UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

TESI DI LAUREA

Dalla αlfa alla βeta

RELATORE

Prof. **Fabio Palomba**

Università degli studi di Salerno

CANDIDATO

Giuseppe Pagano

Matricola: 0512106337

Anno Accademico 2021-2022

INSERIRE QUI UNA DEDICA O UNA CITAZIONE

Sommario

INSERIRE ABSTRACT

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

Gli scacchi sono un gioco di strategia deterministico a somma zero e ad informazione completa che si svolge su una tavola quadrata formata da 64 caselle ,di due colori alternati, detta scacchiera sulla quale ogni giocatore contraddistinto da uno di due colori nero o bianco, dispone di 16 pezzi:un re, regina, due alfieri, due cavalli, due torri e otto pedoni. obiettivo del gioco è dare scacco matto, ovvero minacciare la cattura del re avversario mentre esso non ha modo di rimuovere il re dalla sua posizione di pericolo alla sua prossima semimossa. Ulteriori informazioni verranno fornite nei successivi capitoli quando maggiori conoscenze di teoria si riveleranno necessarie per poter procedere allo sviluppo del motore.

1.2 Motivazioni e obiettivi

Gli scacchi,gioco nato in india attorno al 600 d.C,da gioco utilizzato nelle corti aristocratiche per rappresentare rapporti di potere a campo di battaglia tra uomo e macchina in uno dei primi e più famosi tentativi di far superare ad una macchina l'intelletto umano (Kasparov vs Deep Blue 1996-1997) ,gli scacchi , non hanno mai fallito nel saper cattivare l'attenzione del grande pubblico nonostante abbiano ormai più di 1000 anni sulle spalle.
Quello che agli occhi di un profano potrebbe sembrare un fenomeno stranissimo è in realtà di facile spiegazione se ci si concentra su una delle caratteristiche fondamentali del gioco degli

scacchi questa caratteristica è la **complessità**,in una partita di scacchi fin dalla prima semimossa sono possibili 20 scelte per la seconda semimossa il totale di possibili combinazioni sale a 400, dopo 5 semimosse avremo 119,060,324 possibili risposte, le possibili mosse di una partita si stimano attorno alle 2^{155}.
Con uno spazio di ricerca cosi grande non dovrebbe stupire sapere che è da quando esistono i computer che si cerca un modo di sfruttare la loro potenza di calcolo nel mondo degli scacchi.
La nascita degli scacchi computazionali si deve al lavoro di Claude Shannon, famoso per i suoi innumerevoli contributi al campo della teoria dell'informazione,egli, con il suo paper "Programming a Computer for Playing Chess" del 1950 ha gettato le basi per quello che oggi è il campo conosciuto come scacchi computazionali.
Questa tesi nasce dalla volontà di esplorare questo vasto e interessante campo dell'informatica,e dal voler creare un testo in grado di guidare chiunque lo legga nella creazione di un motore scacchistico spiegando tutte le fasi della progettazione ed illustrando le possibili scelte che condizionano l'efficienza di un motore ,dato che la letteratura su questo fronte è non particolarmente florida e soprattutto quasi esclusivamente in lingua inglese.

1.3 Risultati ottenuti

1.4 Struttura della tesi

1.5 Prefazione

Lo sviluppo di un motore scacchistico è fortemente influenzato dalle scelte progettuali, una di queste è il linguaggio di programmazione che si vuole utilizzare, le prestazioni di un motore possono essere fortemente influenzate dalla natura del linguaggio di programmazione, in particolare l'utilizzo di un linguaggio interpretato e non compilato può impattare notevolmente sulla velocità con la quale il nostro motore è in grado di elaborare le milioni di posizioni con le quali dovrà avere a che fare in una singola partita. Tutti gli esempi di codice all'interno di questa tesi saranno scritti nel linguaggio C, si consiglia quindi di avere almeno una minima familiarità con tale linguaggio.Si segnalano comunque diversi tool per il linguaggio python per chi volesse approcciarsi a questo mondo utilizzando un linguaggio più beginner friendly quali:

- **python-chess**: una libreria di python che contiene funzioni di libreria per la rappresentazione di scacchiera e pezzi e per la generazione e validazione delle mosse,utile se ci si

vuole concentrare esclusivamente sulla parte di ricerca e di valutazione di un motore scacchistico.

- **Sunfish**: un motore scacchistico per principianti scritto nel linguaggio python che in sole 111 linee di codice illustra ,in maniera semplificata, l'implementazione della gran parte dei concetti chiave di un motore scacchistico.

CAPITOLO 2

Rappresentazione della scacchiera e dei pezzi

In questo capitolo viene mostrato passo passo un procedimento guida alla realizzazione di un motore scacchistico

2.1 Introduzione

Il primo passo dello sviluppo di un motore scacchistico è decidere come si vuole rappresentare la scacchiera , si tratta di una scelta fondamentale,non solo perché in seguito ci permetterà di testare le funzioni che andremo a implementare, ma anche perché è nella scacchiera che ,generalmente, viene conservato lo stato generale della partita.[1] Inoltre il tipo di codifica può influenzare la rapidità e la facilità col quale possiamo accedere alle informazioni sullo stato corrente dei pezzi e come vedremo in seguito è in grado di influenzare funzioni come la generazione delle mosse. Non è raro per motori scacchistici particolarmente complessi l'utilizzo di più tipi di board in base al tipo di informazione da conservare e all'utilizzo che se ne vuole fare. Per la rappresentazione di una scacchiera sono chiaramente possibili moltissime scelte, di seguito verranno illustrate alcune tra le più popolari ed utilizzate.

2.2 Rappresentazione della scacchiera Pezzocentrica

Si definisce rappresentazione pezzocentrica, un qualsiasi tipo di rappresentazione della scacchiera che mantiene liste array o set dei pezzi attualmente presenti sulla scacchiera con annesse le informazioni sulle caselle da essi occupate. Le rappresentazioni più comuni sono:

Piece-Lists

liste o array di ogni pezzo sulla scacchiera,ogni elemento della lista o dell'array associa un pezzo alla casella che esso occupa .le caratteristiche di ogni pezzo (colore,tipo etc) possono essere associate all'indice dell'array in cui si trovano o essere presenti in ulteriori array o liste esterne.

Bitboards

Una Bitboard è una struttura dati specifica per i giochi da tavolo, si tratta in sostanza di una struttura dati in grado di immagazzinare lo stato di ogni casella della scacchiera all'interno di una parola [2] di 64 bit. Vediamo un esempio pratico, immaginiamo di avere una scacchiera che si trova nello stato di default di inizio partita:

[1] per stato di una partita si intendono informazioni come informazioni su chi ha diritto di muovere, i permessi di arrocco,lo stato della regola delle 50 mosse etc

[2] Una parola è un gruppo di bit di una determinata dimensione che sono gestiti come unità dal set di istruzioni o dall'hardware di un processore

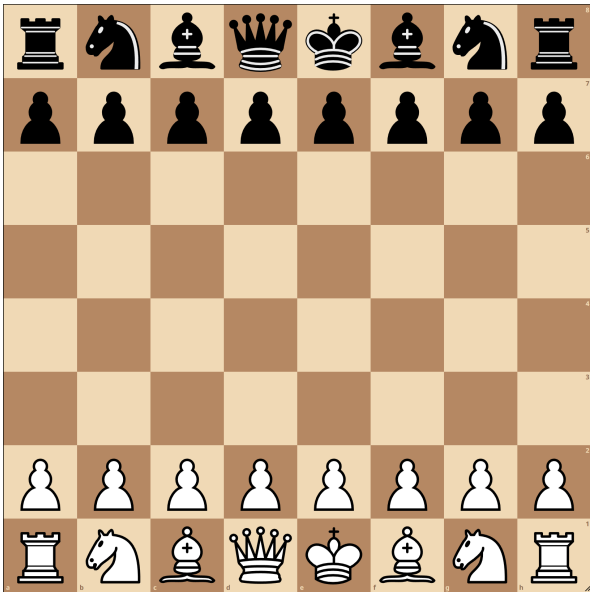

Figura 2.1: Scacchiera nella posizione iniziale di gioco

Una bitboard tipica è quella che ci permette di sapere in quali caselle è presente un pedone nero,per costruirla, operando casella per casella, ci poniamo una domanda "in questa casella è presente un pedone nero?" se si allora quella casella viene marcata con un 1 , altrimenti viene marcata con uno 0, il risultato di questa traduzione diventa in questo caso: La bitboard

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figura 2.2: bitboard

che codifica questa informazione sarà quindi la parola di 64 bit 00000000 11111111 00000000 00000000 00000000 00000000 00000000 00000000

Piece-Sets

rappresentazione con set con un bit per ogni pezzo dentro una parola a 32 bit o 2 parole a 16 bit per ogni lato. i Piece-sets hanno delle somiglianze con le bitboards, ma ogni bit del set non è direttamente correlato ad una casella, ma ad un indice dentro ad una piece-list. Spesso la bit-position di un piece-set implica , di che tipo e colore il pezzo è. - mentre le bitboards solitamente mantengono set distinti per pezzi diversi.

