UNIVERSITÀ DEGLI STUDI DI TRENTO

DIPARTIMENTO DI MATEMATICA

Laurea Triennale in Matematica

Algoritmi e Strutture Dati per il gioco degli scacchi

Candidato:
Andrea Zanin

Relatore:
Alberto Montresor

Anno accademico 2020/21

Indice

Introduzione

Gli scacchi e in particolare la ricerca della mossa migliore in una data posizione sono oggetto di studio rigoroso fin dall'inizio del XX secolo; questo settore di ricerca ha attratto alcuni tra i più brillanti matematici dello scorso secolo, tra questi citiamo Alan Turing e Ernst Zermelo.
In virtù della crescita esponenziale delle varianti possibili all'aumentare del numero di mosse, il gioco degli scacchi non è approcciabile con meri metodi di forza bruta, nemmeno con la potenza computazionale odierna. Infatti i primi algoritmi sviluppati negli anni '50 e '60 hanno risultati mediocri anche se eseguiti su computer moderni. Questo campo è ancora oggi oggetto di ricerca attiva e i migliori engine scacchistici sono in continuo miglioramento, anche escludendo i miglioramenti hardware.
Per affrontare il problema sono stati sviluppati nuovi algoritmi, strutture dati ed euristiche, spesso applicando risultati algebrici (ad esempio le sequenze di De Bruijn) e statistici (ad esempio l'inferenza sulla base di un modello di regressione lineare).

In questa tesi ho voluto presentare una selezione degli elementi più rilevanti nella costruzione di un engine scacchistico moderno, facendo una sintesi della vasta e frammentata letteratura scientifica su questo tema. In particolare ho approfondito le strutture dati per rappresentare la scacchiera, gli algoritmi di esplorazione delle mosse possibili e le euristiche per la valutazione di una posizione.
Ho basato la trattazione di questi argomenti sulla definizione rigorosa di mossa ottima seguendo il metodo dell'induzione a ritroso, come proposto da Von Neumann nei suoi lavori sulla teoria dei giochi.
Successivamente ho trattato principalmente gli algoritmi di ricerca che per la classificazione di Turing sono strategie di tipo A, in quanto questi sono quelli più applicati dagli engine scacchistici moderni.
A corredo di questa tesi ho sviluppato un engine scacchistico con lo scopo di confrontare i vari algoritmi da me studiati e validarne sperimentalmente l'efficacia; il codice è disponibile all'indirizzo github.com/ZaninAndrea/chess_engine.

Rappresentare la scacchiera

1

Nell'implementare il gioco degli scacchi la prima scelta fondamentale è come rappresentare la posizione dei pezzi sulla scacchiera; la scelta della struttura dati utilizzata ha un impatto importante sull'efficienza degli algoritmi di analisi della posizione e di ricerca della mossa migliore che sono oggetto dei capitoli seguenti.

La struttura dati scelta deve permettere implementazioni efficienti delle operazioni utilizzate negli algoritmi di analisi e ricerca, a titolo esemplificativo:

- `Move`: muovere un pezzo sulla scacchiera
- `Count`: contare i pezzi di una certa tipologia (e.g. i cavalli bianchi)
- `IsEmpty`: verificare se una casella è libera
- `IsIsolated`: determinare se un pedone è isolato [1]

1: Un pedone si dice isolato se nelle colonne adiacenti non ci sono pedoni alleati.

Una prima struttura dati naive per rappresentare la scacchiera consiste nell'avere una lista delle posizioni dei pedoni bianchi, una dei pedoni neri, una dei cavalli bianchi, ... Possiamo numerare le caselle della scacchiera e usare un intero a 8 bit per rappresentare una posizione:

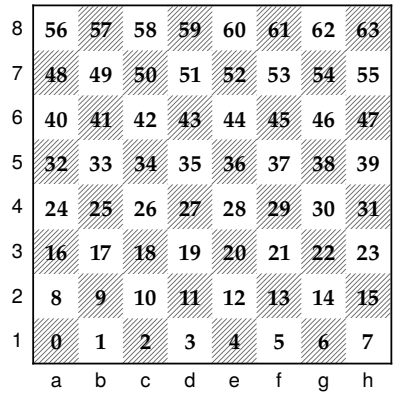

Figura 1.1: Tipicamente le caselle vengono numerate partendo dalla traversa 1 e salendo, questa numerazione è detta Little Endian Rank-File Mapping.

Questa struttura dati è di facile implementazione, ma

gli algoritmi per implementare le operazioni sopracitate sono decisamente inefficienti:

Tabella 1.1: Complessità di alcuni metodi importanti implementati con le strutture dati basate su liste ordinate e non ordinate. Con n indichiamo il numero di pezzi sulla scacchiera.

	Liste non ordinate	Liste ordinate
`Move`	$O(n)$	$O(\log n)$
`Count`	$O(1)$	$O(1)$
`IsEmpty`	$O(n)$	$O(n)$
`IsIsolated`	$O(n)$	$O(\log n)$

1.1 Bitboards

Per conservare le posizioni occupate da ciascuna tipologia di pezzo possiamo alternativamente usare un array di 64 booleani che indicano se in ciascuna casella della scacchiera è presente un pezzo di quel tipo. Per rappresentare un array di 64 booleani posso usare un intero a 64 bit e considerare l'i-esimo bit come il booleano associato all'i-esima casella della scacchiera. [1]

[1]: Browne (2014), «Bitboard Methods for Games»

Generalmente oltre ai 12 interi necessari a rappresentare le posizioni di tutti i pezzi utilizziamo 3 interi per indicare le caselle in cui è presente un qualsiasi pezzo bianco, un qualsiasi pezzo nero e le caselle libere.

Il motivo dell'efficienza di questa struttura dati è che ci permette di sfruttare le operazioni bit-wise tra interi a 64 bit che sono supportate da tutti i processori moderni.

Ad esempio si può verificare con complessità $O(1)$ se una casella è libera:

Pseudocodice

```
bool IsEmpty(int square):
    return emptySquares & (1 << square) != 0
```

Con il comando `1 << square` ottengo l'intero in cui l'unico bit `1` è quello in posizione `square`, successivamente l'operatore `&` mi restituisce un intero in cui ciascun bit è acceso solamente se lo erano anche i bit corrispondenti in `emptySquares` e `1 << square`,

questo intero è diverso da 0 solamente se il bit in posizione `square` nell'intero `emptySquares` è `1`, ovvero se quella casella è libera.

Come approfondiremo in seguito, con questa struttura dati possiamo implementare le operazioni viste sopra in maniera decisamente più efficiente:

	Complessità
`Move`	$O(1)$
`Count`	$O(\log m)$
`IsEmpty`	$O(1)$
`IsIsolated`	$O(1)$

Tabella 1.2: Complessità di alcuni metodi importanti implementati con le bitboards. Con m indichiamo il numero di caselle nella scacchiera.

Proprio in virtù della sua efficienza questa struttura dati viene utilizzata da tutti i più forti engine scacchistici, ad esempio Stockfish [2] .

[2]: Costalba et al. (2021), *Stockfish Engine source code*

Le bitboard sono uno strumento molto flessibile e vengono utilizzate per memorizzare vari tipi di informazioni legate alle caselle della scacchiera, non solo la posizione dei pezzi, ma anche le caselle attaccate da un pezzo, configurazioni pedonali interessanti, lo spazio di un giocatore [2] , ...

2: Con spazio di un giocatore si intende l'insieme delle caselle in cui può muovere almeno un suo pezzo, questo è un parametro interessante per valutare che giocatore è favorito in una data posizione.

Formalmente una bitboard è un elemento di $\mathbb{F}_2^{64}$, ovvero dello spazio vettoriale di dimensione 64 avente come campo $\mathbb{Z}/2\mathbb{Z}$; questo modello matematico ci sarà utile per analizzare algoritmi complessi come l'hash di Zobrist.

1.2 PopCount

Una delle operazioni fondamentali sulle bitboard è contare i bit accesi, nel contesto delle bitboard questa operazione è chiamata Population Count (abbreviato PopCount); interpretando le bitboard come elementi di $\mathbb{F}_2^{64}$ il numero di bit accesi è la distanza di Hamming [3] dall'origine.

3: La distanza di Hamming tra 2 vettori $x, y \in \mathbb{F}_p^n$ è definita come

$$d(x, y) := \sum_{i=1}^{n} \delta_{x_i y_i}$$

dove δ è il delta di Kronecker.

Ci sono vari algoritmi per implementare PopCount, segue un'analisi e un confronto dei 2 algoritmi più

utilizzati: l'approccio lineare di Brian Kernighan e un approccio divide et impera.

Algoritmo di Kernighan

[3]: Anderson (2005), *Bit Twiddling Hacks*

L'algoritmo di Kernighan [3] per PopCount consiste nel trasformare in `0` il bit acceso meno significativo finché la bitboard non diventa vuota, contando il numero di sostituzioni effettuate ottengo il numero di bit accesi.

Sottraendo `1` ad un naturale positivo il bit `1` meno significativo della sua rappresentazione binaria diventa `0` e tutti gli `0` che lo seguono diventano `1`.

