TERMINAL CHESS GAME

3J — A.S. 2024/2025

- Kovalyshyn Dmytriy
- Tognoni Filippo
- Pischettola Luca
- Biondi Leonida

Introduzione

▼ Storia del gioco

Il gioco degli scacchi ha origini antiche e si ritiene comunemente che sia nato in India intorno al VII secolo con il nome di *caturaṅga*, un predecessore degli scacchi moderni. Questo gioco si diffuse successivamente in Persia, dove prese il nome di *shatranj*.

Con la conquista islamica della Persia, gli scacchi vennero adottati dal mondo islamico e, attraverso i contatti culturali e commerciali, raggiunsero l'Europa meridionale.

Nel corso dei secoli, le regole e la forma del gioco vennero progressivamente modificate, fino a consolidarsi nella versione moderna intorno al XVI secolo, simile a quella oggi riconosciuta a livello internazionale.

▼ Regole del gioco

Struttura della scacchiera

La scacchiera è composta da una griglia 8×8, per un totale di 64 caselle, alternate nei colori chiaro e scuro. I pezzi bianchi occupano le due righe inferiori (righe 1 e 2), mentre quelli neri si trovano nelle due superiori (righe 7 e 8).

Il giocatore con i pezzi bianchi inizia sempre per primo.

Descrizione dei pezzi

Pedoni

I pedoni sono i pezzi più numerosi e si trovano disposti in seconda fila, davanti agli altri pezzi.

- Possono muoversi **di una casella in avanti**, oppure **di due caselle** se si trovano ancora nella loro posizione iniziale.
- **Non possono superare altri pezzi**.

- Possono **catturare** solo in diagonale, su una delle due caselle immediatamente davanti a destra o a sinistra.
- Quando un pedone raggiunge l'ultima riga della scacchiera, può essere promosso a qualsiasi altro pezzo, **eccetto un altro pedone o il re**.

Cavalli

I cavalli sono gli unici pezzi in grado di **saltare altri pezzi**. Si muovono con una combinazione di 2 caselle in una direzione e 1 in quella perpendicolare, formando una **"L"**.

Alfieri

Gli alfieri si muovono **in diagonale**, senza limitazioni di distanza, purché non ci siano ostacoli. **Non possono saltare altri pezzi**.

Torri

Le torri si muovono **in verticale o in orizzontale**, su qualsiasi distanza disponibile. Anche le torri **non possono superare altri pezzi**.

Regina

La regina è il pezzo più potente sulla scacchiera. Può muoversi **in verticale, orizzontale o diagonale**, su qualsiasi distanza disponibile. Come gli altri pezzi (esclusi i cavalli), **non può saltare altri pezzi**.

Re

Il re è il pezzo più importante. Può muoversi **di una sola casella** in qualsiasi direzione, ma **non può mai spostarsi su una casella controllata da un pezzo avversario**.

Se il re si trova sotto attacco (scacco), il giocatore **deve obbligatoriamente risolvere la situazione**, spostandolo, bloccando l'attacco o catturando il pezzo avversario.

Una mossa è considerata **illegale** se mette il proprio re sotto scacco (ad esempio, muovere un pezzo che protegge il re e lasciarlo esposto a un attacco).

Obiettivo del gioco

Lo scopo del gioco è mettere il re avversario **sotto scacco matto**, ovvero in una posizione tale che **nessuna mossa legale possa impedirne la cattura** al turno successivo.

Mosse speciali

En Passant

L'**en passant** è una cattura speciale dei pedoni. Se un pedone avversario avanza di due caselle dalla sua posizione iniziale e si ferma accanto a un proprio pedone, quest'ultimo può catturarlo **come se avesse mosso di una sola casella**, spostandosi diagonalmente dietro di lui.

Arrocco (Castling)

L'**arrocco** è una mossa speciale che coinvolge re e torre, utile per mettere il re al sicuro.

Può essere effettuato solo se:

- Né il re né la torre coinvolta si sono mai mossi;
- Tra i due non ci sono pezzi;
- Il re non è sotto scacco e non attraversa o finisce su una casella sotto attacco.

