come una matrice di caratteri in cui 'r' rappresenta una pedina rossa, 'w' una pedina bianca e le dame sono indicate con la corrispondente lettera maiuscola.

Essa, inoltre, implementa i metodi necessari alla classe MinimaxAI.

Infine, contiene la classe interna CheckersMove che implementa l'interfaccia Move e rappresenta una mossa attraverso un ArrayList di Square. La necessità di utilizzare una lista di più coordinate per ogni mossa è data dalla possibilità di eseguire catture multiple nello stesso turno.

La classe Square è un semplice "getter-setter" e rappresenta le coordinate di una casa della damiera.

La classe CheckersGame implementa parte della logica di gioco: come gestire il turno del giocatore o dell'AI e controllare se la mossa scelta è corretta.

Il progetto è organizzato in modo modulare; infatti, grazie all'utilizzo delle interfacce, è possibile utilizzare la classe MinimaxAI per altri giochi a turni, a condizione di implementare i metodi specificati nelle interfacce di Board e Move.

Implementazione dell'algoritmo Minimax

METODO FINDBESTMOVE

```java
public Move findBestMove(boolean isMaximizing, Board board, int depth, Board buboard, Board buboard2) {

    ArrayList<Move> listMoves = (ArrayList<Move>) board.generateNextMoves(isMaximizing);
    Move bestMove;
    ArrayList<Move> backTrackList = new ArrayList<>();
    ArrayList<Integer> evalList = new ArrayList<>();

    int bestScore, eval;
    if (isMaximizing)
        bestScore = -500;
    else
        bestScore = 500;
    buboard2.cloneBoard(board);
    for (Move move : listMoves) {
        board.cloneBoard(buboard2);
        move.execute();
        buboard.cloneBoard(board);
        eval = minimax(board, buboard, depth: depth - 1, alpha: -500, beta: 500, !isMaximizing, backTrackList);
        evalList.add(eval);
        backTrackList.clear();

        if (isMaximizing) {
            bestScore = Integer.max(bestScore, eval);
        } else {
            bestScore = Integer.min(bestScore, eval);
        }
    }
```

Il metodo findBestMove salva all'interno di un ArrayList di Move (listMoves) tutte le possibili mosse in una determinata situazione, in funzione anche del lato che ha il tratto. Ognuna di queste mosse, viene poi eseguita sulla damiera e su quest'ultima viene eseguita la funzione minimax (che verrà analizzata a breve), che restituisce una valutazione della mossa presa in esame. Questa valutazione verrà salvata in evalList, e quindi corrisponderà alla mossa analizzata. Prima di eseguire la successiva mossa di listMoves, la damiera viene ripristinata alla configurazione iniziale.

Dopo aver analizzato tutte le mosse, nella variabile bestScore verrà salvato il punteggio migliore tra le mosse analizzate.

Una possibile variante di questa implementazione consiste nel salvarsi di volta in volta la migliore mossa, e sovrascriverla nel caso di una mossa con valutazione ancora migliore. La scelta di questa implementazione è giustificata dalla seguente immagine, che conclude il metodo findBestMove.

```
//Salva le mosse con punteggio più alto
board.cloneBoard(buboard2);
ArrayList<Move> templist = new ArrayList<>();
for (int i = 0; i < listMoves.size(); i++) {
    if (evalList.get(i) == bestScore)
        templist.add(listMoves.get(i));
}

//Scegli a caso una mossa tra quelle con punteggio più alto
int k = ThreadLocalRandom.current().nextInt( origin: 0, templist.size());
bestMove = templist.get(k);
if (bestMove == null) {
    int i = ThreadLocalRandom.current().nextInt( origin: 0, listMoves.size());
    bestMove = listMoves.get(i);
}
System.out.println("Best move = " + bestMove);

return bestMove;
```

Dato un insieme di mosse analizzate da questo metodo, è possibile che ce ne sia più di una dotata della valutazione migliore. Memorizzando di volta in volta la migliore mossa, se l'utente decide di approcciare la partita con la stessa strategia, essa avrà sempre la stessa sequenza di mosse anche da parte dell'Intelligenza Artificiale.

