# Dama italiana

Giuseppe Miraglia

5 novembre 2012


UNIVERSITÀ DI PISA

# Indice

# 1 Introduzione

Lo scopo di questa relazione è quello di illustrare le metodologie, le scelte di realizzazione e l'implementazione di un programma che cerca di simulare il gioco della *dama italiana*.
Il progetto scelto è stato preferito rispetto ad altri per il suo stretto legame con gli studi sull'*intelligenza artificiale*, porzione dell'informatica che, a mio modesto parere, racchiude diversi aspetti interessanti e che di conseguenza risulta incoraggiante da molti punti di vista.

# 2 Regole del gioco

Le regole di base del gioco che sono state prese in considerazione durante la realizzazione del programma sono le seguenti:

- La damiera è composta da 64 caselle disposte in modo alternato in base al colore. Per convenzione si usano due tonalità differenti dello stesso colore, una chiara e l'altra scura. La casella in alto a sinistra è scura, si prosegue poi per alternanza;
- Ad ogni giocatore vengono assegnate inizialmente 12 *pedine*, tutte dello stesso colore (in genere bianco o nero). Tutte le pedine vengono collocate sulle caselle scure della damiera. Per convenzione quelle nere occupano le prime tre righe in cima, le bianche le ultime tre;
- Inizia il gioco sempre il giocatore avente le *pedine bianche*;
- Le pedine si possono muovere soltanto verso il lato opposto della damiera rispetto a quello dal quale sono partite, ed esclusivamente in diagonale. Ogni qualvolta una pedina raggiunge una delle caselle situate sull'ultima riga nemica della damiera (cioè l'ultima serie di caselle facente parti della difesa nemica) questa diventerà *dama* ed acquisirà la possibilità di muoversi anche dal lato opposto a quello originale;
- Ogni pedina **deve** obbligatoriamente *"prendere"* (o in gergo *"mangiare"*) quelle avversarie che si trovano in avanti, sulla casella diagonale accanto alla propria e che abbiano la casella successiva libera. Dopo aver mangiato la pedina avversaria, se si incontrano ulteriori pedine avversarie che ricadono nuovamente nel caso appena citato, si è costretti a mangiare ricorsivamente finché questo si ripresenterà. Una mossa che comprende più di una mangiata è detta **mangiata multipla**;

- Per ogni dama vale la regola precedente. In più, rispetto alle pedine, ogni dama ha la possibilità di mangiare a ritroso e di mangiare altre dame. Anche per queste, mangiare è obbligatorio;
- Se in un determinato istante della partita si hanno più scelte per mangiare, vanno rispettate obbligatoriamente nell'ordine le seguenti priorità:
    1. mangiare più pezzi possibili;
    2. a parità di numero di pezzi tra mangiate che coinvolgono dama e pedina, mangia la dama;
    3. la dama effettua la mossa che mangia il maggior numero di dame avversarie;
    4. a parità di tutte le precedenti regole, la scelta è libera.
- Vince il giocatore che mangia tutti i pezzi dell'avversario o che mette quest'ultimo nelle condizioni di non poter effettuare alcuna mossa.

# 3 Modulo di intelligenza artificiale

Prima di passare all'implementazione vera e propria del programma, diamo una descrizione del *motore interno* del programma, ovvero delle tecniche utilizzate per far decidere al calcolatore quale mossa effettuare e in quale istante. Possiamo sostanzialmente suddividere il lavoro svolto dal motore interno in 4 "passi":

1. costruzione dell'*albero di ricerca*;
2. applicazione della *funzione di valutazione* ai nodi foglia;
3. applicazione dell'algoritmo *MiniMax*;
4. effettiva esecuzione della migliore mossa trovata.

## 3.1 Generalità

Il programma possiede un interfaccia per avviare una nuova partita secondo due "differenti" modalità:

- umano VS computer;
- computer VS computer.

Nel primo caso, il giocatore umano possiede le pedine bianche, quindi il computer attenderà finché questo non effettuarà la prima mossa. Nel secondo caso, invece, all'avvio sarà direttamente il computer a fare la prima mossa e quindi a giocare sia dalla parte del bianco che dalla parte del nero. In definitiva, nel primo caso il programma si limita ad attendere la mossa dell'avversario ed ad invocare, subito dopo questa, il motore di calcolo. Nel secondo caso, invece, il motore di calcolo viene invocato alternativamente con periodicità "bianco-nero" fino al termine della partita.

## 3.2 Costruzione dell'albero di ricerca

Il motore di calcolo del programma viene invocato quando è il calcolatore a dover muovere. A questo proposito, viene costruito un albero di ricerca avente le seguenti caratteristiche:

- la radice dell'albero rappresenta la configurazione della damiera prima della mossa del computer, ed ha profondità pari a 0;
- a partire dalla radice, viene aggiunto all'albero un nodo figlio per ogni possibile mossa legale da parte del computer applicata alla situazione iniziale (profondità 1);
- per ogni nodo a profondità 1, viene aggiunto all'albero un nodo per ogni possibile mossa legale da parte dell'avversario del computer (umano o computer stesso);
- cosí via fino a giungere ad una configurazione di vittoria per uno dei due giocatori.

In sintesi, viene generato un albero nel quale tutti i nodi a profondità dispari rappresentano le possibili mosse del computer, mentre tutti i nodi a profondità pari rappresentano le possibili mosse effettuate dall'avversario del computer (il giocatore bianco). Ogni mossa è calcolata in base alla configurazione del nodo padre e tutti i nodi foglia rappresentano una possibile terminazione della partita.

Teoricamente la costruzione dell'albero non crea grossi problemi, ma in pratica bisogna tener conto della complessità esponenziale rispetto alla profondità dell'albero. Per andare incontro a questo inconveniente, in generale viene scelta a priori una *profondità massima*, in modo tale da far risultare interattivo il programma.

## 3.3 Funzioni di valutazione

Dopo aver costruito l'albero di ricerca, i suoi nodi foglia vengono valutati. Ogni nodo corrisponde ad una specifica configurazione della damiera che rispecchia un istante futuro della partita, quindi potrà essere favorevole, sfavorevole o stabile rispetto all'attuale configurazione del computer (radice dall'albero).

La stima della bontà di tali configurazioni è calcolata tramite una *funzione di valutazione*

$$f \colon \mathbb{A} \to \mathbb{Z} \tag{1}$$

dove A rappresenta una matrice di interi 8x8 opportunamente codificata per rappresentare lo stato delle celle della damiera come verrà spiegato in seguio. Tale funzione prende in input una configurazione della damiera e restituisce in output un intero che rappresenta la sintesi della bontà di tale configurazione. ciò che l'ouput rappresenta, dipende dal tipo di funzione che è stata realizzata. Le scelte prese in fase di realizzazione sono cruciali, in quanto la funzione rappresenta la strategia con la quale il calcolatore giocherà: possiamo far si che questo abbia un approccio difensivo, e cerchi di proteggere le zone più sensibili della damiera, oppure che giochi in maniera aggressiva, cercando di andare velocemente a dama e preferendo il mangiare piuttosto che la copertura. In questo progetto è stata implementata la funzione di valutazione *"valuteBalanced"*, la quale opera con un approccio abbastanza bilanciato tra aggressività e difesa. Sicuramente il requisito fondamentale comune a queste funzioni è quello di non dover essere "pesanti" da calcolare, poiché in tal caso si penalizzerebbe l'interattività del gioco (a quel punto sarebbe preferibile aumentare la profondità di ricerca dell'albero per avere risultati ancora più precisi). Vediamo ora più in dettaglio la nostra funzione.

### 3.3.1 La funzione "valuteBalanced"

Questa funzione di valutazione associa un peso alle posizioni della damiera e ai pezzi presenti su questa. Il **peso posizionale**, ad esempio dal punto di vista del giocatore nero (che si trova inizialmente nella parte superiore della damiera), cresce in maniera quadratica man mano che si va verso lo

schieramento difensivo del giocatore bianco (la parte inferiore della damiera) ed è quindi calcolato tramite la formula

$$positionalweight = rowindex^2 \tag{2}$$

Dualmente, nella valutazione del giocatore bianco si avrà

$$positionalweight = (7 - rowindex)^2 \tag{3}$$

Per quanto riguarda invece il **peso proprio** dei pezzi, sono state scelte due costanti intere proporzionali alla loro importanza, aventi valori 100 per le pedine e 200 per le dame. Inoltre è stato introdotto un accorgimento di natura difensiva che penalizza il movimento dei pezzi dalla linea difensiva, privilegiando però lo spostamento nelle posizioni "centrali". Soprattutto le dame risentono di questo privilegio in quanto, essendo meno vulnerabili, non avrebbe senso proteggerle lungo i bordi ma al contrario sono utilizzabili al meglio in situazioni di attacco.

Sintetizzando il tutto dal punto di vista algoritmico, la funzione di valutazione *"valuteBalanced"* restituisce in output un numero intero **"score"** ottenuto:

1. sommando il *peso posizionale* dei propri pezzi;
2. sommando il *peso proprio* dei propri pezzi;
3. sottraendo il *peso posizionale* dei pezzi dell'avversario;
4. sottraendo il *peso proprio* dei pezzi dell'avversario;
5. sottranedo un fattore $\Delta$ ogni qualvolta una propria dama si trova su uno dei quattro bordi della damiera;
6. sommando un fattore $\Delta$ ogni qualvolta una dama avversaria si trova su uno dei quattro bordi della damiera.