In tal caso:

- Il re si sposta di **due caselle** verso la torre;
- La torre si posiziona **accanto al re**, dalla parte opposta rispetto alla posizione originale.

Analisi

▼ Problema 1: Gestione File

Il programma è stato organizzato in modo da utilizzare 4 file differenti classificati in base al tipo di funzione:

- `Program.cs` Si occupa degli input/output del programma.
- `Move.cs` Presenta le funzioni per la gestione generale delle mosse.
- `Evaluation.cs` Presenta le funzioni per analizzare la scacchiera ed ottenere il punteggio generale.
- `Minimax.cs` Presenta le funzioni per navigare la struttura ad albero per calcolare la possa migliore per il computer.

Per scrivere il programma utilizzando file diversi ma sempre la stessa classe, abbiamo utilizzato le **classi parziali**.

```
public partial class Program
{...}
```

Questo ovviamente non è il modo ideale per organizzare un progetto. In un contesto reale, sarebbe preferibile utilizzare una classe separata per ciascun file.

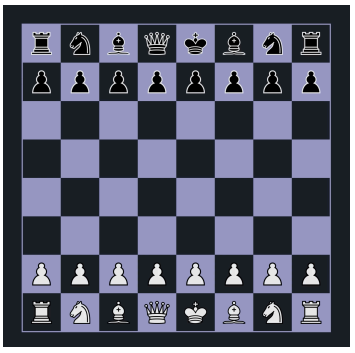
Tuttavia, trattandosi di un progetto scolastico e non avendo ancora affrontato formalmente lo studio delle classi, in C# abbiamo scelto di utilizzare le classi parziali per suddividere il codice in modo più accessibile.

▼ Problema 2: Rappresentazione Scacchiera

Per rappresentare la scacchiera sono state dichiarate **13 variabili** che rappresentano i vari pezzi, ed ognuna assegnato un valore:

```
public const int _  = 0;  // Empty Square
public const int WP = 1;  // White Pawn
public const int WN = 2;  // White Knight
public const int WB = 3;  // White Bishop
public const int WR = 4;  // White Rook
public const int WQ = 5;  // White Queen
public const int WK = 6;  // White King
public const int BP = 7;  // Black Pawn
public const int BN = 8;  // Black Knight
public const int BB = 9;  // Black Bishop
public const int BR = 10; // Black Rook
public const int BQ = 11; // Black Queen
public const int BK = 12; // Black King
```

Mentre per rappresentare e stampare la scacchiera è stata usata una **matrice quadrata 8×8.**



Implementazione in Codice

Per la risoluzione del problema consultare **Funzione 1:** `GetPosition()` **Dmytriy.**

▼ Problema 3: Supporto Caratteri Unicode

Non tutti i font supportano i caratteri **Unicode** utilizzati in questo programma per la visualizzazione dei singoli pezzi.

Per questo motivo, è necessario utilizzare un **font** nel terminale che supporti tutti i caratteri richiesti. Consigliamo l'uso di un font monospace compatibile con l'intero set di caratteri Unicode come **Deja Vu Sans Mono**.

Inoltre, affinché il terminale possa visualizzare correttamente i caratteri Unicode, è necessario aggiungere nel codice la seguente riga:

```
Console.OutputEncoding = Encoding.UTF8;
```

Infine, sarà necessario selezionare il font anche dalle impostazioni del terminale.

▼ Problema 4: Controllo Mossa Legale

Ogni pezzo ha un modo di muoversi unico, quindi è necessario verificare per ogni tipo di pezzo che il movimento effettuato dal giocatore o dal computer sia valido. Inoltre, una mossa è considerata illegale se, dopo averla eseguita, il re del giocatore che ha mosso si trova sotto scacco.

Un ulteriore problema da gestire è quando il giocatore tenta di muovere una casella vuota o fornisce input non validi, come coordinate fuori dalla scacchiera. Questi casi devono essere controllati e bloccati per evitare errori o comportamenti imprevisti durante il gioco.