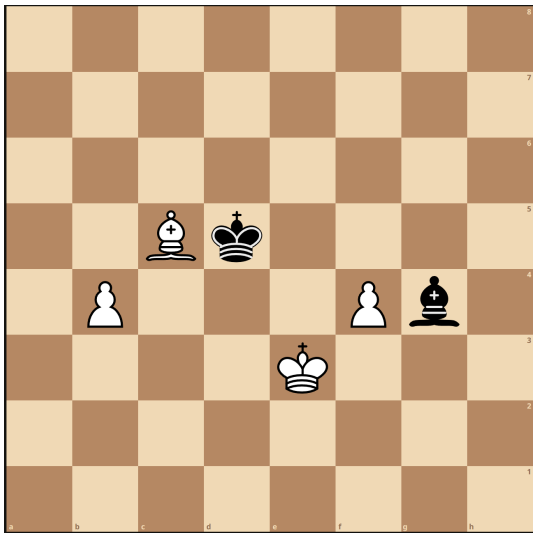

Figura 2.3: Scacchiera nella classica posizione di finale con alfieri di colore opposto

Figura 2.4: Rappresentazione set-wise della scacchiera in figura 2.3

2.3 Rappresentazione della scacchiera Casellocentrica

La rappresentazione casellocentrica mantiene un associazione inversa rispetto a quella pezzocentrica, per ogni casella conserviamo in memoria se è vuota o occupata da un pezzo in particolare. La macro-categoria di rappresentazione più comune è la Mailbox:

Mailbox

La rappresentazione Mailbox è una rappresentazione casellocentrica dove la codifica di ogni casella risiede in una struttura dati che permette l'accesso casuale ,solitamente si utilizza un array con l'indice che codifica dal numero della casella in array monodimensionali o dalla coppia traversa/colonna[3] in array bidimensionali. Il nome deriva dall'associazione di ogni indice al concetto di "indirizzo" di una casella postale.Le implementazioni più famose e comuni del concetto di Mailbox sono la 8x8 Board e la 10x12 Board.

8x8 Board

Una board 8x8,figura 2.5 è una rappresentazione pezzocentrica consistente o in un array bidimensionale di bytes o interi, contenenti rappresentazioni codificate per i pezzi e per la casella vuota, con i due indici ricavati dalla coppia traversa/colonna che identifica la casella sulla scacchiera , o più comunemente un array monodimensionale con indici da 0 a 63,uno per ogni casella della scacchiera. Questo tipo di rappresentazione è usata spesso come rappresentazione ridondante all'interno di programmi che utilizzano bitboards per individuare se e quali pezzi sono presenti su una casella in maniera efficiente.

10x12 Board

Una board 10x12 contorna una board 8x8 con traverse e colonne sentinelle per individuare indici al di fuori della scacchiera durante la generazione delle mosse

[3] termini scacchistici per indicare le righe e le colonne della scacchiera

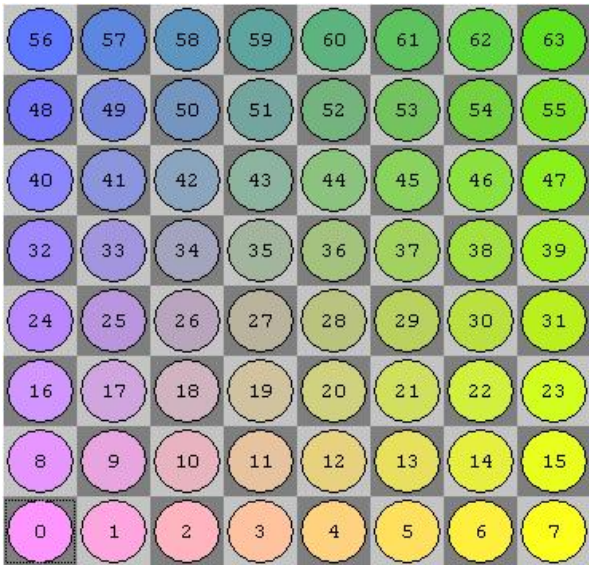

Figura 2.5: board numbering

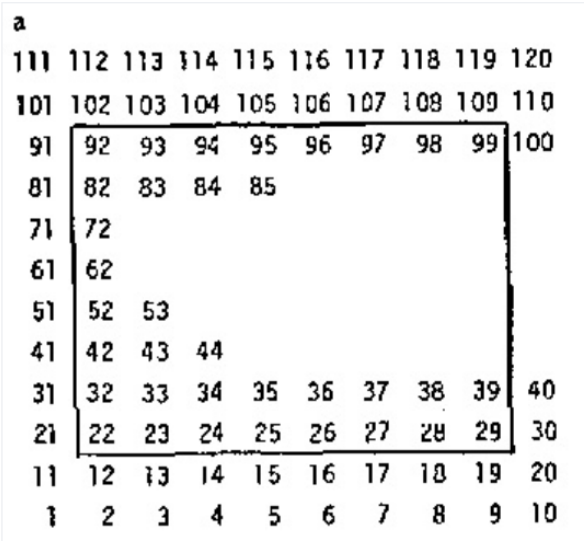

Figura 2.6: Rappresentazione di una board 10x12,notare come le 2 traverse extra sopra e sotto siano necessarie per evitare accessi out of bounds in caso di cavalli posizionati sulla prima o ottava fila

2.4 Rappresentazione dei pezzi

una volta scelto il tipo di rappresentazione della scacchiera si può iniziare a pensare alla rappresentazione dei pezzi, anche se può sembrare controintuitivo i pezzi non hanno bisogno di una struttura elaborata,la generazione delle possibili mosse verrà gestita nella move generation ed il loro spostamento all'interno della scacchiera verrà gestito dalla funzione MakeMove (approfondimenti riguardo questi due argomenti sono presenti nel capitolo 3),per i pezzi quindi abbiamo bisogno di una rappresentazione semplice, di facile interpretazione e che occupi poco spazio.
Rappresentazioni molto comuni sono quella tramite interi, dove ad ogni tipo di pezzo viene assegnato un numero che funge da identificativo univoco e quella tramite caratteri, dove ad ogni tipo di pezzo viene assegnato un carattere che lo identifica.

2.5 Esempio di implementazione

Nel motore che andremo a realizzare verrà adottato un approccio ibrido,con bitboards utilizzate per conservare le posizioni dei pedoni,una board 10x12 per tenere traccia di tutti i pezzi in gioco e delle piece-lists per effettuare più rapidamente operazioni di lookup, la codifica dei pezzi avverrà tramite l'utilizzo di interi e il linguaggio utilizzato come già specificato precedentemente sarà il C.

Codifica dei pezzi

La codifica dei pezzi è realizzata tramite interi,per un uso più agevole all'interno del codice però, si consiglia la realizzazione di un enum col quale mappare gli interi a dei simboli più facili da ricordare mnemonicamente.Si approfitta di questa sezione anche per introdurre ulteriori enum utili per descrivere costanti all'interno del motore.

```
enum { EMPTY, WP, WN, WB, WR, WQ, WK, BP, BN, BB, BR, BQ, BK };
enum { FILE_A, FILE_B, FILE_C, FILE_D, FILE_E, FILE_F, FILE_G, FILE_H, FILE_NONE };
enum { RANK_1, RANK_2, RANK_3, RANK_4, RANK_5, RANK_6, RANK_7, RANK_8, RANK_NONE };
enum { WHITE, BLACK, BOTH };
enum { FALSE, TRUE };
```

Figura 2.7: set di enum consigliato per lo svolgimento di questo progetto

Inoltre,essendo un motore scacchistico un progetto che predilige la rapidità di esecuzione più di ogni altra cosa, anche se sconsigliabile in quasi ogni altro contesto,conserviamo i

dati che caratterizzano i pezzi,sulla base del loro indice,in un insieme di array che verrà condiviso come insieme di variabili esterne dall'intero progetto, permettendo un lookup in tempo costante di tali informazioni.Per fare ciò adoperiamo un file header che conterrà le dichiarazioni ed un unico file sorgente dove saranno presenti le definizioni.Definiamo infine delle macro per rendere il codice più pulito e leggibile.

```
extern const char PieceChar[14];
extern const char SideChar[3];
extern const char RankChar[8];
extern const char FileChar[8];
extern const int PieceBig[13];
extern const int PieceMaj[13];
extern const int PieceMin[13];
extern const int PieceVal[13];
extern const int PieceCol[13];

extern const int PieceKnight[13];
extern const int PieceKing[13];
extern const int PieceRookQueen[13];
extern const int PieceBishopQueen[13];
```

Figura 2.8: PieceData.h

```
#pragma once
#include "Board.h"
#include "PieceData.h"
const char PieceChar[] = ".PNBRQKpnbrqk";
const char SideChar[] = "wb-";
const char RankChar[] = "12345678";
const char FileChar[] = "abcdefgh";
const int PieceBig[13] = { FALSE, FALSE, TRUE, TRUE, TRUE, TRUE,
                           FALSE, FALSE, TRUE, TRUE, TRUE, TRUE, FALSE };
const int PieceMaj[13] = { FALSE, FALSE, FALSE, FALSE, TRUE, TRUE,
                           FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE };
const int PieceMin[13] = { FALSE, FALSE, TRUE, TRUE, FALSE, FALSE,
                           FALSE, FALSE, TRUE, TRUE, FALSE, FALSE, FALSE };
const int PieceVal[13] = { 0, 100, 325, 325, 500, 975, 50000,
                           100, 325, 325, 500, 975, 50000 };
const int PieceCol[13] = { BOTH, WHITE, WHITE, WHITE, WHITE, WHITE, WHITE,
                           BLACK, BLACK, BLACK, BLACK, BLACK, BLACK };
const int PieceKnight[13] = { FALSE, FALSE, TRUE, FALSE, FALSE, FALSE, FALSE,
                              FALSE, TRUE, FALSE, FALSE, FALSE, FALSE };
const int PieceKing[13] = { FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE,
                            FALSE, FALSE, FALSE, FALSE, FALSE, TRUE };
const int PieceRookQueen[13] = { FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE,
                                 FALSE, FALSE, FALSE, TRUE, TRUE, FALSE };
const int PieceBishopQueen[13] = { FALSE, FALSE, FALSE, TRUE, FALSE, TRUE, FALSE,
                                   FALSE, FALSE, TRUE, FALSE, TRUE, FALSE };
const int PieceSlides[13] = { FALSE,FALSE,FALSE,TRUE,TRUE,TRUE,FALSE,
                              FALSE,FALSE,TRUE,TRUE,TRUE,FALSE };

#define IsBQ(p) (PieceBishopQueen[(p)])
#define IsRQ(p) (PieceRookQueen[(p)])
#define IsKn(p) (PieceKnight[(p)])
#define IsKi(p) (PieceKing[(p)])
```