Per evitare ciò, e dare all'utente un'esperienza più ricca ed eterogenea, si è deciso di far scegliere al sistema una mossa a caso tra quelle migliori.

In particolare, il sistema salva nell'ArrayList tutte le mosse con valutazione pari a bestScore, e tra di esse viene restituita una mossa casuale.

METODO MINIMAX

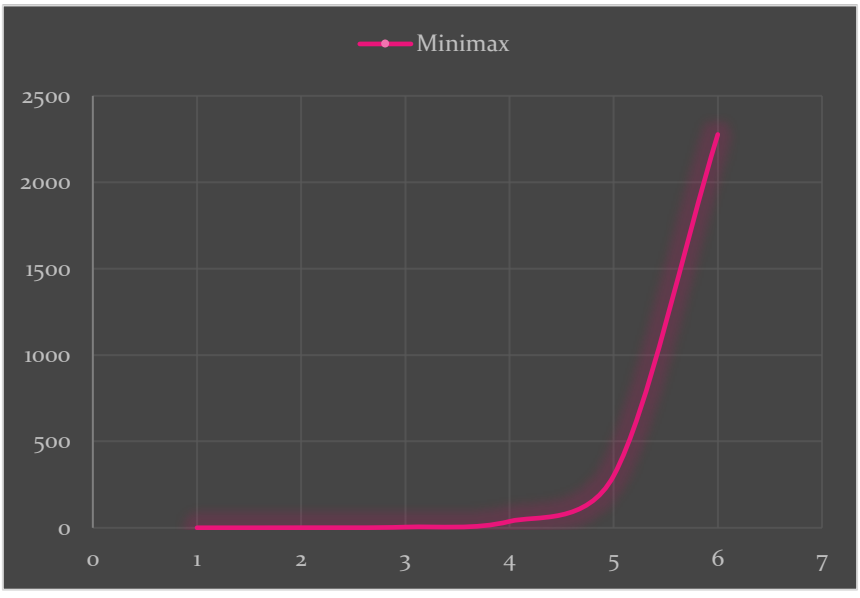
```java
private int minimax(Board board, Board buboard, int depth, int alpha,
                    int beta, boolean isMaximizing, ArrayList<Move> bTrack) {

    //Se la partita è teminata, restituisci il punteggio di fine
    //"Depth" serve a dare la priorità a mosse che terminano la partita più in fretta
    if (board.isGameFinished()) {
        if (board.getWinner() == 1) {
            return 300 + depth;
        }
        return -300 - depth;
    }

    //Se si raggiunge la massima profondità consentita, si restituisce eval della posizione
    if (depth == 0) {
        return board.eval();
    }

    ArrayList<Move> listMoves = (ArrayList<Move>) board.generateNextMoves(isMaximizing);
    //Se non ci sono più mosse disponibili, la partita è finita
    if (listMoves.size() == 0) {
        if (isMaximizing) {
            return -300 - depth;
        }
        return 300 + depth;
    }
```

Il metodo minimax è un metodo ricorsivo, e le seguenti sono le condizioni di terminazione. In particolare, la ricerca finisce nel caso in cui si sta analizzando una configurazione della damiera in cui uno dei giocatori ha vinto (prima e terza condizione if). In questo caso, la valutazione della posizione sarà di +300 o -300 (in valore assoluto, molto più grandi di qualsiasi eval della damiera) a seconda del lato che ha vinto. A questo valore, viene sommato (o sottratto) depth, per dare priorità alle posizioni di vittoria assicurata raggiungibili in meno mosse. Inoltre, la ricerca viene interrotta nel caso in cui si raggiunga la profondità massima. In questo caso, viene restituito il valore eval della damiera.

```java
if (isMaximizing) {
    int maxEval = -500, eval;

    for (Move move : listMoves) {
        bTrack.add(move);
        board.cloneBoard(buboard);
        for (Move move_ : bTrack) {
            move_.execute();
        }

        eval = minimax(board, buboard, depth: depth - 1, alpha, beta, isMaximizing: false, bTrack);
        bTrack.remove( index: bTrack.size() - 1);

        maxEval = Integer.max(maxEval, eval);
        //alpha = Integer.max(alpha, eval);
        //if (beta <= alpha)
            //break;
    }
    return maxEval;
} else {
    int minEval = 500, eval;
    for (Move move : listMoves) {
        bTrack.add(move);
        board.cloneBoard(buboard);
        for (Move move_ : bTrack) {
            move_.execute();
        }
        eval = minimax(board, buboard, depth: depth - 1, alpha, beta, isMaximizing: true, bTrack);
        bTrack.remove( index: bTrack.size() - 1);
        minEval = Integer.min(eval, minEval);
        //beta = Integer.min(beta, eval);
        //if (beta <= alpha)
            //break;
    }
    return minEval;
```