Formalmente, supponendo di eseguire la valutazione per il giocatore nero, si avrà:

$$score \mathrel{+}= \begin{cases} 100\ (\#PedineNere) - 100\ (\#PedineBianche) \\ 200\ (\#DameNere) - 200\ (\#DameBianche) \\ \Delta\ (\#DameNereAiBordi - \#DameBiancheAiBordi) \\ \displaystyle\sum_{\forall\ pezzo\ nero} i^2 \\ -\displaystyle\sum_{\forall\ pezzo\ bianco} (7-i)^2 \end{cases}$$

Se l'output ottenuto è vicino allo zero, vuol dire che la configurazione valutata non porta ad uno stato vantaggioso per nessuno dei due giocatori. Se invece è tanto maggiore di zero, allora risulterà favorevole per il giocatore nero; altrimenti sarà vantaggiosa per il bianco.

## 3.4 Algoritmo MiniMax

Una volta che la funzione di valutazione è stata applicata a tutti i nodi foglia dell'albero di ricerca, bisogna scegliere qual'è per il computer la migliore mossa da realizzare effettivamente. Questa scelta viene adoperata tramite l'algoritmo ***Mini-Max***, il quale segue la filosofia di *"minimizzare la massima perdita possibile"*, basandosi però sul presupposto che anche l'avversario giochi in maniera *ottimale*. Il funzionamento dell'algoritmo, eseguito dal punto di vista del computer (che in questo caso rappresenta il $MAX$), a partire dalle foglie e' il seguente:

1. per ogni sottoalbero relativo a mosse dell'avversario (MIN), si assegna al padre la **minima** valutazione dei nodi figli;

2. per ogni sottoalbero relativo a mosse del computer (MAX), si assegna al padre la **massima** valutazione dei nodi figli;

3. si procede ricorsivamente ripercorrendo l'albero a ritroso fino al livello avente profondità pari ad 1: da questo livello, poiché stiamo ragionando dal punto di vista di MAX, verrà scelta la configurazione che massimizza la nostra scelta, tenendo conto che anche l'avversario ha ragionato in maniera ottimale minimizzandola nei passi precedenti. Proprio quest'ultima configurazione scelta, poiché corrisponde ad una delle prime mosse fatte a partire dalla configurazione originale (rappresentata dalla radice), sarà l'effettiva mossa che il calcolatore eseguirà.

## 3.5 Esecuzione della migliore mossa trovata

Come accennato durante la discussione dell'algoritmo MiniMax, una volta che l'intero albero è stato ripercorso a ritroso riusciamo ad ottenere una stima della bontà di tutte le mosse possibili, e quindi possiamo scegliere quella che ci assicura un "futuro" migliore. Ovviamente l'attendibilità del risultato è direttamente proporzionale alla profondità dell'albero, in quanto con una profondità maggiore riusciamo a migliorare le nostre previsioni. Purtroppo aumentando la profondità di ricerca, il degrado delle prestazioni è evidente: bisogna avere a disposizione dell'hardware più potente.

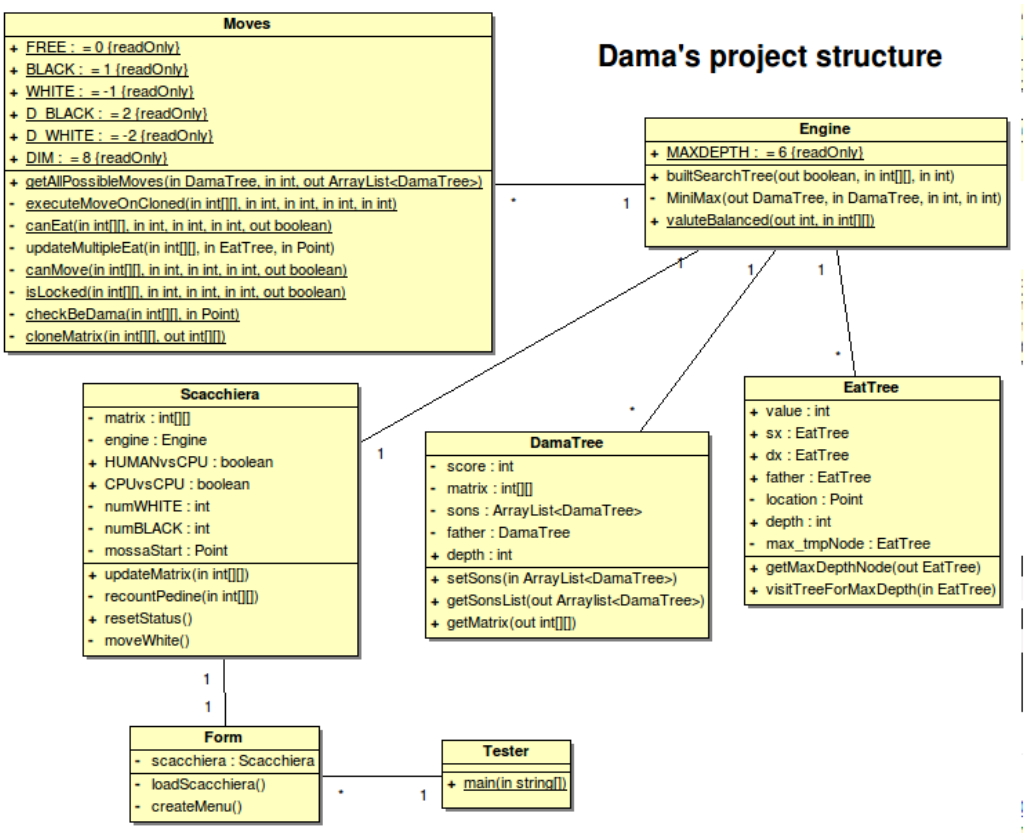
Inoltre, l'algoritmo utilizzato per la ricerca suppone che l'avversario giochi in maniera ottima. ciò implica l'eventualità che, giocando contro un umano che muove le pedine in modo casuale, il computer scelga delle mosse che non risultano essere le migliori e le più intelligenti.

# 4 Implementazione

Il programma è stato implementato in JAVA tramite l'IDE Eclipse e non presta particolare attenzione all'interfaccia grafica, già minimale.

## 4.1 Struttura generale

In seguito è riportato il diagramma delle classi in notazione UML. Dalla figura sono stati omessi i package ed alcuni metodi di banale interpretazione per ovvie motivazioni di spazio.



## 4.2 Package structure

Questo package è stato implementato allo scopo di contenere principalmente due classi, utilizzate come strutture dati fondamentali per la creazione di due differenti tipologie di alberi. Illustriamole singolarmente e cogliamo le loro analogie e differenze.

### 4.2.1 Classe DamaTree

La classe **DamaTree** corrisponde all'implementazione della struttura dati utilizzata per costruire i cosiddetti **alberi di ricerca**, utilizzati per ricercare la migliore mossa da effetture da parte del calcolatore in fase di calcolo. Una generica istanza di questa classe presenta i seguenti attributi:

- *matrix*: configurazione della scacchiera in un certo istante;
- *score*: punteggio associato, tramite la funzione di valutazione, alla configurazione interna al nodo;
- *father*: puntatore al nodo padre, anch'egli istanza della classe stessa;
- *sons*: lista di nodi figli, anch'essi istanze della classe stessa.

I metodi utilizzati sono i classici *"get"* e *"set"* che permettono l'accesso alle variabili d'istanza private.

```
package structure;

import java.util.ArrayList;
import engine.Moves.Action;

public class DamaTree
{
  private int[][] matrix;
  private int score;
  public int depth;
  public Action moveType;
  public DamaTree father;
  private ArrayList<DamaTree> sons;

  /*** Create radix of tree ***/
  public DamaTree(int[][] radix_matrix)
  {
    father = null;
    score = depth = 0;
```

```
    sons = null;
    matrix = radix_matrix;
  }

  public DamaTree(int [][] radix_matrix, DamaTree _father, int
      _depth, int _score, Action type)
  {
    this.matrix = radix_matrix;
    this.father = _father;
    this.score = _score;
    this.depth = _depth;
    this.moveType = type;
    sons = null;
  }

  public int [][] getMatrix()
  {
    return this.matrix;
  }

  public int getDepth()
  {
    return this.depth;
  }

  public void setSons(ArrayList<DamaTree> list)
  {
    this.sons = list;
  }

  public ArrayList<DamaTree> getSonsList()
  {
    return this.sons;
  }

  public void setScore(int s)
  {
    this.score = s;
  }

  public int getScore()
  {
    return this.score;
  }

}
```

DamaTree.java

### 4.2.2 Classe EatTree

Questa classe è apparentemente simile alla precedente, infatti è anch'essa impiegata per la realizzazione di una struttura dati ad albero. Le differenze sostanziali tra un *DamaTree* ed un *EatTree* risiedono:

- nella tipologia dell'albero: nel primo caso si tratta di un albero *n-ario*; nel secondo caso tratta un albero *binario*.
- nella loro costruzione: nel secondo caso l'albero viene costruito in base ad una particolare configurazione della scacchiera, a partire da una ben definita posizione di questa e fino ad una certa profondità variabile; nel primo caso la costruzione dell'albero è gestita manualmente dal package **"engine"** e la sua profondità è fissata a priori in quanto risulta essere un parametro delicato.
- per il loro impiego: gli alberi n-ari vengono utilizzati per scegliere la mossa migliore da intraprendere; gli alberi binari sono utilizzati per individuare l'eventuale presenza di mangiate multiple o singole da parte di pezzi presenti sulla damiera.