Figura 2.9: PieceData.c

Realizzare una bitboard

Come precedentemente spiegato, una bitboard non è altro che una parola di 64 bit, utilizzando il linguaggio C ci basta definire una Bitboard come un unsigned long long int come nella figura 2.10 riga 2. Oltre alla dichiarazione del tipo bitboard possiamo notare la dichiarazione di alcune funzioni indispensabili per operare sulla stessa, le firme sono disponibili nella figura 2.10 ,mentre nella figura 2.11 possiamo vedere delle implementazioni di esempio. Nella figura 2.10 troviamo anche due macro, SETBIT e CLRBIT,che settano ad 1 o 0 uno specifico bit di una bitboard.

```
1  #pragma once
2  typedef unsigned long long int Bitboard;
3  extern void PrintBitBoard(Bitboard bb);
4  extern int Pop(Bitboard* bitboard);
5  extern int Count(Bitboard b);
6
7  #define CLRBIT(bb,sq) ((bb) &= ~(1ULL << sq))
8  #define SETBIT(bb,sq) ((bb) |= 1ULL<<sq)
9
```

Figura 2.10: bitboard.h

```c
#include "stdio.h"
#include "Board.h"
#include "bitboard.h"
const int BitTable[64] = {
  63, 30, 3, 32, 25, 41, 22, 33, 15, 50, 42, 13, 11, 53, 19, 34, 61, 29, 2,
  51, 21, 43, 45, 10, 18, 47, 1, 54, 9, 57, 0, 35, 62, 31, 40, 4, 49, 5, 52,
  26, 60, 6, 23, 44, 46, 27, 56, 16, 7, 39, 48, 24, 59, 14, 12, 55, 38, 28,
  58, 20, 37, 17, 36, 8
};

int Pop(Bitboard* bb) {
    Bitboard b = *bb ^ (*bb - 1);
    unsigned int fold = (unsigned)((b & 0xffffffff) ^ (b >> 32));
    *bb &= (*bb - 1);
    return BitTable[(fold * 0x783a9b23) >> 26];
}


int Count(Bitboard b)
{
    int r;
    for (r = 0;b;r++, b &= b - 1);
    return r;
}


void PrintBitBoard(Bitboard bitboard)
{
    Bitboard shiftME = 1ULL;
    int rank= 0;
    int file = 0;
    int sq = 0;
    int sq64 = 0;

    printf("\n");
    for (rank = RANK_8;rank >= RANK_1;--rank)
    {
        for (file = FILE_A;file <= FILE_H;++file) {
            sq = FR2SQ(file, rank);
            sq64 = Sq64[sq];
            if ((shiftME << sq64) & bitboard)
                printf("X");
            else
                printf("-");
        }
        printf("\n");

    }
    printf("\n\n");
}
```

Figura 2.11: bitboard.c

In particolare PrintbitBoard ci permette di visualizzare in output una bitboard,Pop restituisce l'indice del primo bit impostato a 1 nella bitboard (da meno a più significativo) e lo setta a 0 e Count restituisce il numero di bit settati a 1 di una bitboard. l'array BitTable è utilizzato per la fase di bitscan (la ricerca del primo bit settato ad 1 meno significativo) e risulta troppo complesso da spiegare in quella che è una guida introduttiva,per approfondire si consiglia la lettura della pagina sulla bitscan della ChessProgrammingWiki.

Realizzare la scacchiera

Iniziamo con la definizione di alcune costanti che verranno utilizzate nella creazione della scacchiera:

```c
#ifndef Board_H
#define Board_H
#include "stdlib.h"
#include "bitboard.h"
#define BRD_SQ_NUM 120
#define NAME "Chess Engine"
#define MAXGAMEMOVES 2048
#define MAXPOSITIONMOVES 256
#define START_FEN "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"
```

Figura 2.12: board-costants.png

- **Include guard**:riga 1-2 direttiva rivolta al preprocessore usata nel file header per evitare problemi di doppia definizione in fase di linking.
- **Include(s)**: inclusione delle librerie necessarie alla realizzazione della scacchiera.

- **BRD_SQ_NUM**:numero di casella della scacchiera, nel caso di una rappresentazione 10x12 sono 120.
- **(NAME)**: Nome scelto per il motore scacchistico.
- **MAXGAMEMOVES**: Massimo numero di mosse in una partita,il numero più alto di mosse mai registrato si attesta intorno alle 1000 mosse,2048 quindi è un buon limite .
- **(MAXPOSITIONMOVES)**: massimo numero di mosse eseguibili a partire da una singola posizione,è definibile anche come la massima profondità di ricerca a partire da un nodo N,dove il nodo N è una data posizione.
- **START_FEN**: notazione Forsyth-Edwards di una scacchiera ad inizio partita, per approfondire si legga https://en.wikipedia.org/wiki/Forsyth-Edwards_Notation.

Vediamo ora il codice che implementa la scacchiera,come detto nell'introduzione, la scacchiera oltre a contenere i pezzi contiene anche le informazioni sullo stato delle partita,vedremo quindi anche come gestire questo tipo di dati.

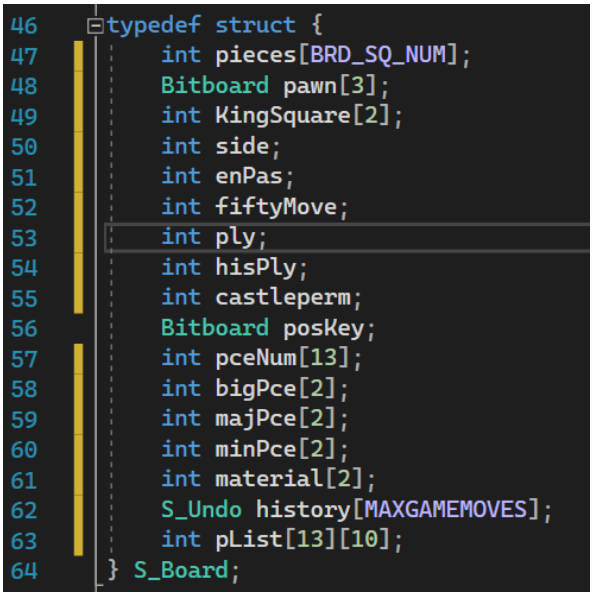

Figura 2.13: board-code.png

- **pieces**:è un array che per ogni casella della scacchiera memorizza se c'è un pezzo, o se la casella è non valida (è quindi una delle caselle "sentinella" proprie della rappresentazione 10x12).
- **pawn**: è un array di 3 Bitboard per memorizzare la posizione dei pedoni bianchi,neri e di entrambi combinati.

- **KingSquare**:è un array che contiene la posizione del re bianco e del re nero,utile per velocizzare delle operazioni di lookup fondamentali in casi nei quali la posizione del re sulla scacchiera gioca un ruolo fondamentale.
- **side**: intero che codifica quale lato può muovere, gli indici si basano sul enum della figura 2.7.
- **enPas**:intero che codifica se è possibile una cattura en passant ed in quale casella, maggiori dettagli nel paragrafo 2.7.
- **(fiftyMove)**: Contatore per la regola delle 50 mosse.
- **ply**: numero di semimosse in una search instance
- **hisPly**:numero totale di semimosse.
- **castleperm**: intero che codifica i permessi di arrocco tramite una rappresentazione sotto forma di stringa di bit, maggiori dettagli nel paragrafo 2.6
- **poskey**: chiave hash unica che codifica la posizione attualmente presente sulla scacchiera,utile per il controllo delle posizioni ripetute.
- **pceNum**: un array che restituisce il numero dei pezzi di quel tipo sulla scacchiera,con gli indici codificati come nell'enum della figura 2.7,ie: pceNum[2] restituisce il numero di cavalli bianchi.
- **bigPce**: array che restituisce il numero di pezzi grandi[4] ancora in gioco per entrambi i colori con gli indici codificati come nell'enum della figura 2.7.
- **majPce**:array che restituisce il numero di pezzi maggiori[5] ancora in gioco per entrambi i colori con gli indici codificati come nell'enum della figura 2.7.
- **minPce**: array che restituisce il numero di pezzi minori[6] ancora in gioco per entrambi i colori con gli indici codificati codificati come nell'enum della figura 2.7.
- **material**: mostra il valore totale del materiale di ogni giocatore con gli indici codificati come nell'enum della figura 2.7.

[4] i pezzi cosiddetti "grandi" sono tutti i pezzi ad esclusione del pedone e del re
[5] i pezzi cosiddetti "maggiori" sono la regina e la torre
[6] i pezzi cosiddetti "minori" sono il cavallo e l'alfiere

- **history**: array di S_Undo Struttura dati che contiene ogni mossa effettuata e lo stato della scacchiera prima che quella mossa venisse fatta,fondamentale per potere effettuare il rollback della scacchiera ad uno stato precedente, maggiori dettagli nel paragrafo 2.8
- **pList**: un array bidimensionale che per ogni tipo pezzo memorizza la posizione di ogni istanza di quel pezzo, fino ad un massimo di 10 possibili istanze (il massimo teorico di copie di un pezzo negli scacchi è 10), il primo indice varia da 0 a 12 come nell'enum della figura 2.7, e codifica ogni tipo possibile di pezzo,il secondo indice indica quale dei 10 pezzi possibili stiamo cercando,non è garantita nessuna forma di ordinamento.

