Universita degli Studi di Salerno

Progettazione e Implementazione di un Agente Intelligente per la Dama Italiana

Antonio Galdo
Matricola: 0512120238

Repository GitHub:

https://github.com/antgaldo/checkers

Gennaio 2026

1 Introduzione

Questo documento descrive la progettazione e l'implementazione di un agente intelligente capace di giocare a Dama con le regole italiane. Il progetto affronta la complessità dello spazio degli stati del gioco utilizzando l'algoritmo di ricerca Minimax ottimizzato con potatura alpha-beta e ordinamento delle mosse. Particolare attenzione è stata posta alla definizione della funzione di valutazione euristica, che integra concetti strategici avanzati quali la mobilità differenziale, la coesione strutturale ("testuggine") e la gestione del finale di partita. Il sistema è stato sviluppato in Java con interfaccia grafica JavaFX. L'obiettivo di questo progetto è sviluppare un'intelligenza artificiale (IA) in grado di competere contro giocatori umani (più o meno esperti), rispettando i vincoli di tempo reale.

2 Specifica dell'Ambiente (PEAS)

Performance Measure (Misura di Prestazione): Il criterio di successo è duplice: Vincere la partita (eliminando tutti i pezzi avversari o bloccandone le mosse) ed avere tempo di risposta per mossa ragionevole (massimo entro 3 secondi).

Environment (Ambiente): L'ambiente in cui opera l'agente è costituito dall'insieme di tutte le possibili configurazioni che la scacchiera può assumere.

Actuators (Attuatori): Il sistema agisce sull'ambiente attraverso: Spostamento delle pedine/-dame sulla scacchiera, rimozione dei pezzi avversari catturati, promozione delle pedine a Dama ed infine aggiornamento dell'interfaccia grafica (GUI).

Sensors (Sensori): L'agente percepisce lo stato attraverso la lettura della matrice 8x8 'Board', che fornisce la posizione di ogni pezzo, il tipo (Pedina/Dama) e il colore.

3 Caratteristiche dell'ambiente

L'ambiente operativo è:

- **Completamente osservabile:** L'agente ha accesso all'intera matrice di gioco.
- **Multiagente competitivo:** Nell'ambiente abbiamo due agenti.
- **Deterministico:** Lo stato successivo dell'ambiente è completamente determinato dallo stato corrente e dalla mossa eseguita dall'agente.
- **Sequenziale:** La decisione corrente influenza le mosse future.
- **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.

4 Sintesi del Regolamento

Per poter contestualizzare meglio il progetto ecco un breve riassunto delle regole della dama italiana (Regole ufficiali della Federazione Italiana Dama):

- **Composizione:** Ogni giocatore possiede 12 pedine di colore nero/bianco.
- **Movimento:** Le pedine si muovono solo in avanti di una casella sulle diagonali chiare. Le Dame possono muoversi anche all'indietro.
- **Obbligo di Presa:** Se un giocatore può catturare un pezzo avversario, è obbligato a farlo.

- **Gerarchia dei Pezzi:** La pedina non può mai catturare una Dama.
- **Priorità di Cattura:** In presenza di più opzioni di presa, il giocatore deve scegliere quella che permette di catturare il maggior numero di pezzi. A parità di numero, la Dama ha la precedenza sulla pedina come pezzo catturante.

5 Analisi del Problema

Il gioco della Dama Italiana rappresenta un classico problema di ricerca in uno spazio degli stati discreto, deterministico e osservabile. Si tratta di un gioco a informazione perfetta per due agenti a somma zero, in cui la vittoria di un giocatore corrisponde necessariamente alla sconfitta dell'altro. La sfida principale è stata nella gestione dei vincoli regolamentari, specificamente l'obbligo di cattura (con eventuale obbligo di cattura multipla). A differenza degli scacchi, nella Dama Italiana la libertà di movimento è fortemente limitata dalle regole di cattura ed inoltre le pedine hanno una direzione e un movimento limitato. Questo aspetto riduce significativamente lo spazio delle mosse possibili, con un impatto rilevante sui tempi di risposta dell'algoritmo. Come mostrato nei risultati sperimentali (indicati nella sezione Permormance), tale aspetto ha ridotto l'efficacia dell'ordinamento delle mosse ,per profondità di ricerca modeste, in quanto la scelta era già limitata dalle regole del gioco e quindi l'ottimizzazione del processo di ricerca si è raggiunta già con la potatura alpha-beta.