Il corpo del metodo consiste nel trovare le mosse possibili, eseguirle sulla configurazione corrente della damiera, e richiamare la funzione minimax, invertendo il lato da analizzare.

L'efficienza e i tempi di risposta (medi ed espressi in ms) in funzione della profondità utilizzata, sono rappresentati di seguito.



E' possibile osservare una complessità esponenziale, esattamente ciò che ci si aspetterebbe da un algoritmo di ricerca in profondità.

La media è stata calcolata sommando i tempi di risposta delle mosse eseguite nel corso di 10 partite (entrambi i lati sono gestiti dal sistema con data profondità) e dividendo il tutto per il doppio del numero totale di turni.

La massima difficoltà accessibile dall'utente corrisponde a una profondità pari a 5. Il tempo di risposta medio corrispondente è di circa 0,3 s. Essendo un valore medio esso varia molto a seconda della situazione. Può assumere valori molto grandi (anche > 7 s) soprattutto in situazioni in cui il valore eval cambia di poco ad ogni mossa.

E' possibile, tuttavia, applicare alcune ottimizzazioni.

OTTIMIZZAZIONE CON POTATURA ALPHA-BETA

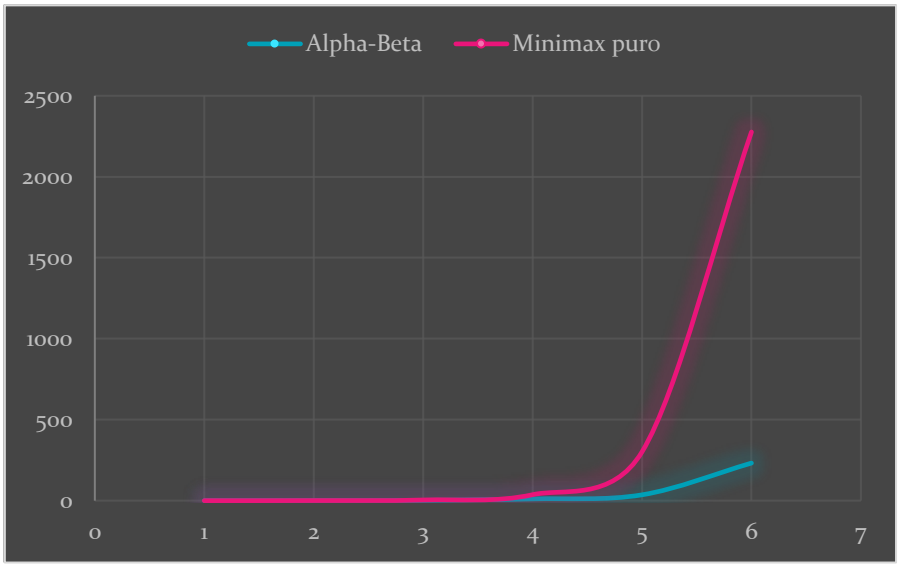
Decommentiamo le righe di codice viste in precedenza.

```
alpha = Integer.max(alpha, eval);
if (beta <= alpha)
    break;
```

```
beta = Integer.min(beta, eval);
if (beta <= alpha)
    break;
```

Questo codice permette di implementare la potatura alpha-beta, un particolare tipo di ottimizzazione per l'algoritmo Minimax che cerca di diminuire il numero di nodi analizzati nell'albero di ricerca. In particolare, ferma la valutazione di una mossa quando è stata trovata almeno una possibilità che prova che in precedenza è stata analizzata una mossa sicuramente migliore. Questo tipo di mosse non necessitano ulteriori analisi. Quando applicato a un albero minimax standard, restituisce la stessa mossa, ma pota i rami che non possono in alcun modo influire su questa decisione.