Definiamo un enum per poter accedere alle caselle della scacchiera in maniera più naturale

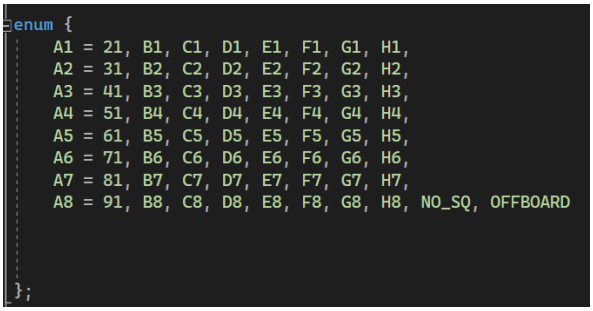

Figura 2.14: board-enum.png

Infine definiamo,come visto nel paragrafo 2.5, degli array ai quali sarà possibile accedere globalmente,per effettuare in tempo costante il lookup delle conversioni dell'indice di una casella da una rappresentazione 10x12 ad una 8x8 e viceversa, e ,della colonna e della traversa di appartenenza di una casella a partire dall'indice della stessa, la dichiarazione è effettuata in Board_Data.h mentre la definizione è in init.c,una raccolta di funzioni che si occupano di inizializzare il programma all'avvio.

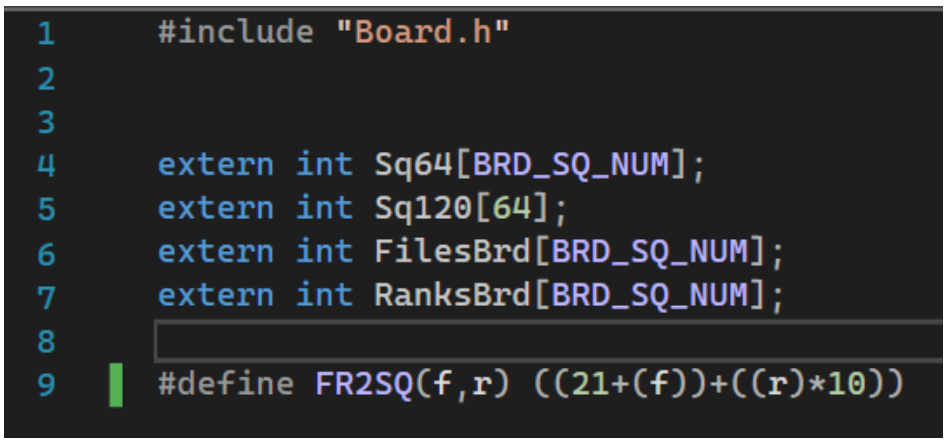

Figura 2.15: Board_Data.h

Inizializzare la scacchiera all'avvio della partita

Abbiamo precedentemente visto che la scacchiera fa un ampio utilizzo di raccolte di dati esterne, per semplificare il processo di inizializzazione di queste componenti creeremo un insieme di funzioni che si occupa di inizializzare atomicamente i vari componenti, tutte queste funzioni verranno dichiarate nel header init.h ed implementate nel file sorgente init.c

```c
#include "Board.h"
#include "stdlib.h"


//init.c
void initHashKeys();
void InitSq120To64();
void InitFilesRanksBrd();
void AllInit();


// *MACROS*//

#define RAND_64 ( (Bitboard)  rand() | \
 (Bitboard) rand()<<15 | \
(Bitboard) rand()<<30 | \
(Bitboard) rand()<<45 | \
((Bitboard) rand() & 0xf) <<60)

```

Figura 2.16: init.h

```c
#include "Board.h"
#include "stdlib.h"
#include "init.h"
#include "Board_Data.h"


int Sq64[BRD_SQ_NUM];
int Sq120[64];
int FilesBrd[BRD_SQ_NUM];
int RanksBrd[BRD_SQ_NUM];
Bitboard PieceKeys[13][120];
Bitboard SideKey;
Bitboard CastleKeys[16];
```

Figura 2.17: definizione dei dati esterni in init.c pre inizializzazione

initHashKeys

InitHashKey inizializza le key univoche randomiche per ogni pezzo in ogni posizione,questa operazione viene effettuata ogni volta ed una volta sola all'avvio del programma per far si che le chiavi siano uniche nella sessione corrente ma internamente consistenti durante il ciclo di vita del software.

```c
void initHashKeys() {
    int Typeindex = 0;
    int Numberindex = 0;
    for (Typeindex = 0;Typeindex < 13;++Typeindex)
    {
        for (Numberindex = 0;Numberindex < 120;++Numberindex) {
            PieceKeys[Typeindex][Numberindex] = RAND_64;
        }
    }

}
```

InitSq120To64

```
27  void InitSq120To64() {
28      int index = 0;
29      int file = FILE_A;
30      int rank = RANK_1;
31      int sq = A1;
32      int sq64 = 0;
33      for (index = 0;index < BRD_SQ_NUM;++index)
34      {
35          Sq64[index] = 65;
36      }
37
38      for (index = 0;index < 64;++index)
39      {
40          Sq120[index] = 120;
41      }
42
43      for (rank = RANK_1;rank <= RANK_8;++rank) {
44          for (file = FILE_A;file <= FILE_H;++file) {
45              sq = FR2SQ(file, rank);
46              Sq120[sq64] = sq;
47              Sq64[sq] = sq64;
48              sq64++;
49          }
50      }
51  }
```

InitSQ120To64 inizializza i due array utilizzati per effettuare comodamente ed in tempo costante traduzioni dall'indice di una casella in una scacchiera in formato 12x10 a quello di una casella in una scacchiera in formato 8x8 e viceversa.

InitFilesRanksBrd

```
void InitFilesRanksBrd() {
    int index = 0;
    int file = FILE_A;
    int rank = RANK_1;
    int sq = A1;
    int sq64 = 0;
    for (index = 0;index < BRD_SQ_NUM;index++) {
        FilesBrd[index] = OFFBOARD;
        RanksBrd[index] = OFFBOARD;
    }
    for (rank = RANK_1;rank <= RANK_8;++rank) {
        for (file = FILE_A;file <= FILE_H;++file) {
            sq = FR2SQ(file, rank);
            FilesBrd[sq] = file;
            RanksBrd[sq] = rank;
        }
    }
}
```

InitFilesRanksBrd inizializza i due array utilizzati per effettuare comodamente ed in tempo costante traduzioni

Funzioni per operare sulla scacchiera

Una volta definita la struttura della scacchiera è il momento di definire alcune funzioni indispensabili per operare sulla stessa, nella figura 2.18 troviamo le firme delle funzioni da porre nel file board.h,di seguito troviamo una descrizione dello scopo di ogni funzione e chiarimenti sulle implementazioni presenti nel file sorgente board.c

```
//board.c
void ResetBoard(S_Board* pos);
int  Parse_Fen(char* fen, S_Board* pos);
void  PrintBoard(const  S_Board* pos);
void UpdateListsMaterial(S_Board* pos);
int CheckBoard(const S_Board* pos);
```

Figura 2.18: board.h

ResetBoard

```
9   void ResetBoard(S_Board* pos) {
10      int index = 0;
11      for (index = 0;index < BRD_SQ_NUM; ++index) {
12          pos->pieces[index] = OFFBOARD;
13      }
14      for (index = 0;index < 64;++index) {
15          pos->pieces[Sq120[index]] = EMPTY;
16      }
17
18      for (index = 0;index < 2;++index) {
19          pos->bigPce[index] = 0;
20          pos->majPce[index] = 0;
21          pos->minPce[index] = 0;
22
23          pos->material[index] = 0;
24      }
25
26      for (index = WHITE;index <= BOTH; ++index) {
27          pos->pawn[index] = 0ULL;
28      }
29
30      for (index = 0;index < 13;++index)
31      {
32          pos->pceNum[index] = 0;
33      }
34
35      pos->KingSquare[WHITE] = NO_SQ;
36      pos->KingSquare[BLACK] = NO_SQ;
37      pos->side = BOTH;
38      pos->enPas = NO_SQ;
39      pos->fiftyMove = 0;
40      pos->ply = 0;
41      pos->hisPly = 0;
42      pos->castleperm = 0;
43      pos->posKey = 0ULL;
44
45
46
47  }
```

ResetBoard è la funzione che riporta la scacchiera ad uno stato neutro,prima ogni riquadro della scacchiera viene settato ad "offboard",il valore corrispondente ad una casella non valida, poi i riquadri appartenenti all'effettiva scacchiera,escluse quindi le caselle sentinella, vengono settate ad empty,in seguito abbiamo l'azzeramento di tutti i valori conservati dalla posizione

PrintBoard

```
155  void  PrintBoard(const  S_Board* pos) {
156      int sq, file, rank, piece;
157      printf("GAME BOARD : \n\n");
158
159      for (rank = RANK_8;rank >= RANK_1;rank--) {
160          printf("%d ", rank + 1);
161          for (file = FILE_A;file <= FILE_H;file++) {
162              sq = FR2SQ(file, rank);
163              piece = pos->pieces[sq];
164              printf("%3c", PieceChar[piece]);
165          }
166          printf("\n");
167      }
168
169      printf("\n  ");
170      for (file = FILE_A;file <= FILE_H;file++) {
171
172          printf("%3c", 'a' + file);
173      }
174      printf("\n");
175      printf("side:%c\n", SideChar[pos->side]);
176      printf("enPas:%d\n", pos->enPas);
177      printf("castleperm:%d\n", pos->castleperm);
178      printf("PosKey:%llX\n", pos->posKey);
179  }
```

PrintBoard è la funzione che permette di visualizzare una scacchiera in stdout,ciclando per ogni casella viene mostrato l'eventuale pezzo presente ed in caso di casella vuota il carattere "." ,infine vengono mostrare le informazioni riguardanti la posizione, quali key univoca che la identifica,stato dei permessi di arrocco etc.

