Università degli Studi di Salerno

Progetto di Fondamenti di Intelligenza Artificiale

Rookie: IA per il gioco degli scacchi

7 settembre 2022

Autore	*Matricola*
Daniele Galloppo	0512106955

`https://github.com/DG266/Rookie`

Indice

1 Introduzione **3**

2 Definizione del problema **4**
2.1 Obiettivi 4
2.2 Specifica PEAS 4
2.2.1 Caratteristiche dell'ambiente 4
2.3 Analisi del problema 5

3 Soluzione del problema **6**
3.1 Tecnologie utilizzate 6
3.2 Rappresentazione della posizione 6
3.3 Funzione di valutazione euristica 7
3.4 Algoritmo di ricerca 10
3.4.1 Minimax "puro" 10
3.4.2 Primo miglioramento: potatura alfa-beta 12
3.4.3 Secondo miglioramento: ordinamento delle mosse 13
3.4.4 Terzo miglioramento: mosse killer 15
3.4.5 Quarto miglioramento: ricerca di quiescenza 17
3.5 Confronto tempi di esecuzione 19
3.5.1 Profondità di ricerca quattro 19
3.5.2 Profondità di ricerca sei 20

1 Introduzione

Il gioco degli scacchi è tanto antico quanto complesso: nato in India intorno al VI secolo d.C., giunse in Europa verso l'anno 1000 e si diffuse nell'intero continente, diventando, negli anni a venire, uno dei giochi più popolari al mondo. Oramai si può giocare a scacchi ovunque, anche al computer, contro un giocatore non umano: un'intelligenza artificiale.

Ma cosa c'è dietro un'intelligenza artificiale capace di giocare a scacchi? I calcolatori odierni, sempre più veloci, possono davvero gestire l'enorme complessità del gioco (si stima che l'albero delle mosse contenga 10^{123} stati)? O c'è qualche trucco?

Proveremo a dare una risposta a questa domanda sviluppando da zero un agente intelligente.

2 Definizione del problema

2.1 Obiettivi

Lo scopo del progetto consiste nella realizzazione di un'**applicazione desktop** che permetta all'utente di **giocare a scacchi contro un'intelligenza artificiale**.

L'applicazione deve consentire la **modifica** di diversi **parametri** che influenzano le performance dell'agente intelligente (algoritmo di ricerca utilizzato, profondità della ricerca, funzione di valutazione euristica utilizzata...), sia per creare livelli di difficoltà personalizzati, sia per scopi didattici ("giocare" con gli algoritmi può essere molto stimolante!).

2.2 Specifica PEAS

Di seguito è riportata la descrizione PEAS dell'ambiente operativo.

- **Performance.** La misura di prestazione adottata prevede la minimizzazione dei tempi per la ricerca di una mossa che sia valida e, al tempo stesso, la migliore possibile.
- **Environment.** L'ambiente in cui opera l'agente è costituito dall'insieme di tutte le possibili configurazioni che la scacchiera può assumere.
- **Actuators.** L'agente agisce sull'ambiente tramite la mossa migliore trovata.
- **Sensors.** L'agente percepisce l'ambiente tramite la scacchiera virtuale a cui sono associate tutte le informazioni della partita (giocatore attuale, mosse possibili...).

2.2.1 Caratteristiche dell'ambiente

L'ambiente operativo è:

- **Completamente osservabile.** I sensori dell'agente gli danno accesso allo stato completo dell'ambiente in ogni momento.
- **Multiagente competitivo.** Banalmente, nell'ambiente abbiamo due agenti.
- **Deterministico.** Lo stato successivo dell'ambiente è completamente determinato dallo stato corrente e dalla mossa eseguita dall'agente.
- **Sequenziale.** Ogni decisione può influenzare tutte quelle successive.
- **Statico.** L'ambiente non può cambiare mentre l'agente sta decidendo come agire.
- **Discreto.** L'ambiente (la scacchiera) ha un numero finito di stati distinti. Nel caso particolare degli scacchi, abbiamo anche un insieme discreto di percezioni e azioni.

Nota: in questo caso si è deciso di non penalizzare l'agente nel caso impieghi troppo tempo a trovare una mossa, perciò l'ambiente è statico (e non semidinamico).

2.3 Analisi del problema

Considerando che nel gioco degli scacchi abbiamo solamente due agenti, di cui uno (non necessariamente) umano, e che l'intelligenza artificiale cerca attivamente di sconfiggere l'avversario, si è deciso di optare per una soluzione basata su alberi di ricerca. Nel nostro caso, parleremo di albero di gioco, cioé di un albero di ricerca che segue ogni sequenza di mosse fino a raggiungere uno stato terminale.

L'agente preleva la mossa migliore dall'albero utilizzando l'algoritmo minimax; per ragioni che verranno esplorate più a fondo nelle prossime sezioni, l'algoritmo minimax "puro", così com'è, non è adatto per raggiungere gli obiettivi prefissati, e per questo motivo è stato necessario effettuare un buon numero di ottimizzazioni. Miglioreremo minimax in maniera graduale e ad ogni step cercheremo di capire perché la modifica effettuata ci porterà un vantaggio (o uno svantaggio).

Alla fine, metteremo a confronto le performance (i tempi di ricerca) di tutte le versioni dell'algoritmo di ricerca implementate.

3 Soluzione del problema

Per sviluppare un agente intelligente capace di giocare a scacchi "from scratch" è necessario implementare prima di tutto una scacchiera virtuale che i giocatori possono consultare: il giocatore umano la consulterà tramite una user interface - ciò significa che ci sarà bisogno di implementare una Graphical User Interface (GUI) - quello virtuale dovrà avere accesso a una serie di strutture dati che forniscono tutte le informazioni necessarie. Per informazioni necessarie si intende le posizioni dei pezzi sulla scacchiera, il turno corrente, le mosse legali e così via. Nelle sezioni successive verranno brevemente esplorati questi aspetti del progetto in maniera da lasciare spazio all'implementazione effettiva dell'agente intelligente.