L'efficienza del sistema dopo aver implementato questa ottimizzazione è rappresentata dal seguente grafico.



E' possibile osservare un miglioramento di circa 89,8% (per quanto riguarda la profondità 6).

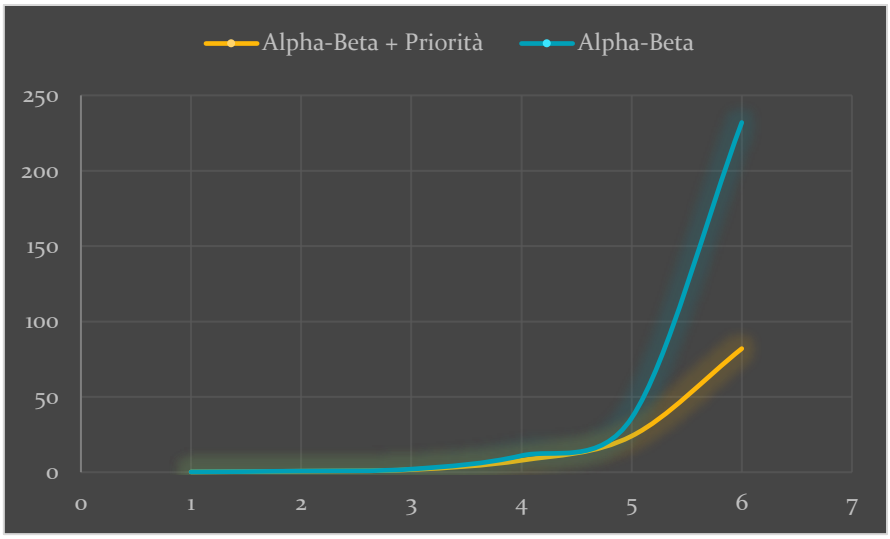
ULTERIORI MIGLIORAMENTI

Come già detto, la potatura taglia i rami dell'albero se in precedenza ha analizzato possibilità migliori. Di conseguenza, il grado di ottimizzazione di alpha-beta migliora quando si analizzano prima mosse migliori. Per fare ciò, nel metodo findValidMoves della classe CheckersBoard è stato implementato il seguente codice:

```
for (Move move : listMoves) {
    CheckersMove move_ = (CheckersMove) move;
    Square s1 = move_.getSquare( i: 0);
    Square s2 = move_.getSquare( i: 1);
    if (isCapture(s1, s2))
        finalMoves.add(move_);
}
for (Move move : listMoves) {
    CheckersMove move_ = (CheckersMove) move;
    Square s1 = move_.getSquare( i: 0);
    Square s2 = move_.getSquare( i: 1);
    if (isMove(s1, s2))
        finalMoves.add(move_);
}
```

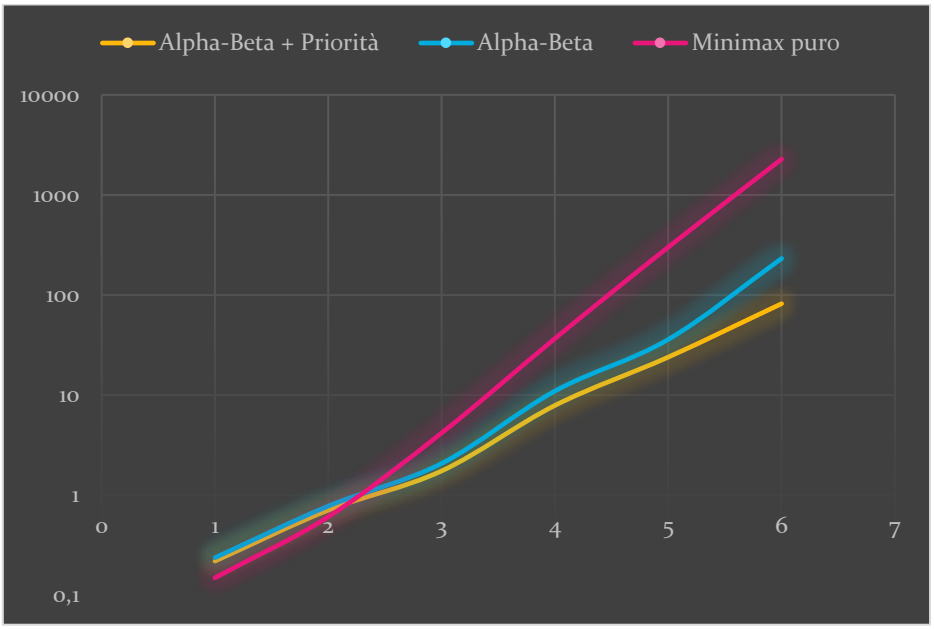
Sia dato l'ArrayList listMoves, che contiene le possibili mosse valide. Viene dichiarato l'ArrayList finalMoves in cui verranno salvate prima le mosse che consistono in una cattura (probabilmente le migliori), e poi quelle che consistono in un semplice spostamento.