6 Architettura della Soluzione

Per l'implementazione dell'intelligenza artificiale, si è optato per un approccio basato su alberi di ricerca. Il processo decisionale dell'agente durante il proprio turno è stato ingegnerizzato secondo il seguente flusso logico:

1. **Generazione e Filtraggio delle Azioni:** Durante il turno dell'ai il sistema scansiona la scacchiera e vede ogni possibile mossa per ogni pedina del proprio colore (bianco). In questa fase, vengono scelte solo le mosse che il regolamento consente: se vengono individuate mosse di cattura l'algoritmo considererà solo quelle altrimenti considererà le mosse semplici. Le mosse possibili vengono inserite nella lista delle mosse legali. Durante questa fase ogni movimento simulato dell'ai viene fatto su una copia della scacchiera (al fine di non modificare lo stato dell'oggetto Board originale) ed inoltre per rendere più efficiente il funzionamento ho sviluppato due funzioni che rendono possibile la generazione della mossa e la cancellazione della mossa senza dover istanziare nuove scacchiere ad ogni livello di profondità.
2. **Ordinamento Euristico (Move Ordering):** La lista delle mosse legali viene processata da una funzione di ordinamento preliminare attraverso una funzione euristica che ha la funzione di guidare la ricerca. Le mosse giudicate "promettenti" (come le catture multiple o le promozioni a Dama) vengono posizionate in testa alla lista.
3. **Esplorazione con potatura Alpha-Beta:** Viene avviato l'algoritmo di ricerca Minimax. Attraverso l'uso della potatura Alpha-Beta, l'agente esplora l'albero di gioco ignorando i rami che, matematicamente, non possono influenzare la scelta finale.
4. **Decisione Ottimale:** Al termine della ricerca, l'agente seleziona l'azione che ha restituito il valore di utilità più elevato.

7 Algoritmi di ricerca

In questa sezione analizzeremo nel dettaglio l'algoritmo di funzionamento di base utilizzato (Min-Max) e le diverse migliorie applicate all'algoritmo per renderlo più efficiente. Per ragioni di

chiarezza viene riportato esclusivamente il codice completo della funzione. Per i dettagli sui tempi di esecuzione si rimanda alla sezione "Confronto tempi di esecuzione".

Algoritmo di base

L'algoritmo di base utilizzato è l'algoritmo di ricerca Minimax che opera costruendo un albero in cui ogni nodo rappresenta uno stato della scacchiera. Il processo si articola in tre fasi principali:

- **Esplorazione Ricorsiva:** L'algoritmo simula le mosse possibili scendendo in profondità nell'albero fino a raggiungere il limite di profondità impostato durante la chiamata alla funzione.
- **Valutazione:** Ai nodi foglia viene assegnato un valore numerico tramite una funzione di utilità (evaluator). Per ragioni strutturali ho ritenuto necessario creare una classe separata (MoveEvaluation) in cui sono inseriti tutti i metodi creati (tra cui evaluator) per valutare lo stato attuale della scacchiera.
- **Backtracking:** I valori risalgono l'albero. Nei livelli "MAX", l'agente sceglie il valore massimo tra i figli; nei livelli "MIN", l'avversario sceglie il valore minimo.

Questo meccanismo permette di determinare la mossa che garantisce il miglior risultato possibile, ipotizzando che l'avversario risponda sempre con la sua mossa migliore.

Ottimizzazione: Potatura Alpha-Beta

Per migliorare l'efficienza computazionale dell'algoritmo Minimax, è stata implementata la tecnica della **potatura Alpha-Beta**. Questa ottimizzazione permette di ignorare interi rami dell'albero di gioco che non influenzerebbero comunque la decisione finale. L'efficacia dell'algoritmo si basa sul mantenimento di due parametri dinamici durante la ricorsione:

α **(Alpha):** Rappresenta la valutazione del miglior scenario (valore massimo) che il giocatore **MAX** è riuscito a garantirsi lungo il percorso corrente. Inizialmente è impostato a $-\infty$.

β **(Beta):** Rappresenta la valutazione del miglior scenario (valore minimo) che il giocatore **MIN** è riuscito a garantirsi. Inizialmente è impostato a $+\infty$.