Implementazione in Codice

Per la risoluzione del problema consultare **Funzione 3:** `LegalMove()` **Luca.**

▼ Problema 5: Valutazione Della Posizione

Come in qualunque gioco, o competizione, c'è sempre qualcuno che è in vantaggio o in svantaggio. Anche nel gioco degli scacchi accade questo; in base alla posizione dei pezzi sulla scacchiera si può determinare quale tra i due giocatori sta vincendo. Quindi guardando solamente lo stato corrente della scacchiera bisogna essere in grado di determinare quale giocatore è in vantaggio e di quanto.

Gestione Del Punteggio

Ogni pezzo ha un proprio valore preciso:

```
- Pawn:   1
- Knight:  3
- Bishop:  3
- Rook:    5
- Queen:   9
```

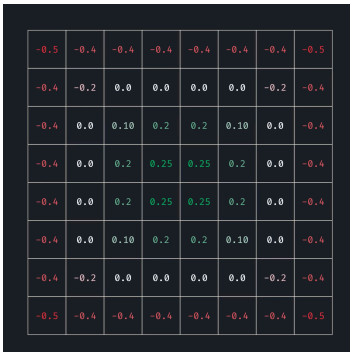
Possiamo quindi **sommare** i valori dei pezzi bianchi e **sottrarre** i valori dei pezzi neri.

Otteniamo così uno score che indica:

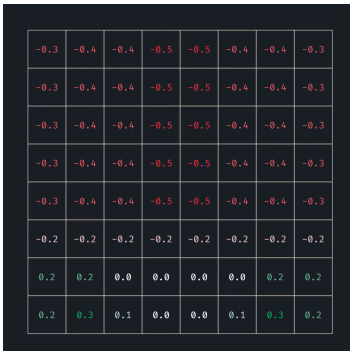
```
= 0 ⟶ Posizione di parità
> 0 ⟶ Bianco in vantaggio
< 0 ⟶ Nero in vantaggio
```

Possiamo inoltre considerare la **posizione** del pezzo sulla scacchiera ed aggiungere o sottrarre un valore in base alla sua posizione.

Questo perché tendenzialmente i pezzi al centro hanno maggiore controllo rispetto ai lati.



Invece per quanto riguarda il Re, egli tendenzialmente è più opportuno che stia ai lati dato che più protetto, perciò:



Anche se questo metodo per valutare la posizione può sembrare “hard-coded”, risulta comunque efficace per sviluppare un computer sufficientemente potente, adatto alle esigenze del nostro progetto.

Implementazione in Codice

Per la risoluzione del problema consultare **Funzione 5:** `Evaluation()` **Filippo.**

▼ Problema 6: Controllo Scacco Matto e Patta

Una partita di scacchi termina quando **il giocatore di turno non ha più alcuna mossa legale disponibile.** Questo può portare a due esiti distinti:

- **Scacco matto**: il re è sotto scacco e non esiste alcuna mossa che permetta di evitare la minaccia.
- **Patta (stallo)**: il re **non è sotto scacco**, ma **non esiste alcuna mossa legale disponibile** per il giocatore. In questo caso la partita termina in parità.

Un altro caso per cui viene dichiarata patta avviene quando il materiale non è sufficiente per effettuare uno stacco matto da entrambi i giocatori.

Soluzione

1. **Conta il numero di pezzi** presenti sulla scacchiera.
2. **Controlla se esiste almeno una mossa legale** per il giocatore di turno.
3. Se esiste **almeno una mossa è legale**, viene controllato che il materiale sia sufficiente.
4. Altrimenti se **nessuna mossa è legale**:
   - Verificare se il re del giocatore è **sotto scacco**.
     - Se **sì** → si tratta di **scacco matto**.
     - Se **no** → si tratta di **patta per stallo**.

Implementazione in Codice

Per la risoluzione del problema consultare **Funzione 7:** `UnderCheck()` **Filippo, Funzione 8:** `Checkmate()` **Filippo.**

▼ Problema 7: Gestione Mosse Speciali

Durante una partita di scacchi, ci sono tre mosse speciali che possono essere eseguite quando si verificano determinate condizioni:

- **Arrocco (Castling)**
- **En Passant**
- **Promozione**

Per il primo caso sono stati inseriti dei costrutti `if - else` all'interno della funzione `LegalKingMove()` (trovata a sua volta dentro alla funzione `LegalMove()`.