Esempio

Confrontiamo ad esempio le rappresentazioni binarie di 5648 e 5647:

```
5648: 1011000010000
5647: 1011000001111
```

Sfruttando questa osservazione possiamo cancellare il bit `1` meno significativo della bitboard mantenendo solo i bit `1` in comune tra `bitboard` e `bitboard - 1`.

Pseudocodice

```
int KernighanPopCount(uint bitboard):
    if bitboard == 0:
        return 0

    count = 1
    bitboard = bitboard & (bitboard - 1)

    while bitboard != 0:
        bitboard = bitboard & (bitboard - 1)
        count=count+1

    return count
```

Detto n il numero di bit `1`, questo algoritmo effettua $3n-1$ operazioni aritmetiche e $n+1$ confronti per calcolare il risultato.

Algoritmo Divide et Impera

Un metodo alternativo [3] consiste nel contare i bit accesi in ciascuna coppia, poi usare questa informazione per contare i bit accesi in ciascun blocco di 4 bit, 8 bit, ... fino a contare i bit accesi nel blocco da 64 bit, ossia in tutta la bitboard.

[3]: Anderson (2005), *Bit Twiddling Hacks*

Notiamo che per contare i bit `1` in ciascuna coppia è sufficiente sommare il primo e il secondo bit della coppia

```
Mask:              01 01 01 01
bb:                11 01 10 00

bb & Mask:         01 01 00 00   +
(bb>>1) & Mask:    01 00 01 00   =
Conta per coppie: 10 01 01 00
```

Esempio Contare i bit `1` in ciascuna coppia di bit del numero `11011000`.

Con l'operatore `>>1` spostiamo tutti i bit di una posizione a destra in modo da poter sommare il bit più significativo e quello meno significativo con la normale addizione, la maschera serve a considerare nel primo caso solo i bit meno significativi delle coppie e nel secondo solo i bit più significativi delle coppie.

Con metodo analogo possiamo ottenere il conteggio a gruppi di 4 bit:

```
Mask:                   00 11  00 11
2bitCount:              10 01  01 00

2bitCount & Mask:       00 01  00 00   +
(2bitCount>>2) & Mask: 00 10  00 01   =
Conta per 4 bit:        00 11  00 01
```

Esempio Contare i bit `1` in ciascuna gruppo di 4 bit del numero `11011000` partendo dal conteggio per coppie.

Per contare i bit `1` in una bitboard è sufficiente ripetere questo procedimento fino ad arrivare al blocco di larghezza 64 bit.

Pseudocodice

```
int DivideEtImperaPopCount(uint bitboard):
    mask2Bits  = 0x5555555555555555
    mask4Bits  = 0x3333333333333333
    mask8Bits  = 0x0f0f0f0f0f0f0f0f
    mask16Bits = 0x00ff00ff00ff00ff
    mask32Bits = 0x0000ffff0000ffff
    mask64Bits = 0x00000000ffffffff

    bb = (bb & mask2Bits)  + ((bb>>1)  & mask2Bits)
    bb = (bb & mask4Bits)  + ((bb>>2)  & mask4Bits)
    bb = (bb & mask8Bits)  + ((bb>>4)  & mask8Bits)
    bb = (bb & mask16Bits) + ((bb>>8)  & mask16Bits)
    bb = (bb & mask32Bits) + ((bb>>16) & mask32Bits)
    bb = (bb & mask64Bits) + ((bb>>32) & mask64Bits)

    return bb
```

Questa implementazione dell'algoritmo richiede 24 operazioni aritmetiche, ma con alcune ottimizzazioni possiamo scendere a 12:

- Nel contare per coppie posso usare una sottrazione al posto dell'addizione per risparmiare un'operazione di masking:

  ```
  bb = bb - ((bb>>1) & mask2Bits)
  ```

  Possiamo infatti verificare che il conteggio rimane giusto per qualsiasi coppia di bit:

Tabella 1.3: Confronto dei risultati del metodo con addizione e sottrazione per tutte le possibili coppie di bit

bitboard	metodo +	metodo -
00	00 + 00 = 00	00 - 00 = 00
01	01 + 00 = 01	01 - 00 = 01
10	00 + 01 = 01	10 - 01 = 01
11	01 + 01 = 10	11 - 01 = 10

- Nel terzo passaggio abbiamo il conteggio per blocchi da 4 bit e vogliamo ottenere il conteggio per gruppi da 8 bit. Su 8 bit al più 8 sono `1` e quindi il risultato ha i 4 bit più significativi sicuramente `0` (perché 8 in binario si esprime come `1000`). Questo ci permette di applicare la mask sul risultato invece che sui 2 addendi:

  ```
  bb = (bb + (bb >> 4)) & mask8Bits
  ```

- Rimangono da sommare i vari blocchi da 8 bit. Dato che il totale è al più 64 il risultato è al più `10000000` (8 bit in totale). Quindi posso sommare la bitboard a se stessa 8 volte traslata a sinistra di 0, 8, 16, ... , 56 bit in questo modo otterrò il risultato negli 8 bit più significativi:

```
bb = bb + (bb << 8) + (bb << 16) + ...
                              + (bb << 56)
bb = bb >> 56
```

  Questa sostituzione apparentemente rende l'algoritmo più pesante, abbiamo infatti sostituito 12 operazioni aritmetiche con 16, ma notiamo che

$$\begin{aligned} \texttt{bb} + \ldots + (\texttt{bb} << 56) &= \texttt{bb} + \texttt{bb} \cdot 2^8 + \ldots + \texttt{bb} \cdot 2^{56} \\ &= \texttt{bb} \cdot (1 + 2^8 + \ldots + 2^{56}) \end{aligned}$$

  Ovvero le operazioni si riducono ad un singolo prodotto[4] (la somma è una costante che precalcoliamo) e una traslazione.

```
bb = bb * 0x0101010101010101
bb = bb >> 56
```

4: I processori moderni sono molto efficienti nel moltiplicare, possiamo quindi considerare la moltiplicazione un'operazione elementare quando studiamo la complessità dell'algoritmo.

Pseudocodice

Applicando tutte queste ottimizzazioni otteniamo la versione più efficiente dell'algoritmo:

```
int DivideEtImperaPopCount(uint bitboard):
    mask2Bits   = 0x5555555555555555
    mask4Bits  = 0x3333333333333333
    mask8Bits  = 0x0f0f0f0f0f0f0f0f

    bb = bb - ((bb>>1) & mask2Bits)
    bb = (bb & mask4Bits)  + ((bb>>2)  & mask4Bits)
    bb = (bb + (bb >> 4)) & mask8Bits
    bb = bb * 0x0101010101010101

    return bb >> 56
```

Confronto tra i due algoritmi

Dall'analisi teorica della complessità dei due algoritmi segue che il metodo Kernighan ha complessità lineare ed è più efficiente per bitboard sparse (con pochi bit 1), mentre il metodo divide et impera ha costo costante ed è più efficiente per bitboard dense.

Per verificare se questa valutazione è corretta anche nella pratica abbiamo implementato i 2 algoritmi nel linguaggio GO e valutato il tempo medio di esecuzione dell'algoritmo nei 2 casi estremi: una bitboard con un singolo bit acceso e una con tutti i bit accesi.[5]

5: Il codice utilizzato per i test è disponibile nel file `bitboard.go` presente nella repository associata alla tesi

Tabella 1.4: Risultati dei test empirici dei 2 metodi

Algoritmo	Bit accesi	Tempo di esecuzione (ns)
Kernighan	1	1.47
Divide et Impera	1	0.254
Kernighan	64	39.9
Divide et Impera	64	0.238

Notiamo che i risultati sul caso con un solo bit 1 sono in disaccordo con l'analisi teorica, questa differenza è dovuta ad un'ottimizzazione del codice svolta dal compilatore: l'inlining[6] ; infatti disabilitando questa ottimizzazione otteniamo risultati in linea con la teoria:

Tabella 1.5: Risultati dei test empirici dei 2 metodi con inlining disabilitato dalla flag `//go:noinline`

Algoritmo	Bit accesi	Tempo di esecuzione (ns)
Kernighan	1	1.73
Parallel	1	2.80
Kernighan	64	27.3
Parallel	64	2.44

In conclusione nella pratica è sempre conveniente utilizzare il metodo divide et impera nonostante questo sia teoricamente subottimale per bitboard sparse.

6: L'inlining consiste nel sostituire una chiamata ad una funzione con il codice della funzione stessa, in questo modo si evitano i costi legati ad aggiungere e togliere una chiamata dalla stack. Il compilatore attua questa operazione solo per funzioni molto semplici, al metodo Kernighan non viene applicato l'inlining perché contiene un ciclo while.

1.3 Least Significant One Bit

Molti algoritmi che vedremo avranno bisogno di ottenere la lista delle posizioni dei bit 1 in una bitboard, ad esempio per calcolare le mosse possibili in una posizione

dobbiamo ottenere dalla bitboard dei pedoni bianchi le posizioni degli stessi.

Il metodo più efficiente per farlo consiste nel calcolare la posizione del bit `1` meno significativo e poi trasformarlo in `0` come nell'algoritmo di Kernighan, per ottenere tutte le posizioni basta ripetere la procedura finché la bitboard non è vuota.

Dobbiamo quindi implementare una funzione che ci restituisca la posizione del bit `1` meno significativo, generalmente abbreviato LS1B dall'inglese Least Significant One Bit.