La logica di interruzione (taglio) si articola in due casi:

Turno di MAX (Taglio Beta) Se durante l'aggiornamento di α si ottiene $\alpha \geq \beta$, l'esplorazione del ramo si interrompe. MAX ha trovato una mossa così favorevole che il giocatore MIN (ai livelli superiori) non permetterà mai di raggiungere questo stato, avendo già a disposizione alternative migliori per i propri interessi.

Turno di MIN (Taglio Alpha) Se durante l'aggiornamento di β si ottiene $\beta \leq \alpha$, si effettua il taglio. MIN ha individuato un'opzione così svantaggiosa per MAX che quest'ultimo eviterà certamente questo percorso, rendendo inutile l'analisi delle restanti mosse del sotto-albero.

Ordinamento delle mosse

Come ultima ottimizzazione, ho implementato una classe dedicata alla logica di ordinamento delle mosse basata su una funzione di supporto (heuristic). Trattandosi di un'ottimizzazione mirata a migliorare il potenziamento della potatura (e non di una funzione di valutazione finale), il sistema premia esclusivamente le mosse che determinano un vantaggio immediato e significativo nello

sviluppo del gioco. Come evidenziato nei risultati sperimentali, tale scelta non produce miglioramenti significativi (per profondità modeste) poiché il numero di mosse legali è spesso già ridotto al minimo dalle regole del gioco (come l'obbligo di cattura), rendendo di fatto superfluo l'ordinamento ed addirittura rende meno efficiente il funzionamento in quanto si aggiunge il costo computazionale dell'ordinamento. L'ordinamento è stato comunque lasciato nell'implementazione finale in quanto siamo in ambito accademico e quindi risulta comunque utile in un contesto accademico a fini sperimentali e comparativi.

```
public class OrderMove {

    public OrderMove(){}

//Ordiniamo le mosse in base all'euristica in modo tale da velocizzare l'ai
    public ArrayList<Move> sortMove(ArrayList<Move> moves){
        ArrayList<Move> ordermoves= new ArrayList<Move>(moves);
        ordermoves.sort((m1, m2) -> Integer.compare(heuristic(m2),
    heuristic(m1))
        );
        return ordermoves;
    }

//funzione per valutare la mossa
    private int heuristic(Move move){
        Piece piece= move.getPiece();
        int direction = piece.getColor()==WHITE ? 1:-1;
        int score=0;
//se diventa dama (solo per le pedine)
            if(!piece.getisKing()) {
                if (direction == 1 && move.getRow() == 7) score += 300;
                if (direction == -1 && move.getRow() == 0) score += 300;
            }
//Per ogni pezzo catturato aggiungi 100 punti
            score += move.getCapturedPieces().size() * 100;
        return score;
    }
}
```

7.1 Implementazione

Inizialmente viene verificato se è stata raggiunta la profondità massima di ricerca: in tal caso la funzione termina restituendo il valore calcolato dalla funzione di valutazione (*Evaluation*) che sarà discussa nella prossima sezione "Evaluation", applicata allo stato corrente della scacchiera.

```
// Serve a valutare la scelta
private int minimax(Board board, int depth, int alpha, int beta, int turn)
    {
// Se abbiamo raggiunto la profondità massima
    if (depth == 0) {
        return moveevaluator.evaluator(board);
    }
```

In seguito viene generata la lista di tutte le **mosse ammissibili** per il giocatore di turno, nel rispetto delle regole del gioco. Se tale lista risulta vuota, significa che il giocatore corrente non ha mosse disponibili e quindi la partita è terminata; in questo caso viene restituito un valore di utilità molto alto o molto basso, a seconda che la situazione rappresenti rispettivamente una vittoria o una sconfitta per il giocatore MAX.

```
// Genera lista delle mosse possibili in base al turno
    ArrayList<Move> moves = board.generateMoves(turn);

```

```
// Se non ci sono mosse, il giocatore ha perso
    if (moves.isEmpty()) {
        return (turn == 0) ? -1000000 : 1000000;
    }
```

Prima di esplorare ricorsivamente l'albero di gioco, le mosse generate vengono ordinate, come spiegato in precedenza. Questo ordinamento non influisce sul risultato finale dell'algoritmo, ma potrebbe rendere più efficiente la potatura Alpha–Beta, riducendo il numero di nodi esplorati (come vedremo nel report finale non sarà così per profondità modeste).

```
// Ordina le mosse per tagliare in maniera ̀piu efficiente
    ArrayList<Move> ordermove = objectorder.sortMove(moves);
```