Di seguito è possibile osservare l'efficienza del sistema rispetto alla potatura alpha-beta senza analizzare la priorità delle mosse.



Con queste impostazioni, il sistema ha dei tempi di risposta più rapidi del 64% rispetto alla potatura senza priorità e del 96,4% rispetto al Minimax puro.

Di seguito è possibile osservare i tre casi a confronto in scala logaritmica.



Integrazione con la piattaforma

La piattaforma è strutturata mediante modello MVC, di conseguenza la logica di gioco è implementata in parte nella servlet “handlePve” e, come già detto, in parte nella classe CheckersGame.

Di seguito è possibile analizzare la servlet in questione.

```
case "/handlePvE":
    cg = (CheckersGame) session.getAttribute( s: "game");
    session.setAttribute( s: "wrongmove",  o: false);
    String colore = (String) session.getAttribute( s: "col");
    boolean lato = colore.equals("Bianco");
    session.setAttribute( s: "lato", lato);
    int d = (int) session.getAttribute( s: "depth");
    String m1 = (String) session.getAttribute( s: "mossa1");
    String m2 = (String) session.getAttribute( s: "mossa2");
    Square sq1 = getSquare(m1);
    Square sq2 = getSquare(m2);
    if (cg.handlePlayerMove(lato, sq1, sq2)) {
        session.setAttribute( s: "wrongmove",  o: true);
        request.getRequestDispatcher( s: "/WEB-INF/views/site/PvEview.jsp").forward(request, response);
    } else if (cg.isFinished()) {
        session.setAttribute( s: "game", cg);
        request.getRequestDispatcher( s: "/WEB-INF/views/site/rematchPvE.jsp").forward(request, response);
    } else {
        cg.handleAiMove(d, !lato);
            ArrayList<Move> listMoves2 = cg.getBoard().generateNextMoves(lato);
            if (listMoves2.size() == 0) {
                if (!lato) cg.getBoard().cannot_move = -1;
                else cg.getBoard().cannot_move = 1;
            }
        if (cg.isFinished()) {
            session.setAttribute( s: "board", cg.getBoard());
            request.getRequestDispatcher( s: "/WEB-INF/views/site/rematchPvE.jsp").forward(request, response);
        } else {
            session.setAttribute( s: "turn", lato);
            session.setAttribute( s: "game", cg);
            session.setAttribute( s: "board", cg.getBoard());
            request.getRequestDispatcher( s: "/WEB-INF/views/site/PvEview.jsp").forward(request, response);
        }
    }
```