3.1 Tecnologie utilizzate

Il linguaggio di programmazione scelto per la soluzione del problema è Java. Per la realizzazione della GUI è stata impiegata la libreria JavaFX.

3.2 Rappresentazione della posizione

La scacchiera virtuale è caratterizzata da un array di pezzi con dimensione 64 (si ricorda che la scacchiera del gioco degli scacchi possiede 8 x 8 caselle), da un riferimento alla mossa che ha generato quella particolare scacchiera (nel caso della scacchiera iniziale, appena inizia la partita, non ci sarà nessuna mossa generatrice) e da altre informazioni come il giocatore che deve effettuare la mossa e dati utili per consentire l'esecuzione di mosse speciali come l'arrocco o la presa al varco.

Ogni pezzo presente sulla scacchiera è caratterizzato da un tipo (pedone, cavallo, alfiere, torre, donna/regina, re), da un colore (bianco o nero), da una posizione e da un booleano che indica se il pezzo è stato mai mosso (anche questo necessario per l'esecuzione di mosse speciali). Inoltre, ogni pezzo contiene tutta la logica necessaria per calcolare le mosse legali su una particolare scacchiera.

Le mosse contengono un riferimento alla scacchiera su cui agire, un riferimento al pezzo da muovere e sorgente e destinazione della mossa. E' importante notare che le scacchiere, una volta create, non vengono mai più modificate: ciò significa che quando si esegue una mossa, la scacchiera originale resta intatta e viene creata una nuova scacchiera in cui è stata eseguita la particolare mossa. Un'alternativa è quella di agire sempre sulla stessa scacchiera conservando delle informazioni che permettano l'esecuzione e l'annullamento della mossa.

Riassumendo:

- la scacchiera è implementata utilizzando un array di oggetti che rappresentano i pezzi del gioco;
- data una particolare scacchiera, ogni pezzo può calcolare le mosse legali a disposizione in rapporto alla sua posizione;
- eseguire una mossa non va a modificare la scacchiera originale ma causa la creazione di un'ulteriore scacchiera.

3.3 Funzione di valutazione euristica

Normalmente, è impossibile prendere in considerazione l'intero albero di gioco, anche considerando tutte le ottimizzazioni applicabili all'algoritmo minimax. Di conseguenza, diventa necessario tagliare la ricerca a un certo punto e applicare una funzione di valutazione euristica che fornisce la stima dell'utilità di uno stato (ossia di una configurazione della scacchiera).

In sostanza, la funzione di valutazione da costruire dovrà essere:

1. quanto più possibile correlata con le effettive probabilità di vincere;
2. rapida da calcolare.

Per quanto riguarda il punto 1, è noto che gli scacchi non siano un gioco basato sul caso (non ci sono aspetti probabilistici) ma la necessità di tagliare la ricerca in stati non terminali implica incertezza sui risultati finali di quegli stati, per questo motivo è desiderabile una funzione che stimi l'utilità di uno stato nella maniera più "corretta" possibile. Tutto ciò si collega direttamente al punto 2: ovviamente una tale funzione dovrà anche essere veloce da calcolare, non deve peggiorare ulteriormente la complessità della ricerca.

La funzione di valutazione implementata considera varie caratteristiche di una configurazione della scacchiera:

- il valore del materiale;
- il numero di mosse legali effettuabili;
- il numero di mosse in cui un pezzo "attaccante" cattura un pezzo "vittima" che ha valore maggiore o uguale a quello dell'attaccante (esempio: pedone che cattura pedone o pedone che cattura regina, ma non regina che cattura pedone);
- se è stato effettuato un arrocco;
- se il re avversario si trova sotto scacco (o scacco matto);
- la struttura dei pezzi (esempio: un cavallo nel mezzo della scacchiera viene valutato positivamente rispetto a un cavallo che si trova ai bordi dove ha mobilità limitata.

Sono state considerate anche funzioni di valutazione con costo minore ma che considerano un numero minore di caratteristiche. Si è deciso di riportare nel documento il codice che implementa la funzione di valutazione più complessa, quella che considera tutte le caratteristiche sopra descritte.

```
public int evaluate(Board board) {
    return getScoreByPlayer(board.getWhitePlayer()) - getScoreByPlayer(board.getBlackPlayer());
  }

  private int getScoreByPlayer(Player player) {
    return player.getMaterialCount()
            + mobility(player)
            + availableGoodAttacks(player)
            + castlingEvaluation(player)
            + kingInCheckBonus(player)
            + generalStructure(player);
  }
```

Nella prossima pagina vedremo nel dettaglio l'implementazione delle singole funzioni.

Una nota: il valore del materiale di ogni giocatore non viene mai ricalcolato, esso viene aggiornato ogni volta che si effettua una mossa (nello specifico, viene aggiornato quando si crea la nuova scacchiera) e la funzione di valutazione non fa altro che consultarlo tramite il metodo getMaterialCount(). Ad esempio, se il giocatore bianco cattura un pezzo del giocatore nero, non andiamo a ricalcolare il valore del materiale da zero, basta sottrarre il valore del pezzo nero catturato al valore del materiale nero.

