

# PROJECT **CH**ESS

*“Un computer una volta mi ha battuto a scacchi, ma non c'è stata partita con la kickboxing.”*

**Emo Philips**

## Introduzione

Fin dagli albori dell'intelligenza artificiale, uno degli obiettivi è stato quello di creare un'intelligenza che riuscisse a battere l'uomo. **Shannon** e **Turing** scrissero sin dal 1950 dei programmi che giocassero a scacchi, Shannon scrisse nel 1950 il primo articolo scientifico su come programmare una macchina scacchistica<sup>1</sup>. Dalla metà degli anni '80, i computer si sono avvicinati al livello dei campioni (grandmaster) e nel 1997 Garry Kasparov, allora campione del mondo, venne sconfitto in una famosa partita con un supercomputer dell'IBM chiamato Deep Blue (il computer vinse due partite, ne perse una e pareggiò le altre tre). Successivamente la ricerca per migliorare le prestazioni e la qualità dei software scacchistici non si ferma, arrivando all'Ottobre 2017 con il progetto AlphaZero, primo elaboratore al mondo che “impara”

---

<sup>1</sup> C. Shannon, Programming a computer for playing chess, Philosophical Magazine, 1950

autonomamente giocando un match dopo l'altro. L'esperienza accumulata gli permette di processare un numero minore di possibili mosse ma di analizzare a fondo quelle che sono più "promettenti". Il risultato è strabiliante: in poche ore il computer sconfigge i migliori software pre-esistenti basati sul vecchio algoritmo (potatura alfa-beta) come **Stockfish**. In questa relazione verrà descritto l'implementazione di un software che utilizza un algoritmo **potatura alfa-beta** con una semplice funzione di valutazione.

## Architettura ed implementazione software

I linguaggi di programmazione usati per questo software sono stati due:

- **C#**: per realizzare l'interfaccia grafica in 3D utilizzando il framework Unity<sup>2</sup>
- **Python**: sfruttando le librerie **python-chess**<sup>3</sup>, è stato utilizzato sia per implementare l'intera logica e regole degli scacchi, sia per implementare l'intelligenza artificiale

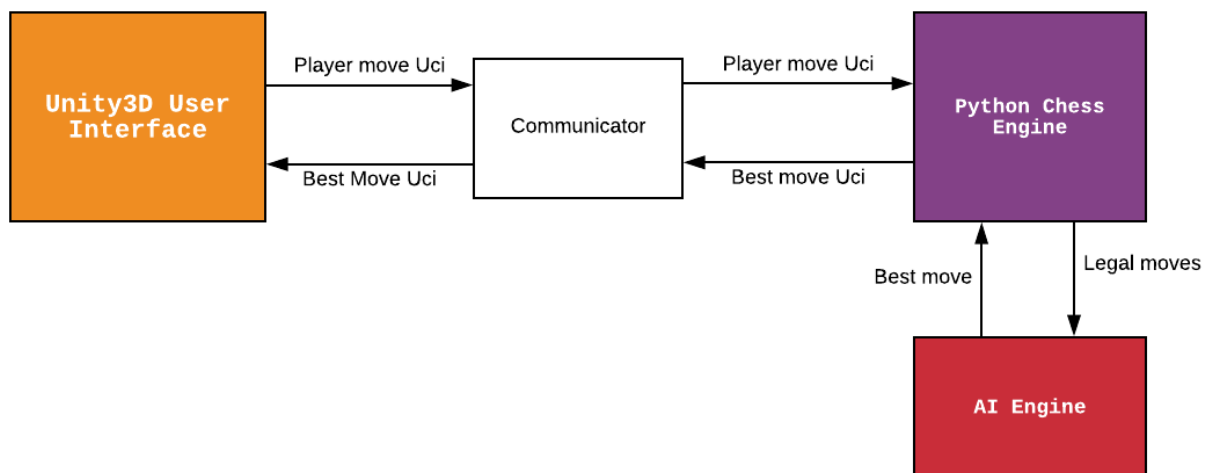


Figura 1

In generale, il programma scritto su Unity riceve dal **Chess Engine** tutte le mosse legali che l'utente può fare, l'utente sceglie quale mossa fare ed essa viene convertita in una stringa in formato **UCI**<sup>4</sup>, questa viene inviata ad una classe **Communicator** che contiene l'implementazione di un socket, dopodiché il Communicator trasferisce la mossa in formato UCI alla classe **Server** implementata in Python. A questo punto la

<sup>2</sup> Sito ufficiale Unity: <https://unity3d.com/>

<sup>3</sup> Repository Github: <https://github.com/niklasf/python-chess>

<sup>4</sup> **Universal Chess Interface** (UCI) è un protocollo di comunicazione libero che permette ad un motore scacchistico di comunicare con la sua interfaccia grafica.