Metodo PopCount

Per identificare il LS1B possiamo trasformare la bitboard in modo che siano accesi solo i bit che a partire dal LS1B e poi applicare la PopCount su questa bitboard, il risultato della PopCount è proprio l'indice del LS1B (con indice 1 per la posizione più a destra).

Per effettuare la trasformazione richiesta possiamo sfruttare l'operazione binaria di or esclusivo tra i bit corrispondenti (che indicheremo con `^`):

```
bb:         11010101010011110010000
bb-1:       11010101010011110001111
bb ^ bb-1:  00000000000000000011111
```

Otteniamo quindi un algoritmo compatto per trovare il LS1B:

```
ls1b = PopCount(bb ^ (bb - 1)) - 1
```

Sottraiamo 1 alla fine per avere il risultato espresso contando gli indici a partire da 0.

Metodo de Bruijn

Un metodo alternativo per identificare il LS1B si basa sulle sequenze di de Bruijn [4] .

[4]: Leiserson et al. (1970), «Using de Bruijn Sequences to Index a 1 in a Computer Word»

Si dice sequenza di de Bruijn una sequenza S di 2^n bit tali che ogni possibile sequenza di n bit appare una e una sola volta come sottosequenza contigua di S, ammettendo anche sottosequenze che stanno "a metà" tra la fine e l'inizio.

Esempio

La seguente è una sequenza di de Bruijn di $2^3 = 8$ bit:

```
00011101
```

Infatti le sue sottosequenze sono:

```
000 001 011 111
110 101 010 100
```

Notiamo che traslando verso sinistra i bit di una sequenza di de Bruijn, gli n bit più significativi sono unici per ciascuna traslazione possibile (di 0 bit, di 1 bit, ...).

La prima idea alla base dell'algoritmo per identificare la posizione LS1B è ottenere una bitboard in cui il LS1B è l'unico acceso. Per farlo possiamo sfruttare l'operazione di complemento a due[7] :

7: Il complemento a 2 di un intero n a N bit è $-n \bmod 2^N$, al computer i numeri negativi vengono rappresentati come complemento a 2 del loro opposto

```
bb:          11010101010011110010000
-bb:         00101010101100001111000
bb & (-bb): 00000000000000000010000
```

Possiamo ottenere una bitboard con solo il LS1B acceso semplicemente calcolando `bb & (-bb)`.

Notiamo che moltiplicare una sequenza di de Bruijn per una bitboard con solo un bit acceso equivale a traslare la sequenza a sinistra di un numero di bit corrispondente all'indice del bit acceso. Quindi trovata una sequenza

di de Bruijn, moltiplicandola per `bb & (-bb)` posso poi identificare di quanto questa è stata traslata e quindi l'indice del LS1B leggendo gli n bit più significativi del risultato.

Pseudocodice

Scegliendo una sequenza di de Bruijn a 64 bit e precalcolando una tabella che associa ai 6 bit più significativi la traslazione subita dalla sequenza posso implementare la ricerca del LS1B [5] :

[5]: Google (2021), *Go Standard Library*

```
deBruijn64 = 0x03f79d71b4ca8b09
deBruijn64table = [
    0, 1, 56, 2, 57, 49, 28, 3, 61, 58, 42, 50,
    38, 29, 17, 4, 62, 47, 59, 36, 45, 43, 51,
    22, 53, 39, 33, 30, 24, 18, 12, 5, 63, 55,
    48, 27, 60, 41, 37, 16, 46, 35, 44, 21, 52,
    32, 23, 11, 54, 26, 40, 15, 34, 20,
    31, 10, 25, 14, 19, 9, 13, 8, 7, 6 ]

int LS1B(int bb):
    if x == 0:
        return 64

    shiftedSequence = (bb & -bb) * deBruijn64
    mostSignificantBits = shiftedSequence >> (64 - 6)

    return deBruijn64table[mostSignificantBits]
```

Nell'articolo in cui viene proposto l'algoritmo [4] , questo viene anche confrontato con alcuni algoritmi alternativi, risultando il più efficiente, ma tra questi non vi è il metodo basato su PopCount proposto sopra. Dai nostri test i 2 metodi che abbiamo descritto risultano analoghi nelle prestazioni:

[4]: Leiserson et al. (1970), «Using de Bruijn Sequences to Index a 1 in a Computer Word»

Algoritmo	Tempo di esecuzione medio (ns)
PopCount	0.238
de Bruijn	0.241

Tabella 1.6: Risultati dei test empirici dei 2 metodi per la ricerca del LS1B, calcolati sulla base di 1.000.000.000 esecuzioni.

1.4 Hash di Zobrist

Nell'esplorare le mosse possibili cercando la mossa ottimale avremo molte posizioni che si ripetono (ad esempio perché sono state fatte le stesse mosse ma in ordine diverso); vogliamo quindi conservare i risultati dell'analisi di una posizione in modo da non doverla ripetere.

Per conservare le analisi passate la struttura dati ideale è la tabella hash,[8] abbiamo quindi bisogno di una funzione di hash che trasformi una posizione della scacchiera in un indice della tabella.

8: Una tabella di hash ci permette di memorizzare associazioni chiave-valore. Per farlo salva in memoria un array che contiene i valori inseriti e con una funzione di hash trasforma le chiavi in indici dell'array; il valore associato ad una data chiave viene scritto e letto dall'elemento dell'array all'indice corrispondente alla chiave.

La funzione di hash della posizione ha dei requisiti particolari che giustificano l'utilizzo di un algoritmo specializzato e non una delle funzioni di hash tipicamente utilizzate in altri contesti:

- La funzione deve essere estremamente veloce da calcolare, perché nelle applicazioni tipiche verrà eseguita milioni di volte al secondo;
- È ammesso che in alcuni casi la tabella di hash restituisca il valore sbagliato, purché questi casi siano rari;
- Non è necessario ricalcolare l'hash di ciascuna posizione da zero, ma si può partire dall'hash della posizione precedente e aggiornarlo sulla base della mossa fatta.

[6]: Zobrist (1990), «A New Hashing Method with Application for Game Playing»

Zobrist propone una funzione di hashing specializzata per questo contesto [6] . L'hash di Zobrist richiede di scegliere 781 naturali ad n bit, ciascuno corrispondente ad una delle informazioni necessarie per descrivere univocamente la posizione:

- $12 \cdot 64$ naturali corrispondenti a ciascuna combinazione pezzo-casella, ad esempio uno di questi rappresenta l'informazione "è presente un cavallo nero in A3"
- un naturale indica che giocatore deve muovere
- 4 naturali indicano i diritti di arrocco, ad esempio "il nero può arroccare sul lato di donna"

- 8 naturali indicano la possibilità di en-passant su ciascuna colonna

L'hash di una data posizione sarà il risultato dell'operazione di or esclusivo bitwise (XOR) tra i naturali corrispondenti alle informazioni corrette.

Ad esempio nella posizione seguente l'hash sarà dato dallo XOR dei naturali corrispondenti a "re bianco in d5", "re nero in f3", "regina nera in e2" e "muove il bianco".

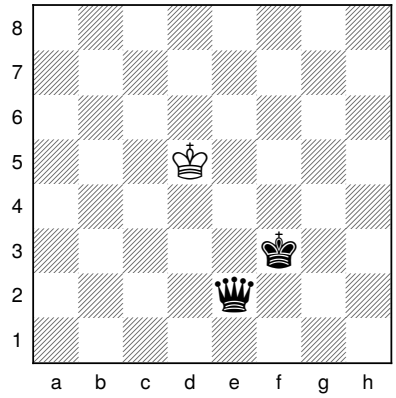

Figura 1.2: Muove il bianco

Il vantaggio dell'hash di Zobrist è che conoscendo l'hash della posizione precedente posso calcolare il nuovo hash facendo l'operazione di XOR solo con gli interi corrispondenti alle informazioni che sono state aggiunte o rimosse. Dato che lo XOR di un intero con se stesso restituisce sempre 0, posso togliere un intero dall'hash semplicemente facendo nuovamente l'operazione di or esclusivo.

Ad esempio per passare dall'hash di Figura 1.2 all'hash della posizione seguente dovrò calcolare lo XOR con i naturali corrispondenti a "re bianco in d5", "re bianco in d6" e "muove il bianco".

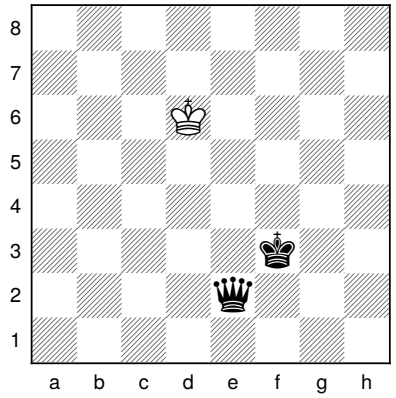

Figura 1.3: Muove il nero

Formalizzazione

Possiamo anche in questo caso interpretare gli interi ad n bit come elementi di $\mathbb{F}_2^n$ e in questo contesto la or esclusivo è semplicemente la somma tra vettori. D'ora in poi con B indicheremo l'insieme dei 781 vettori scelti per rappresentare la posizione.