UpdateListMaterial

```
181  void UpdateListsMaterial(S_Board* pos) {
182      int piece, sq, index, color;
183      for (index = 0;index < BRD_SQ_NUM;++index) {
184          sq = index;
185          piece = pos->pieces[index];
186          if (piece != EMPTY && piece != OFFBOARD) {
187              color = PieceCol[piece];
188              if (PieceBig[piece] == TRUE) { pos->bigPce[color]++; }
189              if (PieceMaj[piece] == TRUE) { pos->majPce[color]++; }
190              else if (PieceMin[piece] == TRUE) { pos->minPce[color]++; }
191              pos->material[color] += PieceVal[piece];
192              pos->pList[piece][pos->pceNum[piece]] = sq;
193              pos->pceNum[piece]++;
194              if (piece == WK) { pos->KingSquare[WHITE] = sq; }
195              else if (piece == BK) { pos->KingSquare[BLACK] = sq; }
196              if (piece == WP) {
197                  SETBIT(pos->pawn[WHITE], Sq64[sq]);
198                  SETBIT(pos->pawn[BOTH], Sq64[sq]);
199              }
200              else if (piece == BP) {
201                  SETBIT(pos->pawn[BLACK], Sq64[sq]);
202                  SETBIT(pos->pawn[BOTH], Sq64[sq]);
203              }
204
205          }
206
207      }
208  }
```

UpdateListMaterial è la funzione che,data una scacchiera dal quale ricavare i dati della posizione,aggiorna in memoria lo stato della posizione,per ogni casella controlla l'eventuale presenza di pezzi,e, nel caso in cui trovi un pezzo,aggiorna tutte le variabili collegate ad esso,in particolare si segnala come per il re venga aggiornato KingSquare e per i pedoni vi sia un aggiornamento delle bitboard

ParseFen

ParseFen si occupa di tradurre una stringa FEN in una posizione all'interno della scacchiera.

```
int  Parse_Fen(char* fen, S_Board* pos)
{
    assert(fen != NULL);
    assert(pos != NULL);
    int rank = RANK_8;
    int file = FILE_A;
    int piece = 0;
    int count = 0;
    int i = 0;
    int sq64 = 0;
    int sq120 = 0;
    ResetBoard(pos);
    while ((rank >= RANK_1 && *fen)) {
        count = 1;
        switch (*fen) {
        case 'p': piece = BP; break;
        case 'r': piece = BR; break;
        case 'n': piece = BN; break;
        case 'b': piece = BB; break;
        case 'k': piece = BK; break;
        case 'q': piece = BQ; break;
        case 'P': piece = WP; break;
        case 'R': piece = WR; break;
        case 'N': piece = WN; break;
        case 'B': piece = WB; break;
        case 'K': piece = WK; break;
        case 'Q': piece = WQ; break;

        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
            piece = EMPTY;
            count = *fen - '0';
            break;

        case '/':
        case ' ':
            rank--;
            file = FILE_A;
            fen++;
            continue;

        default:
            printf("FEN error \n");
            return -1;
        }
```

Figura 2.19: ParseFen1.png

```
        for (i = 0; i < count; i++) {
            sq64 = rank * 8 + file;
            sq120 = Sq120[sq64];
            if (piece != EMPTY) {
                pos->pieces[sq120] = piece;
            }
            file++;
        }
        fen++;
    }

    if (*fen == 'w')
        pos->side = WHITE;
    else if (*fen == 'b')
        pos->side = BLACK;
    else
        return -1;
    fen += 2;

    for (i = 0;i < 4;i++)
    {
        if (*fen == ' ') {
            break;
        }

        switch (*fen)
        {
        case 'K': pos->castleperm |= WKCA;
        case 'Q': pos->castleperm |= WQCA;
        case 'k': pos->castleperm |= BKCA;
        case 'q': pos->castleperm |= BQCA;

        default:
            break;
        }
        fen++;

    }
    assert(pos->castleperm >= 0 && pos->castleperm <= 15);
    fen++;
    if (*fen != '-') {
        file = fen[0] - 'a';
        rank = fen[1] - '1';
        assert(file >= FILE_A && file <= FILE_H);
        assert(rank >= RANK_1 && rank <= RANK_8);
        pos->enPas = FR2SQ(file, rank);
    }
    UpdateListsMaterial(pos);
    pos->posKey = GeneratePosKey(pos);
    return 0;

}
```

Figura 2.20: ParseFen2.png

CheckBoard

```
int CheckBoard(const S_Board* pos) {
    int t_pceNUM[13] = { 0 };
    int t_bigPce[2] = { 0 };
    int t_majPce[2] = { 0 };
    int t_minPce[2] = { 0 };
    int t_material[2] = { 0 };
    Bitboard t_pawn[3] = { 0ULL };
    t_pawn[WHITE] = pos->pawn[WHITE];
    t_pawn[BLACK] = pos->pawn[BLACK];
    t_pawn[BOTH] = pos->pawn[BOTH];


    int sq64, t_piece, t_pce_num, sq120, color, pcount;


    for (t_piece = WP;t_piece <= BK;++t_piece) {
        for (t_pce_num = 0;t_pce_num < pos->pceNum[t_piece];++t_pce_num) {
            sq120 = pos->pList[t_piece][t_pce_num];
            assert(pos->pieces[sq120] == t_piece);
        }
    }
    for (sq64 = 0;sq64 < 64;sq64++) {
        sq120 = Sq120[sq64];
        t_piece = pos->pieces[sq120];
        t_pceNUM[t_piece]++;
        color = PieceCol[t_piece];
        if (PieceBig[t_piece] == TRUE) { t_bigPce[color]++; }
        if (PieceMin[t_piece] == TRUE) { t_minPce[color]++; }
        if (PieceMaj[t_piece] == TRUE) { t_majPce[color]++; }
        t_material[color] += PieceVal[t_piece];
    }
    for (t_piece = WP;t_piece <= BK;++t_piece)
    {

        assert(t_pceNUM[t_piece] == pos->pceNum[t_piece]);
    }

    pcount = Count(t_pawn[WHITE]);
    assert(pcount == pos->pceNum[WP]);
    pcount = Count(t_pawn[BLACK]);
    assert(pcount == pos->pceNum[BP]);
    pcount = Count(t_pawn[BOTH]);
    assert(pcount == (pos->pceNum[WP] + pos->pceNum[BP]));

    while (t_pawn[WHITE]) {
        sq64 = Pop(&t_pawn[WHITE]);
        assert(pos->pieces[Sq120[sq64]] == WP);
    }

    while (t_pawn[BLACK]) {
        sq64 = Pop(&t_pawn[BLACK]);
        assert(pos->pieces[Sq120[sq64]] == BP);
    }

    while (t_pawn[BOTH]) {
        sq64 = Pop(&t_pawn[BOTH]);
        assert((pos->pieces[Sq120[sq64]] == WP) || (pos->pieces[Sq120[sq64]] == BP));
    }

    assert(t_material[WHITE] == pos->material[WHITE] &&
        t_material[BLACK] == pos->material[BLACK]);
    assert((t_minPce[WHITE] == pos->minPce[WHITE]) &&
        (t_minPce[BLACK] == pos->minPce[BLACK]));
    assert(t_majPce[WHITE] == pos->majPce[WHITE] &&
        t_majPce[BLACK] == pos->majPce[BLACK]);
    assert(t_bigPce[WHITE] == pos->bigPce[WHITE] &&
        t_bigPce[BLACK] == pos->bigPce[BLACK]);

    assert(pos->side == WHITE || pos->side == BLACK);
    assert(GeneratePosKey(pos) == pos->posKey);

    assert(pos->enPas == NO_SQ || (RanksBrd[pos->enPas] == RANK_6 && pos->side == WHITE)
        || (RanksBrd[pos->enPas] == RANK_3 && pos->side == BLACK));

    assert(pos->pieces[pos->KingSquare[WHITE]] == WK);
    assert(pos->pieces[pos->KingSquare[BLACK]] == BK);

    return TRUE;
}
```

CheckBoard è la funzione che permette di controllare lo stato della scacchiera, per prima cosa viene effettuato un riscontro tra la piece-list e la posizione dei pezzi sulla scacchiera,viene poi controllato che il numero dei pezzi salvati nella posizione corrisponda al numero dei pezzi presente sulla scacchiera si controlla poi che le bitboard siano accurate rispetto alla presenza dei pedoni,infine si controlla la correttezza delle variabili di stato quali la poskey ed i permessi di castling, se nessuno dei controlli fallisce la funzione restituisce true.

2.6 Permessi di arrocco

L'arrocco è una mossa particolare nel gioco degli scacchi che coinvolge il re e una delle due torri. È l'unica mossa che permette di muovere due pezzi contemporaneamente nonché l'unica in cui il re si muove di due caselle,Consiste nel muovere il re di due caselle a destra o a sinistra in direzione di una delle due torri e successivamente muovere la torre (quella verso la quale il re si è mosso) nella casella compresa tra quelle di partenza e di arrivo del re. Per indicare l'intenzione di effettuare un arrocco si deve prima sollevare il re e muoverlo di due case e solo successivamente muovere la torre nella casa di destinazione. Se si tocca la torre per prima si deve effettuare, secondo la regola che il pezzo toccato dev'essere mosso, una mossa con la sola torre. Se si arrocca muovendo o toccando contemporaneamente il re e la torre, si commette una mossa illegale. La mossa può essere effettuata solo se si è in presenza delle seguenti condizioni:

- il giocatore non ha mai mosso il re;

Figura 2.21: posizione di gioco dove è possibile arroccare

Figura 2.22: posizione della figura 2.21 post arrocco

- il giocatore non ha mai mosso la torre coinvolta nell'arrocco;
- non ci sono pezzi tra il re e la torre coinvolta;
- il re e la torre devono trovarsi sulla stessa traversa;
- il re non deve essere sotto scacco;
- il re, durante il movimento dell'arrocco, non deve attraversare caselle in cui si troverebbe sotto scacco.
- L'arrocco non è vietato se ad essere sotto attacco (prima, durante o al termine della mossa) è la torre.

