

Gli algoritmi di ricerca

PARTE 3

In questo articolo vedremo come Matilde esegue la ricerca in una struttura ad albero per recuperare la mossa migliore.

di **Andrea Lanza** > alanza@infomedia.it

Gli algoritmi utilizzati nella programmazione scacchistica utilizzano la classica struttura ad albero. In pratica, partendo da una determinata posizione, vengono generate tutte le mosse possibili (generazione completa) a profondità 1, poi per ogni mossa successiva si generano tutte le mosse a profondità 2 e così via, fino al raggiungimento della profondità stabilita o fino a quando non si esaurisce il tempo a nostra disposizione (la **Figura 1** rappresenta una parte dell'albero delle mosse). In questo paragrafo prenderemo in considerazione due

algoritmi di ricerca analizzandone le differenze:

- NegaMax
- *AlphaBeta* Pruning

L'algoritmo NegaMax esegue una ricerca esaustiva dell'albero delle mosse.

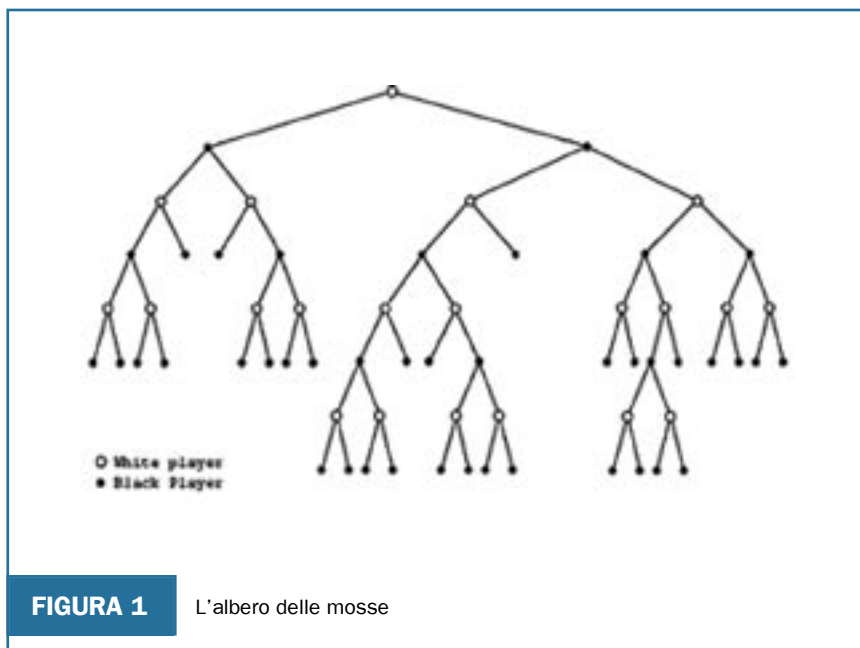
Tutti i nodi dell'albero vengono analizzati ed in teoria è possibile eseguire una scansione dell'intero albero per trovare la mossa migliore da giocare.

In pratica questa soluzione si rivela impossibile non essendo a tutt'oggi un calcolatore in grado di risolvere completamente il gigantesco albero delle mosse di una partita di scacchi.

La soluzione è quindi quella di cominciare la ricerca da una determinata posizione e scendere in profondità fino a quando non si esaurisce il tempo a nostra disposizione ed applicare una funzione di valutazione (la vedremo in dettaglio nel prossimo articolo) che restituisca un punteggio per ogni nodo dell'albero.

Vediamo ora l'algoritmo in pseudocodice:

```
Public Function NegaMax(Profondità as integer) as integer
    Dim Migliore as integer
    Dim I as integer
    Dim Valore as integer
    Migliore = -Infinito
    If Profondità <= 0 then
        NegaMax = <Funzione di Valutazione>
```



```

Exit Function
End if
<Genera Mosse Legali>
For I = 1 to <Numero Mosse>
  <Fai la Mossa>
  Valore = -NegaMax(Profondità - 1)
  <Ritira la Mossa>
  If Valore > Migliore then
    Migliore = Valore
  End if
Next I
NegaMax = Migliore
End Function

```

Questa funzione nega il valore restituito, in quanto ad ogni ricorsione cambia il colore del giocatore.