L'hash della posizione sarà quindi la somma dei vettori corrispondenti alle informazioni che descrivono la scacchiera; per aggiornare l'hash devo sommare le informazioni nuove e sottrarre le informazioni vecchie. Notiamo che ciascun vettore è l'opposto di se stesso[9] , quindi addizione e sottrazione coincidono; di conseguenza, possiamo semplicemente sommare all'hash di partenza i vettori corrispondenti a tutte le informazioni cambiate.

Nelle applicazioni concrete n (la dimensione dello spazio vettoriale) tipicamente sarà 32 o 64, in ogni caso minore di 781, quindi B è un insieme di vettori linearmente dipendenti. Questo implica che ci sono combinazioni lineari diverse che hanno lo stesso risultato, ovvero posizioni diverse che hanno lo stesso hash; chiamiamo questa situazione collisione di tipo 1.

9: Possiamo banalmente verificare che la proprietà vale in $\mathbb{F}_2$ ($0+0=0$ e $1+1=0$), da questo segue che vale anche in $\mathbb{F}_2^n$.

Analisi delle collisioni

Le collisioni sono un problema che deve essere gestito in qualsiasi implementazione delle tabelle hash, l'approccio tipico permette di non leggere mai un dato sbagliato dalla tabella, ma ha un costo in termini di spazio e tempo che non è ammissibile per un engine scacchistico. Negli scacchi il problema viene invece affrontato cercando di ridurre al minimo la probabilità di leggere un dato sbagliato, ma senza portarla a 0.

Nella tabella hash oltre ai dati sull'analisi della posizione salviamo un hash di controllo calcolato analogamente all'hash che usiamo per ottenere l'indice, ma con una scelta dei vettori B diversa.
Per leggere il valore corrispondente ad una chiave ne calcolo l'hash indice, leggo il corrispondente valore nella tabella, verifico se l'hash di controllo salvato corrisponde a quello della chiave; solo in caso affermativo il valore letto è quello corrispondente alla chiave.
Con questo accorgimento posso leggere un dato sbagliato solo quando 2 chiavi diverse hanno lo stesso hash indice e lo stesso hash di controllo.

Poniamoci nel caso pessimo di avere una tabella piena, in questo caso la probabilità di leggere un dato incorretto coincide con la probabilità di collisione dell'hash di controllo; ovvero, supponendo che la funzione di hash sia semplicemente uniforme[10] , la probabilità che il dato sia corretto è:

$$\mathbb{P}(\text{corretto}) = \frac{2^m - 1}{2^m}$$

Quindi supponendo che le letture siano indipendenti tra loro il numero di letture medio prima di avere un errore è 2^m. [11]
Possiamo quindi modulare la frequenza degli errori semplicemente scegliendo il numero di bit m da assegnare all'hash di controllo, ad esempio nell'implementazione associata a questa tesi utilizziamo $m = 42$.

10: Diciamo che una funzione di hash è semplicemente uniforme se, presa per chiave una variabile aleatoria con distribuzione uniforme nel dominio, l'hash della chiave è ancora una variabile aleatoria con distribuzione uniforme.

11: La distribuzione del numero di letture prima di avere un errore è una distribuzione geometrica con probabilità di successo $\frac{1}{2^m}$ ed è noto che la media di una variabile aleatoria con distribuzione geometrica è il reciproco della probabilità di successo.

Scelta della mossa ottima 2

2.1 Definire la mossa migliore

Intuitivamente lo scopo principale di un engine scacchistico è, data una posizione sulla scacchiera, identificare la "mossa migliore"; vogliamo quindi definire in maniera rigorosa cosa significhi "mossa migliore".

Fondamentale per formalizzare questo concetto è la regola della quintupla ripetizione: se una stessa posizione si ripete cinque volte in una partita, allora il risultato è automaticamente patta[1] .

Dato che il numero di posizioni possibili è finito, la regola della quintupla ripetizione implica che esiste un numero massimo di mosse per una partita di scacchi; in particolare gli scacchi sono un gioco finito, ovvero ogni partita ha un numero finito di mosse.
In ogni posizione ci sono un numero finito di mosse possibili, quindi dall'osservazione precedente segue anche che il numero di partite di scacchi possibili è finito.

Definiamo stato della partita l'insieme di tutte le informazioni necessarie a determinare il risultato della partita: posizione della scacchiera, numero di ripetizioni di ciascuna posizione nelle mosse precedenti, quale giocatore deve muovere, ...
Consideriamo ora il grafo avente per vertici tutti i possibili stati della partita e il cui insieme degli archi coincide con le mosse legali in ciascuna posizione; questo è un grafo direzionato e aciclico. Chiameremo foglie del grafo tutti i nodi che non hanno archi uscenti, ovvero i nodi che corrispondono alla fine di una partita.

A ciascuna foglia assegnamo un valore di 1 se ha vinto il Bianco, 0 se è una patta e -1 se ha vinto il Nero. Il

1: La più conosciuta regola della tripla ripetizione invece permette a ciascun giocatore di dichiarare la partita come patta dopo 3 ripetizioni della posizione.

grafo è direzionato e aciclico, quindi possiamo definire il valore di ciascun nodo per induzione a ritroso con la regola minimax: dato un nodo N qualsiasi se è il turno del Bianco allora il suo valore è il massimo tra i valori dei nodi N_i per cui l'arco (N, N_i) appartiene al grafo, analogamente se è il turno del Nero il valore del nodo è dato dal minimo degli stessi. [7]

[7]: Russell et al. (1995), *Artificial Intelligence: A modern approach*

In uno stato N della partita diciamo valore di una mossa il valore dello stato della partita dopo la mossa. Una mossa si dice ottima se è il turno del Bianco e il valore della mossa è massimo tra le mosse legali, oppure se è il turno del Nero e il valore della mossa è minimo tra le mosse legali.

Diversamente da quanto visto sopra, Zermelo ha dimostrato che è possibile definire in maniera rigorosa il concetto di "mossa migliore" anche permettendo partite con infinite mosse [8] . In seguito ignoreremo questa definizione più generale, perché rende praticamente impossibile determinare quale sia la mossa migliore.

[8]: Schwalbe et al. (2001), «Zermelo and the Early History of Game Theory»

2.2 Strategie di tipo A e tipo B

Nel 1949 Claude Shannon pubblica l'articolo "Programming a computer for playing chess" in cui propone la struttura generale per engine scacchistici che viene utilizzata ancora oggi [9] ; più precisamente Shannon propone due classi di strategie per scegliere la mossa: strategie di tipo A e B.

[9]: Shannon (1950), «Programming a computer for playing chess»

Una strategia di tipo A consiste nell'esplorare tutte le mosse possibili per ciascun giocatore fino ad una data profondità, assegnare alla posizione finale di ciascuna variante[2] un valore empirico e ricostruire il valore empirico delle posizioni precedenti in maniera analoga alla definizione data in precedenza.
Nei casi in cui la posizione finale di una variante è una partita conclusa allora il suo valore empirico è uguale al valore teorico (1 se vince il Bianco, 0 se è patta e -1 se vince il nero), in caso contrario la posizione è valutata

2: Negli scacchi una sequenza di mosse è detta variante.

staticamente, cioè senza esplorare le mosse successive, per approssimare il valore teorico. Un esempio di algoritmo per la valutazione statica è calcolare la differenza dei pezzi rimasti a ciascun giocatore.

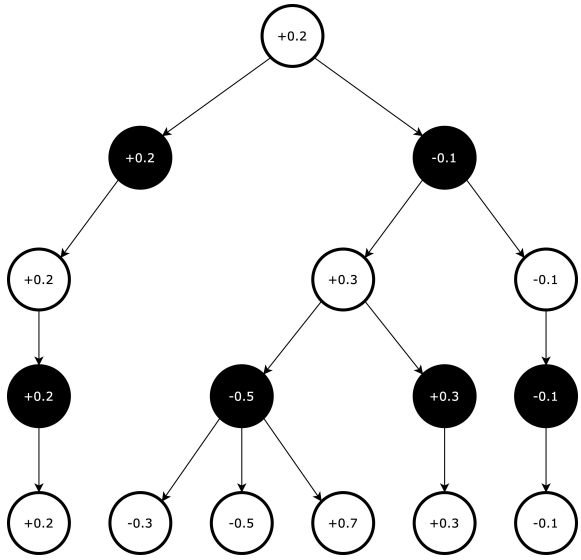

Figura 2.1: Esempio di albero delle mosse possibili fino a profondità 4 con le relative valutazioni.

Le strategie di tipo B sono analoghe, ma non considerano tutte le mosse possibili, bensì solo un sottoinsieme di mosse promettenti.

Il problema fondamentale delle strategie di tipo B è che richiedono regole di decisione molto complesse per determinare le mosse promettenti; il punto debole delle strategie di tipo A è invece che richiedono l'esplorazione di un numero di mosse molto grande. Shannon riteneva che le strategie di tipo B fossero un approccio migliore di quelle di tipo A, in quanto limitano il tempo speso nella valutazione di varianti irrilevanti.