Ecco l'implementazione delle singole funzioni:

```
private int mobility(Player player) {     // Numero mosse legali effettuabili
  return player.getLegalMoves().size();
}

private int availableGoodAttacks(Player player) {     // Numero catture convenienti
  int attackBonus = 0;
  for (Move m : player.getLegalMoves()) {
    Piece attacker = m.getMovedPiece();
    Piece toAttack = player.getPlayingBoard().getPiece(m.getDestination().getValue());
    if (toAttack != null) {
      if (attacker.getType().getValue() <= toAttack.getType().getValue()) {
        attackBonus++;
      }
    }
  }
  return attackBonus;
}

private int castlingEvaluation(Player player) {   // Bonus arrocco
  Move m = player.getPlayingBoard().getGeneratorMove();
  if (m != null) {
    boolean isCastlingMove = (m instanceof CastlingMove);
    boolean playerMadeThisMove = m.getMovedPiece().getColor() == player.getPlayerColor();
    return (isCastlingMove && playerMadeThisMove) ? 500 : 0;
  }
  return 0;
}

private int kingInCheckBonus(Player player) {     // Bonus scacco / scacco matto
  Board board = player.getPlayingBoard();
  Player opponent = player.getOpponentPlayer();
  if (opponent.isKingInCheck()) {
    if (!(board.isCheckMateAvoidable(opponent))) {
      return 10000;
    }
    return 100;
  }
  return 0;
}

private int generalStructure(Player player) {   // Bonus struttura dei pezzi
  int structureScore = 0;
  int totalPieces = player.getPieces().size() + player.getOpponentPlayer().getPieces().size();

  if (player.getPlayerColor() == Color.WHITE) {
    for (Piece p : player.getPieces()) {
      int pos = p.getPosition().getValue();
      switch (p.getType()) {
        case PAWN: structureScore += WHITE_PAWN_STRUCTURE[pos];
        break;
        // ... altri pezzi...
        case KING: {
          int midGameValue = totalPieces * WHITE_KING_MIDDLE_GAME_STRUCTURE[pos];
          int endGameValue = (32 - totalPieces) * WHITE_KING_END_GAME_STRUCTURE[pos];
          structureScore += ((midGameValue + endGameValue) / 32);
        }
        break;
        default:
      }
    }
  } else {
    // ... stesso codice di sopra, ma per il giocatore nero ...
  }
  return structureScore;
}
```

Per valutare la struttura dei pezzi si usano tabelle simili:

```
private static final int[] WHITE_PAWN_STRUCTURE = {
    0,  0,  0,  0,  0,  0,  0,  0,
    50, 50, 50, 50, 50, 50, 50, 50,
    10, 10, 20, 30, 30, 20, 10, 10,
    5,  5, 10, 25, 25, 10,  5,  5,
    0,  0,  0, 20, 20,  0,  0,  0,
    5, -5, -10,  0,  0, -10, -5,  5,
    5, 10, 10, -20, -20, 10, 10,  5,
    0,  0,  0,  0,  0,  0,  0,  0
};
```

La valutazione della struttura dei pezzi così effettuata è molto banale, ma funziona: l'agente inizierà a spostare i pezzi in maniera più ragionata.

Qualcuno (come un giocatore esperto di scacchi!) potrebbe facilmente contestare questa strategia facendo notare che non è un dato di fatto che un particolare pezzo debba sempre trovarsi su una particolare casella. E non sbaglierebbe.

Si consideri il caso del re: nella parte conclusiva della partita (il "finale") il suo comportamento cambia radicalmente e le posizioni che consigliamo all'inizio della partita non vanno più bene (ovviamente questo discorso non vale solo per il re).

Per questo motivo è stato deciso di includere, per il re, due tabelle di valutazione: una per il pre-finale, una per il finale. Più pezzi ci sono sulla scacchiera, più il valore della posizione si avvicinerà alla prima tabella - meno pezzi ci sono sulla scacchiera, più il valore della posizione si avvicinerà alla seconda tabella. Questa tecnica viene chiamata "tapered evaluation" e consente di effettuare una transizione graduale tra le fasi del gioco.

Utilizzare il numero di pezzi per disinguere l'inizio partita dal finale è sicuramente una semplificazione eccessiva, ma è molto semplice da calcolare.

Ora possiamo analizzare l'algoritmo di ricerca.

3.4 Algoritmo di ricerca

In questa sezione analizzeremo nel dettaglio le diverse migliorie applicate all'algoritmo di ricerca minimax. Per i dettagli sui tempi di esecuzione si rimanda alla sezione "Confronto tempi di esecuzione".

3.4.1 Minimax "puro"

Di seguito si riporta l'implementazione dell'algoritmo minimax "puro" (non è propriamente puro in quanto viene tagliata la ricerca e viene utilizzata una funzione di valutazione euristica).

```
public class MiniMaxPlayer implements ArtificialIntelligencePlayer {
    private int depth;
    private int examinedBoards;
    private Evaluator evaluator;

    public MiniMaxPlayer(int depth, Evaluator evaluator) {
        this.depth = depth;
        this.evaluator = evaluator;
        this.examinedBoards = 0;
    }

    @Override
    public Transition play(Board startingBoard) {
        ScoredMove result;
        if (startingBoard.getCurrentPlayer().getPlayerColor() == Color.WHITE) {
          result = max(startingBoard, depth); // White starts as maximizing player
        } else {
          result = min(startingBoard, depth); //Black starts as minimizing player
        }
        Move bestMove = result.getMove();
        return new Transition(startingBoard, bestMove.makeMove(), bestMove);
    }

    private ScoredMove max(Board board, int depth) {
    //...
    }

    private ScoredMove min(Board board, int depth) {
    //...
    }
}
```

Il metodo play, prendendo in input una scacchiera, controlla il colore del giocatore che deve effettuare una mossa: se il colore è bianco, giocherà come MAX, altrimenti, se il colore è nero, giocherà come MIN. In questo modo è possibile far iniziare il gioco all'intelligenza artificiale (che sarà MAX) oppure farlo iniziare al giocatore umano (e quindi l'IA sarà MIN). Si ricordi che MAX al suo turno preferirà muoversi verso uno stato di valore massimo, mentre MIN al suo turno preferirà muoversi verso uno stato di valore minimo.