Caso MAX: Viene inizializzata una variabile `maxEval` al valore minimo possibile. Per ciascuna mossa disponibile:

1. La mossa viene simulata sulla scacchiera.
2. Viene effettuata una chiamata ricorsiva all'algoritmo Minimax per il giocatore avversario (MIN), decrementando la profondità residua.
3. Dopo aver ottenuto il valore di utilità, la mossa viene annullata (*unmake*) per ripristinare lo stato precedente.
4. Il valore ottenuto viene confrontato con `maxEval`, che viene aggiornato mantenendo il massimo.
5. Si aggiorna il parametro α. Se durante l'esplorazione risulta che $\beta \leq \alpha$, l'analisi delle mosse rimanenti viene interrotta tramite **potatura**.

```
// Chiamate per MAX
    if (turn == 0) {
// Impostiamo una variabile a -infinito
        int maxEval = Integer.MIN_VALUE;
// Cicliamo ogni mossa ordinata in precedenza
        for (Move move : ordermove) {
// Eseguiamo la mossa sulla scacchiera
            board.makeMove(move);
// Chiamata ricorsiva per scendere in profondità
            int utility = minimax(board, depth - 1, alpha, beta, 1);
// Ripristino dello stato della scacchiera
            board.undoMove(move);

            maxEval = Math.max(maxEval, utility);
            alpha = Math.max(alpha, utility);

            if (beta <= alpha) break; // Potatura Beta
        }
        return maxEval;
    }
```

Caso MIN: Per il caso Min Viene inizializzata una variabile `minEval` al valore massimo possibile. Per ogni mossa:

1. Si procede alla simulazione e alla chiamata ricorsiva per il giocatore MAX.
2. Il valore minimo ottenuto viene mantenuto in `minEval`.
3. Si aggiorna il parametro β. Se $\beta \leq \alpha$, l'esplorazione del ramo corrente viene interrotta anticipatamente grazie alla potatura Alpha–Beta.

```
// Chiamate per MIN
    else {
        int minEval = Integer.MAX_VALUE;
        for (Move move : ordermove) {
            board.makeMove(move);
            int utility = minimax(board, depth - 1, alpha, beta, 0);
            board.undoMove(move);

            minEval = Math.min(minEval, utility);
            beta = Math.min(beta, utility);

            if (beta <= alpha) break; // Potatura Alpha
        }
        return minEval;
    }
}
```

Conclusione

Al termine dell'esplorazione, la funzione restituisce il valore di utilità migliore (`maxEval` o `minEval`) associato allo stato corrente, permettendo di risalire ricorsivamente l'albero di gioco e individuare la scelta ottimale.

8 Funzione di valutazione

Per valutare lo stato della scacchiera è stata sviluppata una classe apposita, denominata `MoveEvaluator`, contenente il metodo `evaluator`. Tale funzione viene richiamata dall'algoritmo MinMax per assegnare un punteggio euristico ai nodi terminali (foglia) della ricerca, ovvero gli stati situati alla profondità massima impostata. Tali valori vengono poi propagati verso l'alto lungo l'albero per determinare la mossa ottimale. Il calcolo si basa sull'assegnazione di pesi specifici a diverse configurazioni tattiche, quantificando il vantaggio o lo svantaggio della posizione corrente. Di seguito vengono analizzate le porzioni di codice relative al giocatore bianco (AI), fermo restando che la logica applicata al giocatore nero risulta speculare. Le principali metriche di valutazione sono:

- **Condizioni di vittoria:** Se uno dei due giocatori rimane privo di pedine, viene decretata la vittoria dell'avversario. A questa situazione è assegnato un valore estremamente elevato, affinché l'algoritmo le conferisca priorità assoluta, essendo lo stato che conclude la partita.

```
public int evaluator(Board board){
// Assegnazione di un valore elevato alla vittoria per garantire
    priorità assoluta
if (board.getWhite().isEmpty()) return -1000000; if (board.getBlack().
    isEmpty()) return 1000000;
```