È inoltre permesso effettuare un arrocco anche qualora il re sia stato sotto scacco in precedenza e, ovviamente, non sia stato ancora mosso.

In totale sono possibili 4 arrocchi,re bianco dal lato della regina, re bianco dal lato del re, re nero dal lato della regina re nero dal lato del re [7], volendo quindi codificare ogni permesso con un bit, la rappresentazione di tutti i permessi è possibili utilizzando una stringa di 4 bit,la variabile castleperm varrà quindi 1111 (15) se sono possibili tutti i 4 arrocchi,0000 se non sarà possibile effettuare nessuno dei 4 arrocchi, ed assumerà valori intermedi per indicare permessi che si trovano nel mezzo.
Si sottolinea che il permesso di arrocco non è la possibilità di poter effettuare un arrocco in quel preciso momento della partita ma indica la possibilità in generale di poter effettuare un arrocco nel caso in cui si verifichino le altre condizioni necessarie.
In termini pratici la realizzazione si riduce cosi ad un enum WKCA = 1, WQCA = 2, BKCA = 4, BQCA = 8 , la variabile castleperm sarà inizializzata sempre a 15 dato che ad inizio della partita

[7] il lato del re è il lato dove si trova il re rispetto alla regina all'inizio della partita equivale a destra per il bianco e sinistra per il nero, il lato della regina è il lato opposto

tutti gli arrocchi saranno sempre possibili,e, per controllare o eliminare i permessi basterà usare delle operazioni bitwise.

2.7 En passant

Nel gioco degli scacchi la presa en passant o presa al varco è una mossa speciale, cioè un'eccezione alle normali regole del movimento dei pezzi. La presa en passant è una mossa che coinvolge esclusivamente i pedoni e può essere eseguita da ciascun pedone che si trovi nelle condizioni adatte. ed è legata alla caratteristica del pedone di spostarsi di due caselle quando viene mosso per la prima volta. Quando un pedone, muovendosi di due passi (quindi per la prima volta), finisce esattamente accanto (sulla stessa traversa e su colonna adiacente) ad un pedone avversario, nella mossa successiva quest'ultimo può catturarlo come se si fosse mosso di un passo solo. È importante notare che, quando un pedone ha la possibilità di effettuare una presa en passant nei confronti di un pedone avversario, deve realizzarla subito, al verificarsi della posizione, altrimenti perde il diritto a farlo.

Figura 2.23: posizione di gioco dove è possibile catturare con en passant

Figura 2.24: posizione della figura 2.23 post cattura con en passant

Da un attenta lettura delle regole possiamo notare che la presa en passant sarà possibile in al massimo una casella per volta, questo significa che, per indicare la presenza di un possibile presa è sufficiente utilizzare un intero, che avrà il valore della casella dove è possibile catturare con en passant,trattandosi di un elemento della partita strettamente legato alle mosse, l'aspetto implementativo e di gestione verrà approfondito nella sottosezione 3 dedicata alla generazione delle mosse

2.8 Undo

```
35  typedef struct {
36      int move;
37      int castlePerm;
38      int enPas;
39      int fiftyMove;
40      Bitboard posKey;
41  } S_Undo;
```

Struttura dati che contiene ogni mossa effettuata e lo stato della scacchiera prima che quella mossa venisse fatta,fondamentale per potere effettuare il rollback della scacchiera ad uno stato precedente:

- **move**: contiene la mossa svolta al momento di creazione dell'istanza della struttura dati,è utilizzata per riportare la scacchiera allo stato precedente alla mossa tramite una funzione di UnMake,approccio preferibile rispetto al salvataggio e al ripristino dell'intera posizione.
- **castlePerm**: contiene i permessi al momento di creazione dell'istanza della struttura dati.
- **enPas**: contiene la casella di en passant al momento di creazione dell'istanza della struttura dati.
- **fiftyMove**: contiene il contatore per la regola delle 50 mosse al momento di creazione dell'istanza della struttura dati.
- **poskey**: contiene la hashkey della posizione al momento di creazione dell'istanza della struttura dati.

2.9 Generare la chiave hash della posizione

CAPITOLO 3

Move Generation

In questo capitolo viene mostrato passo passo un procedimento guida alla realizzazione di un motore scacchistico

3.1 Introduzione

Una volta stabilito il tipo di rappresentazione il passo successivo è quello della generazione delle mosse,per generazione delle mosse si intende la generazione di tutte le mosse legali eseguibili data una posizione di partenza , la generazione delle mosse è un processo fondamentale in quanto identifica i rami che il nostro algoritmo potrà esplorare nella successiva fase, quella appunto di ricerca, trattata nel capitolo 4.

3.2 Struttura di una mossa

Prima di parlare della generazione delle mosse, iniziamo col parlar e della struttura di una singola mossa,un requisito fondamentale che ogni rappresentazione di una mossa deve avere, è la scelta di una rappresentazione conforme con quella scelta per la codifica della scacchiera e dei pezzi,trattandosi di un elemento realizzabile in moltissimi modi, con tante soluzioni di simile efficienza, non verrà fornito un elenco esaustivo delle possibili scelte ma verrà trattata solo una rappresentazione d'esempio.

```
3   typedef struct {
4       int move;
5       int score;
6   } S_MOVE;
```

Figura 3.1: move.h

Ogni mossa è memorizzata come un intero di 32 bit (con 25 bit effettivamente utilizzati):

- 0000 0000 0000 0000 0000 0**111 1111** -> La casella di partenza è codificata con 7 bit,assume il valore 0-127 dell'indice da cui il pezzo viene mosso
- 0000 0000 0000 00**11 1111 1**000 0000 -> La casella di arrivo è codificata con 7 bit,assume il valore 0-127 dell'indice in cui il pezzo viene mosso
- 0000 0000 00**11 11**00 0000 0000 0000 -> In caso di cattura, il tipo di pezzo catturato viene codificato in 4 bit,assume il valore 0-15 del tipo di pezzo catturato (0 codifica una mossa senza cattura)
- 0000 0000 0**1**00 0000 0000 0000 0000 -> Un bit è utilizzato per codificare se la mossa è una presa en passant
- 0000 0000 **1**000 0000 0000 0000 0000 -> Pawn Start 0x80000, 1 to store if it was a pawn start
- 0000 **1111** 0000 0000 0000 0000 0000 -> In caso di promozione di un pedone,il tipo di pezzo nel quale il pedone è stato promosso è codificato in 4 bit
- 000**1** 0000 0000 0000 0000 0000 0000 -> Un bit è utilizzato per codificare se la mossa è un arrocco

Inoltre insieme alla rappresentazione della mossa viene anche conservato un valore intero nel quale conservare la "bontà" della mossa,ossia quanto riteniamo che la mossa sia promettente nel futuro della ricerca e che verrà calcolato nella fase di valutazione, trattata nel capitolo 5.

Lista delle mosse

Una volta definita la struttura di una singola mossa, definiamo anche una struttura chiamata movelist che consiste nella lista di tutte le mosse registrate dall'inzio della partita e da un contatore delle stesse,infine definiamo delle bitmask utili ad estrarre specifiche informazioni da una mossa

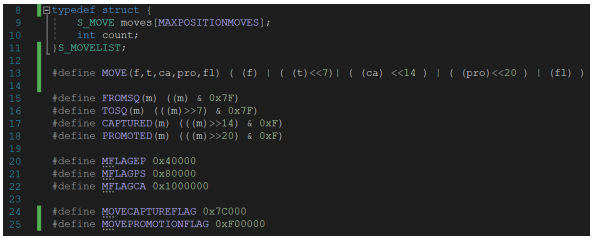

Figura 3.2: struttura movelist e bitmasks presenti in move.h

movegen.c usa attack.c, Va spiegato.

3.3 Generare le mosse

In questa sezione verrà trattata la generazione delle mosse,prima di iniziare a parlare della generazione però è debito soffermarsi su come le mosse ,una volta generate, vengono aggiunte alla lista di possibili mosse. Definiamo in movegen.h una funzione per l'inserimento per ogni tipo di mossa possibile che potrà essere creata durate la generazione delle stesse:

```
void AddQuietMove(const S_Board* pos, int move, S_MOVELIST* list);
void AddCaptureMove(const S_Board* pos, int move, S_MOVELIST* list);
void AddEnPassantMove(const S_Board* pos, int move, S_MOVELIST* list);
void AddWhitePawnCapMove(const S_Board* pos, const  int from, const int to, const int cap, S_MOVELIST* list);
void AddWhitePawnMove(const S_Board* pos, const int from, const int to, S_MOVELIST* list);
void AddBlackPawnCapMove(const S_Board* pos, const  int from, const int to, const int cap, S_MOVELIST* list);
void AddBlackPawnMove(const S_Board* pos, const int from, const int to, S_MOVELIST* list);
void GenerateAllMoves(const S_Board* pos, S_MOVELIST* list);
```

Figura 3.3: movegen.h

Come si può notare dai nomi delle funzioni,negli scacchi le mosse sono divise in due categorie, le mosse possono essere "quiet/silenziose" o meno , sono "silenziose" tutte le mosse che non risultano in una cattura o in una promozione. Abbiamo quindi due funzioni,

una chiamata "AddQuietMove" che si occupa dell'aggiunta delle mosse silenziose ed una chiamata "AddCaptureMove" che gestisce le altre.

```
void AddQuietMove(const S_Board* pos, int move, S_MOVELIST* list) {

    assert(SqOnBoard(FROMSQ(move)));
    assert(SqOnBoard(TOSQ(move)));

    list->moves[list->count].move = move;
    list->moves[list->count].score = 0;
    list->count++;


}

void AddCaptureMove(const S_Board* pos, int move, S_MOVELIST* list){

    assert(SqOnBoard(FROMSQ(move)));
    assert(SqOnBoard(TOSQ(move)));
    assert(PieceValid(CAPTURED(move)));
    list->moves[list->count].move = move;
    list->moves[list->count].score = 0;
    list->count++;


}
```