mossa viene tradotta in mossa effettiva e viene aggiornata la situazione in gioco. In questo momento tocca al computer decidere cosa muovere, vengono calcolate tutte le mosse legali e date in input all'**AI Engine**. In questa classe è implementato l'algoritmo **Minimax** (con il solo scopo di confrontare le prestazioni) e l'algoritmo **Alfa-Beta pruning**. Dopo aver eseguito l'algoritmo, la classe fornirà in output la miglior mossa che possa fare il computer, questa mossa verrà concretamente eseguita dal **Chess Engine** che valuterà se questa nuova situazione è di **Scacco**, **Scacco matto** o **Stallo**. Dopo aver aggiornato lo stato di gioco, invierà la mossa all'interfaccia che aggiornerà la propria situazione di gioco così da mostrarla al giocatore umano. Uno schema riassuntivo è mostrato in *Figura 1*.

Le classi principali in Python sono le seguenti:

- **game.py**, questa è la classe che implementa il **game loop**, essa gestisce il cambio di turno tra bianco e nero, inizializza il Chess engine e l'AIEngine, sta in attesa delle mosse del computer o del giocatore umano e infine aggiorna la situazione attuale. Oltre a questo verifica se c'è una situazione di Scacco, Scacco matto o Stallo.
- **client.py** e **server.py**, implementano rispettivamente il socket che invia le mosse verso l'interfaccia e quello che invece le riceve.
- **AIEngine.py**, come detto in precedenza, implementa l'algoritmo di ricerca della mossa migliore, inoltre contiene il parametro **depth** per impostare il limite di profondità che deve avere l'algoritmo.
- **minmax\_node.py**, questa classe implementa il nodo che verrà usato nell'algoritmo di ricerca. In essa sono contenuti i metodi che calcolano la funzione di valutazione per quella determinata mossa, la funzione di valutazione verrà discussa successivamente in maniera approfondita.

Le classi principali in C# sono invece:

- **BoardManagerUci.cs**, che gestisce tutta la situazione sulla scacchiera e aggiorna l'interfaccia.
- **BoardHighlights.cs**, è la classe che evidenzia le mosse disponibili quando il giocatore umano seleziona un pezzo.
- **Utility.cs**, questa classe traduce le mosse UCI in coordinate cartesiane e viceversa.

# Funzione di valutazione

La funzione di valutazione è l'elemento fondamentale nella ricerca della mossa migliore da fare. Esistono numerosi articoli scientifici che propongono vari modi per calcolare questa funzione. In questo software viene calcolata usando una versione simile a quella proposta da **Tomasz Michniewski**<sup>5</sup>. Questa funzione di valutazione calcola il valore basandosi in due tipi di valori: **valore material** e **valore posizionale**.

## Valore material

Il valore material viene calcolato assegnando un peso in **centipawn**<sup>6</sup> ad ogni pezzo:






Pezzo	Peso in centipawn
 Pedone	100
 Cavallo	320
 Alfiere	330
 Torre	500
 Regina	900

Tabella 1

Questi pesi sono stati calcolati per raggiungere i tre seguenti risultati:

1. Evitare lo scambio di un pezzo minore con tre pedoni
2. Incoraggiare l'engine ad avere una coppia di Alfieri
3. Evitare lo scambio di due pezzi minori per una torre e un pedone

Il punto (1) può essere soddisfatta semplicemente ponendo:

$$B > 3P$$

<sup>5</sup> <https://chessprogramming.wikispaces.com/Tomasz%20Michniewski>

<sup>6</sup> Il centipawn è l'unità di misura utilizzata negli scacchi come misura del vantaggio. Un centipawn è uguale a 1/100 di un pedone. Quindi 100 centipawn = 1 pedone

$$N > 3P$$

Dove  $B$ ,  $N$  e  $P$  sono rispettivamente i pesi da dare ad alfiere, cavallo e pedone.

Il punto (2) può essere ottenuto riformulando la (1) come:

$$B > N > 3P$$

Il punto (3) viene ottenuto tramite la relazione ( $R$  è il valore da dare alla torre):

$$B + N > R + 1.5P$$

Il valore 1.5 viene inserito per evitare la simmetria nel bilanciamento, in quanto più simmetria è presente e più probabile che partita finirà con un pareggio. Moltiplicando per 1.5, l'engine tenderà ad avere due pezzi minori contro una torre più un pedone oppure una torre più due pedoni. Come ultima cosa viene aggiunta una relazione basata sull'esperienza umana ( $Q$  è il valore da dare alla regina):

$$Q + P = 2R$$

Ricapitolando le relazioni da soddisfare sono:

$$B > N > 3P$$

$$B + N = R + 1.5P$$

$$Q + P = 2R$$

I valori che soddisfano queste relazioni sono quelli rappresentanti nella *Tabella 1*.