Una generica istanza di questa classe presenta i seguenti attributi:

- *value*: valore intero che rappresenta il colore della pedina che occupa la cella della damiera rappresentata dal nodo;
- *sx*: figlio sinistro, anch'egli istanza della classe stessa;
- *dx*: figlio destro, anch'egli istanza della classe stessa;
- *father*: nodo padre, anch'egli istanza della classe stessa;
- *location*: coordinate della cella della damiera rappresentata dal nodo; le coordinate sono date in forma di coppia *(indice_riga, indice_colonna)*;
- *depth*: valore che indica a che profondità dell'albero si trova il nodo;
- *maxtmpNode*: nodo foglia di massima profondità all'interno dell'albero.

In questa classe è di particolare importanza il metodo *getMaxDepthNode*, il quale visita l'albero fino alle foglie individuando e restituendo in output il nodo foglia alla massima profondità. Questa tecnica è utilizzata, dopo che l'albero è stato opportunamente creato, per capire qual'è il pezzo che può effettuare il maggior numero di catture multiple in un sol colpo.

```
package structure;
import java.awt.Point;

public class EatTree
{
  public int value;
  public EatTree sx = null;
  public EatTree dx = null;
  public EatTree father = null;
  private Point location;
  public int depth;
  private boolean evitasx = false;
  private boolean evitadx = false;
  private EatTree max_tmpNode = null;

  public EatTree(int[][] matrix, Point current_loc, boolean up,
      EatTree _father, int maxNumOfLevels)
  {
    this.location = current_loc;
    this.value = matrix[current_loc.x][current_loc.y];
    this.father = _father;
    this.depth = -1;

    // i can go (up || down) ?
    if ((up && current_loc.x == 0) || (!up && current_loc.x ==
        7) || (maxNumOfLevels == 0))
    {
      this.sx = this.dx = null;
      return;
    }

    // i can go right?
    if ((current_loc.y + 1) > 7)
    {
      this.dx = null;
      evitadx = true;
    }

    // i can go left?
    if ((current_loc.y - 1) < 0)
    {
      this.sx = null;
      evitasx = true;
    }

    if (!evitasx)
    {
      Point nextsx;
```

```
      if (up)
        nextsx = new Point(current_loc.x - 1, current_loc.y - 1)
            ;
      else
        nextsx = new Point(current_loc.x + 1, current_loc.y - 1)
            ;

      this.sx = new EatTree(matrix, nextsx, up, this,
          maxNumOfLevels - 1);
    }

    if (!evitadx)
    {
      Point nextdx;

      if (up)
        nextdx = new Point(current_loc.x - 1, current_loc.y + 1)
            ;
      else
        nextdx = new Point(current_loc.x + 1, current_loc.y + 1)
            ;

      this.dx = new EatTree(matrix, nextdx, up, this,
          maxNumOfLevels - 1);
    }
  }

  public EatTree getMaxDepthNode()
  {
    max_tmpNode = null;
    visitTreeForMaxDepth(this);    //initialize "max_tmpNode" to
        max depth
    return max_tmpNode;
  }

  private void visitTreeForMaxDepth(EatTree node)
  {
    if (max_tmpNode == null || (max_tmpNode.depth < node.depth))
      max_tmpNode = node;

    if (node.sx != null)
      visitTreeForMaxDepth(node.sx);

    if (node.dx != null)
      visitTreeForMaxDepth(node.dx);
  }

  public Point getLocation()
```

```
  {
    return this.location;
  }
}
```

EatTree.java

## 4.3 Package engine

Questo package rappresenta il cuore del progetto, nonché il fulcro del modulo d'intelligenza artificiale implementato. Le due classi presenti in questo package comunicano tra loro tramite metodi pubblici, costruendo ed esaminando l'*albero di ricerca* per trovare la migliore mossa possibile da effettuare ed utilizzando gli *EatTree* per attuare le varie considerazioni sulle "mangiate" secondo le regole a priorità previste dal gioco.

### 4.3.1 Classe Moves

Si tratta della classe che gestisce il sistema di movimento dei pezzi sulla damiera e definisce un *sistema di rappresentazione* per questi tramite alcune sue variabili d'istanza intere:

- *FREE*: rappresenta le celle vuote della damiera;
- *BLACK*: rappresenta le celle occupate dalle pedine di colore nero;
- *WHITE*: rappresenta le celle occupate dalle pedine di colore bianco;
- *D_BLACK*: rappresenta le celle occupate dalle dame di colore nero;
- *D_WHITE*: rappresenta le celle occupate dalle dame di colore bianco;
- *INACCESIBLE*: rappresenta le celle che non possono essere occupate da alcun pezzo (celle chiare della damiera).

Vediamo ora alcuni metodi fondamentali per la realizzazione del sistema di movimento e per il rilevamento delle mosse.

**Metodi generali di utilità** Tra i metodi più generali e di ovvia intepretazione troviamo:

- *cloneMatrix*: effettua una copia esatta della matrice passata; è necessario clonare la matrice di partenza per simulare le mosse future;

- *isOfColor*: verifica se la casella rappresentata dall' *EatTree* è occupata oppure no da una pedina avente un determinato colore;
- *isNodeFree*: verifica se la casella rappresentata dall' *EatTree* è libera oppure occupata;
- *inRange*: verifica se la coppia di indici passata è contenuta in una matrice 8x8 con indici da 0 a 7;
- *isDama*: verifica se in una determinata posizione della matrice passata è presente una dama;
- *isLocked*: verifica se il pezzo presente sulla damiera, in posizione specificata dalle coordinate passate, è impossibilitato a muoversi;
- *canMoveSx e Dx*: verifica se il pezzo presente sulla damiera, in posizione specificata dalle coordinate passate, può muoversi a sinistra o a destra.

I rimanenti metodi meritano un'attenzione particolare.

**setLastEatPosition** è quel metodo che viene invocato quando si vuole calcolare, a partire da una certa posizione della damiera, la posizione di arrivo di un pezzo in seguito ad una "mangiata". Il metodo percorre l'*EatTree* corrispondente alla cella di partenza desiderata e, in base alla configurazione della damiera, setta la massima profondità alla quale si può arrivare in fondo all'albero mangiando. Il nodo corrispondente alla cella libera sulla quale il pezzo che ha eseguito la mangiata approderà, potrà essere recuperato successivamente tramite il metodo *getMaxDepthNode* della classe *EatTree*.

**updateMultipleEat** Questo metodo viene invocato in seguito ad una mangiata multipla ed ha il compito di aggiornare le posizioni intermedie della matrice eliminando le pedine mangiate ed aggiornando la posizione di quella che ha mangiato. Come detto in precedenza, la posizione finale è rappresentata da un nodo della classe *EatTree*. A partire da quest'ultimo, sfruttando il puntatore al padre, si risale l'albero delle mangiate modificando iterativamente le celle della damiera interessate fino a ritornare nella posizione di partenza (radice dell'albero).

**executeMoveOnCloned** Questo metodo si limita ad invocare il metodo *cloneMatrix* e ad usare il suo output per simulare una delle mosse possibili.