Caratteristiche degne di nota

EVAL

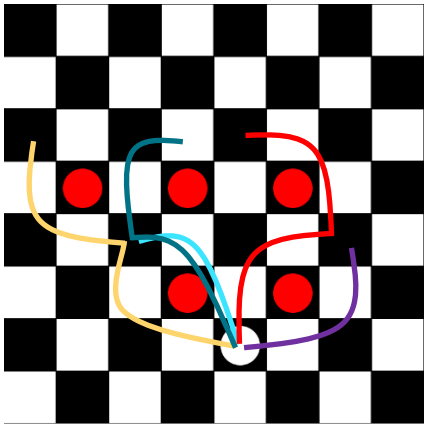
```java
public int eval() {
    int w = 0, r = 0;
    for (int i = 0; i < res; i++) {
        for (int j = 0; j < res; j++) {
            if (board[i][j] == 'w')
                w++;
            if (board[i][j] == 'W')
                w += 2;
            if (board[i][j] == 'r')
                r++;
            if (board[i][j] == 'R')
                r += 2;
        }
    }

    w *= 2;
    r *= 2;
    w += findValidMoves( isMax: true).size();
    r += findValidMoves( isMax: false).size();
    return w - r;
}
```

Una prima implementazione del metodo eval consisteva nel contare il numero di pedine bianche e quelle rosse e restituire la loro differenza. Tuttavia, è stato osservato che fornendo una euristica più mirata, il sistema è in grado di scegliere mosse più efficaci. In particolare, le pedine valgono un punto e le dame due punti. Queste quantità verranno raddoppiate e ad esse verrà aggiunto il numero di mosse disponibili al relativo giocatore. Il raddoppio serve a sottolineare che il numero di pedine in campo ha comunque un peso maggiore rispetto al numero di mosse. Il risultato della valutazione consiste nella differenza dei punteggi dei due lati.

FINDVALIDMOVES

Con tutta probabilità, la componente del progetto più difficile da risolvere è stata la stesura del metodo per trovare le possibili mosse in una determinata situazione. La principale difficoltà deriva dalla possibilità di eseguire catture multiple, ma comunque lasciare al giocatore l'opportunità di fermare la pedina tra due catture.

Di seguito è illustrato un esempio.



E’ possibile notare tutte le mosse possibili per il bianco, evidenziate da colori diversi. Com’è facilmente intuibile, si crea un vero e proprio albero delle mosse, e quindi è necessario il metodo ricorsivo mostrato di seguito:

```java
private void recur(CheckersMove singleList, ArrayList<Move> totalList, boolean player, char c) {
    Square currSquare = singleList.getSquare( i: singleList.size() - 1);
    CheckersMove validMoves = new CheckersMove();
    if (singleList.size() == 1) {
        validMoves = new CheckersMove((ArrayList<Square>) showMoves(currSquare, player, multiCatch: false, c));
    } else if (singleList.size() > 1) {
        Square prevSquare = singleList.getSquare( i: singleList.size() - 2);
        if (isCapture(prevSquare, currSquare)) {
            validMoves = new CheckersMove((ArrayList<Square>) showMoves(currSquare, player, multiCatch: true, c));
            validMoves = onlyCaptures(validMoves, currSquare, prevSquare);
        } else
            System.out.println("ERRORE!");
    }
    for (Square x : validMoves.move) {
        char[][] temp = new char[res][res];
        copyBoard(board, temp);
        if (isMove(currSquare, x)) {
            singleList.addSquare(x);
            totalList.add(singleList.cloneMove());
            singleList.removeMove();
        } else if (isCapture(currSquare, x)) {
            singleList.addSquare(x);
            CheckersMove lastMove = new CheckersMove();
            lastMove.addSquare(singleList.getSquare( i: singleList.size() - 2));
            lastMove.addSquare(singleList.getSquare( i: singleList.size() - 1));
            lastMove.execute();
            totalList.add(singleList.cloneMove());
            recur(singleList, totalList, player, c);
            singleList.removeMove();
        } else {
            System.out.println("ERRORE!");
        }
        copyBoard(temp, board);
```

Considerazioni finali

L'obiettivo del progetto è stato ampiamente raggiunto, sia per quanto riguarda la sua struttura e funzionamento che per i tempi di risposta che il team si era prefissato di ottenere. Durante il corso dello sviluppo, il team ha appreso nuovi concetti e sperimentato con nuovi tool, e ciò è stato possibile non solo grazie a un accesa motivazione di ogni individuo, ma anche grazie all'interesse nel migliorare il risultato e l'efficienza del sistema.

Note bibliografiche

[1] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis: "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play", Science, 2017.