Come risultato avremo che tanto più il valore sarà positivo e tanto più la mossa risulterà buona per il giocatore bianco; viceversa ad una mossa negativa corrisponderà una buona mossa per il giocatore nero.

La funzione può essere richiamata nel nostro codice in questo modo:

```
Valore = NegaMax(4)
```

In questo caso il nostro motore si spingerà fino a 4 livelli di profondità.

Questo algoritmo appena descritto trova pochissime applicazioni in pratica; il fatto di eseguire una ricerca esaustiva (cioè una scansione di tutti i nodi dell'albero) comporta una limitata profondità di ricerca in quanto l'albero delle mosse cresce in maniera esponenziale.

Una variante molto più efficiente di questo algoritmo è la cosiddetta funzione *AlphaBeta*.

Questa funzione elimina i nodi dell'albero che sono ininfluenti nella ricerca, rendendo la stessa molto più veloce. Alpha e Beta rappresentano i limiti minimo e massimo della nostra finestra utile; il valore assegnato al nodo sarà tenuto in considerazione se andrà a cadere entro questo intervallo.

Se il valore sarà minore di Alpha, il nodo e tutto il suo sottoinsieme saranno eliminati in quanto avremo trovato una brutta mossa.

Se il valore sarà maggiore di Beta, anche in questo caso tutta la ramificazione del nostro albero al di sotto di questa mossa sarà da buttare in quanto il nostro avversario sicuramente non terrà da conto questa mossa essendo per lui molto sfavorevole.

Questo è l'algoritmo *AlphaBeta* descritto in pseudo-codice:

```

Public Function AlphaBeta(Profondità as integer,
  Alpha as integer, Beta as integer) as integer
  Dim I, Valore as integer
  If Profondità <= 0 then
    AlphaBeta = <Funzione di Valutazione>
  Exit Function
End if
<Genera Mosse Legali>

```

```

For I = 1 to <Numero Mosse>
  <Fai la Mossa>
  Valore = -AlphaBeta(Profondità - 1, -Beta, -Alpha)
  <Ritira la Mossa>
  If Valore >= Beta then
    AlphaBeta = Beta
  Exit Function
End if
If Valore > Alpha then Alpha = Valore
Next I
AlphaBeta = Alpha
End Function

```

Questa funzione sarà richiamata nel seguente modo:

```
Valore = AlphaBeta(4, -Infinito, Infinito)
```

Anche qui la ricerca si fermerà al quarto livello di profondità, ma il tempo impiegato sarà nettamente inferiore ed avremo ottenuto lo stesso identico risultato.

Un aspetto molto importante da tener presente è che l'efficienza dell'algoritmo *AlphaBeta* è strettamente correlato con un buon ordinamento delle mosse.

Se noi utilizziamo questo algoritmo senza ordinare in maniera efficiente le mosse generate dal nostro motore, avremo un'efficienza paragonabile a quella dell'algoritmo NegaMax; al contrario, con un buon ordinamento la velocità della nostra funzione aumenterà notevolmente, in quanto avremo ottime probabilità di avere un taglio nel nostro albero di ricerca dopo le prime 5, 6 mosse, contro la media di 35 che avremmo utilizzando l'algoritmo NegaMax.

La Quiescenza

Supponete che la nostra ricerca debba terminare a livello di profondità pari a 5 e che, proprio al quinto livello la nostra Regina catturi la Torre avversaria.

Benissimo, il nostro motore restituirà una buona valutazione in modo che questa mossa venga giocata.

Peccato però che a livello 6 (cioè immediatamente successivo), l'Alfiere avversario mangi la nostra Regina.

Questo errore viene chiamato "effetto orizzonte", cioè l'impossibilità di vedere le catture ai livelli successivi perché siamo stati costretti a fermarci al livello prestabilito.

La soluzione per ovviare a questo inconveniente ci viene data dalla *Quiescenza*.