**getAllPossibleMoves** è il metodo principale del sistema: la sua invocazione provoca una scansione completa della matrice, limitata ovviamente ai pezzi del giocatore che deve muovere nel corrente turno. Il metodo alloca una lista di oggetti istanze della classe *DamaTree*, dove ognuno dei quali rappresenta una **mossa** fattibile nel turno corrente e contiene la copia della matrice del turno corrente sulla quale, però, è stata eseguita la mossa suddetta. Ovviamente, poiché le regole parlano di "mangiate obbligatorie", è stato implementato un meccanismo di interruzione di ciclo che viene attivato nel caso in cui non si debbano valutare più possibili mangiate.

```
package engine;

import java.awt.Point;
import java.util.ArrayList;

import structure.*;

public class Moves
{
  private static int tmpx,tmpy;

  public final static int DIM = 8;
  public final static int FREE = 0;
  public final static int BLACK = 1;
  public final static int WHITE = -1;
  public final static int INACCESSIBLE = -5;
  public final static int D_BLACK = 2;
  public final static int D_WHITE = -2;
  public final static int EATFACTOR = 50;

  /*** Effettua una copia esatta della matrice ***/
  private static int[][] cloneMatrix(int[][] matrix)
  {
    int[][] cloned = new int[DIM][DIM];
    for (int row = 0; row < DIM; row++)
      for (int col = 0; col < DIM; col++)
        cloned[row][col] = matrix[row][col];

    return cloned;
  }

  /**** methods for EAT TREE ****/
  private static boolean isOfColor(EatTree node, int color)
  {
    return (node != null && node.value == color);
```

```
  }

  private static boolean isNodeFree(EatTree node)
  {
    return (node != null && node.value == FREE);
  }

  public static void setLastEatPosition(EatTree start_cell, int
      depth, int enemy)
  {
    boolean eat_sx = (isOfColor(start_cell.sx, enemy) &&
        isNodeFree(start_cell.sx.sx));
    boolean eat_dx = (isOfColor(start_cell.dx, enemy) &&
        isNodeFree(start_cell.dx.dx));

    if (!eat_sx && !eat_dx)
    {
      start_cell.setDepth(depth);
      return;
    }
    else
    {
      if (eat_sx)
        setLastEatPosition(start_cell.sx.sx, depth + 1, enemy);

      if (eat_dx)
        setLastEatPosition(start_cell.dx.dx, depth + 1, enemy);
    }
  }

  /*** Ricerca della mangiata migliore per la dama
   *
   * @param value colore della dama da cui partire;
   * @param loc posizione della scacchiera della dama di
       partenza;
   * @param matrix matrice completa (clone);
   * @return la casella di arrivo oppure null se non vi sono
       mangiate disponibili.
   */
  public static EatTree searchMaxDepthDamaEat(int value, Point
      loc, int[][] matrix)
  {
    int enemy_pedina = (value == D_WHITE) ? BLACK : WHITE;
    int enemy_dama = (value == D_WHITE) ? D_BLACK : D_WHITE;
    EatTree up_treePED = new EatTree(matrix, loc, true, null,
        Engine.MAXDEPTH);
    EatTree down_treePED = new EatTree(matrix, loc, false, null,
         Engine.MAXDEPTH);
    EatTree up_treeDAMA = new EatTree(matrix, loc, true, null,
```

```
        Engine.MAXDEPTH);
    EatTree down_treeDAMA = new EatTree(matrix, loc, false, null
        , Engine.MAXDEPTH);

    setLastEatPosition(up_treePED, 0, enemy_pedina);
    setLastEatPosition(down_treePED, 0, enemy_pedina);
    setLastEatPosition(up_treeDAMA, 0, enemy_dama);
    setLastEatPosition(down_treeDAMA, 0, enemy_dama);

    EatTree max_depthUP_PED = up_treePED.getMaxDepthNode();
    EatTree max_depthDOWN_PED = down_treePED.getMaxDepthNode();
    EatTree max_depthUP_DAMA = up_treeDAMA.getMaxDepthNode();
    EatTree max_depthDOWN_DAMA = down_treeDAMA.getMaxDepthNode()
        ;

    int max = Math.max(Math.max(max_depthUP_PED.depth,
        max_depthDOWN_PED.depth), Math.max(max_depthUP_DAMA.depth
        , max_depthDOWN_DAMA.depth));

    if (max <= 0)
      return null;
    else
    {
      if (max == max_depthUP_PED.depth)
        return max_depthUP_PED;
      else
        if (max == max_depthDOWN_PED.depth)
          return max_depthDOWN_PED;
        else
          if (max == max_depthUP_DAMA.depth)
            return max_depthUP_DAMA;
          else
            if (max == max_depthDOWN_DAMA.depth)
              return max_depthDOWN_DAMA;
            else
              return null;
    }
}

/**** END methods for EAT TREE ****/

public static boolean inRange(int i, int j)
{
  return (i > -1 && i < 8 && j > -1 && j < 8);
}

private static boolean isDama(int[][] matrix, int i, int j)
{
  return (matrix[i][j] == D_BLACK || matrix[i][j] == D_WHITE);
```

```
}

private static boolean isFree(int[][] matrix, int i, int j)
{
  return (inRange(i,j) && matrix[i][j] == FREE);
}

/*** Controlla se la pedina di turno "turn" spostandosi in
    posizione "end_pos" nella matrice "cloned_matrix"
 * diventa oppure no una dama.
 * @param cloned_matrix
 * @param turn
 * @param end_pos
 */
private static void checkBeDama(int[][] cloned_matrix, Point
    end_pos)
{
  for (int y = 0; y < 8; y++)
  {
    if ((y % 2) != 0 && cloned_matrix[7][y] == BLACK)
      cloned_matrix[end_pos.x][end_pos.y] = D_BLACK;
    else
      if ((y % 2) == 0 && cloned_matrix[0][y] == WHITE)
        cloned_matrix[end_pos.x][end_pos.y] = D_WHITE;
  }
}

/*** Controlla se la pedina in posizione matrix[i][j] Ã¨
    impossibilitata a muoversi a causa:
 * del bordo scacchiera;
 * della presenza di una pedina della propria squadra che la
    blocca;
 * NON vengono invece considerate gli "blocchi" da parte di
    pedine della squadra avversaria, in quanto potrebbero
 * risultare mangiate.
 * @param matrix
 * @param i
 * @param j
 * @param turn
 * @return
 */
private static boolean isLocked(int[][] matrix, int i, int j,
    int turn)
{
  if (!isDama(matrix,i,j))
  {
    boolean locked_dx, locked_sx;

    if (matrix[i][j] == BLACK)
```

```
      {
        locked_dx = (!inRange(i+1,j+1) || matrix[i][j] == matrix
            [i+1][j+1]);
        locked_sx = (!inRange(i+1,j-1) || matrix[i][j] == matrix
            [i+1][j-1]);
      }
      else
      {
        locked_dx = (!inRange(i-1,j+1) || matrix[i][j] == matrix
            [i-1][j+1]);
        locked_sx = (!inRange(i-1,j-1) || matrix[i][j] == matrix
            [i-1][j-1]);
      }

      return (locked_dx && locked_sx);
    }
    else
      return false;
  }

  /*** Verifica la possibilitÃ  di spostare la pedina nella
      casella di sx, in base al proprio turno.
   *
   * @param matrix
   * @param i
   * @param j
   * @param turn
   * @return true  se la casella alla propria sx Ã¨ vuota.
   * @return false se la pedina Ã¨ impossibilitata a spostarsi a
       sx.
   */
  private static boolean canMoveSx(int[][] matrix, int i, int j,
      int turn)
  {
    tmpy = (j - 1);

    if (turn == BLACK)
      tmpx = (i + 1);
    else
      tmpx = (i - 1);

    return isFree(matrix,tmpx,tmpy);
  }

  /*** Verifica la possibilitÃ  di spostare la pedina nella
      casella di dx, in base al proprio turno.
   *
   * @param matrix
   * @param i
```

```
     * @param j
     * @param turn
     * @return true  se la casella alla propria dx Ã¨ vuota.
     * @return false se la pedina Ã¨ impossibilitata a spostarsi a
         dx.
     */
    private static boolean canMoveDx(int[][] matrix, int i, int j,
        int turn)
    {
      tmpy = (j + 1);

      if (turn == BLACK)
        tmpx = (i + 1);
      else
        tmpx = (i - 1);

      return isFree(matrix,tmpx,tmpy);
    }

    private static boolean canEatDama(int[][] matrix, int i, int j
        , int turn)
    {
      int enemy_pedina = (turn == WHITE) ? BLACK : WHITE;
      int enemy_dama = (turn == WHITE) ? D_BLACK : D_WHITE;
      boolean eatdownsx, eatdowndx, eatupsx, eatupdx;

      eatdownsx = (inRange(i+1,j-1) && (matrix[i+1][j-1] ==
          enemy_pedina || matrix[i+1][j-1] == enemy_dama) && isFree
          (matrix, i+2, j-2));
      eatdowndx = (inRange(i+1,j+1) && (matrix[i+1][j+1] ==
          enemy_pedina || matrix[i+1][j+1] == enemy_dama) && isFree
          (matrix, i+2, j+2));
      eatupsx =   (inRange(i-1,j-1) && (matrix[i-1][j-1] ==
          enemy_pedina || matrix[i-1][j-1] == enemy_dama) && isFree
          (matrix, i-2, j-2));
      eatupdx =   (inRange(i-1,j+1) && (matrix[i-1][j+1] ==
          enemy_pedina || matrix[i-1][j+1] == enemy_dama) && isFree
          (matrix, i-2, j+2));