Per i primi decenni di sviluppo degli algoritmi scacchistici vennero adottate quasi esclusivamente strategie di tipo B, ad esempio nel 1966 Greenblatt usando questo metodo creò un computer in grado di competere in torneo con degli umani (il suo ELO massimo fu 1450) [10] . Con lo sviluppo di computer sempre più potenti e di ottimizzazioni sempre migliori (in particolare la potatura alfa-beta) le strategie di tipo B sono diventate sempre meno comuni e oggi tutti i migliori engine usano strate-

[10]: Greenblatt et al. (1967), «The Greenblatt Chess Program»

gie di tipo A, ad esempio DeepBlue[3] usava una strategia di questo tipo [11] .
Negli ultimi anni stanno emergendo vari engine basati su un approccio completamente differente: le ricerche Monte Carlo. I più noti engine di questa categoria sono AlphaZero e Leela Chess Zero. Questi algoritmi hanno avuto risultati molto promettenti, ma non hanno ancora sorpassato gli algoritmi basati su strategie di tipo A: nelle classifiche del Computer Engines Grand Tournament e delle Computer Chess Ratings Lists mantengono le prime posizioni Stockfish, Fat Fritz e Komodo, tutti algoritmi di tipo A.

In questa tesi ci concentriamo sulle strategie di tipo A, ma la maggior parte delle tecniche illustrate sono applicabili anche a strategie di tipo B.

3: DeepBlue nel 1997 vinse contro Garry Kasparov, diventando così il primo engine scacchistico ad aver vinto contro un campione del mondo in carica.

[11]: Campbell et al. (2002), «Deep Blue»

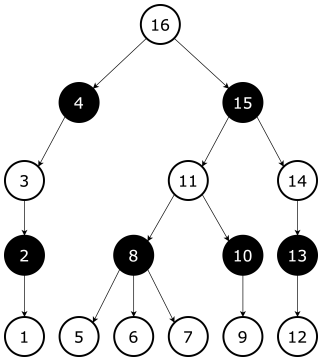

Figura 2.2: Ordine in cui vengono assegnate le valutazioni ai vari nodi dall'algoritmo.

Pseudocodice

Le strategie di tipo A sono tipicamente implementate in maniera ricorsiva come esplorazioni in profondità del grafo delle mosse possibili; segue un esempio di implementazione.
Indichiamo con 1 il Bianco e con −1 il Nero, inoltre per semplicità la funzione restituisce la valutazione della posizione attuale e non la mossa migliore.

```
float positionScore(node, movesLeft, player):
    if isGameFinished(node):
        return gameScore(node)
    if movesLeft == 0:
        return staticEvaluation(node)

    legalMoves = getLegalMoves(node, player)
    if player == 1:
        value = -1

        for move of legalMoves:
            newPosition = playMove(node, move)
            value = max(value, positionScore(
                     newPosition, movesLeft - 1, -1))

        return value
    else:
```

```
        value = +1

        for move of legalMoves:
            newPosition = playMove(node, move)
            value = min(value, positionScore(
                     newPosition, movesLeft - 1, 1))

        return value
```

2.3 Potatura alfa-beta

La potatura alfa-beta è un'ottimizzazione branch and bound[4] per le strategie descritte da Shannon che permette di ottenere la valutazione di una data posizione senza considerare tutte le possibili varianti. [12] [13]

Chiamiamo variante principale una successione delle mosse ottime a partire da una data posizione. La variante principale non è unica, potremmo ad esempio avere 2 mosse diverse per dare matto ed entrambe sarebbero varianti principali.

Dato che il valore di una posizione è definito come il valore della posizione successiva alla sua mossa ottimale, tutte le posizioni della variante principale hanno lo stesso valore.
Il concetto alla base della potatura alfa-beta è che non è necessario valutare una posizione nell'albero delle varianti possibili se siamo sicuri che quella posizione non appartiene ad una variante principale.

La potatura alfa-beta consiste nel tenere traccia, durante l'esplorazione di ciascuna variante dell'albero, del valore α della migliore mossa che il Bianco poteva fare uscendo da questa variante e del valore β dell'analoga mossa per il Nero. Quando valutando una posizione in cui muove il Bianco troviamo una mossa con valore $v > \beta$ siamo sicuri che questa non è la variante principale: se fosse la variante principale allora tutte le mosse della variante principale avrebbero valore v, ma sappiamo che il Nero poteva scegliere una mossa con valore $\beta < v$ e quindi abbiamo un assurdo.

4: Con branch and bound si indicano le ottimizzazioni ad algoritmi di ricerca dell'ottimo in un albero che si basano sull'identificare un intervallo che contiene la soluzione e scartare i rami dell'albero la cui valutazione sicuramente non è contenuta nell'intervallo.

[12]: Fulda (1985), «Alpha-Beta Pruning: They've Seen It Before»

[13]: Marsland (1992), «Computer Chess Methods»

Analogamente quando muove il Nero e $v < \alpha$ la posizione attuale non appartiene alla variante principale.

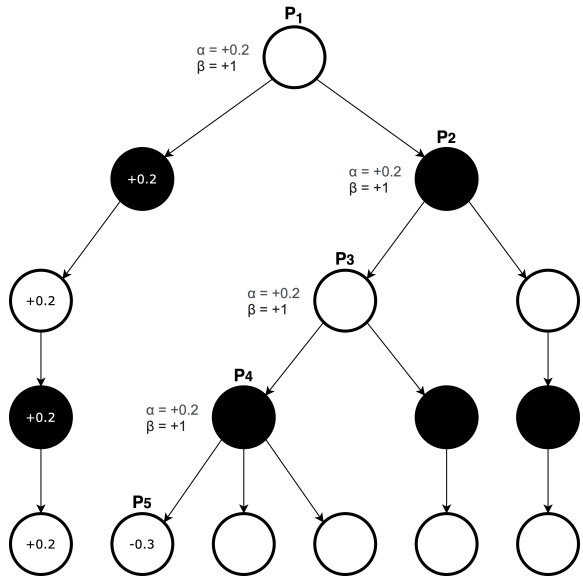

Figura 2.3: Esempio di taglio con la potatura alfa-beta. Il colore del nodo indica il giocatore che deve muovere in quella posizione.

Prendiamo ad esempio l'albero delle varianti di Figura 2.3 e applichiamo l'algoritmo alfa-beta per trovare il valore della posizione P_1. L'algoritmo valuta la prima mossa disponibile in P_1 e ottiene +0.2, quindi aggiorna α a +0.2; successivamente valuta la seconda mossa possibile: ricorsivamente arriva alla posizione P_4 e ad ogni chiamata il nodo padre passa al figlio i valori di α e β. A questo punto considera la prima mossa disponibile in P_4 e ottiene -0.3; in P_4 muove il Nero e $-0.3 < \alpha$ quindi il nodo P_4 non appartiene alla variante principale e l'algoritmo ne interrompe la valutazione, assegnandogli valore -0.3, senza nemmeno considerare le altre mosse possibili. L'algoritmo poi procederà valutando la seconda mossa possibile in P_3.

Pseudocodice

Nella seguente implementazione quando interrompiamo la valutazione della posizione per via di una potatura ritorniamo il valore che ha causato la potatura, questo non è il valore esatto della posizione valutata, ma non è importante perché in una delle

chiamate ricorsive precedenti questa variante verrà comunque scartata.

```
float positionScore(node,movesLeft,alpha,beta,player):
    if isGameFinished(node):
        return gameScore(node)
    if movesLeft == 0:
        return staticEvaluation(node)

    legalMoves = getLegalMoves(node, player)
    if player == 1:
        value = -1

        for move of legalMoves:
            newPosition = playMove(node, move)
            value = max(value, positionScore(
                      newPosition, movesLeft - 1,
                      alpha, beta, -1))

            if value >= beta:
                break

            alpha = max(alpha, value)

        return value
    else:
        value = +1

        for move of legalMoves:
            newPosition = playMove(node, move)
            value = min(value, positionScore(
                      newPosition, movesLeft - 1,
                      alpha, beta, 1))

            if value <= alpha:
                break

            beta = min(beta, value)

        return value
```

La funzione va richiamata passando alfa e beta rispettivamente $-\infty$ e $+\infty$.

2.4 Ottimizzazioni per strategie di tipo A

Il successo delle strategie di tipo A rispetto alle strategie di tipo B è dovuto principalmente al gran numero di ottimizzazioni sviluppate sull'algoritmo di base, queste hanno reso fattibile analizzare una posizione a profondità elevata (>20) in pochi secondi su computer moderni. Tra queste ottimizzazioni alcune rendono l'algoritmo più rapido senza modificarne il risultato finale, mentre altre in alcuni casi portano al risultato sbagliato, ma questi casi sono statisticamente minoritari.

In questa sezione analizziamo una selezione non esaustiva delle ottimizzazioni più utilizzate dagli engine moderni, ad esempio Stockfish utilizza tutte le ottimizzazioni trattate [2] .

[2]: Costalba et al. (2021), *Stockfish Engine source code*

Depth First Iterative Deepening

Tipicamente un engine scacchistico deve giocare con le stesse regole degli scacchi umani e quindi ha a disposizione tempo limitato per giocare la partita; vorremmo quindi imporre all'algoritmo un tempo massimo di esecuzione.
Utilizzando l'algoritmo di esplorazione descritto nella sezione precedente dovremmo stimare la profondità di analisi in modo da sfruttare il tempo a disposizione, ma non eccederlo; nella pratica è sostanzialmente impossibile determinare la profondità corretta. Notiamo che non si può interrompere l'esecuzione dell'algoritmo a metà, perché potremmo non aver ancora trovato la variante ottimale.