Vediamo l'implementazione del metodo max() :

```
private ScoredMove max(Board board, int depth) {
    if (depth == 0 || board.matchIsOver()) {
      this.examinedBoards++;
      return new ScoredMove(evaluator.evaluate(board), null);
    }

    int highestScore = Integer.MIN_VALUE;
    Move bestMove = new Move();

    for (Move move : board.getCurrentPlayer().getLegalMoves()) {
      Board transitionedBoard = move.makeMove();
      if (transitionedBoard.getOpponentPlayer().isKingInCheck()) {
        continue;
      }
      ScoredMove scoredMove = min(transitionedBoard, depth - 1);
      if (scoredMove.getScore() > highestScore) {
```

```
17          highestScore = scoredMove.getScore();
18          bestMove = move;
19        }
20      }
21      return new ScoredMove(highestScore, bestMove);
22    }
```

Prima di tutto, si effettuano i controlli necessari per tagliare la ricerca: se è stata raggiunta la profondità massima (scelta dall'utente tramite la GUI) oppure è stato raggiunto uno stato terminale, si taglia la ricerca e si restituisce il valore della funzione di valutazione relativo allo stato/scacchiera raggiunto. Proseguendo, se non abbiamo tagliato la ricerca, iniziamo ad analizzare tutte le mosse effettuabili dal giocatore MAX sulla scacchiera in input. Se una mossa genera una scacchiera in cui il giocatore MAX è sotto scacco, essa viene subito scartata (righe 12-13), non sarebbe una mossa legale per le regole degli scacchi. Se la mossa è legale, allora si procede ricorsivamente e si calcolano i valori minimax. Ovviamente MAX sceglierà (grazie al ciclo for) il più grande tra questi valori: è a questo che serve il successivo confronto (riga 16), dove viene salvata la mossa migliore trovata. Alla fine di tutto, viene restituita la mossa migliore trovata con il punteggio relativo (se positivo, indica un vantaggio per MAX, se negativo indica un vantaggio per MIN).

Di seguito viene riportata l'implementazione del metodo min():

```
1  private ScoredMove min(Board board, int depth) {
2      if (depth == 0 || board.matchIsOver()) {
3        this.examinedBoards++;
4        return new ScoredMove(evaluator.evaluate(board), null);
5      }
6
7      int lowestScore = Integer.MAX_VALUE;
8      Move bestMove = new Move();
9
10     for (Move move : board.getCurrentPlayer().getLegalMoves()) {
11       Board transitionedBoard = move.makeMove();
12       if (transitionedBoard.getOpponentPlayer().isKingInCheck()) {
13         continue;
14       }
15       ScoredMove scoredMove = max(transitionedBoard, depth - 1);
16       if (scoredMove.getScore() < lowestScore) {
17         lowestScore = scoredMove.getScore();
18         bestMove = move;
19       }
20     }
21     return new ScoredMove(lowestScore, bestMove);
22   }
```

L'implementazione è praticamente identica a quella del metodo max(), bisogna solamente fare in modo che venga scelto il valore minimax più basso (invece del più alto).

Purtroppo l'algoritmo così com'è non ha dato buoni risultati: come già anticipato, l'albero di ricerca del gioco degli scacchi ha dimensioni molto elevate e minimax non fa nulla per "risparmiare" sul numero di scacchiere analizzate. Ciò non consente di analizzare l'albero di gioco a profondità elevate, fattore molto importante per ottenere un buon giocatore.

3.4.2 Primo miglioramento: potatura alfa-beta

E' possibile migliorare l'algoritmo appena visto con una tecnica particolare: la potatura alfa-beta. Con questa tecnica è possibile ottenere un algoritmo che calcola le stesse mosse ottime di minimax ma ha un'efficienza migliore in quanto elimina sottoalberi che sono sicuramente irrilevanti.

Metodo max() modificato:

```
private ScoredMove max(Board board, int depth, int alpha, int beta) {
  if (depth == 0 || board.matchIsOver()) {
    this.examinedBoards++;
    return new ScoredMove(evaluator.evaluate(board), null);
  }

  int highestScore = Integer.MIN_VALUE;
  Move bestMove = new Move();

  for (Move move : board.getCurrentPlayer().getLegalMoves()) {
    Board transitionedBoard = move.makeMove();
    if (transitionedBoard.getOpponentPlayer().isKingInCheck()) {
      continue;
    }
    ScoredMove scoredMove = min(transitionedBoard, depth - 1, alpha, beta);
    if (scoredMove.getScore() > highestScore) {
      highestScore = scoredMove.getScore();
      bestMove = move;
      alpha = Math.max(alpha, highestScore);
    }
    if (highestScore >= beta) {
      return new ScoredMove(highestScore, bestMove);
    }
  }
  return new ScoredMove(highestScore, bestMove);
}
```

Metodo min() modificato:

```
private ScoredMove min(Board board, int depth, int alpha, int beta) {
  if (depth == 0 || board.matchIsOver()) {
    this.examinedBoards++;
    return new ScoredMove(evaluator.evaluate(board), null);
  }

  int lowestScore = Integer.MAX_VALUE;
  Move bestMove = new Move();

  for (Move move : board.getCurrentPlayer().getLegalMoves()) {
    Board transitionedBoard = move.makeMove();
    if (transitionedBoard.getOpponentPlayer().isKingInCheck()) {
      continue;
    }
    ScoredMove scoredMove = max(transitionedBoard, depth - 1, alpha, beta);
    if (scoredMove.getScore() < lowestScore) {
      lowestScore = scoredMove.getScore();
      bestMove = move;
      beta = Math.min(beta, lowestScore);
    }
    if (lowestScore <= alpha) {
      return new ScoredMove(lowestScore, bestMove);
    }
  }
  return new ScoredMove(lowestScore, bestMove);
}
```

La potatura alfa-beta prende il suo nome dai due parametri addizionali (vedere le firme dei metodi, sono stati aggiunti alpha e beta, due interi) che descrivono i limiti sui valori "portati su" in qualsiasi punto del cammino.