- **Mobilità (Branching Factor):** Qualora l'AI disponga di un numero maggiore di mosse legali rispetto all'avversario, vengono aggiunti 5 punti per ogni opzione supplementare. Questa metrica è cruciale, poiché una riduzione della mobilità porta spesso a situazioni di blocco, prefigurando una sconfitta nelle fasi avanzate del gioco.

```
int whiteScore=0;
int blackScore=0;
//Se l'AI ha ùiu scelte del nero aggiungiamo 5 punti per ogni scelta
    in ùiu
whiteScore= 5 * (board.generateMoves(0).size() - board.generateMoves
    (1).size());
```

- **Penalità per blocco:** Se il numero di mosse disponibili per l'AI scende sotto la soglia critica di 3, viene applicato un malus. Ciò impedisce al sistema di evolvere verso stati in cui il controllo della scacchiera è minimo.

```
//Se l'AI ha poche mosse di scelta verrà penalizzata
if (board.generateMoves(0).size() < 3) whiteScore -= 150;
```

- **Valutazione dinamica della Dama:** Il valore della dama è variabile in base alla fase della partita. Nelle fasi finali (*End Game*), il pezzo assume un'importanza strategica maggiore, riflessa in un incremento del punteggio assegnato.

```
//Diamo un valore maggiore alla dama nelle fasi finali del gioco
int totalPieces = board.getWhite().size() + board.getBlack().size();
boolean isEndGame = totalPieces < 8;
int kingValue = isEndGame ? 500 : 350;
```

- **Valutazione del materiale e della posizione:** In questa parte di codice valutiamo ogni singola pedina
  - Alle dame viene assegnato il valore dinamico calcolato in precedenza, mentre alle pedine normali sono attribuiti 100 punti base, partendo dal presupposto che la superiorità numerica sia un fattore determinante per la vittoria.

```
for(Piece piece: board.getWhite()){
    if(piece.getisKing()) whiteScore += kingValue;
    if(!piece.getisKing()) whiteScore += 100;
```

  - Bonus Difensivo: Alle pedine che occupano la riga di fondo (difesa) vengono assegnati 70 punti, incentivando il mantenimento di tale posizione

```
    if(piece.getRow() == 0) whiteScore += 70;

```

  - Protezione e Coesione: Viene premiata la protezione tra pezzi vicini (30 punti se la pedina sta proteggendo una dama e 15 se protegge una pedina semplice). Questa logica favorisce formazioni compatte "a testuggine", scoraggiando l'avanzamento isolato di singoli pezzi.

```
//Se la pedina sta proteggendo una dama 30 punti altrimenti 15
    punti
    whiteScore += isProtectingSomeone(piece, board, 0);

```

  - Avanzamento e Promozione: Per incentivare il raggiungimento della riga di promozione a dama, vengono assegnati 10 punti per ogni riga avanzata verso il bordo opposto.

```
    whiteScore += getPromotionBonus(piece);

```

  - Controllo dei bordi e del centro: Le pedine situate sui bordi laterali ricevono un bonus di 15 punti per la loro stabilità difensiva. Viceversa, le pedine in posizione centrale ricevono 20 punti per premiare il controllo del cuore della scacchiera e la conseguente ampiezza di scelta.

```
    if (piece.getCol() == 0 || piece.getCol() == 7) { whiteScore
    += 15; }

```

```
// Valorizzazione del controllo delle caselle centrali
if (isCentral(piece)) { whiteScore += 20; }
```

- Nell'ultima riga di codice vediamo quale giocatore ha la situazione migliore

```
return whiteScore-blackScore;
```

9 Funzione per gestire le eventuali mosse multiple per l'AI

Nel gioco della dama, la cattura multipla è obbligatoria ogni volta che la posizione dei pezzi lo permette. Tecnicamente, il sistema controlla la possibilità di una cattura analizzando le diagonali:

- **Per le pedine:** il controllo avviene solo sulle due diagonali anteriori (in avanti).
- **Per le dame:** il controllo viene esteso a tutte e quattro le direzioni (avanti e indietro).