Richard Korf descrive un algoritmo di esplorazione del grafo, detto Depth First Iterative Deepening (DFID), che risolve esattamente questo problema [14] . L'algoritmo consiste nel calcolare `positionScore` con profondità crescente, partendo da profondità 1 e incrementando ogni volta di 1; quando finisce il tempo a disposizione

[14]: Korf (1985), «Depth-first iterative-deepening: An optimal admissible tree search»

per l'analisi interrompiamo l'esecuzione e ritorniamo il valore dell'analisi completata di profondità maggiore.

Questo approccio ha lo svantaggio che tutte le esecuzioni prima dell'ultima completata sono tempo di calcolo sprecato; possiamo però dimostrare che, nel caso degli scacchi, la complessità asintotica di DFID è la stessa dell'algoritmo base.

Dimostrazione. Per semplicità supponiamo che il fattore di diramazione b dell'albero, ovvero il numero di mosse possibili in ciascuna posizione, sia una costante.
Calcoliamo quindi il numero N_{base} di nodi esplorati dall'algoritmo base in funzione della profondità d: ciascun nodo dell'albero viene visitato una sola volta, quindi N_{base} è semplicemente la somma del numero di nodi a ciascuna profondità dell'albero.

$$\begin{aligned} N_{\text{base}} &= b^0 + b^1 + \cdots + b^d \\ &= b^d \left(1 + \frac{1}{b} + \frac{1}{b^2} + \ldots \frac{1}{b^d}\right) \\ &\leq b^d \sum_{i=0}^{+\infty} \frac{1}{b^i} \\ &= b^d \frac{1}{1 - \frac{1}{b}} \\ &= O(b^d) \end{aligned}$$

Analogamente calcoliamo N_{DFID}: in DFID ciascun nodo viene visitato una volta per ciascuna esplorazione con profondità maggiore o uguale al suo livello nell'albero.

$$
\begin{aligned}
N_{\text{DFID}} &= (d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d \\
&= b^d \left(1 + \frac{2}{b} + \frac{3}{b^2} + \cdots + \frac{d+1}{b^d}\right) \\
&\leq b^d \sum_{i=0}^{+\infty} \frac{i+1}{b^i} \\
&= b^d \frac{1}{(1-\frac{1}{b})^2} \\
&= O(b^d)
\end{aligned}
$$

□

Nella potatura alfa-beta se esploriamo prima le mosse più forti ci saranno più tagli, perché una ramo dell'albero viene tagliato solo quando è già stata esplorata una mossa migliore. L'algoritmo DFID ci fornisce anche un'euristica per la valutazione di una mossa: la valutazione ottenuta nell'esplorazione precedente.

Durante l'esplorazione dell'albero salviamo quindi la valutazione di ciascun nodo, nell'esplorazione successiva l'algoritmo valuta le mosse possibili in ordine decrescente di valutazione per le mosse del Bianco e crescente per le mosse del Nero. In questo modo molto spesso esploreremo la mossa ottima per prima, avendo quindi il massimo dei tagli.
Con questo accorgimento DFID risulta sperimentalmente più efficiente rispetto ad una singola esplorazione con potatura alfa-beta a profondità massima, perché il costo delle iterazioni precedenti è più che compensato dall'aumento dei tagli nell'esecuzione finale [13] .

[13]: Marsland (1992), «Computer Chess Methods»

Per salvare la valutazione di ciascuna posizione si usa una tabella hash con l'algoritmo di Zobrist descritto nella Sezione 1.4.

Quiescent search

Una debolezza di tutti gli algoritmi trattati precedentemente è che quando raggiungono la profondità massima

nell'albero la posizione viene valutata staticamente, ma questa valutazione può essere completamente errata se ci sono mosse successive che cambiano radicalmente la posizione.

Questi errori di valutazione possono portare l'algoritmo a favorire mosse controproducenti; infatti l'algoritmo preferirà delle mosse di temporeggiamento che spostano una perdita inevitabile ad una profondità oltre l'orizzonte delle mosse esplorate; questo comportamento è detto horizon effect [13] .
Ad esempio piuttosto che perdere una regina subito l'algoritmo preferisce sacrificare inutilmente una torre, che ha un valore inferiore alla regina, e rimandare l'inevitabile cattura della regina ad una mossa non esplorata.

[13]: Marsland (1992), «Computer Chess Methods»

Per limitare questo problema si può implementare una quiescent search: una volta raggiunta profondità massima nell'esplorazione dell'albero, si continuano ad esplorare tutte le mosse che potrebbero cambiare pesantemente la posizione. Nelle implementazioni più semplici si considerano solo le mosse di cattura di un pezzo, ma ci sono anche implementazioni che selezionano le mosse con criteri decisamente più sofisticati [15] .

[15]: Kaindl (1983), «Searching to Variable Depth in Computer Chess»

La scelta di quali mosse esplorare durante la quiescent search deve bilanciare due aspetti: una selezione troppo ampia aumenta eccessivamente la complessità dell'algoritmo, mentre una selezione troppo ristretta non risolve i problemi della valutazione statica.

Scout

L'algoritmo Scout si basa sulla considerazione che, una volta valutata la prima mossa in una data posizione, è più facile dimostrare che le mosse successive sono peggiori della prima piuttosto che calcolare il valore esatto di ciascuna di queste. [16]

[16]: Pearl (1980), «SCOUT: A Simple Game-Searching Algorithm with Proven Optimal Properties»

Chiamiamo v il valore di una data posizione P e v_i i valori delle posizioni dopo ciascuna delle mosse possibili

in P, allora fissato $d \in \mathbb{R}$ qualsiasi valgono le seguenti implicazioni:

1. Se in P deve muovere il Bianco allora $\exists\, i \mid v_i > d \Rightarrow v > d$
2. Se in P deve muovere il Nero allora $v > d \Rightarrow \forall i.\, v_i > d$

Pseudocodice

Sulla base di queste condizioni possiamo costruire una funzione che valuta se $v > d$:

```
boolean scoreIsGreater(node, player, d):
    legalMoves = getLegalMoves(node, player)
    if player == 1:
        for move of legalMoves:
            newNode = playMove(node, move)

            if scoreIsGreater(newNode,-player,d):
                return True

        return False
    else:
        for move of legalMoves:
            newNode = playMove(node, move)

            if not scoreIsGreater(newNode,-player,d):
                return False

        return True
```

Possiamo fare un ragionamento analogo per $v < d$ e quindi definire `scoreIsLesser`.

Per valutare una posizione l'algoritmo Scout valuta ricorsivamente la prima mossa possibile e salva il risultato come d, poi per ciascuna altra mossa verifica se la sua valutazione è migliore di d usando `scoreIsGreater` se muove il Bianco e `scoreIsLesser` se muove il Nero, se la mossa è migliore viene valutata con una chiamata ricorsiva e d viene aggiornato, altrimenti si passa alla mossa successiva. Le foglie dell'albero vengono valutate staticamente come negli altri algoritmi.

Il vantaggio di Scout è che una volta valutata la prima mossa non spreca tempo a valutare precisamente le alternative, ma dimostra solamente che sono inferiori, se però trova una mossa migliore il calcolo di `scoreIsGreater` o `scoreIsLesser` è stato inutile.

Nonostante il rischio di dover riesplorare alcuni nodi, Scout ha complessità asintotica uguale alla potatura alfa-beta anche se le mosse vengono esplorate in ordine casuale [13] . Se invece le mosse sono esplorate nell'ordine dato da un'euristica, come nel caso di DFID, Scout è empiricamente risultato più efficiente di alfa-beta. [16]

[13]: Marsland (1992), «Computer Chess Methods»

[16]: Pearl (1980), «SCOUT: A Simple Game-Searching Algorithm with Proven Optimal Properties»

ProbCut

L'algoritmo ProbCut utilizza un approccio statistico per decidere quando potare un ramo dell'albero: una mossa viene scartata quando la probabilità che la sua valutazione sia nell'intervallo $[\alpha, \beta]$ è minore di una certa soglia. [17]

[17]: Buro (1995), «ProbCut: An Effective Selective Extension of the α-β Algorithm»

ProbCut si basa sul costruire un modello di regressione lineare tra la valutazione v a profondità d di una data posizione e la valutazione v' a profondità $d' < d$ della stessa posizione:

$$v = a \cdot v' + b + e$$

Ci poniamo nelle ipotesi classiche per la costruzione di una regressione lineare, quindi e è una variabile casuale con distribuzione normale $N(0, \sigma^2)$. Dato che la valutazione della posizione tende a convergere quando aumentiamo la profondità di esplorazione, ci aspettiamo $a \approx 1$, $b \approx 0$ e che la varianza σ^2 sia piccola.

Fissando d e d' e analizzando molte posizioni ad entrambe le profondità possiamo ottenere un campione casuale sul quale costruire gli stimatori ai minimi quadrati $\hat{a}$ e $\hat{b}$, questi ci definiscono uno stimatore per v:

$$\hat{v} = \hat{a}v' + \hat{b}$$

Possiamo verificare la correttezza della nostra ipotesi che v e v' siano correlate calcolando il coefficiente di determinazione $\hat{r}^2$, che dovrebbe risultare vicino ad 1.