      return (eatdownsx || eatdowndx || eatupdx || eatupsx);
    }

    private static boolean canEat(int[][] matrix, int i, int j,
        int turn)
    {
      int enemy = (turn == WHITE) ? BLACK : WHITE;
      boolean eatdx, eatsx;

      if (turn == BLACK)
```

```
    {
      eatdx = (inRange(i+1,j+1) && matrix[i+1][j+1] == enemy &&
          isFree(matrix, i+2, j+2));
      eatsx = (inRange(i+1,j-1) && matrix[i+1][j-1] == enemy &&
          isFree(matrix, i+2, j-2));
    }
    else
    {
      eatdx = (inRange(i-1,j+1) && matrix[i-1][j+1] == enemy &&
          isFree(matrix, i-2, j+2));
      eatsx = (inRange(i-1,j-1) && matrix[i-1][j-1] == enemy &&
          isFree(matrix, i-2, j-2));
    }

    return (eatdx || eatsx);
  }

  private static void updateMultipleEat(int[][] cloned_matrix,
      EatTree end_node, Point start_pos)
  {
    Point end_pos = end_node.getLocation();
    cloned_matrix[end_pos.x][end_pos.y] = cloned_matrix[
        start_pos.x][start_pos.y];
    checkBeDama(cloned_matrix, end_pos);     //Ã¨ diventata dama
        ??

    while (end_node.father != null)
    {
      end_node = end_node.father;
      end_pos = end_node.getLocation();
      cloned_matrix[end_pos.x][end_pos.y] = FREE;
    }
  }

  /*** Esegue la mossa sulla matrice clonata.
   *
   * @param s_i row start index
   * @param s_j column start index
   * @param f_i row end index
   * @param f_j column end index
   * @return cloned  la matrice sulla quale ho eseguito la mossa
       (s_i, s_j) --> (f_i, f_j)
   */
  private static int[][] executeMoveOnCloned(int[][] matrix, int
      s_i, int s_j, int f_i, int f_j)
  {
    int[][] cloned = cloneMatrix(matrix);
    cloned[f_i][f_j] = cloned[s_i][s_j];
    cloned[s_i][s_j] = FREE;
```

```
    checkBeDama(cloned, new Point(f_i, f_j));     //Ã¨ diventata
        dama??

    return cloned;
  }

  /*** Calcola i figli di "radix" in base alle mosse possibili e
      restituisce il vettore di figli **/
  public static ArrayList<DamaTree> getAllPossibleMoves(DamaTree
      radix, int turn)
  {
    ArrayList<DamaTree> moves = new ArrayList<DamaTree>();
    int damaOfTurn, opponent;

    if (turn == BLACK)
    {
      damaOfTurn = D_BLACK;
      opponent = WHITE;
    }
    else
    {
      damaOfTurn = D_WHITE;
      opponent = BLACK;
    }

    int[][] tmp_matrix = radix.getMatrix();
    boolean force_eat = false;

    for (int row = 0; row < DIM; row++)
      for (int col = 0; col < DIM; col++)
        if ((row % 2) == (col % 2) && (tmp_matrix[row][col] ==
            turn || tmp_matrix[row][col] == damaOfTurn) && !
            isLocked(tmp_matrix, row, col, turn))
        {
          if (isDama(tmp_matrix, row, col)) // DAME
          {
            if (canEatDama(tmp_matrix, row, col, turn))
            {
              force_eat = true;
              int[][] cloned = cloneMatrix(tmp_matrix);
              EatTree max_depth_node = searchMaxDepthDamaEat(
                  cloned[row][col], new Point(row,col), cloned);

              int score = 0;
              if (turn == WHITE)
                score = -(max_depth_node.depth * EATFACTOR);  //
                    force eating select in tree
              else
```

```
            score = max_depth_node.depth * EATFACTOR;

        updateMultipleEat(cloned, max_depth_node, new
            Point(row,col));
        score += Engine.valuteBalanced(cloned); //applico
            la funzione di valutazione per lo stato in
            questione
        moves.add(new DamaTree(cloned, radix, radix.
            getDepth() + 1, score));
    }

    if (!force_eat)
    {
        if (canMoveSx(tmp_matrix, row, col, turn))
        {
            int[][] cloned = executeMoveOnCloned(tmp_matrix,
                row, col, tmpx, tmpy);
            int score = Engine.valuteBalanced(cloned);  //
                applico la funzione di valutazione per lo
                stato in questione

            moves.add(new DamaTree(cloned, radix, radix.
                getDepth() + 1, score));
            //System.out.println("Move sx: (" + row + "," +
                col + ") --> (" + tmpx + "," + tmpy + ")");
        }

        if (canMoveDx(tmp_matrix, row, col, turn))
        {
            int[][] cloned = executeMoveOnCloned(tmp_matrix,
                row, col, tmpx, tmpy);
            int score = Engine.valuteBalanced(cloned);  //
                applico la funzione di valutazione per lo
                stato in questione

            moves.add(new DamaTree(cloned, radix, radix.
                getDepth() + 1, score));
            //System.out.println("Move dx: (" + row + "," +
                col + ") --> (" + tmpx + "," + tmpy + ")");
        }

        if (canMoveDx(tmp_matrix, row, col, opponent))
        {
            int[][] cloned = executeMoveOnCloned(tmp_matrix,
                row, col, tmpx, tmpy);
            int score = Engine.valuteBalanced(cloned);  //
                applico la funzione di valutazione per lo
                stato in questione
```

```
            moves.add(new DamaTree(cloned, radix, radix.
                getDepth() + 1, score));
            //System.out.println("Move dx: (" + row + "," +
                col + ") --> (" + tmpx + "," + tmpy + ")");
          }

          if (canMoveSx(tmp_matrix, row, col, opponent))
          {
            int[][] cloned = executeMoveOnCloned(tmp_matrix,
                row, col, tmpx, tmpy);
            int score = Engine.valuteBalanced(cloned);  //
                applico la funzione di valutazione per lo
                stato in questione

            moves.add(new DamaTree(cloned, radix, radix.
                getDepth() + 1, score));
            //System.out.println("Move dx: (" + row + "," +
                col + ") --> (" + tmpx + "," + tmpy + ")");
          }
        }
      }
      else     //PEDINE
      {
        if (canEat(tmp_matrix, row, col, turn))
        {
          force_eat = true;
          int[][] cloned = cloneMatrix(tmp_matrix);
          EatTree tree = new EatTree(cloned, new Point(row,
              col), (turn == WHITE), null, 4);
          setLastEatPosition(tree, 0, (turn == BLACK) ?
              WHITE : BLACK); //switch enemy
          EatTree max_depth_node = tree.getMaxDepthNode();
          int score = 0;

          if (turn == WHITE)
            score = -(max_depth_node.depth * EATFACTOR);  //
                force eating select in tree
          else
            score = max_depth_node.depth * EATFACTOR;

          updateMultipleEat(cloned, max_depth_node, new
              Point(row,col));  //application of tree results
              -eats
          score += Engine.valuteBalanced(cloned); //applico
              la funzione di valutazione per lo stato in
              questione

          moves.add(new DamaTree(cloned, radix, radix.
              getDepth() + 1, score));
```

```
                //System.out.println("Eat sx: (" + row + "," + col
                    + ") --> (" + tmpx + "," + tmpy + ")");
            }

            if (!force_eat)
            {
              if (canMoveSx(tmp_matrix, row, col, turn))
              {
                int[][] cloned = executeMoveOnCloned(tmp_matrix,
                    row, col, tmpx, tmpy);
                int score = Engine.valuteBalanced(cloned);  //
                    applico la funzione di valutazione per lo
                    stato in questione

                moves.add(new DamaTree(cloned, radix, radix.
                    getDepth() + 1, score));
                //System.out.println("Move sx: (" + row + "," +
                    col + ") --> (" + tmpx + "," + tmpy + ")");
              }

              if (canMoveDx(tmp_matrix, row, col, turn))
              {
                int[][] cloned = executeMoveOnCloned(tmp_matrix,
                    row, col, tmpx, tmpy);
                int score = Engine.valuteBalanced(cloned);  //
                    applico la funzione di valutazione per lo
                    stato in questione

                moves.add(new DamaTree(cloned, radix, radix.
                    getDepth() + 1, score));
                //System.out.println("Move dx: (" + row + "," +
                    col + ") --> (" + tmpx + "," + tmpy + ")");
              }
            }
          }
        }

    return moves;
  }
}
```