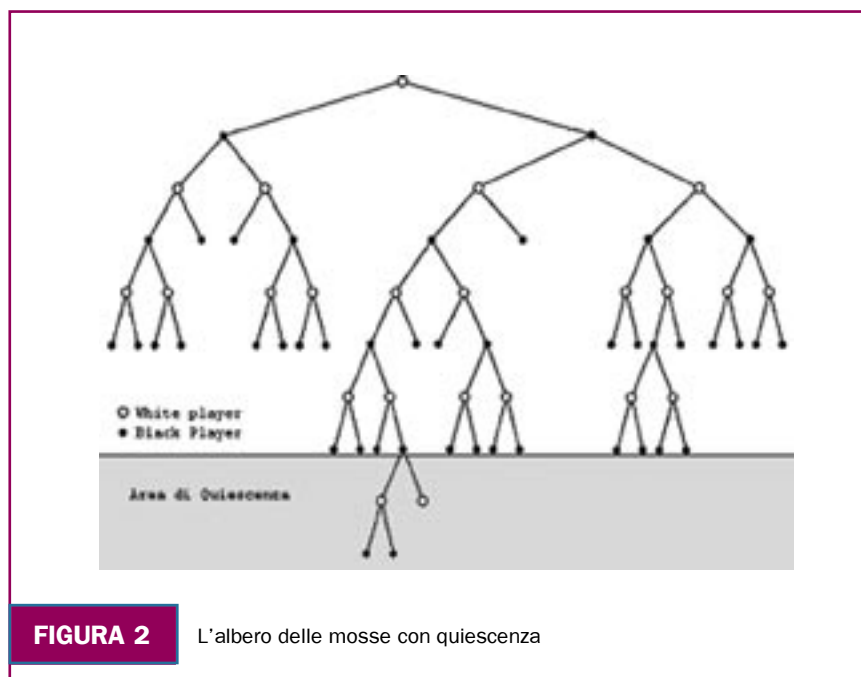
La classica *Quiescenza* coinvolge solo le mosse dove sono presenti delle catture; essa sarà chiamata al posto della nostra funzione di valutazione in modo da spingersi ulteriormente in profondità fintantoché tutte le catture non siano state risolte.

Un esempio di albero con *Quiescenza* è quello di **Figura 2**:

Come detto in precedenza questa funzione sarà richiamata all'interno del nostro algoritmo *AlphaBeta* al posto della nostra funzione di valutazione.

Di seguito è descritto il codice della *Quiescenza*:

```
Public Function Quiescenza(Alpha as integer,
```



grande dispendio di risorse è quello che in inglese viene chiamato *MVV/LVA* (Most Valuable Victim/Least Valuable Attacker).

In questo caso le mosse saranno ordinate in base al pezzo utilizzato per catturare e il pezzo catturato; ad esempio PxR (Pedone cattura Regina), CxR (Cavallo cattura Regina), ecc...

A seguire PxT (Pedone cattura Torre), CxT (Cavallo cattura Torre), ecc...

A queste mosse sarà assegnato un punteggio, utile in questo caso solamente per l'ordinamento (non è il punteggio della funzione di valutazione); alla mossa PxR (Pedone cattura Regina) sarà assegnato il punteggio più alto, a quella RxP (Regina cattura Pedone) quello più basso.

Altre mosse da tenere in considerazione per l'ordinamento a mio avviso sono sicuramente quelle che portano ad una

promozione (ad esempio, se il Pedone promuove a Regina catturando la Regina avversaria, questa mossa deve essere ordinata per prima).

Ci sono anche altri algoritmi di ordinamento come ad esempio lo SEE (Static Exchange Evaluation) che producono un ordinamento più efficiente scartando quelle catture che alla fine risultano inutili, ma l'algoritmo da implementare è molto più complicato del precedente.

Iterative Deepening e gestione del tempo

Generalmente quando avviamo la funzione di ricerca non impostiamo una profondità fissa da raggiungere, ma continuiamo a scendere in profondità nell'albero delle mosse fino a quando non si esaurisce il tempo a nostra disposizione, dopodichè eseguiamo la nostra funzione di valutazione che restituirà in pratica la mossa migliore trovata fino a quel momento.

Un metodo per gestire il tempo che ci viene fornito dall'interfaccia Winboard è quello nell'utilizzo dell'Iterative Deepening.