Il secondo, l' en passant, è controllato in maniera simile all'interno della sottofunzione `LegalPawnMove()`.

Per la promozione, è stata implementata una semplificazione che trasforma automaticamente il pedone in regina quando raggiunge l'ultima riga, essendo questa la scelta più comune e vantaggiosa nella maggior parte delle situazioni di gioco.

▼ Problema 8: Verifica Posizioni Sotto Attacco

Per controllare se una cella è sotto attacco sono state implementate numerose funzioni all'interno del file `Evaluation`:

Nome Funzione	Scopo
`AttackedByPawn()`	Verifica l'attacco da parte dei pedoni
`AttackedByKnight()`	Verifica l'attacco da parte dei cavalli
`AttackedByKing()`	Verifica l'attacco da parte del re
`AttackedVerticallyTop()`	Verifica l'attacco verticale dall'alto
`AttackedVerticallyBottom()`	Verifica l'attacco verticale dal basso
`AttackedVertically()`	Verifica l'attacco verticale complessivo
`AttackedHorizontallyLeft()`	Verifica l'attacco orizzontale da sinistra
`AttackedHorizontallyRight()`	Verifica l'attacco orizzontale da destra
`AttackedHorizontally()`	Verifica l'attacco orizzontale complessivo
`AttackedDiagonallyTopLeft()`	Verifica l'attacco diagonale dall'alto a sinistra
`AttackedDiagonallyTopRight()`	Verifica l'attacco diagonale dall'alto a destra
`AttackedDiagonallyBottomLeft()`	Verifica l'attacco diagonale dal basso a sinistra
`AttackedDiagonallyBottomRight()`	Verifica l'attacco diagonale dal basso a destra
`AttackedDiagonally()`	Verifica l'attacco diagonale complessivo
`AttackedBy()`	Funzioni generali per verificare gli attacchi

Queste funzioni esaminano se una data cella è sotto attacco controllando ogni possibile direzione di movimento dei pezzi avversari. Restituiscono true se la cella è minacciata o false se è al sicuro.

▼ Problema 9: Determinazione del Tipo di Patta

Più che "problema" si potrebbe definire impegnativo dato la moltitudine di tipi ci patta he esistono, per capire il tipo di patta sono stati utilizzati più costrutti `if - else` all'interno della funzione `Checkmate()` situata a sua volta all'interno del file `Evaluation`. I diversi tipi di patta sono:

- **Stallo**: non possono essere effettuate mosse, perché tutte porterebbero alla cattura del proprio re. Però a differenza dello scacco matto il re non è sotto scacco in questo momento.
- **Materiale insufficiente**: i pezzi non sono sufficienti per nessuno dei due giocatori per dare scacco matto all'avversario. Questo può accadere se sulla scacchiera sono presenti solo:
  - Re VS Re
  - Re e cavallo VS Re
  - Re e alfiere VS Re
  - Re e alfiere VS Re e alfiere (con gli alfieri su caselle dello stesso colore)

- **Tripla ripetizione**: una posizione viene ripetuta per tre volte durante la partita.

Implementazione

▼ Funzione 1: `GetPosition()` Dmytriy

Scopo

La funzione `GetPosition()` si occupa della stampa della scacchiera, iterando attraverso le righe e le colonne della matrice di gioco, scrivendo poi il pezzo giusto chiamando la funzione `WriteSquare()`.

Signature

```
static void GetPosition(bool white)
```

Parametri

- `white`: indica se la scacchiera deve essere stampata dalla parte del bianco.

Descrizione

1. **Ruota la scacchiera** in base al colore del giocatore, posizionando i pezzi del giocatore nella parte inferiore dello schermo.
2. Scrive le lettere e i numeri per indicare righe (1-8) e colonne (a-h).
3. **Evidenzia la casella** di partenza e di arrivo dell'ultima mossa per aumentarne la visibilità.