Moves.java

### 4.3.2 Classe Engine

Possiamo immaginare la classe Engine come colei che, data una configurazione della damiera, esegue i seguenti passi:

1. richiede alla classe *Moves*, tramite il metodo *getAllPossibleMoves*, quali sono le possibili mosse possibli per l'attuale turno;

2. si serve dell'array di mosse allocato dal metodo precedente per creare ricorsivamente (tramite il metodo *recursiveSonsInitialization*) l'albero di ricerca per la migliore mossa, limitandosi alla profondità specificata dalla variabile d'istanza *MAXDEPTH*;

3. costruito l'albero, si occupa della valutazione dei nodi foglia invocando la funzione di valutazione *valuteBalanced* discussa nel paragrafo (3.3.1);

4. valutati i figli, applica l'algoritmo *MiniMax* per ottenere la mossa che minimizza la massima perdita e invoca il metodo *updateMatrix* della classe *Scacchiera* per aggiornare il display della scacchiera con la matrice ottenuta dall'esecuzione della mossa cercata.

**MiniMax** L'algortimo MiniMax, già discusso nel paragrafo (3.4), è stato realizzato tramite l'utilizzo di code di priorità già implementate nell'ambiente Java (*java.util.PriorityQueue*). Di queste code di priorità è stato sfruttato il potente potere estrattivo degli elementi, ordinati rispetto ad uno specifico *Comparator*. poiché l'algoritmo MiniMax prevede di assegnare al padre la minima e la massima valutazione delle mosse alternativamente, sono state allocate alternativamente due code di priorità aventi i comparator specializzati per tali estrazioni.

```
package engine;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.PriorityQueue;
import javax.swing.JOptionPane;
import engine.Moves.Action;
import launcher.Scacchiera;
import structure.DamaTree;

public class Engine
{
  public final static int MAXDEPTH = 6; // max profonditÃ  di
      ricerca dell'albero
  private final static int MAXDIM_QUEUE = 20;
  private DamaTree radixOfTree;
```

```
private Scacchiera scacchiera;
private final static int PEDINA = 100;     // Pedina's Weigth
private final static int DAMA = 200;     // Dama's Weight
private final static int JPOSFACTOR = 1;         //one matrix
    along the -j worth 1
private final static int BOARD = 10;     // fattore per dama
    sui bordi

public Engine(Scacchiera s)
{
  this.scacchiera = s;
}

private static int coverBLACK(int[][] matrix, int i, int j)
{
  int score = 0, tmp = 0;

  if (!Moves.inRange(i,j) || ((Moves.inRange(i-1,j-1) &&
      matrix[i-1][j-1] != Moves.BLACK ) && (Moves.inRange(i-1,j
      +1) && matrix[i-1][j+1] != Moves.BLACK)))
    if (i == -1)
      return 0;
    else
      return -1;
  else
  {
    tmp = coverBLACK(matrix, i-1, j-1);
    if(tmp!=-1)
      score += tmp + 2;

    tmp = coverBLACK(matrix, i-1, j+1);
    if(tmp!=-1)
      score += tmp + 2;
  }
  return score;
}

private static int coverWHITE(int[][] matrix, int i, int j)
{
  int score = 0, tmp = 0;

  if ( !Moves.inRange(i,j) || ((Moves.inRange(i+1,j-1) &&
      matrix[i+1][j-1] != Moves.WHITE) && (Moves.inRange(i+1,j
      +1) && matrix[i+1][j+1] != Moves.WHITE)))
    if(i == 8)
      return 0;
    else
      return -1;
  else
```

```
    {
      tmp = coverWHITE(matrix, i+1, j-1);
      if(tmp != -1)
        score += tmp + 2;

      tmp = coverWHITE(matrix, i+1, j+1);

      if(tmp != -1)
        score += tmp + 2;
    }

    return score;
  }

  public static int valuteDefensed(int[][] matrix)
  {
    int score = 0, numDameW = 0, numDameB = 0;
    int[][] posDamaW = new int[12][2];
    int[][] posDamaB = new int[12][2];

    for (int i = 0; i < Moves.DIM; i++)
      for (int j = 0; j < Moves.DIM; j++)
      {
        if (matrix[i][j] == Moves.WHITE)
        {
          score -= PEDINA;
          score -= JPOSFACTOR*i*i;
          score -= coverWHITE(matrix,i,j);
        }
        else
          if (matrix[i][j] == Moves.BLACK)
          {
            score += PEDINA;
            score += JPOSFACTOR*(7-i)*(7-i);
            score += coverBLACK(matrix,i,j);
          }
          else
            if (matrix[i][j] == Moves.D_WHITE)
            {
              posDamaW[numDameW][0]=i;
              posDamaW[numDameW][1]=j;
              numDameW++;
              score -= DAMA;
              if (i==0 || i==7)
                score += BOARD;
              if (j==0 || j==7)
                score += BOARD;
            }
            else
```

```
                if (matrix[i][j] == Moves.D_BLACK)
                {
                  posDamaB[numDameB][0]=i;
                  posDamaB[numDameB][1]=j;
                  numDameB++;
                  score += DAMA;
                  if (i==0 || i==7)
                    score -= BOARD;
                  if (j==0 || j==7)
                    score -= BOARD;
                }
    }

    score += (int)(Math.random() * RANDOMFACTOR);
  return score;
}


/*** Funzione di valutazione Peso + Posizione :
   Il peso dei pezzi viene moltiplicato per il quadrato
      dellâ indice relativo alla riga in cui si trova il
      pezzo.  ***/
public static int valuteBalanced(int[][] matrix)
{
  int score = 0;

  for (int i = 0; i < Moves.DIM; i++)
    for (int j = 0; j < Moves.DIM; j++)
    {
      if (matrix[i][j] == Moves.WHITE)
        {
        score -= PEDINA;
        score -= JPOSFACTOR * (7-i) * (7-i);  //duale: riga i
           = 7 <==> 7 - 7 = riga 0
        }
        else
          if (matrix[i][j] == Moves.D_WHITE)
          {
            score -= DAMA;

          if (i == 0 || i == 7)    //check board (WHITE)
            score += BOARD;

          if (j == 0 || j == 7)
            score += BOARD;
          }
          else
            if (matrix[i][j] == Moves.D_BLACK)
            {
```

```
                score += DAMA;

              if (i == 0 || i == 7)    //check board (BLACK)
                score -= BOARD;

              if (j == 0 || j == 7)
                score -= BOARD;
              }
              else
                if (matrix[i][j] == Moves.BLACK)
                {
                score += PEDINA;
                score += JPOSFACTOR * i * i;
                }
    }

  score += (int)(Math.random() * RANDOMFACTOR);
  return score;
  }

/*** DFS initialization of tree
 *   L'albero, a partire dal figlio subito DOPO la radice, Ã¨
     formato da nero-bianco alternati per livello
 * @param node
 */
private void recursiveSonsInitialization(DamaTree node)
{
  if (node != null && node.getDepth() < (MAXDEPTH - 1) && node
      .getSonsList().size() > 0)  //c'Ã¨ almeno una mossa
      disponibile
  {
    for (DamaTree son : node.getSonsList())
    {
      ArrayList<DamaTree> sons_list = Moves.
          getAllPossibleMoves(son, ((son.getDepth() % 2) == 0)
          ? Moves.BLACK : Moves.WHITE);
      son.setSons(sons_list);
      recursiveSonsInitialization(son);
    }
  }
}

private DamaTree MiniMax(DamaTree node, int depth, int turn)
{
  if (node != null)
  {
    if (node.getDepth() != depth)
    {
      for (DamaTree son : node.getSonsList())
```

```
            MiniMax(son, depth, turn);
        }
        else
        {
            PriorityQueue<DamaTree> queue;

            if ((depth % 2 != 0 && turn == Moves.BLACK) || (depth %
                2 == 0 && turn == Moves.WHITE))    //get min
            {
                queue = new PriorityQueue<DamaTree>(MAXDIM_QUEUE, new
                    Comparator<DamaTree>() {
                                                public int compare(
                                                    DamaTree o1, DamaTree
                                                     o2)
                                                {
                                                  return o1.getScore() -
                                                      o2.getScore();

                                                } });
            }
            else            //get max
            {
                queue = new PriorityQueue<DamaTree>(MAXDIM_QUEUE, new
                    Comparator<DamaTree>() {
                                                public int compare(
                                                    DamaTree o1, DamaTree
                                                     o2)
                                                {
                                                  return o2.getScore() -
                                                      o1.getScore();

                                                } });
            }

            for (DamaTree son : node.getSonsList())    //figli giÃ
                valutati
                queue.add(son);

            DamaTree el = queue.poll();
            if (el == null)
                return null;

            node.setScore(el.getScore());
            return el;
        }
    }

    return null;
}
```

```
private void detectMoveDeletingForEat(ArrayList<DamaTree> list
    )
{
  boolean exists_eat = false;

  for (DamaTree el : list)
  {
    if (el.moveType == Action.EAT)
    {
      exists_eat = true;
      break;
    }
  }

  if (!exists_eat)
    return;

  for (int i = list.size() - 1; i >= 0; i--)
    if (list.get(i).moveType == Action.MOVE)
      list.remove(i);

}

public boolean builtSearchTree(int[][] start_matrix, int turn)
{
  /*** set time ***/
  long Tempo1, Tempo2;
  Tempo1 = System.currentTimeMillis();

  this.radixOfTree = new DamaTree(start_matrix);
  ArrayList<DamaTree> sons_list = Moves.getAllPossibleMoves(
      this.radixOfTree, turn);  //inizializza figli

  //se vi sono mangiate, elimino tutte le mosse di tipo "move"
       dalla lista in quanto le prime sono prioritarie
  detectMoveDeletingForEat(sons_list);

  if (sons_list.size() > 0)
  {
    /*** very tree built ***/
    this.radixOfTree.setSons(sons_list);
    recursiveSonsInitialization(this.radixOfTree);

    /*** Selecting best max for black move ***/
    DamaTree max_depth_node = null;
    for (int i = MAXDEPTH - 1; i >= 0; i--)
      max_depth_node = MiniMax(this.radixOfTree, i, turn);
```

```
        int[][] m = null;
        if (max_depth_node != null)
        {
          m = max_depth_node.getMatrix();
          /*** get time ***/
          Tempo2 = System.currentTimeMillis();
          System.out.println(max_depth_node.moveType + " - " + (
              Tempo2 - Tempo1) + "ms");
        }

        scacchiera.updateMatrix(m);
        return true;
      }
      else
      {
        if (turn == Moves.WHITE)
        {
          System.out.println("no moves for WHITE player: BLACK
              WINS");
          JOptionPane.showMessageDialog(null, "Nessuna mossa
              disponibile per il BIANCO: Vince il NERO");
        }
        else
        {
          System.out.println("no moves for BLACK player: WHITE
              WINS");
          JOptionPane.showMessageDialog(null, "Nessuna mossa
              disponibile per il NERO: Vince il BIANCO");
        }

        return false;
      }
    }

    private static void print_matrix(int[][] m)
    {
      for (int i=0; i < 8; i++)
      {
        for (int j = 0; j < 8; j++)
          System.out.print(m[i][j] + ", ");
        System.out.print("\n");
      }

      System.out.print("\n\n");
    }
}
```

Engine.java

## 4.4 Package launcher

è il package utilizzato per realizzare la finestra di gioco, la damiera e l'avvio del programma.

### 4.4.1 Classe Scacchiera

è la classe che realizza l'interfaccia grafica del programma. Si occupa di ridisegnare la configurazione della damiera ogni qualvolta viene invocato il suo metodo *updateMatrix*. Inoltre gestisce l'invocazione alternata dell'engine del programma rispetto alla mossa dell'utente umano. è anche colei che si accorge di quando la partita è finita e a causa di quale condizione (vittoria, patta o esaurimento mosse). Le modalità di gioco che mette a disposizione sono due:

- CPUvsCPU;
- HUMANvsCPU.