Figura 3.4: movegen.h

Discorso a parte va fatto per i pedoni,i pedoni sono il pezzo più difficile da trattare in quanto, come vederemo anche nel paragrafo 3.3.1 è l'unico pezzo che può catturare con en passant e che può promuovere,creiamo quindi delle funzioni che gestiscono la possibilità del pedone di poter essere promosso in uno qualsiasi dei 4 pezzi aggiungendo di fatto 4 mosse invece di 1,di seguito sono riportate le funzioni per il pedone bianco, quelle del pedone nero sono identiche con ovviamente gli enum cambiati per rispecchiare la differenza.

```
void AddEnPassantMove(const S_Board* pos, int move, S_MOVELIST* list) {
    list->moves[list->count].move = move;
    list->moves[list->count].score = 0;
    list->count++;
}

void AddWhitePawnCapMove(const S_Board* pos, const  int from, const int to, const int cap, S_MOVELIST* list) {

    assert(PieceValidEmpty(cap));
    assert(SqOnBoard(from));
    assert(SqOnBoard(to));

    if (RanksBrd[from] == RANK_7) {
        AddCaptureMove(pos, MOVE(from, to, cap, WQ, 0), list);
        AddCaptureMove(pos, MOVE(from, to, cap, WR, 0), list);
        AddCaptureMove(pos, MOVE(from, to, cap, WB, 0), list);
        AddCaptureMove(pos, MOVE(from, to, cap, WN, 0), list);
    }
    else {
        AddCaptureMove(pos, MOVE(from, to, cap, EMPTY, 0), list);
    }

}


void AddWhitePawnMove(const S_Board* pos, const int from, const int to, S_MOVELIST* list) {
    if (RanksBrd[from] == RANK_7) {
        AddQuietMove(pos, MOVE(from, to, EMPTY, BQ, 0), list);
        AddQuietMove(pos, MOVE(from, to, EMPTY, BR, 0), list);
        AddQuietMove(pos, MOVE(from, to, EMPTY, BB, 0), list);
        AddQuietMove(pos, MOVE(from, to, EMPTY, BN, 0), list);
    }
    else {
        AddQuietMove(pos, MOVE(from, to, EMPTY, EMPTY, 0), list);
    }
}
```

Figura 3.5: move.h

3.3.1 Pedoni

Il pedone muove solo in avanti, mai indietro o di lato. Un pedone può avanzare di due caselle se è la prima volta che viene mosso,altrimenti può avanzare solo di una casella. Il un pedone mangia un pezzo avversario spostandosi diagonalmente di una casella,sempre soltanto in avanti, o a destra o a sinistra, se un pedone raggiunge la traversa finale della scacchiera rispetto alla sua direzione di movimento allora viene promosso, diventa quindi,a scelta del giocatore che possiede la pedina, uno qualsiasi degli altri pezzi (ad eccezione del re).

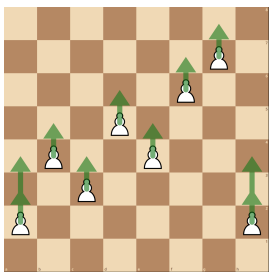

Figura 3.6: aaaa

Figura 3.7: aaaa

generazione delle mosse di un pedone

Di seguito è presentato un esempio di implementazione per la generazione delle mosse dei pedoni bianchi, per quelli neri il codice si presenta identico ma con i valori di movimento adattati di conseguenza.

```
if (side == WHITE) {
    for (pieceNum = 0;pieceNum < pos->pceNum[WP];++pieceNum) {
        sq = pos->pList[WP][pieceNum];
        assert(SqOnBoard(sq));

        if (pos->pieces[sq + 10] == EMPTY) {
            AddWhitePawnMove(pos, sq, sq + 10, list);
            if (RanksBrd[sq] == RANK_2 && pos->pieces[sq + 20] == EMPTY) {
                AddQuietMove(pos, MOVE(sq, (sq + 20), EMPTY, EMPTY, MFLAGPS), list);
            }
        }
        if (SqOnBoard(sq + 9) && PieceCol[pos->pieces[sq + 9]] == BLACK) {
            AddWhitePawnCapMove(pos, sq, sq + 9, pos->pieces[sq + 9], list);
        }
        if (SqOnBoard(sq + 11) && PieceCol[pos->pieces[sq + 11]] == BLACK) {
            AddWhitePawnCapMove(pos, sq, sq + 11, pos->pieces[sq + 11], list);
        }
        if (pos->enPas != NO_SQ) {
            if (sq + 9 == pos->enPas) {
                AddEnPassantMove(pos, MOVE(sq, sq + 9, EMPTY, EMPTY, MFLAGEP), list);
            }
            if (sq + 11 == pos->enPas) {
                AddEnPassantMove(pos, MOVE(sq, sq + 11, EMPTY, EMPTY, MFLAGEP), list);
            }
        }

    }
```

Per generare le mosse dei pedoni,prendiamo la posizione di ogni pedone sulla scacchiera,se il pedone può avanzare in avanti di una casella allora aggiungiamo quella mossa all'elenco,se il pedone non era ancora stato mosso (e si trova quindi sulla traversa di partenza) e può muoversi di 2 caselle in avanti, aggiungiamo anche questa variante all'elenco,controlliamo poi se il pedone può catturare può catturare un pezzo diagonalmente sia destra sia a sinistra ed aggiungiamo queste eventuali mosse all'elenco,infine controlliamo se è possibile per il pedone effettuare una cattura en passant ed in tal caso aggiungiamo quest'ultima mossa alla lista.

3.3.2 Pezzi scorrevoli

Si definiscono pezzi scorrevoli i pezzi che possono spostarsi di un numero non prefissato di caselle lungo l'asse orizzontale,verticale o diagonale,fino a raggiungere il bordo della scacchiera o un altro pezzo. I pezzi scorrevoli consistono di alfiere,torre e regina,la generazione delle mosse di questo tipo di pezzi è più complessa rispetto a quella degli altri pezzi, quanto bisogna controllare la presenza di pezzi,propri o avversari che siano, in grado di fermare il movimento del pezzo e bisogna assicurarsi che il pezzo non superi il bordo della scacchiera.

alfieri

The bishop chess piece moves in any direction diagonally. Chess rules state that there is no limit to the number of squares a bishop can travel on the chessboard, as long as there is

not another piece obstructing its path. Bishops capture opposing pieces by landing on the square occupied by an enemy piece.

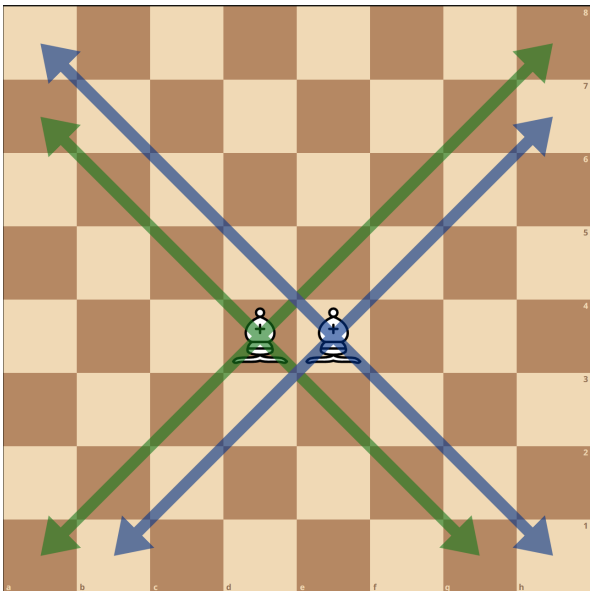

Figura 3.8: move.h

generazione delle mosse di un alfiere

torri

The rook moves horizontally or vertically, through any number of unoccupied squares (see diagram). The rook cannot jump over pieces. As with captures by other pieces apart from en passant, the rook captures by occupying the square on which the enemy piece stands.

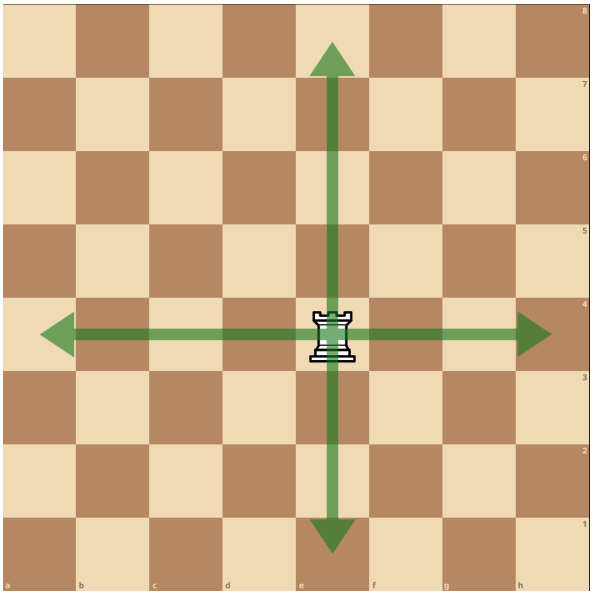

Figura 3.9: move.h

generazione delle mosse di una torre

regine

The queen can be moved any number of unoccupied squares in a straight line vertically, horizontally, or diagonally, thus combining the moves of the rook and bishop. The queen captures by occupying the square on which an enemy piece sits.