▼ Funzione 2: `Move()` Leonida

Scopo delle funzioni

Le funzioni `Move()` e `UndoMove()` gestiscono rispettivamente l'applicazione e l'annullamento delle mosse sulla scacchiera. Sono fondamentali sia per il gioco in tempo reale che per la simulazione all'interno dell'algoritmo Minimax.

Funzione `Move()`

Scopo

Esegue la mossa indicata, aggiornando la scacchiera e lo stato di gioco.

Signature

```
public static bool Move(
    int startX,
    int startY,
    int endX,
    int endY,
    bool isWhiteTurn,
    int promoteTo
)
```

Parametri

- `startX`, `startY`: coordinate della casella di partenza;
- `endX`, `endY`: coordinate della casella di destinazione;
- `isWhiteTurn`: indica se tocca al bianco;

- `promoteTo`: pezzo a cui promuovere un pedone (valido solo se la mossa è una promozione).

Descrizione

1. Verifica la legalità della mossa tramite `LegalMove()`.
2. Gestisce la cattura en passant:
   - Rimuove il pedone catturato nella posizione dietro la destinazione.
3. Gestisce la promozione:
   - Se un pedone raggiunge l'ultima traversa, viene trasformato nel pezzo specificato da `promoteTo`.
   - Controlli aggiuntivi assicurano che un bianco non promuova a pezzo nero e viceversa.
4. Gestisce l'arrocco:
   - Sposta anche la torre coinvolta.
5. Aggiorna i flag di arrocco e le coordinate en passant.
6. Aggiorna la scacchiera con il nuovo stato.

Valore di ritorno

- `true`: la mossa è stata applicata con successo;
- `false`: mossa non valida o promozione errata.

Funzione `UndoMove()`

Scopo

Ripristina lo stato precedente della scacchiera e delle variabili globali dopo una mossa temporanea.

Signature

```
static void UndoMove(
    int startX,
    int startY,
    int endX,
    int endY,
    int startPiece,
    int endPiece,
    bool oldWhiteCastleKing,
    bool oldWhiteCastleQueen,
    bool oldBlackCastleKing,
    bool oldBlackCastleQueen,
    int oldEnPassantX,
    int oldEnPassantY
)
```

Parametri

- `startX`, `startY`: coordinate della casella iniziale del pezzo;
- `endX`, `endY`: coordinate della casella in cui si era mosso il pezzo;
- `startPiece`: tipo di pezzo che ha effettuato la mossa;
- `endPiece`: pezzo eventualmente catturato;
- `oldWhiteCastleKing`, `oldWhiteCastleQueen`: stato originale del diritto all'arrocco per il bianco;
- `oldBlackCastleKing`, `oldBlackCastleQueen`: stato originale del diritto all'arrocco per il nero;
- `oldEnPassantX`, `oldEnPassantY`: coordinate originali per la cattura en passant.

Descrizione

1. Ripristina le pedine in caso di arrocco ed en passant.
2. Reimposta i flag di arrocco e le coordinate en passant.
3. Riporta i pezzi alle loro posizioni originali sulla scacchiera.

▼ Funzione 3: `LegalMove()` Luca

Scopo

La funzione `LegalMove` verifica la validità di una mossa sulla scacchiera controllando che il movimento richiesto rispetti le regole degli scacchi.

Signature

```
static bool LegalMove(int startX, int startY, int endX, int endY, bool isWhiteTurn)
```

Parametri

- `startX`, `startY`: coordinate della casella di partenza (colonna e riga);
- `endX`, `endY`: coordinate della casella di destinazione;
- `isWhiteTurn`: indica se è il turno del bianco (`true`) o del nero (`false`).

Descrizione

1. Validazione delle Coordinate

Vengono eseguiti controlli per:

- garantire che tutte le coordinate siano comprese tra 0 e 7;
- evitare che una mossa lasci il pezzo nella stessa posizione.