- alpha = il valore della scelta migliore (valore più alto) per MAX che abbiamo trovato sin qui in un qualsiasi punto di scelta lungo il cammino.
- beta = il valore della scelta migliore (valore più basso) per MIN che abbiamo trovato sin qui in qualsiasi punto di scelta lungo il cammino.

In sostanza, la ricerca alfa-beta aggiorna i valori di alpha e beta a mano a mano che procede e pota i rami restanti che escono da un nodo (fa terminare le chiamate ricorsive) non appena determina che il valore del nodo è peggiore di quello di alpha per MAX o, rispettivamente, di beta per MIN. La potatura avviene nelle righe 21-23.

L'applicazione di questa tecnica ha abbassato sensibilmente il numero di nodi da analizzare e, di conseguenza, ha permesso di aumentare la profondità di ricerca consentendo all'agente di giocare mosse migliori. Anche qui c'è un però: l'efficacia della potatura alfa-beta dipende fortemente dall'ordine in cui sono esaminate le mosse. Perché? Supponiamo, ad esempio, che MAX debba scegliere la mossa da eseguire: se MAX analizzasse per prima la mossa che porta al suo massimo guadagno, non ci sarebbe bisogno di analizzare tutte le restanti mosse, ossia, possiamo subito restituire la mossa trovata senza andare ad analizzare le altre (che poteremo). Purtroppo, come si può facilmente immaginare, non abbiamo nessuna garanzia sul fatto che la prima mossa analizzata sia sempre la migliore, anzi, spesso l'algoritmo analizzerà mosse sub-ottime (e quindi interi sottoalberi) prima di giungere alla mossa che causa il cut-off.

3.4.3 Secondo miglioramento: ordinamento delle mosse

Arrivati a questo punto, dovrebbe essere chiaro che l'ordine in cui si analizzano le mosse gioca un ruolo non da poco nella riduzione della complessità di minimax con potatura alfa-beta. Ma secondo quale criterio andrebbero ordinate le mosse? Un inizio potrebbe essere quello di ordinarle in maniera casuale:

```
1  // Recupera le mosse legali...
2  ArrayList<Move> legalMoves = board.getCurrentPlayer().getLegalMoves();
3  // ... e ordinale in maniera casuale
4  Collections.shuffle(legalMoves);
5
6  for (Move move : legalMoves) {
7    //Analisi mosse...
8  }
```

Sebbene molto semplice, è un approccio che grossomodo sembra funzionare bene, anche se poco affidabile. E' importante notare che questo ordinamento casuale viene fatto in ogni singola chiamata a max() e a min(): ciò significa che un pessimo ordinamento casuale "in alto", negli stati più vicini alla radice, potrebbe addirittura peggiorare le prestazioni rispetto a un algoritmo alfa-beta senza ordinamento. Si pensi a un pessimo ordinamento quando l'algoritmo parte dal livello 0 - ha un impatto sicuramente maggiore rispetto a quello di un ordinamento mediocre effettuato al penultimo livello dell'albero di ricerca.

Guardando l'altra faccia della medaglia, nel migliore dei casi, l'ordinamento casuale potrebbe portare all'ordine "perfetto", ciò a cui stiamo cercando di avvicinarci. Ma chi può garantirlo?

Una tecnica simile porterebbe alla realizzazione di un agente con tempi di risposta molto altalenanti, quindi è stato deciso di scartare quest'idea.

Sappiamo che non è possibile ottenere l'ordinamento perfetto delle mosse (altrimenti basterebbe usare la funzione di ordinamento per giocare una partita perfetta): e se ci affidassimo a qualche euristica?

L'euristica MVV-LVA (Most Valuable Victim - Least Valuable Aggressor) dà priorità alle mosse che catturano pezzi, con un dettaglio in più: catture in cui "l'aggressore" vale molto meno della "vittima" saranno più importanti di catture in cui accade il contrario.

Chiariamo con un esempio: la cattura di una regina da parte di un pedone avrà valore molto più alto della cattura di un pedone da parte di un cavallo.

L'implementazione di quest'idea è molto semplice:

```
1  // Recupera le mosse legali...
2  ArrayList<Move> legalMoves = board.getCurrentPlayer().getLegalMoves();
3  // ... e ordinale seguendo l'euristica MVV-LVA
4  Collections.sort(legalMoves, new MoveComparator());
5
6  for (Move move : legalMoves) {
7    //Analisi mosse...
8  }
```

Di seguito è riportata l'implementazione del MoveComparator():

```
1  public class MoveComparator implements Comparator<Move> {
2    private static final int[][] MVV_LVA = {
3          //  Aggressori
4          //  P   N   B   R   Q   K    // Vittime
5            { 6,  5,  4,  3,  2,  1},  // P
6            {12, 11, 10,  9,  8,  7},  // N
7            {18, 17, 16, 15, 14, 13},  // B
8            {24, 23, 22, 21, 20, 19},  // R
9            {30, 29, 28, 27, 26, 25},  // Q
10           { 0,  0,  0,  0,  0,  0},  // K
11   };
12
13   public int mvvlva(Move m) {
14     Piece victim = m.getBoard().getPiece(m.getDestination().getValue());
15     Piece aggressor = m.getMovedPiece();
16     if (victim != null) {
17       return MVV_LVA[victim.getType().getId()][aggressor.getType().getId()];
18     }
19     return 0;
20   }
21
22   @Override
23   public int compare(Move m1, Move m2) {
24     // Ordine decrescente
25     if (mvvlva(m1) > mvvlva(m2)) {
26       return -1;
27     } else if (mvvlva(m1) == mvvlva(m2)) {
28       return 0;
29     } else {
30       return 1;
31     }
32   }
33 }
```

La tabella (righe 2-11) dovrebbe rendere più chiara l'idea che c'è dietro MVV-LVA. Va notato che il re ha sempre valore MVV-LVA 0 in quanto è un pezzo che non può essere catturato.