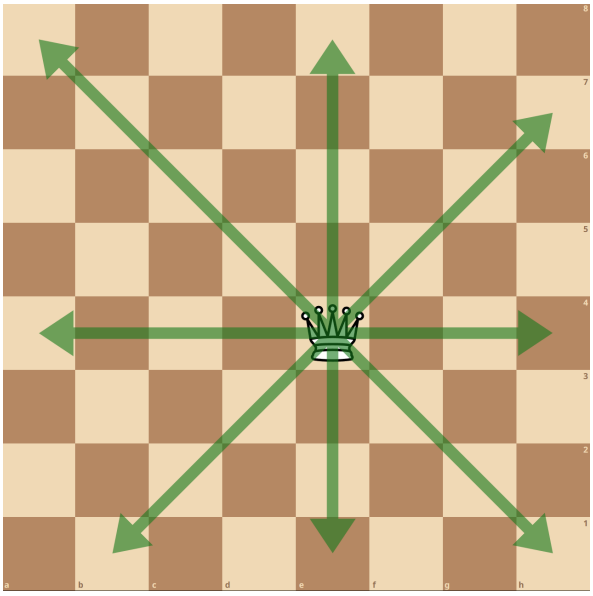

Figura 3.10: move.h

generazione delle mosse di una regina

non sliding

```
void AddQuietMove(const S_Board* pos, int move, S_MOVELIST* list);
void AddCaptureMove(const S_Board* pos, int move, S_MOVELIST* list);
void AddEnPassantMove(const S_Board* pos, int move, S_MOVELIST* list);
void AddWhitePawnCapMove(const S_Board* pos, const  int from, const int to, const int cap, S_MOVELIST* list);
void AddWhitePawnMove(const S_Board* pos, const int from, const int to, S_MOVELIST* list);
void AddBlackPawnCapMove(const S_Board* pos, const  int from, const int to, const int cap, S_MOVELIST* list);
void AddBlackPawnMove(const S_Board* pos, const int from, const int to, S_MOVELIST* list);
void GenerateAllMoves(const S_Board* pos, S_MOVELIST* list);
```

Figura 3.11: struttura movelist e bitmasks presenti in move.h

```
41
42  void AddQuietMove(const S_Board* pos, int move, S_MOVELIST* list) {
43
44      assert(SqOnBoard(FROMSQ(move)));
45      assert(SqOnBoard(TOSQ(move)));
46
47      list->moves[list->count].move = move;
48      list->moves[list->count].score = 0;
49      list->count++;
50
51
52  }
53
54  void AddCaptureMove(const S_Board* pos, int move, S_MOVELIST* list) {
55
56      assert(SqOnBoard(FROMSQ(move)));
57      assert(SqOnBoard(TOSQ(move)));
58      assert(PieceValid(CAPTURED(move)));
59      list->moves[list->count].move = move;
60      list->moves[list->count].score = 0;
61      list->count++;
62
63
64  }
65
66  void AddEnPassantMove(const S_Board* pos, int move, S_MOVELIST* list) {
67      list->moves[list->count].move = move;
68      list->moves[list->count].score = 0;
69      list->count++;
70  }
```

Figura 3.12: bitboard.c

```
void AddEnPassantMove(const S_Board* pos, int move, S_MOVELIST* list) {
    list->moves[list->count].move = move;
    list->moves[list->count].score = 0;
    list->count++;
}

void AddWhitePawnCapMove(const S_Board* pos, const  int from, const int to, const int cap, S_MOVELIST* list) {

    assert(PieceValidEmpty(cap));
    assert(SqOnBoard(from));
    assert(SqOnBoard(to));

    if (RanksBrd[from] == RANK_7) {
        AddCaptureMove(pos, MOVE(from, to, cap, WQ, 0), list);
        AddCaptureMove(pos, MOVE(from, to, cap, WR, 0), list);
        AddCaptureMove(pos, MOVE(from, to, cap, WB, 0), list);
        AddCaptureMove(pos, MOVE(from, to, cap, WN, 0), list);
    }
    else {
        AddCaptureMove(pos, MOVE(from, to, cap, EMPTY, 0), list);
    }

}


void AddWhitePawnMove(const S_Board* pos, const int from, const int to, S_MOVELIST* list) {
    if (RanksBrd[from] == RANK_7) {
        AddQuietMove(pos, MOVE(from, to, EMPTY, BQ, 0), list);
        AddQuietMove(pos, MOVE(from, to, EMPTY, BR, 0), list);
        AddQuietMove(pos, MOVE(from, to, EMPTY, BB, 0), list);
        AddQuietMove(pos, MOVE(from, to, EMPTY, BN, 0), list);
    }
    else {
        AddQuietMove(pos, MOVE(from, to, EMPTY, EMPTY, 0), list);
    }
}
```

Figura 3.13: bitboard.c

3.4 MakeMove

3.5 Perft

Perft è una funzione di debugging che attraversa l'albero delle mosse legali generate e conta tutti i nodi foglia fino ad una data profondità n.Il numero viene poi comparato con dei valori noti per controllare la presenza di bug. I nodi vengono contati alla fine della generazione, dopo l'ultimo makemove,non vengono quindi contati i nodi foglia che si trovano a profondità superiori di n.Perft inoltre ignora i pareggi per ripetizione, per la regola delle 50 mosse, e per materiale insufficiente.Utilizzando la stessa versione di perft, o In alternativa una versione estremamente simile di perft,è possibile comparare il tempo impiegato da diversi generatori di mosse.

```
typedef unsigned long long u64;

u64 Perft(int depth)
{
  MOVE move_list[256];
  int n_moves, i;
  u64 nodes = 0;

  if (depth == 0)
    return 1ULL;

  n_moves = GenerateLegalMoves(move_list);
  for (i = 0; i < n_moves; i++) {
    MakeMove(move_list[i]);
    nodes += Perft(depth - 1);
    UndoMove(move_list[i]);
  }
  return nodes;
}
```

Figura 3.14: codice di una funzione basica di perft

CAPITOLO 4

Ricerca

In questo capitolo viene mostrato passo passo un procedimento guida alla realizzazione di un motore scacchistico

4.1 Esempio di implementazione

CAPITOLO 5

Valutazione

Questo capitolo illustra lo stato dell'arte e i lavori presenti in letteratura sugli aspetti di ricerca trattati nel nostro studio. ECC ECC...

5.1 Esempio di implementazione

CAPITOLO 6

Libro delle aperture,Tablebases

Questo capitolo illustra lo stato dell'arte e i lavori presenti in letteratura sugli aspetti di ricerca trattati nel nostro studio. ECC ECC...

6.1 Esempio di implementazione

CAPITOLO 7

Validazione preliminare

BREVE SPIEGAZIONE CONTENUTO CAPITOLO

Introduzione al concetto di ELO,rappresentazione ELO stimata del motore,metriche prestazionali varie

CAPITOLO 8

Stato dell'arte per Algoritmi di intelligenza artificiale e motori scacchistici per il gioco degli scacchi

Questo capitolo illustra lo stato dell'arte e i lavori presenti in letteratura sugli aspetti di ricerca trattati nel nostro studio. ECC ECC...

8.1 Pre Neural Network Stockfish

8.1.1 Board Representation

8x8 Board Bitboards with Little-Endian Rank-File Mapping (LERF) Magic Bitboards BMI2 - PEXT Bitboards (not recommend for AMD Ryzen [28] prior to Zen 3)

8.1.2 Search

Iterative Deepening Aspiration Windows Parallel Search using Threads YBWC prior to Stockfish 7 Lazy SMP since Stockfish 7, January 2016 Principal Variation Search Transposition Table Shared Hash Table 10 Bytes per Entry, 3 Entries per Cluster Depth-preferred Replacement Strategy No PV-Node probing Prefetch Move Ordering Countermove Heuristic Counter Moves History since Stockfish 7, January 2016 [32] History Heuristic Internal Iterative Deepening Killer Heuristic MVV/LVA SEE Selectivity Extensions Check Extensions if SEE >= 0 Restricted Singular Extensions Pruning Futility Pruning Move Count Based Pruning Null Move Pruning Dynamic Depth Reduction based on depth and value Static Null Move Pruning Verification search at high depths ProbCut SEE Pruning Reductions Late Move Reductions Razoring Quiescence Search

8.1.3 Evaluation

8.1.3.1 Material

Bishop Pair Imbalance Tables Material Hash Table

8.1.3.2 Mobility

Trapped Pieces Rooks on (Semi) Open Files

8.1.3.3 Pawn Structure

Pawn Hash Table

Backward Pawn

Doubled Pawn

Isolated Pawn

Phalanx

Connected Pawns

Passed Pawn

King Safety

Attacking King Zone Pawn Shelter Pawn Storm Square Control

Tapered Eval

Evaluation Patterns

Piece-Square Tables

Space

Outposts

8.2 Google AlphaZero

8.3 Post Neural Networ Stockfish-NNUE e LCZero

Nonostante le critiche ed i dubbi sulla performance di AlphaZero, gli appassionati di scacchi computazionali non rimasero impassibili davanti ai meriti di un approccio orientato alle reti neurali,il 6 agosto del 2020, un anno e mezzo dopo la pubblicazione definitiva dell'articolo su AlphaZero da parte di DeepMind e dopo un anno di lavoro, viene ufficialmente introdotta ,all'interno della repo di Stockfish,NNUE. NNUE, acronimo di "Effeciently Upgradable Neural Network" scritto da sinistra a destra, è una rete neurale per la valutazione di posizioni shogi, alle quali assegna un punteggio utilizzato poi in fase di potatura,adattata per operare sugli scacchi ed essere integrata in Stockfish. In questa versione Stockfish mantiene le caratteristiche principali che contraddistiguono la sua versione precedente, e la valutazione NNUE viene utilizzata solo in posizioni materialmente bilanciate

CAPITOLO 9

Conclusioni e Sviluppi Futuri

BREVE SPIEGAZIONE CONTENUTO CAPITOLO

Ringraziamenti

INSERIRE RINGRAZIAMENTI QUI