2. Verifica del Colore del Turno

Controlla che il pezzo presente nella casella di partenza appartenga al colore corretto del turno

3. Controllo del Tipo di Movimento

Viene determinato il tipo di pezzo e chiamata la sottofunzione corrispondente:

3.1 `LegalPawnMove`

- Movimento in avanti di una o due caselle dalla posizione iniziale;
- Cattura diagonale di un pezzo avversario;
- Cattura en passant: consentita se un pedone avversario ha appena effettuato un doppio passo accanto al pedone corrente;
- Verifica che il pedone non muova in diagonale su una casella vuota, a meno che non si tratti di en passant;

3.2 `LegalKnightMove`

- Movimento a forma di "L" (due caselle in una direzione, una nell'altra);
- I cavalli possono saltare sopra altri pezzi.

3.3 `LegalKingMove`

- Movimento di una sola casella in qualsiasi direzione;
- Arrocco re e torre: validato tramite `LegalCastle`, che verifica:
  - che il re e la torre coinvolti non siano mai stati mossi;
  - che le caselle tra re e torre siano libere;
  - che nessuna delle caselle attraversate dal re sia sotto attacco;

- che il re non sia attualmente sotto scacco.

3.4 `LegalSlidingMove`

- Valido per torri e regine che si muovono in orizzontale o verticale;
- Tutte le caselle tra partenza e destinazione devono essere vuote.

3.5 `LegalDiagonalMove`

- Valido per alfieri e regine che si muovono in diagonale;
- Le caselle tra partenza e destinazione devono essere libere.

4. Simulazione della Mossa e Controllo di Auto-Scacco

Dopo che la mossa è risultata conforme alle regole del pezzo:

- viene temporaneamente fatta scacchiera;
- si verifica con `UnderCheck(isWhiteTurn)` se il re del giocatore è sotto attacco;
- se il re è sotto scacco, la mossa è rifiutata;
- viene quindi ripristinato lo stato originale con `UndoMove`.

Valore di Ritorno

- `true`: se la mossa è valida;
- `false`: se la mossa è irregolare, non valida per il pezzo, o causa auto-scacco.

▼ Funzione 5: `Evaluation()` Filippo

Scopo

Assegnare un valore numerico alla posizione corrente.

Signature

```
static double Evaluation()
```

Descrizione

La funzione scorre tutte le caselle della scacchiera e in base al pezzo somma il valore assegnato ad esso se bianco e lo sottrae se nero. In seguito somma un valore in base alla posizione seguendo le tabelle sopra riportate per prendere in considerazione anche il posizionamento dei pezzi sulla scacchiera, e non solo la loro quantità.

Problemi riscontrati nell'implementazione

Nell'implementazione è stato necessario moltiplicare tutti i valori dei pezzi per `100` per poi solo alla fine dividere tutto per `100` riportando la valutazione con i valori prestabiliti. Questo per rimuovere la virgola che altrimenti portava ad un aumento significativo degli errori, fino a determinare un giocatore migliore dell'altro anche in una posizioni totalmente pari.

▼ Funzione 6: `AttackedBy()` Filippo

Scopo

Questa funzione controlla se una casella è attaccata da uno dei due giocatori. Divide il lavoro tramite sotto funzioni per maggiore leggibilità.

Signature

```
static bool AttackedBy(int x, int y, bool white, ref int fromX, ref int fromY)

static bool AttackedBy(int x, int y, bool white)
```

Descrizione

Le seguenti sotto-funzioni sono utilizzate per determinare se una casella è sotto attacco da parte di uno specifico giocatore. Ognuna si occupa di un tipo specifico di pedina, in base alle regole di movimento dei pezzi:

- `AttackedByPawn()`

  Verifica se un pedone avversario attacca la casella indicata, controllando le due caselle diagonali immediatamente adiacenti.

- `AttackedByKnight()`

  Controlla tutte le otto possibili posizioni da cui un cavallo potrebbe attaccare la casella corrente. Se uno di questi punti contiene un cavallo del giocatore specificato, la casella è considerata sotto attacco.

- `AttackedByKing()`

  Verifica le otto caselle adiacenti per determinare se un re avversario si trova in una posizione da cui potrebbe attaccare la casella corrente.