Con queste modifiche, diminuisce ancora una volta il numero di stati analizzati, infatti controllare per prime queste mosse porta l'algoritmo a considerare stati "più interessanti", che a loro volta causano dei cut-off e, quindi, a diverse potature dell'albero di ricerca. E' sicuramente una strategia più stabile dell'ordinamento casuale, ma possiamo fare ancora di più.

3.4.4 Terzo miglioramento: mosse killer

Le mosse di cattura non sono le uniche mosse interessanti e non sono le uniche che portano alla potatura dell'albero di gioco. Ci sono mosse - apparentemente "innocue" - che non catturano nulla, ma che portano un grosso vantaggio al giocatore che le effettua: queste sono le mosse killer. Vengono chiamate così perché sono mosse che l'avversario non può ignorare e che "uccidono" molte delle mosse che potrebbe effettuare (potatura!). Anche qui l'idea è semplice: ogni volta che viene effettuato un alpha cut-off o un beta cut-off (una potatura alfa o una potatura beta) controlliamo se la mossa che l'ha causato è una cattura o meno, se non è una cattura, la salviamo come mossa killer e salviamo anche il livello dell'albero (ply o strato) in cui è stata considerata. Il ply può anche essere visto come la distanza dalla radice dell'albero di ricerca. Successivamente, mentre si analizzano altre porzioni dell'albero di ricerca, se ci si trova alla stessa distanza dalla radice e tra le mosse legali è presente la mossa killer, la si fa "salire" nell'ordinamento, in modo da provarla subito dopo le mosse di cattura (quelle valore MVV-LVA > 0). Vediamo l'implementazione.

Queste sono variabili che vanno aggiunte per tenere traccia delle mosse killer.

```
1  private Move[][] killerMoves;
2
3  private static final int MAX_DISTANCE_FROM_ROOT = 64;
4  private static final int KILLER_MOVES_SLOTS = 2;
```

In questo caso l'array bidimensionale avrà un numero di righe pari a MAX_DISTANCE_FROM_ROOT che rappresenta il numero massimo di livelli per i quali salveremo le mosse (è un numero parecchio elevato, nel nostro caso non verranno mai effettuate ricerche così profonde, basterebbero anche 10 righe) e un numero di colonne pari a KILLER_MOVES_SLOTS (per ogni livello salviamo 2 mosse killer).

Modificheremo max() così:

```
1  for (Move move : legalMoves) {
2    //...
3    if (highestScore >= beta) {
4      // Se la mossa non cattura nulla, salvala come mossa killer
5      if (!bestMove.isCaptureMove()) {
6        saveKillerMove(currentPly, bestMove);
7      }
8      return new ScoredMove(highestScore, bestMove);
9    }
10 }
```

e min() così:

```
1  for (Move move : legalMoves) {
2    //...
3    if (lowestScore <= alpha) {
4      // Se la mossa non cattura nulla, salvala come mossa killer
5      if (!bestMove.isCaptureMove()) {
6        saveKillerMove(currentPly, bestMove);
7      }
8      return new ScoredMove(highestScore, bestMove);
9    }
10 }
```

Questa è l'implementazione del metodo saveKillerMove():

```
1  private void saveKillerMove(int ply, Move candidateKiller) {
2    Move firstKiller = killerMoves[ply][0];
3    if (!(candidateKiller.equals(firstKiller))) {
4      for (int i = KILLER_MOVES_SLOTS - 2; i >= 0; i--) {
5        killerMoves[ply][i + 1] = killerMoves[ply][i];
6      }
7      killerMoves[ply][0] = candidateKiller;
8    }
9  }
```

Appena una mossa (che non è di cattura) causa un cut-off, si controlla se è già presente nell'array, se non lo è, si spostano in avanti tutti le mosse già salvate e si salva "in cima" la nuova mossa trovata. Ciò significa che se a quel ply abbiamo consumato tutti gli slot, una delle mosse verrà cancellata in modo da far spazio a quella nuova. Quanti slot vanno assegnati a ogni ply/livello? Nella pratica, i risultati migliori si ottengono con due slot; aumentare il numero di mosse killer salvate non implica necessariamente un abbassamento dei tempi di ricerca.

Manca l'ultimo tassello: come facciamo a posizionare queste mosse subito dopo le mosse con valore MVV-LVA > 0? Abbiamo bisogno di un nuovo Comparator():

```
public class KillerMovesComparator implements Comparator<Move> {
  private Move firstKillerMove;
  private Move secondKillerMove;

  public KillerMovesComparator(Move firstKillerMove, Move secondKillerMove) {
    super();
    this.firstKillerMove = firstKillerMove;
    this.secondKillerMove = secondKillerMove;
  }

  private static final int[][] MVV_LVA = {
        //  Aggressori
        //  P   N   B   R   Q   K    // Vittime
          {16, 15, 14, 13, 12, 11},  // P
          {22, 21, 20, 19, 18, 17},  // N
          {28, 27, 26, 25, 24, 23},  // B
          {34, 33, 32, 31, 30, 29},  // R
          {40, 39, 38, 37, 36, 35},  // Q
          { 0,  0,  0,  0,  0,  0},  // K
  };

  public int mvvlva(Move m) {
    Piece victim = m.getBoard().getPiece(m.getDestination().getValue());
    Piece aggressor = m.getMovedPiece();
    if (victim != null) {
      return MVV_LVA[victim.getType().getId()][aggressor.getType().getId()];
    }
    return 0;
  }

  public int score(Move m) {
    if (m.isCaptureMove()) {
      return mvvlva(m);
    } else if (m.equals(firstKillerMove)) {
      return 6;
    } else if (m.equals(secondKillerMove)) {
      return 4;
    } else {
      return 0;
    }
  }

  @Override
  public int compare(Move m1, Move m2) {
    // Ordine decrescente
    if (score(m1) > score(m2)) {
      return -1;
    } else if (score(m1) == score(m2)) {
      return 0;
    } else {
      return 1;
    }
  }
}
```

Così facendo otteniamo un nuovo ordinamento delle mosse: prima mosse di cattura, poi mosse killer, infine le

restanti. Confrontando questa versione dell'algoritmo con la ricerca alfa-beta senza ordinamento, è possibile notare miglioramenti impressionanti: otteniamo lo stesso risultato ma con un tempo di molto inferiore.