Nel primo caso viene avviato l'engine con alternanza "bianco - nero"; nel secondo caso attende l'azione del giocatore umano e subito dopo invoca l'engine per l'azione del computer.

```
package launcher;

import javax.swing.*;
import structure.EatTree;
import engine.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.ArrayList;

@SuppressWarnings("serial")
public class Scacchiera extends JPanel     // scacchiera classic
    8 x 8
{
  private int[][] matrix = new int[Moves.DIM][Moves.DIM];
  private final static int CELLSPACE = 80;
  private final static int INNERRAY = 20;
  private Engine engine;
  public boolean HUMANvsCPU = false;
  public boolean CPUvsCPU = false;
  private int numWHITE = 12;
  private int numBLACK = 12;
```

```
//Graphics
private ArrayList<Rectangle> coord_list = new ArrayList<
    Rectangle>(); //for pick correlation
private Point mossaStart;
private Point mossaEnd;
private Color color1 = new Color(139,69,19);     //dark brown
private Color color2 = new Color(210,180,140);     //light
    brown

public Scacchiera()
{
  super();
  this.setBackground(color2);
  initializeInternal();

  this.addMouseListener(new MouseAdapter() {
          public void mousePressed(MouseEvent evt)
          {
            if (HUMANvsCPU)
            {
                for (Rectangle r : coord_list)
                  if (r.contains(evt.getPoint()))
                  {
                    mossaStart = new Point(r.y / CELLSPACE, r.
                        x / CELLSPACE);
                    break;
                  }
            }
          }
          public void mouseReleased(MouseEvent evt)
          {
            if (HUMANvsCPU && (matrix[mossaStart.x][mossaStart
                .y] == Moves.WHITE || matrix[mossaStart.x][
                mossaStart.y] == Moves.D_WHITE))
            {
              for (Rectangle r : coord_list)
                  if (r.contains(evt.getPoint()))
                  {
                    mossaEnd = new Point(r.y / CELLSPACE, r.x
                        / CELLSPACE);
                    moveWhite();
                    break;
                  }
            }
          }
      });
}
```

```
/*** Effettua una mossa del bianco e fa partire il calcolo del
    nero ***/
private void moveWhite()
{
  EatTree checker_tree = new EatTree(this.matrix, mossaStart,
      true, null, Engine.MAXDEPTH);
  Moves.setLastEatPosition(checker_tree, 0, Moves.BLACK); //
      switch enemy
  EatTree max_depth_node = checker_tree.getMaxDepthNode();
  EatTree end_eat_dama = Moves.searchMaxDepthDamaEat(matrix[
      mossaStart.x][mossaStart.y], mossaStart, matrix);

  if (matrix[mossaStart.x][mossaStart.y] == Moves.D_WHITE &&
      end_eat_dama != null && end_eat_dama.depth > 0)     // >=1
       of eat dama!
  {
    Point end_position = end_eat_dama.getLocation();
    if (!end_position.equals(mossaEnd))
    {
      JOptionPane.showMessageDialog(null, "Mangiata migliore
          con arrivo in (" + end_position.x + "," +
          end_position.y + ")");
      return;
    }
    else
    {
      matrix[end_position.x][end_position.y] = matrix[
          mossaStart.x][mossaStart.y];

      while (end_eat_dama.father != null)
      {
        end_eat_dama = end_eat_dama.father;
        end_position = end_eat_dama.getLocation();
        matrix[end_position.x][end_position.y] = Moves.FREE;
      }
    }
  }
  else
    if (max_depth_node.depth > 0) // >=1 of eat pedina!
    {
      Point end_position = max_depth_node.getLocation();
      if (!end_position.equals(mossaEnd))
      {
        JOptionPane.showMessageDialog(null, "Mangiata migliore
            con arrivo in (" + end_position.x + "," +
            end_position.y + ")");
        return;
      }
      else
```

```
        {
          matrix[end_position.x][end_position.y] = matrix[
              mossaStart.x][mossaStart.y];

          while (max_depth_node.father != null)
          {
            max_depth_node = max_depth_node.father;
            end_position = max_depth_node.getLocation();
            matrix[end_position.x][end_position.y] = Moves.FREE;
          }
        }
      }
      else
      {
        if (matrix[mossaStart.x][mossaStart.y] == Moves.WHITE)
        {
          // evito spostamento in basso e quando resto sulla
              stessa casella
          if (mossaStart.equals(mossaEnd) || (mossaStart.x -
              mossaEnd.x) != 1 || Math.abs((mossaStart.y -
              mossaEnd.y)) != 1 )
            return;
        }
        else
          if (matrix[mossaStart.x][mossaStart.y] == Moves.
              D_WHITE)
          {
            if (mossaStart.equals(mossaEnd) || Math.abs((
                mossaStart.x - mossaEnd.x)) != 1 || Math.abs((
                mossaStart.y - mossaEnd.y)) != 1 )
              return;
          }

        // rilascio su posizione libera ==> switch
        if (matrix[mossaEnd.x][mossaEnd.y] == Moves.FREE)
        {
          matrix[mossaEnd.x][mossaEnd.y] = matrix[mossaStart.x][
              mossaStart.y];  //switch pedina
          matrix[mossaStart.x][mossaStart.y] = Moves.FREE;
        }
      }

      if (mossaEnd.x == 0)
        matrix[mossaEnd.x][mossaEnd.y] = Moves.D_WHITE;



      this.paint(getGraphics());           //repaint scacchiera
      recountPedine(matrix);
```

```
      if (this.numBLACK == 0)
        JOptionPane.showMessageDialog(null, "Vince il BIANCO");
      else
        engine.builtSearchTree(matrix, Moves.BLACK);
    }

  public void playCPUvsCPU()
  {
    for (int i = 0; i < 200; i++)   //###DEBUG:: upper bound
        evita loop ###
    //while (this.numBLACK > 0 && this.numWHITE > 0)
    {
      if (this.numWHITE > 0 && this.numBLACK > 0)
      {
        if (!engine.builtSearchTree(matrix, Moves.WHITE))
          return;
      }
      else
        return;

      if (this.numWHITE > 0 && this.numBLACK > 0)
      {
        if (!engine.builtSearchTree(matrix, Moves.BLACK))
          return;
      }
      else
        return;
    }
  }

  private void initializeInternal()
  {
    for (int row = 0; row < matrix.length; row++)
      for (int col = 0; col < matrix.length; col++)
        if ((col % 2) == (row % 2))
        {
          coord_list.add(new Rectangle(col * CELLSPACE, row *
              CELLSPACE, CELLSPACE, CELLSPACE));

          if (row < 3)
            matrix[row][col] = Moves.BLACK;
          else
            if (row >= 5)
              matrix[row][col] = Moves.WHITE;
            else
              matrix[row][col] = Moves.FREE;
        }
        else
```

```
            matrix[row][col] = Moves.INACCESSIBLE;

    engine = new Engine(this);
    this.numBLACK = this.numWHITE = 12;
    //matrix[5][1] = Moves.D_WHITE;
}

public void resetStatus()
{
    this.CPUvsCPU = this.HUMANvsCPU = false;
    initializeInternal();
    paintComponent(this.getGraphics());
}

public void updateMatrix(int[][] new_matrix)
{
    if (new_matrix != null)
    {
        this.matrix = new_matrix;
        this.paint(getGraphics());            //repaint scacchiera
        recountPedine(new_matrix);

        if (numWHITE == 0)
            JOptionPane.showMessageDialog(null, "Vince il NERO");
        else
            if (numBLACK == 0)
                JOptionPane.showMessageDialog(null, "Vince il BIANCO")
                    ;
    }
    else
        JOptionPane.showMessageDialog(null, "ERROR: null matrix
            -.-");
}

private void recountPedine(int[][] new_matrix)
{
    numBLACK = numWHITE = 0;
    for (int i = 0; i < Moves.DIM; i++)
        for (int j = 0; j < Moves.DIM; j++)
        {
            if (new_matrix[i][j] == Moves.WHITE || new_matrix[i][j]
                == Moves.D_WHITE)
                numWHITE++;
            else
                if (new_matrix[i][j] == Moves.BLACK || new_matrix[i][j
                    ] == Moves.D_BLACK)
                    numBLACK++;
        }
}
```

```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D)g;
  g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);

  /*** Drawing scacchiera ***/
  g2.setColor(color1);
  for (int row = 0; row < matrix.length; row++)
    for (int col = 0; col < matrix.length; col++)
      if ((row % 2) == (col % 2))
        g2.fillRect(col * CELLSPACE, row * CELLSPACE,
            CELLSPACE, CELLSPACE);

  /*** Drawing pedine **/
  for (int row = 0; row < matrix.length; row++)
    for (int col = 0; col < matrix.length; col++)
      if (matrix[row][col] == Moves.BLACK)
      {
        g2.setColor(Color.black);
        g2.fillOval(col * CELLSPACE, row * CELLSPACE,
            CELLSPACE, CELLSPACE);
      }
      else
        if (matrix[row][col] == Moves.WHITE)
        {
          g2.setColor(Color.yellow);
          g2.fillOval(col * CELLSPACE, row * CELLSPACE,
              CELLSPACE, CELLSPACE);
        }
        else
          if (matrix[row][col] == Moves.D_BLACK)
          {
            int x = col * CELLSPACE;
            int y = row * CELLSPACE;
            g2.setColor(Color.black);
            g2.fillOval(x, y, CELLSPACE, CELLSPACE);
            g2.setColor(Color.red);
            g2.fillOval(x + INNERRAY, y + INNERRAY, 2*INNERRAY
                , 2*INNERRAY);
          }
          else
            if (matrix[row][col] == Moves.D_WHITE)
            {
              int x = col * CELLSPACE;
              int y = row * CELLSPACE;
              g2.setColor(Color.yellow);
```

```
                    g2.fillOval(x, y, CELLSPACE, CELLSPACE);
                    g2.setColor(Color.DARK_GRAY);
                    g2.fillOval(x + INNERRAY, y + INNERRAY, 2*
                        INNERRAY, 2*INNERRAY);
                }
    }

}
```

Scacchiera.java

### 4.4.2 Classe Form

Implementa la finestra di visualizzazione della scacchiera. è stata definita ereditando da *javax.swing.JFrame*.

```
package launcher;
import java.awt.Point;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

import javax.swing.*;

@SuppressWarnings("serial")
public class Form extends JFrame{

  private final int squaredim = 80;
  private Scacchiera scacchiera;

  public Form(String txt)
  {
    super(txt);
    this.setLocation(new Point(300, 15));
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    createMenu();
    this.setSize(this.squaredim * 8, (this.squaredim * 8) + 45);
    loadScacchiera();
  }

  private void loadScacchiera()
  {
    scacchiera = new Scacchiera();
    this.getContentPane().add(scacchiera);
  }

  private void createMenu()
```

```
  {
    JMenuBar menuBar = new JMenuBar();
    JMenu menu = new JMenu("Game");
    menuBar.add(menu);

    //submenu's
    JMenuItem menuItem1 = new JMenuItem("Play Human vs CPU");
    menuItem1.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent evt)
            {
              scacchiera.resetStatus();
              scacchiera.HUMANvsCPU = true;
              System.out.println("HUMANvsCPU mode running...\n\n
                  ");
            }
    });

    JMenuItem menuItem2 = new JMenuItem("Play CPU vs CPU");
    menuItem2.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent evt)
            {
              scacchiera.resetStatus();
              scacchiera.CPUvsCPU = true;
              System.out.println("CPUvsCPU mode running...\n\n")
                  ;
              scacchiera.playCPUvsCPU();
            }
    });
    JMenuItem menuItem3 = new JMenuItem("Stop game");
    menuItem3.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent evt)
            {
              scacchiera.resetStatus();
              System.out.println("Game stopped\n\n");
            }
    });
    menu.add(menuItem1);
    menu.add(menuItem2);
    menu.add(menuItem3);

    this.setJMenuBar(menuBar);
  }
}
```