- `AttackedVertically()`

  Analizza la colonna verticale della casella, controllando:

  - Dall'alto verso il basso: partendo dalla casella immediatamente sopra e proseguendo finché non si incontra un pezzo o si esce dai limiti della scacchiera.
  - Dal basso verso l'alto: stesso procedimento in direzione opposta.

    La funzione è suddivisa in due sotto-funzioni che gestiscono le due direzioni.

- `AttackedHorizontally()`

  Segue la stessa logica di `AttackedVertically()`, ma esplora le righe orizzontali:

  - Verso sinistra e verso destra rispetto alla casella di partenza.

    Anche questa funzione è divisa in 2 sotto-funzioni per semplificare la gestione delle due direzioni.

- `AttackedDiagonally()`

  Applica lo stesso schema di ricerca delle funzioni precedenti per le quattro diagonali.

  Viene verificato se lungo ciascuna diagonale si trova un pezzo avversario in grado di minacciare la casella specificata.

▼ Funzione 7: `UnderCheck()` Filippo

Scopo

La funzione `UnderCheck()` verifica se il re del colore specificato è attualmente sotto scacco. Questa verifica è essenziale per la validazione delle mosse e per la logica di fine partita (scacco matto o patta).

Signature

```
static bool UnderCheck(bool isWhite)
```

Parametri

- `isWhite`: `true` se si vuole controllare se il re bianco è sotto scacco, `false` per il re nero.

Descrizione

1. **Identificazione del Re**
   - Scorre l'intera scacchiera per trovare la posizione del re del colore `isWhite`.
   - Se non viene trovato, la funzione può restituire `false` per evitare errori (caso eccezionale).
2. **Verifica di Attacco da Parte dell'Avversario**
   - Una volta localizzato il re, si utilizza la funzione `AttackedBy(x, y, !isWhite)` per verificare se la sua posizione è sotto attacco da parte di un pezzo avversario.
3. **Risultato**
   - Se la posizione è attaccata, restituisce `true` indicando che il re è sotto scacco.
   - Altrimenti, restituisce `false`.

▼ Funzione 8: `Checkmate()` Filippo

Scopo

La funzione `Checkmate()` controlla se uno dei due giocatori (specificato tramite un parametro di tipo `bool`) non può effettuare nessuna mossa. Se questo accade viene eseguito un ulteriore controllo, ovvero se il re è sotto scacco o meno; se il re è sotto scatto allora è scacco matto, altrimenti è patta per stallo, ovvero il re del giocatore non è catturabile, ma se facesse qualunque mossa questa porterebbe alla cattura del proprio re.

Se si fosse in una situazione in cui è possibile effettuare anche solo una mossa viene controllata la possibilità di non avere materiale sufficiente per effettuare uno scacco matto, questo succede se:

- re VS re
- re e cavallo VS re
- re e alfiere VS re
- re e alfiere VS re e alfiere (con gli alfieri su caselle dello stesso colore)

Signature

```
static bool Checkmate(bool white, ref bool draw)
```

Parametri

- `white` : `true` se si vuole controllare se il re bianco è sotto scacco matto/patta, `false` per il re nero.
- `draw` : `true` se la partita è terminata con una patta. Nota: **In caso di patta la funzione ritornerà** `false` **!**

Descrizione

1. **Conta pezzi sulla scacchiera**
   - Scorre l'intera scacchiera e aggiorna le variabili che tengono conto dei pezzi.
   - Gli alfieri sono divisi anche in base al colore della casella su cui si trovano. Questa distinzione è necessaria in seguito per determinare lo stato di patta per materiale insufficiente.
2. **Controlla se esiste almeno una mossa legale**
   - Mentre scorre la scacchiera determina se è un pezzo del giocatore (secondo il parametro `white`) e se può essere mosso.
3. **Se esiste almeno una mossa legale**
   - Mentre scorre la scacchiera determina se è un pezzo del giocatore (secondo il parametro `white`) e se può essere mosso.

Problemi nell'implementazione

La patta per la tripla ripetizione non è stata implementata perché richiedeva di ricordarsi tutte le posizioni precedenti.