Dato che possiamo generare un campione casuale arbitrariamente grande, da questo punto in poi supporremo che $\hat{a} = a$ e $\hat{b} = b$. Vale quindi l'uguaglianza $\hat{v} = v - e$ e di conseguenza $\hat{V}$ è una variabile casuale con distribuzione normale $N(v, \sigma^2)$. Possiamo inoltre stimare σ^2 con la varianza campionaria ed ottenere la statistica pivot $\frac{\hat{V}-v}{S_n}$ avente distribuzione t di Student ad $n-1$ gradi di libertà[5] .

5: Nell'articolo in cui viene proposto ProbCut Buro approssima la distribuzione t di Student ad una distribuzione normale; questo è giustificato dal fatto che il campione è molto popoloso.

La costruzione del modello che ci fornisce la statistica pivot $\frac{\hat{V}-v}{S_n}$ è un'operazione molto costosa, ma che deve essere effettuata solo una volta e ci permette poi di fare inferenza sulla base di una singola valutazione a profondità d', operazione dal costo trascurabile rispetto alla valutazione a profondità d.

Consideriamo di voler valutare a profondità d una data posizione in cui muove il bianco e in particolare vogliamo determinare se $v > \beta$. Valutiamo la posizione a profondità d' e otteniamo il valore v', questo ci determina il valore della statistica pivot $\frac{\hat{V}-v}{S_n}$.
Possiamo quindi costruire un test del seguente sistema di ipotesi:

$$\begin{cases} H_0 : \ v \leq \beta \\ H_1 : \ v > \beta \end{cases}$$

Saremo disposti a rigettare H_0, ovvero a tagliare questo ramo dell'albero, quando il p-value è inferiore ad una certa soglia p_{CUT} fissata. Indicata con $F_{t,n-1}$ la funzione di ripartizione della distribuzione t di Student ad $n-1$ gradi di libertà, possiamo calcolare il p-value partendo dalla definizione

$$\begin{aligned}\text{p-value} &= \mathbb{P}\left[\frac{\hat{V}-v}{S_n} > \frac{\hat{v}-v}{S_n} \mid H_0\right] \\ &= \mathbb{P}\left[\frac{\hat{V}-\beta}{S_n} > \frac{\hat{v}-\beta}{S_n}\right] \\ &= 1 - F_{t,n-1}\left(\frac{\hat{v}-\beta}{S_n}\right)\end{aligned}$$

Quindi effettueremo una potatura su una mossa del Bianco, quando $p_{\text{CUT}} >$ p-value, ovvero quando

$$\begin{aligned}p_{\text{CUT}} &> 1 - F_{t,n-1}\left(\frac{\hat{v}-\beta}{S_n}\right) \\ \hat{v} &> S_n F^{-1}_{t,n-1}(1-p_{\text{CUT}}) + \beta \\ v' &> \frac{S_n t_{n,\,1-p_{\text{CUT}}} + \beta - \hat{b}}{\hat{a}}\end{aligned}$$

Analogamente possiamo costruire un test di ipotesi per $v \geq \alpha$ e otteniamo una regola per potare una mossa del nero:

$$v' < \frac{\alpha - S_n t_{n,\,1-p_{\text{CUT}}} - \hat{b}}{\hat{a}}$$

La soglia p_{CUT} va decisa sperimentalmente bilanciando da un lato la necessità di non fare troppi tagli errati, ma al contempo di non tagliare così raramente che il costo della valutazione a profondità d' non sia compensato. ProbCut, al contrario degli algoritmi visti precedentemente, accetta di sbagliare valutazioni in alcuni casi, ma in compenso taglia più rami e quindi può valutare le mosse rimanenti a profondità maggiore.

Una variante migliorata di ProbCut è MultiProbCut, che consiste nel calcolare più regressioni per diverse coppie (d, d') e per diverse fasi di gioco (e.g. apertura, mediogioco e finale). MultiProbCut ha sperimentalmen-

te portato ad un miglioramento quando applicato ad algoritmi scacchistici [18] .

[18]: Jiang et al. (2004), «First Experimental Results of ProbCut Applied to Chess»

Valutazione della posizione

3

In questo capitolo esploriamo brevemente alcune euristiche che vengono utilizzate nella valutazione statica della posizione.
Tipicamente un engine scacchistico utilizza molte euristiche e il valore assegnato alla posizione è una media pesata delle varie valutazioni.

3.1 Differenza di materiale

Il criterio più intuitivo per valutare una posizione è la differenza di materiale tra i due giocatori: ad ogni pezzo viene assegnato un valore e si calcola la differenza tra il valore totale dei pezzi del Bianco e del Nero.

Questo criterio viene utilizzato anche dai giocatori umani nell'analizzare una posizione e tipicamente i valori assegnati per pedone, cavallo, alfiere, torre e regina sono rispettivamente 1, 3, 3, 5 e 9 [19] .
Ciascun engine scacchistico utilizza dei valori leggermente diversi ottenuti tramite sperimentazione empirica. Ad esempio l'algoritmo di Greenblatt utilizza i pesi 1, 3.25, 3.5, 5 e 9.75 [10] ; Stockfish invece utilizza pesi diversi in base alla fase del gioco, quelli per il mediogioco sono 0.77, 3.19, 3.26, 4.96 e 9.84 [2] .

[19]: Wild (2011), *Giocare a scacchi*

[10]: Greenblatt et al. (1967), «The Greenblatt Chess Program»

[2]: Costalba et al. (2021), *Stockfish Engine source code*

Pseudocodice

Questa euristica può essere implementata molto efficientemente sfruttando la rappresentazione della posizione tramite bitboards e l'algoritmo PopCount (vedi Sezioni 1.1 e 1.2).

```
float materialDifference(pos):
    pawnsDiff = KernighanPopCount(pos.whitePawns)
```

```
                -KernighanPopCount(pos.blackPawns)
    knightsDiff = KernighanPopCount(pos.whiteKnights)
                 -KernighanPopCount(pos.blackKnights)
    bishopsDiff = KernighanPopCount(pos.whiteBishops)
                 -KernighanPopCount(pos.blackBishops)
    rooksDiff = KernighanPopCount(pos.whiteRooks)
                -KernighanPopCount(pos.blackRooks)
    queensDiff = KernighanPopCount(pos.whiteQueens)
                 -KernighanPopCount(pos.blackQueens)

    return pawnsDiff + knightsDiff*3.25
              + bishopsDiff*3.5 + rooksDiff*5
              + queensDiff*9.75
```

3.2 Strutture pedonali

Nell'analisi di una posizione gioca un ruolo fondamentale il riconoscimento di strutture pedonali interessanti, sia strutture desiderabili, sia strutture deboli.

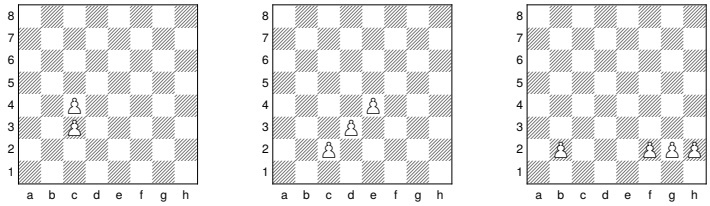

Figura 3.1: Alcune strutture pedonali rilevanti nell'analisi della posizione.

In figura 3.1 vediamo tre esempi di strutture pedonali interessanti [19] :

[19]: Wild (2011), *Giocare a scacchi*

(**a**) Due pedoni sulla stessa colonna si dicono impedonati, questa è una condizione svantaggiosa, perché non possono difendersi a vicenda e il pedone più avanti blocca l'avanzata di quello più indietro

(**b**) Due o più pedoni contigui lungo una diagonale formano una catena di pedoni, questa è una struttura utile, perché ciascun pedone difende il successivo

(**c**) Un pedone che non ha pedoni alleati nelle colonne adiacenti si dice isolato, questa è una debolezza perché quel pedone non può essere difeso da un altro pedone

Possiamo costruire euristiche che analizzano la posizione contando il numero di occorrenze di ciascuna struttura pedonale di interesse. L'utilizzo delle bitboard per rappresentare la scacchiera ci permette di sfruttare le operazioni binarie bitwise per implementare efficientemente queste euristiche.

3.3 Simultaneous Perturbation Stochastic Approximation

Dato che la maggior parte degli engine scacchistici utilizzano varie euristiche contemporaneamente, è necessario un metodo per decidere il peso di ciascuna euristica sul risultato finale. Problema analogo si pone anche per scegliere i pesi utilizzati internamente a ciascuna euristica, ad esempio i valori dei vari pezzi nella differenza di materiale.

Generalmente i pesi vengono inizialmente scelti seguendo il buon senso e successivamente vengono migliorati iterativamente con degli algoritmi di ottimizzazione. Un algoritmo utilizzato per svolgere questa ottimizzazione iterativa di tutti i parametri contemporaneamente è Simultaneous Perturbation Stochastic Approximation (SPSA) [20] .

[20]: Spall (1992), «Multivariate stochastic approximation using a simultaneous perturbation gradient approximation»

Formalmente vogliamo trovare il valore $\theta^\star$ per il vettore dei parametri che massimizza la funzione $L(\theta)$ che rappresenta la bravura dell'engine. Non abbiamo modo di calcolare direttamente $L(\theta)$, possiamo però ottenerne un'approssimazione attraverso una simulazione Monte Carlo nella quale l'engine gioca molte partite contro se stesso o contro degli altri engine; assumiamo che questa simulazione restituisca una misurazione $y = L(\theta) + e$, dove e è una variabile casuale di media 0.
Notiamo che non possiamo usare i tipici algoritmi di ottimizzazione deterministica (e.g. il metodo di Newton-Raphson), perché le misurazioni contengono una componente d'errore.