3.4.5 Quarto miglioramento: ricerca di quiescenza

A questo punto, abbiamo sensibilmente ridotto il numero di stati da esplorare ma c'è ancora un problema che non va trascurato. Se l'agente viene configurato in maniera tale da esplorare al massimo n livelli dell'albero di ricerca, esso andrà incontro a un grande numero di scacchiere/posizioni non quiescenti, cioè stati in cui vi sono mosse pendenti che causerebbero grandi variazioni di valore nella valutazione. Attualmente l'algoritmo non è per niente flessibile, appena raggiunge la profondità predefinita (o uno stato terminale) si ferma e valuta quella scacchiera. E se su quella scacchiera ci fosse un pedone avversario che al prossimo turno potrebbe catturare la regina dell'agente? L'agente dovrebbe approfondire la ricerca in modo da appurare che quello stato non gli è per niente favorevole. In generale, sarebbe desiderabile che l'agente applichi la funzione di valutazione euristica solamente a stati quiescenti.

Modifichiamo max():

```
ScoredMove scoredMove = min(transitionedBoard,
                            getQuiescenceDepth(transitionedBoard,depth),
                            alpha,
                            beta
                          );
```

Modifichiamo min():

```
ScoredMove scoredMove = max(transitionedBoard,
                            getQuiescenceDepth(transitionedBoard,depth),
                            alpha,
                            beta
                          );
```

Questo è il metodo getQuiescenceDepth():

```
private static final int MAX_QUIESCENCE_SEARCHES = 10000;

private int getQuiescenceDepth(Board transitionedBoard, int currentDepth) {
    if (currentDepth == 1 && this.numberOfQuiescenceSearches < MAX_QUIESCENCE_SEARCHES) {
      int notQuiescentCounter = 0;
      Move lastMove = transitionedBoard.getGeneratorMove();
      if (lastMove.isCaptureMove()) {
        notQuiescentCounter += 1;
      }

      Move lastLastMove = lastMove.getBoard().getGeneratorMove();
      if (lastLastMove.isCaptureMove()) {
        notQuiescentCounter += 1;
      }

      if (transitionedBoard.getCurrentPlayer().isKingInCheck()) {
        notQuiescentCounter += 1;
      }

      if (notQuiescentCounter >= 2) {
        this.numberOfQuiescenceSearches++;
        return currentDepth + 1;  // Approfondisci la ricerca
      }
    }
    return currentDepth - 1;  // Procedi normalmente
  }
```

Il metodo è semplice, se alla prossima chiamata di max o min la ricerca verrà tagliata (se currentDepth == 1 allora alla prossima chiamata diventerà 0 e scatterà il taglio), allora si controlla il livello di "non-quiescenza" della scacchiera. Viene controllato se il re del giocatore che deve muovere è sotto scacco, se l'ultima mossa è una cattura e se la penultima mossa è una cattura. Se almeno due di queste condizioni si verificano, allora si

approfondisce la ricerca, in modo da trovare uno stato quiescente a cui applicare la funzione di valutazione. MAX_QUIESCENCE_SEARCHES limita il numero di "ricerche approfondite" da effettuare.

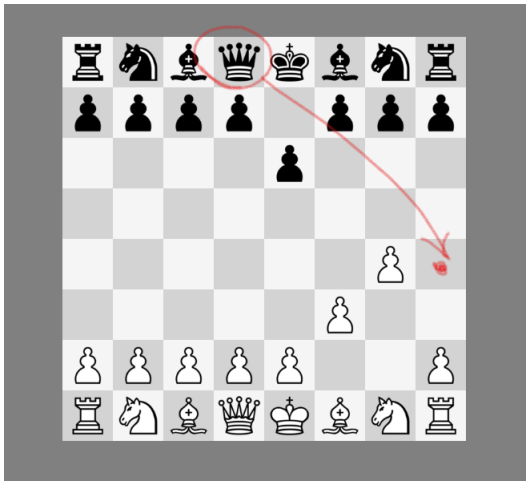
Anche qui non possiamo sapere con certezza quando uno stato non è quiescente, quindi si è deciso di effettuare questi semplici test ma, ovviamente, non è l'unico modo (e non è il modo migliore!).

A differenza dei precedenti miglioramenti, qui non è stato ridotto il numero di stati da visitare, anzi, è stato aumentato. Il vantaggio che si spera di ottenere è quello di una maggiore qualità delle mosse scelte dall'agente.

3.5 Confronto tempi di esecuzione

Dopo aver illustrato i miglioramenti applicabili all'algoritmo minimax, è giunto il momento di mettere a confronto i tempi di esecuzione.

Per confrontare i tempi è stata scelta la seguente scacchiera:



E' il turno del giocatore nero e la mossa migliore consiste nel muovere la regina dalla casella D8 alla casella H4, infatti ciò porterà alla fine della partita con una vittoria del giocatore nero (scacco matto). Curiosità: è lo scacco matto più veloce possibile nel gioco degli scacchi, il *matto dell'imbecille*.

L'applicazione verrà configurata in maniera tale che

- la funzione di valutazione sia quella descritta nelle sezioni sopra;
- l'agente intelligente impersoni il giocatore nero;

Vogliamo controllare se l'IA riesce a considerare questa mossa come la migliore e, soprattutto, quanto tempo impiega per trovarla. Tutti i test sono stati effettuati su una CPU Intel® Core™ i5-8300H.