Form.java

### 4.4.3 Classe Tester

è il main dal quale viene lanciato il programma. Si limita a creare una nuova istanza della classe *Form* e a renderla visibile.

```
package launcher;

public class Tester {

  public static void main(String[] args)
  {
    Form f = new Form("DamaGame");
    f.setVisible(true);
  }
}
```

Tester.java

# 5 Test e prestazioni

Sono stati eseguiti diversi tipi di test:

- test di valutazione della bontà della *funzione di valutazione*;
- test di valutazione del *tempo richiesto* per scegliere la mossa migliore da parte del calcolatore, nel caso HUMANvsCPU;
- $\beta$-test (incluso ovviamente anche nei precedenti).

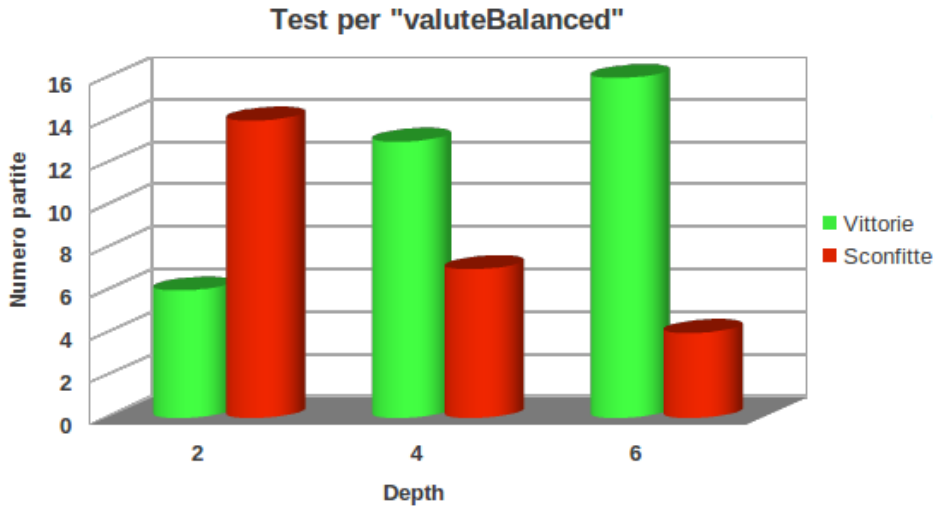
I risultati ottenuti sono stati trasformati poi in *istogrammi*, aggiunti in un secondo momento alla documentazione.

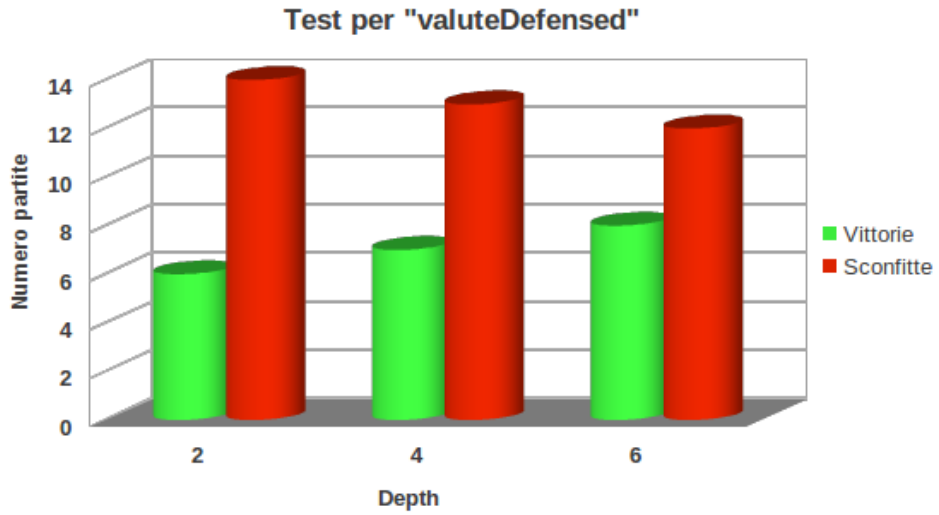
## 5.1 Test per la funzione di valutazione

In principio erano state implementate *due* differenti funzioni di valutazione:

1. *"valuteBalanced"*, ovvero quella attualmente utilizzata, basata su una strategia bilanciata tra attacco e difesa;
2. *"valuteDefense"*, una funzione che curava di più l'aspetto difensivo, soprattutto quello di copertura delle retrovie, a discapito però di quel minimo di aggressività che non può mancare ad un giocatore di dama.

È stata infine scelta la prima tra le due a causa del buon "rendimento" ottenuto nel confronto con la seconda in modalità HUMANvsCPU. I risultati sono stati ottenuti in base ad un campione di circa **20 partite**, eseguite con profondità dell'albero di ricerca crescente fino a 6. In seguito sono riportati gli istogrammi corrispondenti a tali risultati, costruiti ignorando le situazioni di pareggio.




Possiamo quindi notare il successo di un approccio leggermente più aggressivo rispetto a quello prettamente difensivo.

## 5.2 Tempo richiesto per scegliere la mossa migliore

Il tempo impiegato dal calcolatore per capire qual'è la migliore mossa da effettuare è direttamente proporzionale alla profondità dell'albero. I test sono stati effettuati prendendo in esame la profondità pari a 6, valore di default all'interno del codice:

- per valori *inferiori* a 6, il programma risulta estremamente rapido, ma ovviamente le scelte intraprese non sono mediamente buone;
- per valori *superiori* a 6, il programma inizia a perdere interattività. Addirittura in alcuni casi limite, come la presenza di molte dame sulla damiera da parte di entrambi i giocatori, si sono manifestate situazioni di *"Java Heap full"* che hanno impedito il normale proseguimento della partita.

Ritornando alla profondità pari 6, è stata attuata una valutazione per capire, in media, quanto tempo impiega il calcolatore a scegliere la mossa migliore e come varia questo tempo in funzione della profondità dell'albero di ricerca. Un modo per misurare approsimativamente tale tempo si ottiene utilizzando il metodo *"currentTimeMillis"* oppure il metodo della classe *"nanoTime"*, entrambi della classe *System*:

```
long Tempo1, Tempo2;
Tempo1 = System.currentTimeMillis();

/**** ........... ****/
/**** costruzione albero + scelta mossa ****/
/**** ........... ****/

Tempo2 = System.currentTimeMillis();
System.out.println(Tempo2 - Tempo1 + "ms");
```

Il precedente codice genera un listato contenente tutte le mosse effettivamente eseguite dal calcolatore durante una partita contro un giocatore umano, ed i relativi tempi impiegati per eseguirla:

```
HUMANvsCPU mode running...
MOVE - 106ms
EAT  - 64ms
MOVE - 239ms
MOVE - 200ms
EAT  - 46ms
```

```
MOVE - 411ms
MOVE - 626ms
EAT  - 58ms
MOVE - 1215ms
EAT  - 13ms
MOVE - 1222ms
MOVE - 168ms
EAT  - 34ms
MOVE - 241ms
MOVE - 1079ms
MOVE - 399ms
MOVE - 1196ms
MOVE - 1026ms
MOVE - 102ms
MOVE - 81ms
MOVE - 56ms
MOVE - 89ms
MOVE - 37ms
MOVE - 85ms
MOVE - 56ms
MOVE - 47ms
MOVE - 5ms
MOVE - 26ms
MOVE - 20ms
MOVE - 18ms
MOVE - 2ms
MOVE - 5ms
MOVE - 7ms
MOVE - 5ms
no moves for BLACK player: WHITE WINS
```

Dal precedente listato si può notare come le mangiate siano molto rapide rispetto ai semplici spostamenti, in quanto è stato introdotto il meccanismo di interruzione di ciclo che esclude da questo le valutazioni dei successivi possibili spostamenti.

## 5.3 $\beta$-testing

Il $\beta$-test è stato effettuato, oltre che dal realizzatore del programma, da due volontari che si sono gentilmente prestati allo scopo. Il suo compito è proprio quello di individuare e correggere vari ed eventuali bug che lo sviluppatore non è riuscito a trovare. Sin ora non vi sono stati particolari situazioni spiacevoli, ad esclusione di un grave errore che permetteva ad una pedina di non mangiare pur essendone obbligata.

# 6 Conclusioni

Addentrarsi nel campo dell'intelligenza artificiale non è stata un esperienze semplice, ma al contrario è stata molto piacevole. L'idea di progetto ha ispirato il sottoscritto sin dall'inizio.

Particolari attenzioni e gran parte del tempo è stato dedicato alla realizzazione delle funzioni di valutazione e dell'algoritmo MiniMax, entrambi colonne portanti della struttura del progetto. Non è stato dato molto peso all'implementazione dell'interfaccia grafica, creata in modo minimale giusto per fornire una accessibilità diretta.