Ad ogni iterazione SPSA genera un vettore di perturba-

zioni Δ_k con lo stesso numero di componenti di θ_k; le componenti di Δ_k devono essere indipendenti tra loro e le distribuzioni devono avere momenti di ordine -1 finiti; una scelta che soddisfa questi criteri è la distribuzione di Bernoulli ± 1 con probabilità $1/2$ [21] .

[21]: Spall (1998), «Implementation of the simultaneous perturbation algorithm for stochastic optimization»

SPSA calcola due misurazioni $L_k^{(1)} = L(\theta_k + c_k\Delta_k)$ e $L_k^{(2)} = L(\theta_k - c_k\Delta_k)$, dove c_k è uno scalare che dipende dall'implementazione, una scelta empiricamente efficiente è $c_k = c/(k+1)^{1/6}$ con c arbitrario. Da queste due misurazioni SPSA calcola una stima del gradiente di L in $\hat{\theta}_k$:

$$\hat{l}_k\left(\hat{\theta}_k\right) = \frac{L^{(1)} - L^{(2)}}{2c_k} \begin{bmatrix} \Delta_{k,1}^{-1} \\ \Delta_{k,2}^{-1} \\ \vdots \\ \Delta_{k,p}^{-1} \end{bmatrix}$$

Infine SPSA ottiene una nuova stima per $\theta^\star$ muovendosi nella direzione del gradiente stimato:

$$\hat{\theta}_{k+1} = \hat{\theta}_k + a_k\hat{l}_k(\hat{\theta}_k)$$

Nell'equazione precedente a_k è uno scalare che va determinato empiricamente e deve convergere a 0, ad esempio $a_k = a/(A+k+1)^{0.601}$ con a e A scalari arbitrari.
L'algoritmo termina quando si raggiunge un limite di iterazioni o quando le variazioni di θ_k in iterazioni successive diventano molto piccole.

L'algoritmo SPSA permette di ottimizzare contemporaneamente molti parametri dell'engine e per farlo richiede solo due misurazioni per ogni iterazioni, queste caratteristiche lo rendono adatto ad essere utilizzato nel campo degli engine scacchistici. Si può inoltre dimostrare che sotto blande condizioni di regolarità θ_k converge in probabilità a $\theta^\star$ [21] .

[21]: Spall (1998), «Implementation of the simultaneous perturbation algorithm for stochastic optimization»

Conclusione

In questa tesi abbiamo tracciato una mappa dei principali algoritmi utilizzati nella costruzione di engine scacchistici, riorganizzando sistematicamente la frammentata letteratura accademica in merito. Abbiamo inoltre mostrato come in questo campo trovino applicazione varie branche della matematica: da costruzioni algebriche come le sequenze di De Bruijn a metodi di inferenza statistica come in ProbCut e tecniche di calcolo numerico come SPSA.

Come esempio di implementazione degli algoritmi trattati in questa tesi ho realizzato un engine scacchistico nel linguaggio Go, il codice sorgente è disponibile all'indirizzo github.com/ZaninAndrea/chess_engine. Oltre agli algoritmi descritti nei capitoli precedenti ho implementato anche ulteriori aspetti che per brevità non ho inserito nella tesi, ad esempio la generazione delle mosse legali.
L'engine gioca al livello di un giocatore amatoriale umano e la sua debolezza principale è la limitatezza delle euristiche implementate.

Data la complessità esponenziale di analizzare una posizione a profondità k, il problema della costruzione di un engine scacchistico è rimasto rilevante nonostante gli avanzamenti nella potenza computazionale degli ultimi decenni, anzi la maggior parte del progresso in questo campo è attribuibile a miglioramenti degli algoritmi e non dei computer.
In questa tesi abbiamo trattato gli algoritmi basati su strategie di tipo B, che sono stati sviluppati e migliorati continuamente dal 1950 ad oggi; negli ultimi anni però sono diventati rilevanti (seppur non ancora dominanti) algoritmi basati su metodi di Machine Learning e sulla Monte Carlo Tree Search.
Ad oggi i più forti engine adottano un approcio ibrido; ad esempio, Stockfish ha sostituito la sua funzione di valutazione statica con una rete neurale; i metodi che abbiamo descritto sono quindi tutt'ora rilevanti e probabilmente continueranno ad esserlo in futuro.

Bibliografia

[1] Cameron Browne. «Bitboard Methods for Games». In: *ICGA journal* 37 (giu. 2014), pp. 67–84. DOI: 10.3233/ICG-2014-37202 (citato a pag. 2).

[2] Marco Costalba, Tord Romstad e Joona Kiiski. *Stockfish Engine source code*. 2021. URL: https://github.com/official-stockfish/Stockfish (citato a pagg. 3, 24, 33).

[3] Sean Eron Anderson. *Bit Twiddling Hacks*. 2005. URL: https://graphics.stanford.edu/~seander/bithacks.html (citato a pagg. 4, 5).

[4] Charles Leiserson, Harald Prokop e Keith Randall. «Using de Bruijn Sequences to Index a 1 in a Computer Word». In: (feb. 1970) (citato a pagg. 10, 11).

[5] Google. *Go Standard Library*. 2021. URL: https://github.com/golang/go (citato a pag. 11).

[6] Albert Zobrist. «A New Hashing Method with Application for Game Playing». In: *ICGA Journal* 13 (1990), pp. 69–73 (citato a pag. 12).

[7] S. Russell e P. Norvig. *Artificial Intelligence: A modern approach*. Inglese. Prentice-Hall, 1995. Cap. 5.2 (citato a pag. 18).

[8] Ulrich Schwalbe e Paul Walker. «Zermelo and the Early History of Game Theory». In: *Games and Economic Behavior* 34.1 (gen. 2001), pp. 123–137 (citato a pag. 18).

[9] Claude E. Shannon. «Programming a computer for playing chess». In: *London Edinburgh Philos. Mag. J. Sci.* 41.314 (1950), pp. 256–275. DOI: 10.1080/14786445008521796 (citato a pag. 18).

[10] Richard D. Greenblatt, Donald E. Eastlake e Stephen D. Crocker. «The Greenblatt Chess Program». In: *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*. AFIPS '67 (Fall). Anaheim, California: Association for Computing Machinery, 1967, pp. 801–810. DOI: 10.1145/1465611.1465715 (citato a pagg. 19, 33).

[11] Murray Campbell, A.Joseph Hoane e Feng-hsiung Hsu. «Deep Blue». In: *Artificial Intelligence* 134.1 (2002), pp. 57–83. DOI: https://doi.org/10.1016/S0004-3702(01)00129-1 (citato a pag. 20).

[12] Joseph S. Fulda. «Alpha-Beta Pruning: They've Seen It Before». In: *SIGART Bull.* 94 (ott. 1985), p. 26. DOI: 10.1145/1056313.1056315 (citato a pag. 21).

[13] T. Antony Marsland. «Computer Chess Methods». In: *Encyclopedia of Artificial Intelligence*. A cura di Inc. John Wiley & Sons. USA, 1992 (citato a pagg. 21, 26, 27, 29).

[14] Richard E. Korf. «Depth-first iterative-deepening: An optimal admissible tree search». In: *Artificial Intelligence* 27.1 (1985), pp. 97–109. DOI: https://doi.org/10.1016/0004-3702(85)90084-0 (citato a pag. 24).

[15] Hermann Kaindl. «Searching to Variable Depth in Computer Chess». In: gen. 1983, pp. 760–762 (citato a pag. 27).

[16] J. Pearl. «SCOUT: A Simple Game-Searching Algorithm with Proven Optimal Properties». In: *AAAI*. 1980 (citato a pagg. 27, 29).

[17] Michael Buro. «ProbCut: An Effective Selective Extension of the α-β Algorithm». In: *ICGA journal* 18 (giu. 1995), pp. 71–76. DOI: 10.3233/ICG-1995-18202 (citato a pag. 29).

[18] A. X. Jiang e M. Buro. «First Experimental Results of ProbCut Applied to Chess». In: *Advances in Computer Games: Many Games, Many Challenges*. A cura di H. Jaap Van Den Herik, Hiroyuki Iida e Ernst A. Heinz. Boston, MA: Springer US, 2004, pp. 19–31. DOI: 10.1007/978-0-387-35706-5_2 (citato a pag. 32).

[19] Alexander Wild. *Giocare a scacchi*. Edizioni Ediscere, 2011 (citato a pagg. 33, 34).

[20] J.C. Spall. «Multivariate stochastic approximation using a simultaneous perturbation gradient approximation». In: *IEEE Transactions on Automatic Control* 37.3 (1992), pp. 332–341. DOI: 10.1109/9.119632 (citato a pag. 35).

[21] J.C. Spall. «Implementation of the simultaneous perturbation algorithm for stochastic optimization». In: *IEEE Transactions on Aerospace and Electronic Systems* 34.3 (1998), pp. 817–823. DOI: 10.1109/7.705889 (citato a pag. 36).