3.5.1 Profondità di ricerca quattro

Per iniziare, è stato eseguito ogni algoritmo dieci volte sulla stessa scacchiera di partenza con profondità pari a quattro. Questi sono i tempi (in millisecondi) ottenuti:

Minimax	Alpha-Beta	Ord. casuale	Ord. MVV-LVA	Mosse killer	Quiescenza
2343	86	91	76	48	74
2336	84	63	78	44	70
2386	80	75	77	45	62
2326	84	69	77	47	58
2319	82	150	78	43	60
2327	83	89	76	47	62
2320	79	104	81	43	59
2327	84	47	79	50	60
2329	85	63	79	43	58
2368	79	99	76	45	60

Tabella 1: Tempi di ricerca con profondità 4.

Minimax	Alpha-Beta	Ord. casuale	Ord. MVV-LVA	Mosse killer	Quiescenza
2338,1	82,6	85	77,7	45,5	62,3

Tabella 2: Media aritmetica dei valori della Tabella 1

Minimax	Alpha-Beta	Ord. casuale	Ord. MVV-LVA	Mosse killer	Quiescenza
367562	6360	Numero variabile	5502	1411	2049

Tabella 3: Numero stati valutati

Nota bene: ogni algoritmo presenta tutte le migliorie che si trovano alla sua sinistra (quindi, ad esempio, per "Mosse killer" si intende l'algoritmo di ricerca minimax, con potatura alfa-beta, ordinamento delle mosse MVV-LVA e salvataggio delle mosse killer - "Quiescenza" avrà tutte queste migliorie con l'aggiunta della ricerca di quiescenza). Ovviamente, a partire dalla quarta colonna, l'ordinamento casuale delle mosse viene scartato e sostituito con l'euristica MVV-LVA.

In tutti i casi, l'agente ha deciso di spostare la regina dalla casella D8 alla casella H4. Dai risultati ottenuti, possiamo notare come la sola potatura alfa-beta migliora in maniera decisiva i tempi di ricerca: dalle quasi quattrocentomila scacchiere di minimax si passa a sole seimila scacchiere valutate. Proseguendo, è possibile constatare quanto siano "altalenanti" le prestazioni dell'ordinamento casuale delle mosse - a volte migliora i tempi della potatura alfa-beta pura, a volte li peggiora (anche di molto) - è proprio per questo motivo che questa tecnica è stata scartata. L'ordinamento con l'euristica MVV-LVA presenta lievi miglioramenti rispetto alle precedenti tecniche (bisogna comunque osservare che questa scacchiera non è la più adatta per mostrare i vantaggi dell'utilizzo di MVV-LVA). Aggiungendo le mosse killer riduciamo ancor di più i tempi, ottenendo quelli più bassi in assoluto. Infine, come ci aspettavamo, la ricerca di quiescenza (con al massimo 10000 ricerche di quiescenza, vedi sezione 3.4.5) tende ad allungare i tempi di ricerca (ma come specificato nelle sezioni precedenti, lo fa a fin di bene: vogliamo che la funzione di valutazione venga applicata a stati quiescenti!).

3.5.2 Profondità di ricerca sei

Adesso proviamo con profondità di ricerca uguale a sei:

Minimax	Alpha-Beta	Ord. casuale	Ord. MVV-LVA	Mosse killer	Quiescenza
/	9005	6536	6102	1208	3294
/	8662	7936	6098	1203	3167
/	8679	6739	6145	1209	3181
/	8730	6329	6179	1200	3171
/	8669	3722	6116	1209	3223
/	8731	7660	6098	1203	3197
/	8706	4755	6177	1210	3197
/	8699	8848	6159	1202	3214
/	8667	3998	6132	1202	3242
/	8702	3902	6121	1199	3197

Tabella 4: Tempi di ricerca con profondità 4.

Minimax	Alpha-Beta	Ord. casuale	Ord. MVV-LVA	Mosse killer	Quiescenza
/	8725	6042,5	6132,7	1204,5	3208,3

Tabella 5: Media aritmetica dei valori della Tabella 4

Minimax	Alpha-Beta	Ord. casuale	Ord. MVV-LVA	Mosse killer	Quiescenza
/	759118	Num. Variabile	434011	51070	114768

Tabella 6: Numero stati valutati

Anche qui l'agente ha sempre scelto la mossa migliore. Con profondità di ricerca uguale a sei, l'algoritmo minimax diventa particolarmente oneroso e si è deciso di non proseguire oltre con i suoi dieci test (l'agente impiega diverse ore per trovare una singola mossa!). Per gli altri casi, possiamo notare come viene confermato ciò che è stato già detto per la profondità quattro.

Potremmo proseguire con ricerche ancora più profonde ma il risultato non cambierebbe.

Chiaramente, i dati appena analizzati non ci possono dire molto sulle prestazioni "a tutto tondo" del nostro agente - è un caso troppo specifico. Arrivati a questo punto, l'IA andrebbe testata a fondo e per fare ciò, c'è bisogno di una grande quantità di tempo, di risorse di calcolo e di dati.

Considerando che l'obiettivo principale di questo progetto è quello di mostrare l'efficacia delle euristiche e delle ottimizzazioni che riguardano l'algoritmo di ricerca minimax, è stato deciso di non proseguire oltre per evitare di sfociare in questioni troppo vicine al "piccolo" mondo degli scacchi ma troppo lontane dall'ampio mondo dell'intelligenza artificiale.

Riferimenti bibliografici

[1] Russell SJ, Norvig P. Artificial Intelligence A Modern Approach. Third edition ed. Pearson Education; 2020.

[2] Mark Lefler. Chess Programming Wiki. https://www.chessprogramming.org