Questo primo pezzo di codice riguarda proprio questo e termina con il primo controllo di cattura

```
1
2       public ArrayList<Move> getMoveCapture(Piece piece, int turn) {
3           ArrayList coordinate = new ArrayList<Move>();
4           int[] dirRow = {2, -2};
5           int[] dirCol = {2, -2};
6           int startrow = piece.getRow();
7           int startcol = piece.getCol();
8           int landingRow;
9           int landingCol;
10          int enemyRow;
11          int enemyCol;
12          for (int dirrow : dirRow) {
13              landingRow = piece.getRow() + dirrow;
14              if (!isInside(landingRow)) continue;
15              enemyRow = piece.getRow() + dirrow / 2;
16              for (int dircol : dirCol) {
17                  landingCol = piece.getCol() + dircol;
18                  if (!isInside(landingCol)) continue;
19                  enemyCol = piece.getCol() + dircol / 2;
20                  if ((getPiece(landingRow, landingCol) != null)) continue;
21                  if ((getPiece(enemyRow, enemyCol) == null)) continue;
22                  if ((getPiece(enemyRow, enemyCol).getColor() == piece.
    getColor())) continue;
23                  if (canPieceCapture(piece, enemyRow, enemyCol, landingRow,
    landingCol, turn)) {
```

Poiché il numero di catture consecutive non è prevedibile a priori, la logica è stata implementata tramite una funzione ricorsiva. Il processo continua a esplorare le diagonali finché non vengono più rilevate condizioni per ulteriori catture.

Durante ogni passaggio della ricorsione, il sistema tiene traccia dei pezzi catturati e della sequenza di spostamenti effettuati dal pezzo attivo. Al termine del calcolo, la funzione restituisce una mossa "atomica": l'intera sequenza di catture multiple viene infatti trattata come un'unica operazione, fornendo come parametri finali le coordinate d'arrivo definitive e l'elenco completo di tutti i pezzi rimossi dalla scacchiera.

```
24  //Simulazione per verificare se ci sono altre possibili mangiate dopo la
        prima
25  //rimozione del nemico mangiato e sposto la pedina in avanti
26                      Piece enemy = getPiece(enemyRow, enemyCol);
27                      board[enemyRow][enemyCol] = null;
28                      piece.setRow(landingRow);
29                      piece.setCol(landingCol);
30  //Chiamata ricorsiva per vedere se ci sono altre possibili catture
31  //Controlliamo se da questa nuova posizione può mangiare ancora
32                      ArrayList<Move> nextMoves = getMoveCapture(piece, turn)
        ;
33
34  //Se la lista è vuota vuol dire che non può ̀piu mangiare
```

```
                    if (nextMoves.isEmpty()) {
// Aggiungo questa casella come destinazione finale.
                        Move finalMove = new Move(piece, true, landingRow,
    landingCol,startrow,startcol);
                        finalMove.addCapturedPiece(enemy);
                        coordinate.add(finalMove);
                    } else {
// BIVIO O CONTINUAZIONE: la ricorsione ha trovato la fine della strada
                        for (Move m : nextMoves) {
// Creiamo la mossa atomica che arriva alla destinazione finale
                            Move atomicMove = new Move(piece, true, m.
    getRow(), m.getCol(), startrow, startcol);
// Copiamo tutti i pezzi mangiati nei salti successivi futuri
                            atomicMove.getCapturedPieces().addAll(m.
    getCapturedPieces());
// Aggiungiamo il nemico mangiato in QUESTO salto specifico
                            atomicMove.addCapturedPiece(enemy);
                            coordinate.add(atomicMove);
                        }
                    }
//Ripristino della posizione iniziale della pedina e del nemico
                    piece.setRow(startrow);
                    piece.setCol(startcol);
                    board[enemyRow][enemyCol] = enemy;
                }
            }
        }
        return coordinate;
    }
```

10 Performance

Dopo aver illustrato l'algoritmo è opportuno illustrarne le performance. Per valutare l'efficienza degli algoritmi implementati, ho analizzato il tempo di esecuzione e il numero di nodi visitati. Quest'ultimo parametro è fondamentale per quantificare l'efficacia delle ottimizzazioni (Potatura e Ordinamento) nel ridurre lo spazio di ricerca esplorato.

I test sono stati effettuati eseguendo le stesse 5 mosse (ovviamente per ogni profondità in quanto l'AI in base alla profondità farà scelte diverse).