▼ Funzione 9: `Minimax()` Dmytriy

Scopo

La funzione `Minimax` valuta ricorsivamente tutte le possibili sequenze di mosse fino a una profondità specificata, alternando i turni tra il computer e il giocatore. Utilizza il principio del Minimax insieme al taglio alpha-beta per determinare in modo efficiente la migliore mossa per il computer.

Signature

```
static double Minimax(
    int depth,
    double alpha,
    double beta,
    ref int bestStartX,
```

```
    ref int bestStartY,
    ref int bestEndX,
    ref int bestEndY,
    bool isMaximizing,
    bool playerWhite
)
```

Parametri

Nome	Tipo	Descrizione
`depth`	`int`	Profondità da esplorare.
`alpha`	`double`	Punteggio migliore del computer trovata fino ad un determinato punto
`beta`	`double`	Punteggio migliore del giocatore trovata fino ad un determinato punto
`bestStartX`	`ref int`	Output: riga della casella di partenza della mossa migliore trovata.
`bestStartY`	`ref int`	Output: colonna della casella di partenza della mossa migliore trovata.
`bestEndX`	`ref int`	Output: riga della casella di arrivo della mossa migliore trovata.
`bestEndY`	`ref int`	Output: colonna della casella di arrivo della mossa migliore trovata.
`isMaximizing`	`bool`	`true` se è il turno del computer (massimizzatore), `false` se del giocatore (minimizzatore).
`isPlayerWhite`	`bool`	`true` se il giocatore gioca con i bianchi, `false` con i neri (il computer gioca il colore opposto).

Valore di ritorno

- Ritorna un `double` che rappresenta il punteggio di valutazione della migliore sequenza di mosse trovata.
  - Valori positivi favoriscono il computer; valori negativi favoriscono il giocatore umano.
  - Valori di grande entità (±10000 ± depth) indicano scacchi matti.
  - Zero indica una valutazione in pareggio.

Descrizione dell'algoritmo

1. **Controllo di profondità**: se `depth` è 0, chiama `Evaluation()` per ottenere una valutazione della posizione corrente.
2. **Determinazione del colore**:
   - `currentPlayerIsWhite` è impostato in base a `isMaximizing`:
     - Se massimizzante, è il turno del computer; altrimenti, del giocatore.
3. **Controllo se la partita è terminata**:
   - Chiama `Checkmate(currentPlayerIsWhite, ref draw)`:
     - Se `draw` è `true`, restituisce 0.
     - Se scacco matto, restituisce un valore negativo o positivo (±10000 ± depth) in base alla variabile `isMaximizing` .
4. **Inizializzazione del punteggio**:
   - Se sta massimizzando: inizia con `-Infinito`.
   - Se sta minimizzando: inizia con `+Infinito`.
5. **Generazione mosse e ricorsione**:
   - Scorre tutte le caselle `(i, j)` della matrice (scacchiera).
   - Per ciascun pezzo, scorre tutte le possibili destinazioni `(k, l)`.
   - Se `Move(i, j, k, l, currentPlayerIsWhite)` restituisce `true`, chiama ricorsivamente `Minimax` con:
     - `depth - 1`
     - Alpha e beta aggiornati
     - `isMaximizing` invertito
     - Stesso `playerWhite`

- Dopo la ricorsione, chiama `UndoMove(i, j, k, l, capturedPiece)` per ripristinare lo stato della scacchiera.

6. **Confronto punteggi e taglio dell'albero**:
   - **Nodo massimizzante**:
     - Se `currentEvaluation > bestEvaluation`, aggiorna `bestEvaluation` e le coordinate della mossa.
     - Imposta `alpha = max(alpha, bestEvaluation)`.
   - **Nodo minimizzante**:
     - Se `currentEvaluation < bestEvaluation`, aggiorna `bestEvaluation` e le coordinate della mossa.
     - Imposta `beta = min(beta, bestEvaluation)`.
   - Se `beta <= alpha`, interrompe i cicli per il taglio.
7. **Restituzione** del miglior punteggio trovato.