## Valore posizionale

Nel calcolo del valore posizionale, verranno dati dei bonus e dei malus proporzionali a quanto un pezzo sia in una buona posizione o meno. Ciò viene ottenuto usando delle tabelle 8x8 chiamate **piece-square tables** (nel software sono contenute nella classe **pieces\_tables.py**). In seguito saranno discusse le varie tabelle e saranno mostrate solo quelle per la valutazione del bianco, la valutazione del nero è ottenuta semplicemente considerando la tabella in modo speculare. La tabella usata per il pedone è la seguente:

0	0	0	0	0	0	0	0
50	50	50	50	50	50	50	50
10	10	20	30	30	20	10	10
5	5	10	25	25	10	5	5
0	0	0	30	30	0	0	0
5	5	-10	0	0	-10	5	5
5	10	10	-20	-20	10	10	5
0	0	0	0	0	0	0	0

*Tabella (Pedone) 2*

Tramite questa tabella vengono incoraggiati i pedoni ad avanzare cercando di preferire quelli centrali, in modo da avere un buon controllo del centro (risultato molto importante da raggiungere) e si scoraggia a far avanzare quelli laterali, soprattutto quelli che possono lasciare buchi per un eventuale attacco al re. La riga 7 è impostata tutta a 50, in quanto nella prossima mossa si può ottenere una promozione che farà ottenere un notevole vantaggio. Adesso si passerà ad analizzare la tabella relativa al cavallo, essa è mostrata nella *Tabella 3*. In questa tabella forziamo il cavallo ad occupare le case centrali cercando di evitare quelle laterali in quanto diminuirebbero il numero di mosse possibili del cavallo (soprattutto agli angoli) e rallenterebbero il movimento.

-50	-40	-30	-30	-30	-30	-40	-50
-40	-20	0	0	0	0	-20	-40
-30	0	10	15	15	10	0	-30
-30	5	15	20	20	15	5	-30
-30	0	15	20	20	15	0	-30
-30	5	10	15	15	10	5	-30
-40	-20	0	5	5	0	-20	-40
-50	-40	-30	-30	-30	-30	-40	-50

*Tabella (Cavallo) 3*

Nella *Tabella 4* è mostrata la tabella relativa all'alfiere. Similmente al cavallo, verranno penalizzate le caselle laterali e gli angoli in modo da avere più mobilità. Inoltre le case **b2** e **g2** hanno un valore leggermente più alto per favorire il **fianchetto**.

-20	-10	-10	-10	-10	-10	-10	-20
-10	0	0	0	0	0	0	-10
-10	0	5	10	10	5	0	-10
-10	5	5	10	10	5	5	-10
-10	0	10	10	10	10	0	-10
-10	10	10	10	10	10	10	-10
-10	20	0	0	0	0	20	-10
-20	-10	-10	-10	-10	-10	-10	-20

*Tabella (Alfiere) 4*

Per quanto riguarda la torre si cercherà di conquistare la fila 7 in modo da poter mangiare facilmente i pedoni. Inoltre le colonne **a** e **h** riceveranno un malus in quanto meno importante perché danno meno mobilità e all'inizio c'è il rischio di essere mangiati dalle torri nemiche, infine le case **d1** e **e1** ricevono un bonus per incoraggiare l'arrocco. Ciò è mostrato in *Tabella 5*.

0	0	0	0	0	0	0	0
5	10	10	10	10	10	10	5
-5	0	0	0	0	0	0	-5
-5	0	0	0	0	0	0	-5
-5	0	0	0	0	0	0	-5
-5	0	0	0	0	0	0	-5
-5	0	0	0	0	0	0	-5
0	0	0	10	10	0	0	0

*Tabella (Torre) 5*

Tramite la tabella della regina, si cercano di evitare i bordi, gli angoli e si cerca di ottenere il controllo del centro.

-20	-10	-10	-5	-5	-10	-10	-20
-10	0	0	0	0	0	0	-10
-10	0	5	5	5	5	0	-10
-5	0	5	5	5	5	0	-5
0	0	5	5	5	5	0	-5
-10	5	5	5	5	5	0	-10
-10	0	5	0	0	0	0	-10
-20	-10	-10	-5	-5	-10	-10	-20

*Tabella (Regina) 6*



Infine allo score finale della funzione vengono aggiunti i seguenti valori se le condizioni sono rispettate:

- È possibile arroccare: **50**
- Scacco: **200**
- Scacco matto: **250**

## Miglioramenti

I miglioramenti da fare alla funzione di valutazione sono molteplici anche per via della semplicità di quella implementata in questo progetto. Esistono migliaia di articoli scientifici che propongono euristiche per calcolare il valore di questa funzione, oltre quelli discussi in questa relazione, si possono aggiungere e migliorare i seguenti aspetti:

- I pesi sia dei singoli pezzi che delle posizioni possono essere calcolate tramite learning attraverso una rete neurale (AlphaZero)
- Miglioramento del fianchetto
- **Color weakness**, ovvero quando un giocatore non ha abbastanza controllo sulle case di un determinato colore
- Migliorare del controllo del centro
- Implementazione di tutte le aperture<sup>7</sup> conosciute in modo da avere un'ottima risposta nelle fasi iniziali di gioco
- Aggiunta di un database di situazioni di endgame

## Prestazioni

In questa sezione verranno analizzate le prestazioni del software. Il test è stato eseguito in un hardware con le seguenti caratteristiche hardware:

- Sistema operativo: Windows 10
- CPU: Intel Core i7-5500U 2.4 GHz
- RAM: 8 GB
- Scheda video: AMD Radeon R5 M330

Come prima cosa verranno analizzate le prestazioni utilizzando l'algoritmo **Minimax**, di seguito l'estratto di codice che lo implementa:

---

<sup>7</sup> il termine **apertura** indica la prima fase della partita - <https://chessprogramming.wikispaces.com/Opening%20Book>

```

def minimax(self, configuration, depth, maximizingPlayer):
    if depth == 0 or configuration.isCheckMate:
        evalMove = BestMove()
        evalMove.move = configuration
        evalMove.value = configuration.evaluation_function()
        return evalMove
    if maximizingPlayer:
        bestMove = BestMove()
        bestMove.value = -float("inf")
        bestMove.move = None
        for child in configuration.generate_children():
            move = self.minimax(child, depth - 1, False)
            if move.value > bestMove.value:
                bestMove = move
        return bestMove
    else:
        minMove = BestMove()
        minMove.value = float("inf")
        minMove.move = None
        for child in configuration.generate_children():
            move = self.minimax(child, depth - 1, True)
            if move.value < minMove.value:
                minMove = move
        return minMove

```

Qui di seguito è mostrata una sequenza di mosse e il tempo di esecuzione per esse da parte dell'AI con profondità settata a 3:

Mossa	Tempo di esecuzione
g8f6	16.6299999876
b8c6	33.5629999638
c6e5	44.8500001431
d7d6	66.5469999313
c8f5	84.5299999714
f5g6	103.084000111

f6d7	73.7939999104
e5c6	57.0710000992

*Tabella 7*

Come si può ben notare, le prime mosse hanno un tempo di esecuzione minore in quanto il **branching factor** è minore prima di sviluppare i vari pezzi. La media del tempo di esecuzione è all'incirca 52 secondi.

Per ottenere un notevole miglioramento delle prestazioni, viene implementata la **potatura alfa beta**. L'estratto di codice che implementa l'algoritmo è il seguente:

```
def alphabeta(self, configuration, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or configuration.isCheckMate:
        evalMove = BestMove()
        evalMove.move = configuration
        evalMove.value = configuration.evaluation_function()
        return evalMove
    if maximizingPlayer:
        bestMove = BestMove()
        bestMove.value = -float("inf")
        bestMove.Move = None
        for child in configuration.generate_children():
            move = self.alphabeta(child, depth-1, alpha, beta, False)
            if move.value > bestMove.value:
                bestMove = move
                alpha = move.value
            if beta <= alpha:
                break
        return bestMove
    else:
        minMove = BestMove()
        minMove.value = float("inf")
        minMove.Move = None
        for child in configuration.generate_children():
            move = self.alphabeta(child, depth-1, alpha, beta, True)
            if move.value < minMove.value:
                minMove = move
                beta = move.value
            if beta <= alpha:
                break
        return minMove
```

Nella seguente tabella verranno mostrati i tempi di esecuzione a seguito dell'aggiunta della potatura alfa beta:

Mossa	Tempo di esecuzione
<b>g8f6</b>	2.51099991798
<b>b8c6</b>	5.78099989891
<b>c6e5</b>	7.35800004005
<b>d7d5</b>	22.9470000267
<b>c8g4</b>	46.6270000935
<b>g4h5</b>	64.9499998093
<b>h5g6</b>	68.5510001183

I risultati sono decisamente migliorati in quanto in media il computer sta 30 secondi per rispondere. Le prestazioni potrebbero essere migliorate parallelizzando l'algoritmo, cercando di usare strutture dati e algoritmi efficienti per il calcolo della funzione di valutazione. Inoltre potrebbero essere introdotte le tabelle di endgame per ottenere un notevole aumento delle prestazioni nelle fasi finali del gioco, le librerie usate per questo software contengono già all'interno di esse le **Gaviota endgame tablebase**<sup>8</sup> e le **Syzygy endgame tablebase**<sup>9</sup>.

---

<sup>8</sup> <https://python-chess.readthedocs.io/en/latest/gaviota.html>

<sup>9</sup> <https://python-chess.readthedocs.io/en/latest/syzygy.html>