In pratica eseguiamo un ciclo "For" da 1 a infinito dove avvieremo la funzione di ricerca, prima fino a livello di profondità 1, poi 2, poi 3 ecc... Quando il tempo a disposizione sarà terminato usciremo dal ciclo e prenderemo la mossa migliore trovata fino a quel momento:

```

For I = 1 to Infinito
  Valore = AlphaBeta(I, -Infinito, Infinito)
  If <Tempo Esaurito> then
    Exit for
  End if
Next I
    
```

Questa tecnica si rivela estremamente efficiente nonostante il fatto (sicuramente lo avrete già notato) che questa funzione sia ripetitiva (1, 1-2, 1-2-3, 1-2-3-4, 1-2-3-4-5 ecc...).

```

Beta as integer) as integer
Dim I, Valore as integer
Valore = <Funzione di Valutazione>
If Valore >= Beta then
  Quiescenza = Beta
  Exit Function
End if
If Valore > Alpha then Alpha = Valore
For I = 1 to <Numero Catture>
  <Fai la Mossa>
  Valore = -Quiescenza(-Beta, -Alpha)
  <Ritira la Mossa>
  If Valore >= Beta then
    Quiescenza = Beta
    Exit Function
  End if
  If Valore > Alpha then Alpha = Valore
Next I
Quiescenza = Alpha
End Function
    
```

Come avrete notato in questa funzione non esiste una profondità massima da raggiungere, questo perché la quiescenza termina dopo aver analizzato tutte le catture.

Di conseguenza anche qui è importante dare un ordine alle catture generate, per evitare che la nostra funzione perda in efficienza.

L'ordinamento delle mosse

Come abbiamo visto in precedenza, sia l'algoritmo *Alpha-Beta* che la *Quiescenza* sono strettamente legate ad un buon ordinamento delle mosse.

È pertanto fondamentale nella realizzazione del nostro software ordinare le mosse prima di iniziare la nostra ricerca mettendo al primo posto le mosse che effettuano le catture migliori.

Un metodo abbastanza efficace e che non richiede un

In realtà, alla fine di ogni iterazione del nostro ciclo, le mosse del livello appena analizzato saranno ordinate mettendo al primo posto la mossa migliore ricavata a quella data profondità.

In pratica noi eseguiamo la ricerca a livello 1; una volta finita la ricerca ordiniamo le mosse a quel livello; poi eseguiamo la ricerca fino a livello 2 ma in questo caso ci troveremo le mosse del livello precedente già ordinate, aumentando la probabilità di tagli all'interno del nostro albero di ricerca.

La gestione del tempo utilizzata da Matilde è quella che in Inglese viene chiamata "Sudden Death".

Si utilizza una costante che rappresenta le mosse che ci restano da fare (supponiamo che siano 20).

Il tempo residuo a nostra disposizione viene sempre diviso per questa costante; supponiamo di giocare una partita dove i giocatori hanno ciascuno un'ora di tempo a disposizione.

Bene, la prima mossa sarà giocata da Matilde dopo esattamente 3 minuti ($60 \text{ minuti} / 20 = 3 \text{ minuti}$).

A mano a mano che diminuisce il tempo a nostra disposizione le mosse saranno giocate in maniera sempre più rapida (ad esempio, con 1 minuto a disposizione avremo $60 \text{ secondi} / 20 = 3 \text{ secondi}$).

Questo sistema induce a pensare maggiormente all'inizio e a metà partita, mentre avremo maggiore rapidità nel finale dove generalmente si hanno pochi pezzi a disposizione.

Conclusioni

Con questo articolo abbiamo descritto che cos'è l'albero delle mosse e come viene utilizzato dalla funzione di ricerca per recuperare la mossa migliore.

È stato applicato il metodo "AlphaBeta Pruning", cioè quell'algoritmo che permette di eliminare i nodi inutili dell'albero e di aumentare quindi la profondità di analisi.

È stato inoltre illustrato come evitare gli errori dovuti al cosiddetto "Effetto Orizzonte" e come utilizzare al meglio il tempo a nostra disposizione per permettere al nostro motore di calcolare la mossa migliore.

Bibliografia

- [1] Bruce Moreland, "Programming Topics",
<http://www.brucemo.com/compchess/programming/index.htm>

Andrea Lanza

Lavora da 18 anni come programmatore in un'azienda che si occupa di analisi ambientali. Utilizza il Visual Basic 6, il Delphi e il Visual C++ per realizzare programmi per la gestione di sistemi di rilevamento ambientali e Gas Serra in ambienti Windows e Linux. È membro del GSei (Gruppo Scacchi e Informatica), dedicato allo studio e sviluppo di software scacchistico.