10.1 Primo confronto, profondità 6

Il primo confronto è stato eseguito a profondità 6. Dai dati è emerso che la potatura aiuta a ridurre il carico di circa il 60% mentre l'effetto dell'ordinamento è marginale per il numero dei nodi visitati. Per quanto riguarda il tempo l'ordinamento è addirittura peggiorativo in quanto si viene ad aggiungere il peso appunto dell'ordinamento. Le mosse che hanno pochi nodi visitati e il tempo di esecuzione è nettamente ridotto rispetto alle altre mosse sono mosse cattura dove quindi c'è uno stop obbligato (non ha senso andare oltre)

Table 1: Confronto delle prestazioni tra le varianti dell'algoritmo (Profondità 6)

	MinMax		MinMax + Potatura		MinMax + Pot. + Ord.	
Mosse	**Tempo**	**Nodi**	**Tempo**	**Nodi**	**Tempo**	**Nodi**
1	231ms	45.956	80ms	13.618	78ms	13.377
2	145ms	31.017	54ms	10.142	56ms	10.039
3	9ms	1.580	2ms	203	2ms	203
4	156ms	41.443	62ms	16.561	64ms	16.521
5	61ms	18.361	26ms	7.645	28ms	7.803
Media	**120,4ms**	**27.671**	**44,8ms**	**9.633**	**45,6ms**	**9.588**

10.2 Secondo confronto, profondità 10

Il secondo confronto è stato eseguito a profondità 10. In questo caso la potatura non è utile per l'ottimizzazione ma è proprio necessaria per il funzionamento poiché il carico computazionale eccede le capacità della CPU utilizzata per i test (AMD Ryzen 7 serie 7000). L'ordinamento in questo caso ha un effetto tangibile sia sui nodi visitati che sul tempo impiegato (tranne per la mossa 5, probabilmente l'euristica usata per l'ordinamento porta a scelte premature in questo caso) e comporta un risparmio di un ulteriore 10%-20% circa sia sul tempo che sui nodi visitati.

Table 2: Confronto delle prestazioni tra le varianti dell'algoritmo (Profondità 10)

	MinMax		MinMax + Potatura		MinMax + Pot. + Ord.	
Mosse	**Tempo**	**Nodi**	**Tempo**	**Nodi**	**Tempo**	**Nodi**
1	—	—	3.568ms	1.317.348	3.251ms	1.167.157
2	—	—	1.622ms	627.879	1.576ms	600.278
3	—	—	472ms	182.854	468ms	175.760
4	—	—	3.470ms	1.431.216	3.316ms	1.350.794
5	—	—	9.963ms	4.239.233	16.762ms	6.928.639
Media	**N.A. (Timeout)**		**3.819ms**	**1.559.706**	**5.074ms**	**2.044.525**

11 Conclusione

In questo lavoro è stata progettata e implementata un'AI per il gioco della Dama Italiana. L'obiettivo era realizzare un agente in grado di rispettare il regolamento del gioco e di prendere decisioni efficaci entro tempi compatibili con un utilizzo interattivo.

Per la scelta delle mosse è stato utilizzato l'algoritmo Minimax, affiancato dalla potatura Alpha-Beta, che si è rivelata fondamentale per ridurre il numero di stati esplorati e consentire ricerche a profondità maggiori. L'ordinamento delle mosse risulta invece generalmente poco efficace nelle fasi iniziali della ricerca, principalmente a causa delle regole della Dama Italiana, in particolare dell'obbligo di cattura, che riduce significativamente il numero di mosse legali disponibili. Solo a profondità di ricerca più elevate l'ordinamento mostra un contributo più evidente.

Un aspetto rilevante dell'implementazione riguarda la gestione della cattura multipla, affrontata tramite una procedura ricorsiva che individua tutte le sequenze di cattura obbligatorie e le tratta come un'unica mossa. Questa scelta consente di integrare correttamente tale vincolo all'interno dell'algoritmo Minimax, mantenendo la correttezza rispetto alle regole del gioco.

La funzione di valutazione euristica permette all'agente di stimare la qualità delle posizioni considerando diversi fattori strategici, come il materiale, la mobilità e la fase della partita, rendendo possibile un comportamento coerente anche con profondità di ricerca limitate.

In futuro, si potrebbe permettere all'utente di scegliere il livello di difficoltà dell'agente, modificando quanto approfondita è la ricerca, e di migliorare la funzione di valutazione aggiungendo nuovi fattori strategici che influenzano il punteggio finale. Queste modifiche potrebbero rendere le

decisioni dell'agente più efficaci, ma richiederebbero più partite di prova per capire bene vantaggi e limiti.

Nel complesso, il progetto ha raggiunto gli obiettivi prefissati e ha consentito di applicare in modo concreto i principali concetti dell'intelligenza artificiale studiati.